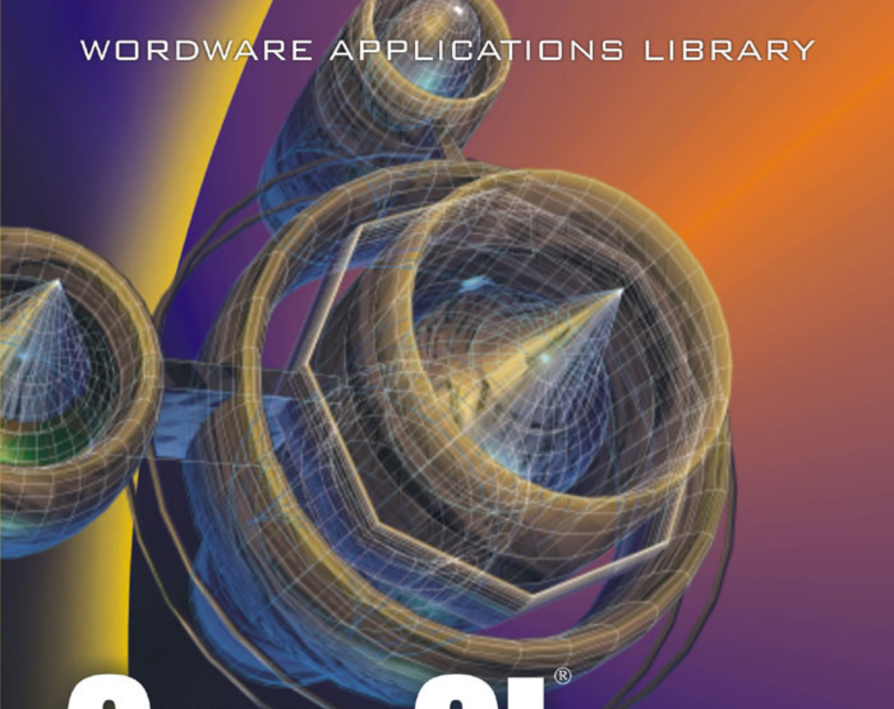
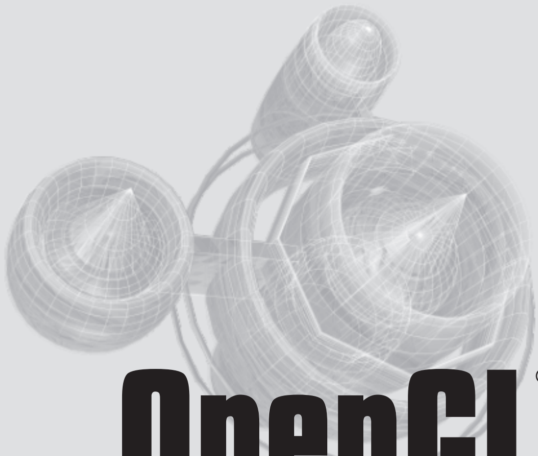
WORDWARE APPLICATIONS LIBRARY



OpenGL® Game Development



OpenGL® Game Development

Chris Seddon

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Seddon, Chris.
OpenGL game development / by Chris Seddon.
p. cm.
Includes bibliographical references and index.
ISBN 1-55622-989-5 (pbk.)
1. Computer games--Programming. 2. OpenGL. 3. Computer graphics. I. Title.
QA76.76.C672S43 2004
794.8'1526--dc22 2004024961
CIP

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-989-5

10 9 8 7 6 5 4 3 2 1
0411

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

To Tim, Mom, and my wonderful wife, Dina.

This page intentionally left blank.

Contents

Introduction

Objectives

OpenGL is the industry standard for 3D graphics. The SDK has been used in many leading applications, games, CAD, and military and medical simulations over the past eight years. In *OpenGL Game Development*, we'll explore the immense 3D processing power of OpenGL while writing a basic map editor and 3D video game. Why not focus on the game itself and forgo the map editor? Because, although writing the game's 3D engine is important, it is also a good idea to know how to write an application that uses OpenGL to draw data onto the screen.

With the knowledge gained from this book, you'll be able to write your own 3D first-person shooter video games using some of the advanced features of your video card. You'll also be able to load 3D models exported from 3D Studio MAX (using the .ase format), add 3D positional surround sound into your levels, create head-to-head multiplayer games, and much more.

System Requirements

The system requirements for the book's source code are fairly standard. A video card that supports OpenGL version 1.3 or higher is desired. Because some of the features the book discusses are not part of the OpenGL standard, it would be a good idea for you to update your video card drivers before running the advanced game demos.

Other requirements:

- 64 MB of RAM (or higher)
- 100 MB of disk space (or higher)
- A DirectX-compatible sound card for the audio components

Software Requirements

This book was written and tested with Microsoft Visual C++ 6.0 Professional. The source code included in the book should recompile with newer Microsoft compilers; however, you might need to create new workspaces.

Library Requirements

You will need the following libraries installed before running the game engine source code components discussed in the book:

- OpenAL 1.0 (or higher) library
- Microsoft DirectX 8.0 (or higher) library

Companion Files

The companion files can be downloaded from www.wordware.com/files/openglgd. The "ex" directories contain the scource code and executables for the examples discussed in the book, and the media directory contains resources needed to complete some of the examples.

Abbreviations

The following abbreviations are used in this book:

2D — two-dimension

3D — three-dimension

AI — artificial intelligence

GLU — GL Utility library

GLUT — GL Utility Toolkit

MMORPG — massively multiplayer online role-playing game

NPC — non-player character

OpenAL — Open Audio Library

OpenGL — Open Graphics Library

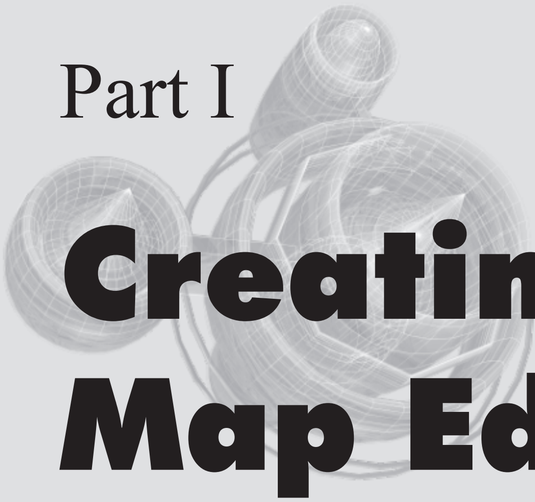
RAM — random access memory

RGB — red, green, blue

RGBA — red, green, blue, alpha

SGI — Silicon Graphics, Inc.

UI — user interface



Part I

Creating the Map Editor

In Part I, we will learn how Windows applications are created, how to add interfacing controls, i.e., buttons, menus, and dialog boxes, and how to use common controls such as saving controls.

We will also learn about the Open Graphics Library (OpenGL), how to initialize the API for use in Windows, and finally, how to draw basic objects.

Chapter 1

Introducing Windows Programming

In Chapter 1, we will learn about event-driven programming and how it differs from conventional programming, the basic structure of a Windows program, how to add buttons and menus, how to add functionality to buttons, what resources are and how to use them, and finally, how to use common controls in dialog boxes.

By the end of the chapter, the beginning structure for our map editor will take shape and you will have the knowledge to write basic Windows applications.

Concepts of Event-Driven Programming

Event-driven programming is a simple but powerful concept. When something is done, the computer will send an event to the handler, which will process the event and do the actions associated with it. In Windows programming, event-driven programming is used to create graphical user interfaces (GUIs) without the painstaking work of polling for every possible click, press, and move the program allows. Instead, the software will automatically send a message saying, “Event X has happened.” A simple example is where the developer has a button that will play a sound when pressed. In some cases, the software specifications may require that the developer continuously check if the button is pressed; however, when programming with events, the software could automatically send an event message to the message handler saying, “Button X is down,” in which case the sound would be played.

Unfortunately, the Windows operating system has more than just a ButtonDown event. In a typical Windows program you could use dozens of different events to create the perfect application or game. Almost every corner of the Win32 graphical user interface has been covered with message events, from program activation and keypresses to program display painting and background timers. And just when you thought there wasn’t a specified event, many Windows libraries have the capability of creating their own specific Windows message events.

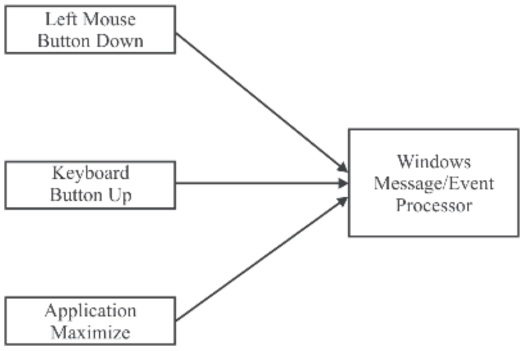


Figure 1.1: Event flow

In the Win32 SDK, we use the Window Procedure (WndProc) callback for message handling. The declaration will be discussed in more detail later in this chapter.

Introducing the Win32 SDK

When developing Windows applications in Microsoft Visual C++, there are two different methods we can use to create the application. The first method is using the Microsoft Foundation Classes (MFC) framework. MFC is a robust object-oriented library that encapsulates a large portion of the Windows application programming interface (API) into a C++ class. Generally, when developing smaller applications, MFC code is bulkier. For simplicity, in this book we will use the alternative method of writing applications for Windows using the Win32 Software Development Kit (SDK). The Win32 SDK in some cases is less robust feature wise, but is far easier for beginners to understand.

When using the Win32 SDK, there are two different types of applications you can create: Win32 console and Win32. A Win32 console application is similar in design to a regular C program in DOS. You can use the Win32 SDK functionality within the console application; however, it must be run from within an MS-DOS prompt, as opposed to straight DOS. A Win32 application is a regular Windows graphical application.

Creating Win32 applications is much different from creating the typical C program that many programmers are familiar with. A good example is the main entry point of code in a C program. In Windows applications, there is no main but rather a WinMain, and instead of optionally supplying the parameters to the WinMain, it is mandatory. The return type for the WinMain is still of the int type; however, we must add the standard calling convention to the Windows API. The default parameters are very different. There are now four parameters: two of the HINSTANCE type, an LPSTR, and an int. With many Win32 variables you'll notice an H in the front of the name. For instance, the HINSTANCE type is a handle of an instance. An instance is the program itself. For the first two parameters we use the HINSTANCE type with the names hInstance and hPrevious, respectively. The hInstance is the instance of the current program running, and the hPrevious in 32-bit applications is always NULL. The third parameter is of the LPSTR type. The LPSTR type is a pointer to a null-terminated string of 8-bit characters. The variable passes program arguments when started. This is similar to the argv variable in regular C programs, with the exception that it does not include the program name. The final parameter defines how the program should be displayed when started.

The following code snippet displays the main entry point for Windows applications.

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
           int CmdShow)
```

Now that we have a main, we're almost ready to start building a Windows application. But first, we must discuss which headers to include. Rather than including the standard I/O (stdio.h) and standard library (stdlib.h), we need to include the Windows header (windows.h). This header includes links to all relevant headers to compile a basic Windows program. With that in mind, let's create a new Windows application and use the code snippet from above to begin programming. The following code snippet shows the main entry point for a Windows program.

```
#include <windows.h>

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
           int CmdShow)
{
   return (1);
}
```

This code will compile in Visual C++; however, we are far from creating a fully functional Windows application. Before creating a fully functional Windows program, we must register a window class, which defines the attributes of the window.

Registering a Window Class

The window class contains details such as the class name, menus, the callback for messages, and several other pieces of data. To declare a window class, we use the WNDCLASS type. A typical window class would look like the following.

```
WNDCLASS wc;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
wc.hInstance     = hInstance;
wc.lpfnWndProc   = WndProc;
wc.lpszClassName = "ME";
wc.lpszMenuName  = NULL;
wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
```

Starting from the top, we set the variable cbClsExtra to 0 because we do not want to allocate extra bytes of memory beyond the WNDCLASS size. The cbWndExtra variable is set to 0 because we do not need extra memory allocated for the window instance. The hbrBackground variable sets the background color for the window. We use the function GetStockObject to retrieve a predefined stock brush, which in this case was light gray, and cast it as an HBRUSH type. The HBRUSH is a handle to a brush object. Rather than using LTGRAY_BRUSH, we could have used one of the following colors:

- Dark gray (DKGRAY_BRUSH)
- Black (BLACK_BRUSH)
- White (WHITE_BRUSH)

However, light gray is one of the routine background colors, so let's stick to the basics.

The hCursor variable sets the default cursor for the window. In our case we load a default cursor using the function LoadCursor. This function requires two parameters, the first being the current instance, which should be NULL, and the second being the resource ID of the cursor, which in this case is IDC_ARROW. The hIcon variable sets the default icon for the application. To set this variable we must use the function LoadIcon, which returns the handle of an icon. The first parameter for the LoadIcon function is the current program instance, which is NULL. The second parameter is the resource ID of the icon, which is IDI_APPLICATION. In both the LoadIcon and LoadCursor functions, the first parameter is of the HINSTANCE type. We set this first parameter as NULL to indicate we want to use the standard Windows icons/cursors rather than specifying custom resources. The hInstance variable requires the handle of this instance, which is the hInstance from our WinMain.

The lpfnWndProc variable is the pointer to the window procedure, which is our message handler. As discussed before, our message handler will process any messages sent by the system. The lpszClassName variable is the name of the window class, which in our case is ME, short for Map Editor. Later, when we are going to create a window, we will need to reuse this name, so it is a good idea to keep it simple. The next variable, lpszMenu-Name, controls the menu that is to be loaded for the window class. We do not need this parameter, so we can set it to NULL. And finally, the last variable in the WNDCLASS structure we must fill in indicates the style of the window class. A wide variety of options can be used for the style. I've chosen the basic values of CS_HREDRAW and CS_VREDRAW, which will redraw the screen when the width or height of the client window has been changed, and I've also requested CS_OWNDC, which gives each registered class a unique device context.

Now that all the fields in the WNDCLASS structure are filled in, it is time to register a new window class by calling the function RegisterClass and passing the wc variable we just filled in. If the registration of the class is successful, a non-zero number will be returned. If the function fails, 0 will be returned. With this in mind, the next piece of source code attempts to register a window class and displays an error message if it fails.

```
if (!RegisterClass(&wc))
{
  MessageBox (NULL,"Error: Cannot Register Class", "ERROR!", MB_OK);
  return (0);
}
```

As you can see, registering the window class is fairly simple. To display the error message, I simply used the MessageBox function to display a basic error message. The MessageBox function requires four parameters: a handle to a window (HWND), two null-terminated strings of the LPCTSTR type, which include the main message and the caption text respectively, and an int for the type. We don't need any elaborate message boxes for the time being, so a basic MB_OK or OK button message box will suit our current needs.

Creating a Window

After registering the window class we must create a window on the screen using the CreateWindow function. There are 11 simple parameters in this function, the first being the LPCTSTR lpClassName, which as the parameter name suggests is our class name, "ME". The second parameter is another LPCTSTR called lpWindowName, which is a string identifying the name of the window in the title bar. For this parameter we will use the value "Map Editor". After the lpWindowName parameter, we set the style of the window using DWORD styles. There are many options to choose from when setting the style of the window; we will use the default options WS_OVER-LAPPEDWINDOW and WS_VISIBLE for simplicity's sake. The WS_OVERLAPPEDWINDOW style creates a window with a title bar and Minimize and Maximize buttons. The WS_VISIBLE style allows the window to be visible upon creation. The next two parameters are of the int type and identify the starting X and Y coordinates for the window. After the starting locations are the width and height of the window as the int type. The eighth parameter is hWndParent, which is of the HWND type. This parameter is used when a child window must be created. Since this window is the parent, we can keep this parameter as NULL. The next parameter is of the HMENU type. If we wanted to load a menu we would use the LoadMenu function and return a value into this parameter; however, we do not need this functionality at the moment and therefore this parameter should be NULL for the time being. Later we will discuss how to create, display, and add functionality to menus. After the hMenu parameter we include the current instance, which in our case is hInstance. The final parameter is an LPVOID called lpParam. This parameter is used when creating a multiple-document interface (MDI). This is beyond the topics discussed in this book and therefore we will make this parameter NULL.

Now that we've filled in the CreateWindow function, it is a good idea to declare our handle of a window variable at the top of the source code as a global variable. In the global section we will declare an HWND called Window. We will place our CreateWindow code after the RegisterClass function is called.

```
if (!RegisterClass (&wc))
{
  MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
  return (0);
}

Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE, 0, 0,
          640, 480, NULL, NULL, hInstance, NULL);
```

It is a good idea to test the returned window value for NULL after calling the CreateWindow function. If the creation of the window fails, NULL is returned and we must display an error message and exit the program. As with all errors, we will use the MessageBox function to display the error and return 0 to indicate an error occurred. The following code displays our window creation error checking.

```
if (Window == NULL)
{
  MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
  return (0);
}
```

The Main Loop

After creating the window we must write the main loop for the software. The main loop we will create will run indefinitely. With each pass through the loop we will check for a WM_QUIT message, which we will program as a break out of the loop. To accomplish this we must first declare a variable called msg of the MSG data type. The MSG data type contains information about messages being sent from the program. We will use the variable as one of the parameters to peek for messages. In the while loop we will have a hardcoded true condition (1) to allow it to continuously loop. Within the loop we will use the function PeekMessage to check for messages in the message queue.

The PeekMessage function has five parameters that need to be filled in. The first is the address of the msg variable, and the second is the handle of the window we are getting messages from. This value will be NULL. The next two parameters are unsigned integers that specify the range of the messages we can use, which in our case will be 0 as the default. And finally, the fifth parameter is an unsigned integer that describes how the message will be handled. There are two options for the value: PM_NOREMOVE, which does not remove the message from the message queue, and PM_REMOVE, which removes messages from the queue. We will use the latter of the two. When PeekMessage is called and a message is in the queue, it is put into our msg variable and a non-zero number is returned. If there is no data in the queue, the return value is 0.

When PeekMessage succeeds we must check to see if the message sent is WM_QUIT, which is the designated quit message for ending our program. A simple if statement comparison between the unsigned integer message member of the msg variable and WM_QUIT would suffice; if the values are equal, use the break keyword to break out of the while loop. If the message is not WM_QUIT, then we must call the TranslateMessage and DispatchMessage functions, which will translate a message from virtual keypresses to character messages and dispatch the new message to the window procedure. The main-loop code when completed will look like the following:

```
while (1)
{
   if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
   {
      if (msg.message == WM_QUIT) break;
      TranslateMessage(&msg);
      DispatchMessage (&msg);
   }
}
```

The Message Handler

The next section that needs to be addressed before our application will compile is the message handler. As stated earlier, the message handler processes any messages such as keypresses, mouse movements, button clicks, and other types of interactivity. The return value for the message handler is of the LRESULT type, which is a 32-bit integer. We also use the CALLBACK data type in place of FAR PASCAL. The message handler has four parameters. The first is of the HWND type and is a handle to the current window.

The second parameter is an unsigned integer (UINT), which contains the type of message being sent. The third parameter is a WPARAM data type, which is a 32-bit integer. This variable contains additional information about the messages being sent. The fourth and final parameter in the message handler is an LPARAM data type. The LPARAM is similar to the WPARAM in that it is a 32-bit variable that contains additional message information. With the return type being LRESULT, it is a good idea to return the returned value from the DefWindowProc function. The DefWindowProc function will ensure that all messages are processed either through their respective message handler or through a default message. Each message produces different return values; therefore using this function will ease the burden of message return values and default message handling. The parameters for the DefWindowProc function are exactly the same as for the WndProc callback, making it easy to remember the parameters for the function.

Handling messages in the message handler is fairly simple. We use a switch statement on the msg variable to control each message. The first message we will handle will be the WM_DESTROY message, which is sent by the application when the "X" button is clicked in the top-right corner of all Windows applications. This message does not necessarily mean we have to quit the program; however, it is good to follow general standards and add the quit functionality. To quit our application we will use the function PostQuitMessage, which will send a WM_QUIT message to the application. The only parameter required for PostQuitMessage is an integer, which is the error code you'll return. In our case this number will be the magical number 0. The message handler is now complete and the code for it is shown below.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}
```

With the addition of the Windows message handler we have finished our first application.

Your First Windows Application

Now that we've finished our first application, it is time to review our source code and try out the program. In many cases throughout this book, I won't provide the complete source code for each example due to size constraints (although all the examples are provided in the downloadable file available at www.wordware.com/files/openglgd); however, this is the base code for both our map editor and game engine, so it is essential to discuss this source code in the example.

As you can see, this example is exactly how we wrote the individual sections earlier in this chapter. Our header information only requires the windows.h file for the moment; however, this will change soon enough. Below the header information are our global variables, which at the moment is only Window. The message handler (WndProc) is below the global variables. And finally, we've put WinMain at the bottom of the source code. You can alternatively put it at the top of the application if you write the appropriate function prototypes before any functions are defined. As a personal preference I like the WinMain at the bottom of the main source file, because I find it's easier to build the software from the ground up (bottom of the source code) as opposed to top down (top of the source code). Both styles accomplish the same goals in the end, but the latter requires function prototypes for the source code to compile.

Example ex1_1.cpp:

```
#include <windows.h>

HWND Window;

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
           int CmdShow)
{

   WNDCLASS  wc;
   MSG       msg;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
```

```
      wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
      wc.hInstance     = hInstance;
      wc.lpfnWndProc   = WndProc;
      wc.lpszClassName = "ME";
      wc.lpszMenuName  = NULL;
      wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
      if (!RegisterClass(&wc))
      {
         MessageBox (NULL,"Error: Cannot Register Class", "ERROR!", MB_OK);
         return (0);
      }

      Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                  0, 0, 640, 480, NULL, NULL, hInstance, NULL);

      if (Window == NULL)
      {
         MessageBox (NULL,"Error: Failed to Create Window", "ERROR!", MB_OK);
         return (0);
      }


      while (1)
      {
         if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
         {
            if (msg.message == WM_QUIT) break;
            TranslateMessage(&msg);
            DispatchMessage (&msg);
         }
      }

      return (1);
   }
```

Compiling Notes

As you can see, the source code for our first application is rather simple now that all the structures and functions have been explained. To compile and test the application, use the ex1_1 example located in the source directory of the downloadable files (available at www.wordware.com/files/openglgd). If you are using Visual C++ 6.0 or higher, there is a workspace file available that has all the project information for quick compiling.

Adding Windows

As noted earlier in the chapter, many parts of a graphical user interface are handles to a window. Objects such as buttons, slide bars, combo boxes, list boxes, and edit boxes are all created using the same HWND data type as a regular window. Furthermore, we still use the CreateWindow function to create the window. The primary difference between writing interface windows and regular windows is that you must specify the window type in the first parameter (ClassName). Once the window is created, other initialization functions may need to be called depending on the type of window you have created.

When declaring any interface windows, I have a general rule of declaring the variables with their associated function in mind. Similar to Hungarian notation, this focuses more on the mechanics of what the variable does as opposed to what the variable is, in terms of data types. For instance we'll be creating a button called "Create Wall." A logical name for this button would be bCreateWall. The lowercase b would indicate it's a button, and the action it would perform would be creating a wall, hence bCreateWall. I have found this simple technique fairly useful when writing programs where I have text boxes, statics, and buttons that all used the same function. In this situation, rather than having the variables create_wall_static, create_wall_edit, and create_wall_button, we simply have the letters s, b, or e in front of the CreateWall function. With all this in mind, let's begin adding interface windows!

Buttons

The next window data type handle we'll add to our map editor is a button. Although it doesn't sound very exciting, this is a major step toward writing an event-driven program and, more importantly, useful functionality in our map editor.

Before we create a button, it is a good idea to define the default dimensions (i.e., width and height) as constants. This will allow us to have a default look for each button without hardcoding values, which could be annoying to update manually. Our new constants section will contain the dimension definitions and will look like the following:

```
#define DEFAULT_BUTTON_WIDTH   100
#define DEFAULT_BUTTON_HEIGHT  20
```

The values chosen for the dimensions simply looked good in the example, so I kept them. After defining the default dimensions, we need to declare a new window handle that will be our new button. We will declare our new button in the globals section, underneath the Window variable. Remember that buttons, just like other interface windows, are regular handles of a window and we can use the HWND data type to declare them.

Our new button will be called bCreateWall. If you haven't guessed by now, the button will create a wall in the map editor, and follows my general rule for naming variables. With that in mind, our new globals section will look like the following:

```
HWND Window;
HWND bCreateWall;
```

After declaring the bCreateWall variable, we need to use the CreateWindow function to create the physical button. We could simply copy and paste the previous CreateWindow function call we used to create our window; however, many of the parameters must be changed to create the button.

Obviously, the return parameter for the CreateWindow function must be bCreateWall rather than Window, so the handler can be altered later on. The first parameter must be changed to "BUTTON" to indicate our handle of a window will be a button as opposed to the default, which is a window. The second parameter, once again, is the caption of the handle, which in this case is "Create Wall". When specifying a button or any other type of interface handle, we must use the WS_CHILD window style to allow the handle to be "attached" to the window. If this is not declared, the button may be spawned in another window because the program will interpret it as a separate button from the main window. Any window-specific styles such as WS_OVERLAPPEDWINDOW will be automatically disregarded when "BUTTON" is specified in the first parameter; however, we still need WS_VISIBLE to allow the button to be seen.

The fourth and fifth parameters declare the starting X and Y coordinates, which locate the button at 0 pixels horizontally and 100 pixels vertically. The left side of the application will be our function bar and will contain buttons for specific functionality we'll be adding later. The sixth and seventh parameters are the width and height of our button, which were filled in using our default button constants DEFAULT_BUTTON_WIDTH and DEFAULT_BUTTON_HEIGHT. We use these two constants to give us the freedom to change the size without using the time-consuming method of changing all the values and because it adds standardized dimensions for all buttons.

The final parameter that must be changed is the eighth parameter, which was originally NULL. This parameter is the handle of the parent window. With the first CreateWindow call we are creating the parent window and therefore we cannot specify one. In this new CreateWindow function call we are creating a child and can specify the Window variable as the parent.

This concludes the discussion of how to create buttons in a Windows program. Next, we'll learn how to add functionality to a button click.

Adding Functionality to Button Clicks

Button clicks are the fundamentals of event-driven programming. When a button is clicked, we can program something to happen. I've discussed this earlier in the chapter and it's time to put those words into action by writing your first event for the button bCreateWall.

When the user presses our newly created bCreateWall button, a WM_COMMAND message is sent to our WndProc message handler. The WM_COMMAND message is sent to the message handler on several events, such as when a menu item is selected from a menu, a button is clicked, or an accelerated keystroke is translated. To begin, we must add the WM_COMMAND message to the case statement below the WM_DESTROY message that is already present. It is a good idea to write a new function called WMCommand to handle the messages as opposed to handling them directly in the case statement. For smaller projects, handling the message may be easier in the case statement; however, mid-sized to larger projects will have many buttons to check and will therefore clutter the otherwise simple message handler. For this reason we will write a WMCommand function that will contain exactly the same parameters as the WndProc message handler. To check for the bCreateWall button click, we can write a simple if statement casting the bCreateWall handle as an LPARAM data type, which converts the data of bCreateWall to a message parameter and compares it to lParam. If the data is equal, then the button is clicked and we'll display a message using the MessageBox function. The following source code displays the WMCommand function and checks to see if the bCreateWall button is clicked.

```
void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (lParam == (LPARAM)bCreateWall) MessageBox (Window, "You Pressed
             bCreateWall", "Congrats!", MB_OK);
{
```

Next we must add the WM_COMMAND message to our message handler so when any WM_COMMAND messages are sent they will be redirected to the WMCommand function for processing. We can do this by adding the WM_COMMAND message below the WM_DESTROY message and add the WMCommand function with the proper parameters to the condition, allowing any WM_COMMAND messages to be processed through WMCommand when they are sent. The final code for the example (ex1_2.cpp) has been provided for reference.

Example ex1_2.cpp:

```
#include <windows.h>

#define DEFAULT_BUTTON_WIDTH   100
#define DEFAULT_BUTTON_HEIGHT  20

HWND Window;
HWND bCreateWall;

void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (lParam == (LPARAM)bCreateWall) MessageBox (Window, "You Pressed
             bCreateWall", "Congrats!", MB_OK);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
          int CmdShow)
{

   WNDCLASS   wc;
   MSG        msg;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "ME";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);
   }

   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
         0, 0, 640, 480, NULL, NULL, hInstance, NULL);
```

```
    bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE,
              0, 100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window,
              NULL, hInstance, NULL);

    if (Window == NULL)
    {
       MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
       return (0);
    }


    while (1)
    {
       if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
       {
          if (msg.message == WM_QUIT) break;
          TranslateMessage(&msg);
          DispatchMessage (&msg);
       }
    }

    return (1);
 }
```

The example is fairly simple in design. The button dimensions are defined and the button is declared, then created. When the button is clicked, the WM_COMMAND message is sent, which calls the WMCommand function to process any possible button clicks. The WMCommand finds that bCreateWall has been clicked, and therefore the MessageBox function is called to display the message “You Pressed bCreateWall” on the screen.

Next we’ll be discussing resources, what they are, and why we want to use them.

Resources

A resource allows the programmer to include data in the binary executable of a program. Resources can contain user interface-related resources like dialog boxes, menus, and toolbars, along with general-use data like bitmaps, icons, and cursors. The programmer can also insert user-defined data into the resource and load data dynamically from the executable. It is much easier to distribute one executable than a mound of bitmaps and icon files. Another reason a programmer may decide to use resources for raw data is for security. Although it is still possible to take the raw data out, the average computer user will not have the knowledge of how to do so.

The most commonly used resources for applications are dialog boxes, menus, and icons. Beyond those resource types, we will also make use of the cursor type and make a customized cursor for one of the functions of the map editor.

There are two methods to create a resource file: by hand or with the Visual C++ resource editor. Both methods produce exactly the same results, but one is less complicated than the other. Typing the resource file source by hand allows the developer to write exactly what he or she wants with little to no overhead. This process can be tiresome in large projects where menus and dialog boxes are rather complicated in their design. For example, a simple File menu in an application looks like the following:

```
IDR_MAINMENU MENU DISCARDABLE
BEGIN
   POPUP "&File"
   BEGIN
      MENUITEM "&Open",  ID_OPEN
      MENUITEM "&Save",  ID_SAVE
      MENUITEM SEPARATOR
      MENUITEM "&Print", ID_PRINT
      MENUITEM SEPARATOR
      MENUITEM "E&xit",  ID_FILE_EXIT
   END
END
```

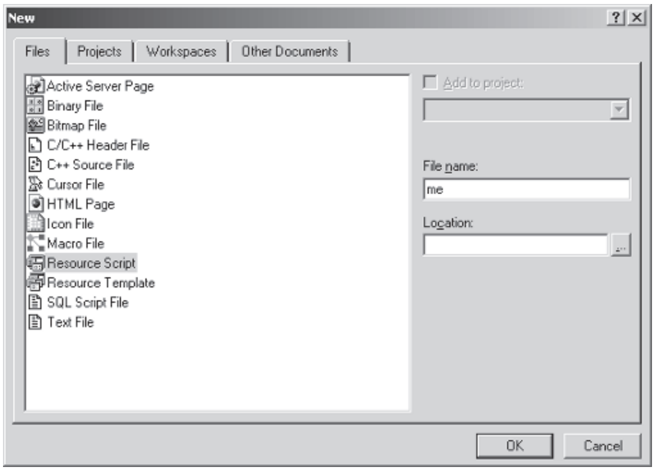
Each menu must have a begin/end pair. Each menu can have menu items such as “Open” or “Save” along with pop-ups, which move data into a subsection of the pop-up name. You can also add separators, which are horizontal lines to break up items in each menu. These help organize a pop-up window into common tasks such as I/O, configuration, and exiting. Each menu item has a unique identifying number associated with it, which is generally defined in a resource.h. This identifying number is used when you need to perform many tasks on that resource.

Now imagine you’re writing the next blockbuster first-person shooter. A professional map editor isn’t going to have one menu with four menu items. In some cases there could be 20 menus with 25 to 50 items each. Some of those items may have pop-ups to other menus as well. This could become very time consuming to organize and type by hand. Although some developers like writing code manually, it would become chaotic at times to update and add additional features.

The alternative method to writing resources by hand is through the use of the resource editor included in Visual C++. The resource editor is a drag-and-drop interface that writes all the resource code and definitions for you, while you add details to each resource. The downside to using the resource editor over the traditional typing method is that it adds a lot of overhead to each resource. The overhead includes things such as comments, compiler directives, and other non-required text in the file. If you’re never going to open the resource in a text editor, it will never make a difference for you. If you like looking at the code from the resource file to understand the commands, it may be slightly confusing. There is plenty of documentation in the Microsoft Developer Network (MSDN) libraries on resource files. In this book we will focus on using the resource editor because it's the easiest and quickest solution for us when making our resources.

The Resource Editor

To use the resource editor we must create a resource script file in Visual C++. To create the file, click the **File** menu item, followed by **New**. Select the **Resource Script** option when prompted with choices of new files to create. In the File name section, type **me**. Figure 1.2 displays the New dialog box with the proper information filled in.

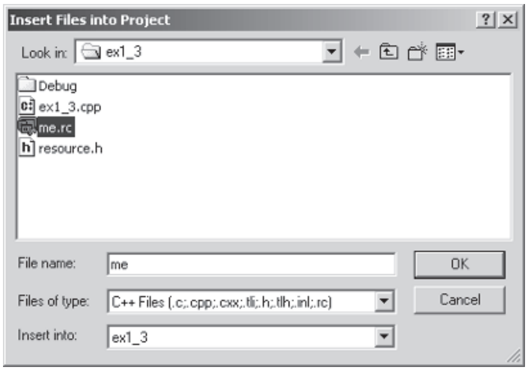


File name: me

Location:

Figure 1.2: New resource script

To confirm the creation of the file, click the **OK** button. The resource script file has now been created. Before we can use the file we must include it in the Visual C++ workspace. A resource file must be included into the workspace so it can be compiled and the program can access its contents. To include the resource file, simply click the **Project** menu item, scroll down to the option **Add To Project**, and then click the **Files** item. A dialog box will appear with the title Insert Files into Project. Change the Files of type combo box to **resource *.rc**, and select the file **me.rc**. Figure 1.3 displays the dialog box used to select our resource file.



File name: me

Files of type: C++ Files (.c;.cpp;.cxx;.tli;.h;.tlh;.inl;.rc)

Insert into: ex1_3

Figure 1.3: Selecting our resource file

Click the **OK** button to finalize the operation, and the resource file will be included in the workspace. If you have successfully included the resource file, the file should be located among the files in the workspace window. Generally it appears in either the Resource Files section or the Source Files section. You can access the resource data by simply double-clicking on the file.

Adding Menus

The first resource we will add to our resource file is a menu. The menu is one of the most important resources we will use because it allows us to add functionality to an otherwise boring application. This first example will add a menu with an exit option. To start building the menu, double-click on the **me.rc** file in the workspace. The resource editor will pop up with the resource filename as a folder. Figure 1.4 displays our empty resource script.

By default the resource script is empty. Any default resource can be added in the resource editor by simply right-clicking on the resource filename and selecting **Insert**. A dialog box will pop up, displaying the types of resources you can add. In this case, select the **Menu** option and click the **New** button to finalize the operation. Figure 1.5 displays the Insert Resource dialog box.

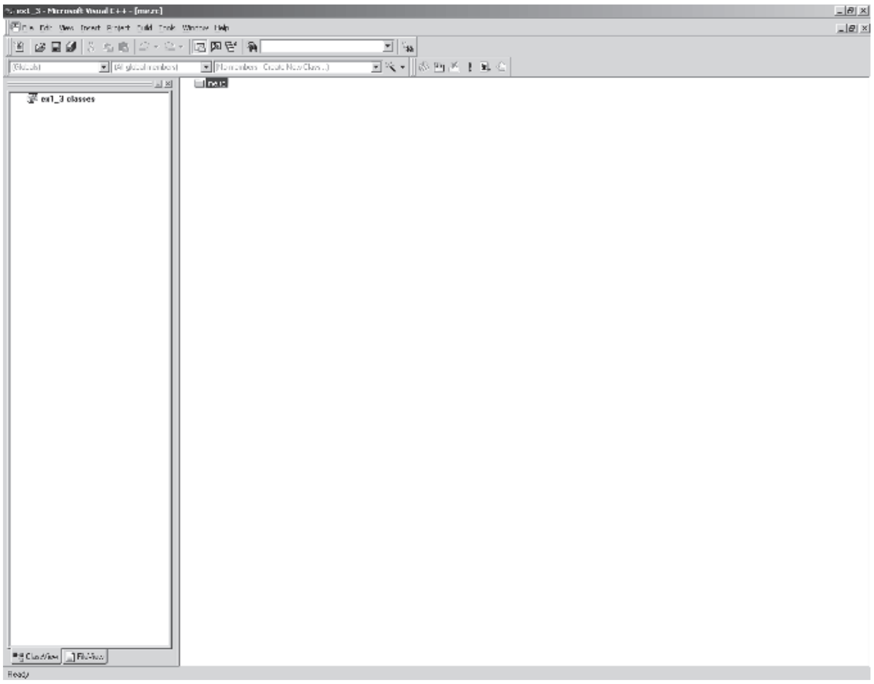


Figure 1.4: An empty resource script

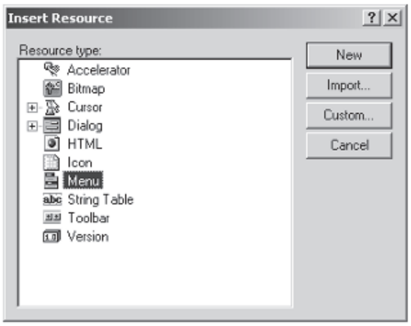


Figure 1.5: Selecting a resource type

Our menu for the map editor has been created. The next step is to add menu items and pop-ups to the menu, allowing functions to eventually be put in place. Notice our newly created menu resource is completely blank. There are two types of menu items you can add to the root menu of a menu resource: a regular menu item and a pop-up item. A regular menu item has a unique value associated with it that allows the user to track when the item has been clicked. The pop-up item creates a submenu with the name given. By clicking or highlighting the item, a menu will appear with other items available within it. This allows the programmer to organize menus neatly.

A separator is also available that can be added to pop-up items to break up items within a specific menu without resorting to having multiple pop-up menus. Separators are commonly used to help organize multifunction menus, such as the File menu, into sections including Save/Open, Import/Export, Printing, and Exit options. Obviously, the programmer could write the application without the separators, but by separating the data, you can sort items into logical groups. There are no rules or guidelines as to how menus should be created; however, it is a good idea to keep things as simple as possible so the user doesn't get frustrated by a poor application layout.

Adding Item Click Functions

With all this in mind, let's add our first menu item by double-clicking the highlighted item in the resource editor. A dialog box will appear and prompt you for a caption for the menu. The default item type is a pop-up, which is what we want, and the caption should be set to **&File**. The ampersand in front of the F represents a character hotkey for that menu item. When the resource is compiled and run, the user can press Alt+F to bring up the options for the File menu item instead of clicking the item. This is not specific to pop-up items and can be used in regular items, allowing quick access for functionality. Press **Enter** when you've typed the caption, and the new menu item File will appear with space for adding menu items.

Add a menu item to the File menu following the same procedure, this time setting the caption to **E&xit**. Before pressing Enter and accepting the changes, make sure the Pop-up option is *not* checked and then press **Enter**. Your Exit menu item should appear, and we're now ready to write the source code to handle resource menu item clicks.

When a menu item is clicked, it sends the WM_COMMAND to the message handler. The wParam variable contains the ID value of the menu item clicked. As discussed earlier, the ID value is a unique ID that identifies the specific menu item. A message is not sent when the menu item has the pop-up option flagged true. In that situation a new set of menus would be displayed rather than sending the message.

In this example we'll add functionality to the Exit menu item located in the File menu. When the menu item is clicked, the program will send the WM_COMMAND message with wParam as ID_FILE_EXIT. The program will then compare to see if wParam equals ID_FILE_EXIT (which it does) and proceed through the if statement. In this case, the program will use the function PostQuitMessage to send a generic quit message. This will allow the program to exit, giving it the desired functionality.

A new header has been included in this example. The new header file is resource.h. This file contains all the relevant information required to use our resources. More specifically, the resource.h file contains the values for all the unique IDs in the resource script and must be included in the source code to allow the compiler to know the value of every unique ID.

Example ex1_3.cpp:

```
#include <windows.h>

#include "resource.h"

#define DEFAULT_BUTTON_WIDTH   100
#define DEFAULT_BUTTON_HEIGHT  20

HMENU   Menu;
HWND    Window;
HWND    bCreateWall;

void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (lParam == (LPARAM)bCreateWall) MessageBox (Window, "You Pressed
             bCreateWall","Congrats!", MB_OK);
   else if (wParam == ID_FILE_EXIT) PostQuitMessage(0);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
          int CmdShow)
{

   WNDCLASS  wc;
   MSG       msg;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "ME";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);
   }

   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
               0, 0, 640, 480, NULL, NULL, hInstance, NULL);


   bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE,
               0, 100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window,
               NULL, hInstance, NULL);

   Menu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_MENU));
   SetMenu (Window, Menu);

   if (Window == NULL)
   {
      MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
      return (0);
   }


   while (1)
   {
      if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
      {
         if (msg.message == WM_QUIT) break;
         TranslateMessage(&msg);
         DispatchMessage (&msg);
      }
   }

   return (1);
}
```

Congratulations! You've created your first menu. With this information you are on your way to writing your first map editor. The next section will discuss how to checkmark and uncheckmark specific menu items.

Checkmarking Items

The checkmarking functionality is a simple way of displaying an enabled function among many options. In the case of our map editor, we'll need the option of displaying the game content in either wireframe or solid color mode. (Both drawing types will be discussed in detail later in the book.) By default, the wireframe drawing type will be selected; the solid color will be unchecked. If the user selects the solid color item, we must uncheck the wireframe drawing type and check the newly select option, solid color. The same is true for wireframe; if it's clicked, then the wireframe option is the newly selected item.

Before continuing we'll need to add a new pop-up menu called Drawing to our existing menu. Within this new menu we create two items: Wireframe and Solid. The function CheckMenuItem is used to check and uncheck a specific menu item. The first parameter in the function is a handle to a menu (our Menu variable), which contains the currently loaded menu. The second parameter is the ID of the item to be modified. In this case we'll use our newly created items ID_DRAWING_WIREFRAME and ID_DRAWING_SOLID. The third parameter specifies the state of the menu item. A menu item has two simple states available for use. The first is MF_CHECKED, which just as it implies checks a specific menu item. The other state is MF_UNCHECKED, which unchecks the given item.

If the user clicks the Wireframe menu item, a simple checked swap could look like the following:

```
CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);
```

This is great when an application only has two options that need to be flipped. However, if our application had four options to select from, it would be much easier to save the currently selected item in a long data type, uncheck all the options, then check the appropriate value.

Getting the Check State

When dealing with menu items that can be both checked and unchecked, we need a function that will allow us to retrieve the current state of the item to allow the specific functionality to happen. There are two drawing states in our map editor: Solid and Wireframe. When drawing a map, we need to check whether Solid or Wireframe is selected. To see if the menu item is selected, we would simply send a message using the SendMessage function. The SendMessage function sends a user-specified message to the message handler, WinProc. In this case, we specify that we want the BM_GETCHECK message to be sent, which will return the current state of the menu item.

The SendMessage function requires four unique parameters. The first parameter is the handle of the destination window, the Window variable. The second parameter is the user-specified message, which in this case is BM_GETCHECK. The third and fourth parameters are of the WPARAM and LPARAM data types, respectively. These two parameters are used to specify extra details about the message. In the case of the BM_GETCHECK message, both values should be set to 0. The return value for this particular message will be the current item state, which will be either BST_CHECKED or BST_UNCHECKED, respectively. This function will be used in later chapters, when we are drawing our map using OpenGL.

Making a Pop-up Menu

Pop-up menus are a wonderful way to increase functionality through simple clicks. It's a good idea to use pop-up menus for commonly used functions to increase the overall productivity of the user. For instance, in Microsoft Word a user can highlight a sentence, right mouse click, and bring up a menu for quick access to functions like Copy and Paste. Now imagine if Word didn't have that functionality. Every time you wanted to copy and paste you'd have to move the mouse away from the highlighted area to the Edit menu item, then select the specific function you wanted. Although this may seem like a minor interface detail, it could decrease your overall productivity when developing big projects.

This is where pop-up menus come into play. Although we are not specifically writing a word processor, we still need quick paths to core functionality. Functions such as Duplicate, Move, and Delete are common actions we would want quick access to. For this reason, we'll add these functions and a Texture item to our pop-up menu.

To begin, you'll need to create a new menu resource named IDR_POP-UP_MENU, following the same steps as discussed earlier in this chapter in the "Adding Item Click Functions" section. Add the Move, Duplicate, Delete, and Texture items to the menu. It's a good idea to space each item out using the separator, but this is not required. Moving back to the example source code, we must add a new global variable called PopupMenu, which is of the HMENU data type. This handle of a menu will contain our pop-up menu information.

In the WinMain of our program, we once again need to use the LoadMenu function to load the menu resource. By placing the PopupMenu variable below the original function call and simply copying and pasting the LoadMenu function call, we can easily load the menu resource. We don't need to use the SetMenu function because this menu will not be a main menu of a window, so we are finished with the WinMain section. Moving to the WinProc message handler, we must add a new message to the message case statement. The new message is WM_RBUTTONUP. The message is activated when the right mouse button is released. This message uses the lParam variable to obtain the X and Y coordinates of the mouse location upon release. To extract the X coordinate from the lParam variable we use the LOWORD macro and supply lParam as the parameter. The return value is the cursor X coordinate. To extract the Y coordinate we use the HIWORD macro and once again supply the lParam variable as the parameter. Both return values are integers.

We'll need the X and Y coordinates of the cursor location as parameters for a new function we're going to create called DisplayPopupMenu. The DisplayPopupMenu function has two parameters, x and y, both of which are of the long data type. In the function we must first declare a new temporary menu called temp, which will store a submenu of the PopupMenu variable we loaded earlier. We cannot access the items in the PopupMenu variable directly so we must use the GetSubMenu function to retrieve the list of items. There are two parameters for the GetSubMenu function: The first is the handle of the current menu, and the second is the position of the menu you want to retrieve. The return value is the new menu handle.

After returning the new menu we need to call the function TrackPopup-Menu, which will display our pop-up menu. The first parameter for the function is the menu handle you want to display. The second parameter specifies the flags for the pop-up menu. In this example we will hardcode the value TPM_LEFTALIGN|TPM_RIGHTBUTTON, which specifies that the menu is aligned to the left of the current cursor position and both the left and right buttons can be used to select items. After the flags are the X and Y coordinates, which are passed to the function from the message handler itself. The fifth parameter is a reserved value that has no relevance to us, so we set the value to 0. The sixth parameter is the handle of the window, which is our Window variable, and the final parameter is a pointer to a RECT structure. This parameter is set to NULL since it is ignored in the API at the moment.

Now when the user clicks the right mouse button, our pop-up menu will appear at the current mouse position. To write events for the specific items clicked, simply add the item ID to the else-if statement in the WMCommand function and add the appropriate functionality.

In the next example, each menu item has a default message box that brings up its appropriate function name. For example, if you click the Texture item, a message box will pop up with the text Texture, as per its function.

The following source code displays several functions, including checking/unchecking menu items and creation/manipulation of pop-up menus. Take a look at the code and see how the new functions have been created.

Example ex1_4.cpp:

```
#include <windows.h>
#include <winbase.h>

#include "resource.h"

#define DEFAULT_BUTTON_WIDTH   100
#define DEFAULT_BUTTON_HEIGHT  20

HMENU   Menu;
HMENU   PopupMenu;
HWND    Window;
HWND    bCreateWall;

void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (lParam == (LPARAM)bCreateWall) MessageBox (Window, "You Pressed
             bCreateWall", "Congrats!", MB_OK);
   else if (wParam == ID_FILE_EXIT) PostQuitMessage(0);
   else if (wParam == ID_DRAWING_WIREFRAME)
   {
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);
   }
   else if (wParam == ID_DRAWING_SOLID)
   {
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_UNCHECKED);
   }


   else if (wParam == ID_POPUP_MOVE) MessageBox (Window, "Move", "Click", MB_OK);
   else if (wParam == ID_POPUP_DELETE) MessageBox (Window, "Delete", "Click",
             MB_OK);
   else if (wParam == ID_POPUP_TEXTURE) MessageBox (Window, "Texture", "Click",
             MB_OK);
   else if (wParam == ID_POPUP_DUPLICATE) MessageBox (Window, "Duplicate",
             "Click", MB_OK);
}

void DisplayPopupMenu(long x, long y)
{
   HMENU temp = GetSubMenu(PopupMenu, 0);
   TrackPopupMenu(temp, TPM_LEFTALIGN|TPM_RIGHTBUTTON, x, y, 0, Window, NULL);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
      case WM_RBUTTONUP: DisplayPopupMenu(LOWORD(lParam), HIWORD(lParam)); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
          int CmdShow)
{
   WNDCLASS  wc;
   MSG       msg;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "ME";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);
   }

   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
            0, 0, 640, 480, NULL, NULL, hInstance, NULL);

   if (Window == NULL)
   {
      MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
      return (0);
   }

   bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE,
               0, 100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window,
               NULL, hInstance, NULL);

   Menu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_MENU));
   SetMenu (Window, Menu);

   PopupMenu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_POPUP_MENU));


   while (1)
   {
      if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
      {
         if (msg.message == WM_QUIT) break;
         TranslateMessage(&msg);
         DispatchMessage (&msg);
```

```
        }
    }

    return (1);
}
```

Dialog Boxes

Dialog boxes are windows that appear in a program to elicit information from the user.

A dialog box window is created either using the drag-and-drop interface in the resource editor or through the resource script itself. In either case, a dialog box provides a quick and easy solution when creating interface windows that require input from the user. For windows with purposes like entering map details, selecting the type of map, and choosing what texture blending functions you want, dialog boxes are the best option. Creating a dialog box is a matter of simply dragging and dropping a window into the resource editor and changing the name of the title bar and the ID.

We'll use dialog boxes extensively throughout this book because they are easy to use and quick to develop. Unlike other resources, a dialog box can specify a new message handler to deal with messages specific to the dialog box. This is rather handy because you can limit functionality to specific areas in your program, allowing for cleaner code. Other benefits of the dialog box include easy setup of the default font, auto mouse and window centering, and even the option to add a menu resource directly to the window. With all these attributes you may be wondering why we don't use dialog boxes as our primary windowing system instead of registering a window class and creating the window manually. Since dialog boxes were originally created for the purpose of user input, they don't have the same basic guidelines a regular window may have. For instance, if you create a window manually you can resize the window on the fly without any problems. If you want to resize a similar dialog box, you'll need to write the code. For quick utilities, however, dialogs are by far the easiest way to go.

Differences between Windows and Dialog Controls

At the beginning of the chapter you learned that many things in a Windows application are handles of a window. Menus, combo boxes, list boxes, edit boxes, and buttons were all made from that magical HWND data type. We can create a dialog box equivalent by using dialog controls in the resource editor. Rather than typing lengthy CreateWindow functions, the controls are added to the dialog box resource script and are compiled in the resource code automatically.

When accessing data in any dialog control, we need to use the SendDlg-ItemMessage function instead of the regular SendMessage function. The difference between the two functions is an extra parameter added to the new function. The new parameter is the unique ID that specifies to which ID the message will be sent. Configuring dialog controls is as simple as double-clicking the control and selecting the appropriate options from a list.

Using Controls

To create a dialog box, double-click on the workspace file **me.rc** to open our resource script. Next, select **me resource** using your left mouse button, then right-click to bring up a pop-up menu. In the middle of the pop-up menu is an Insert option. Select this option and press the left mouse button. A new dialog box will pop up with a list of different resources you can insert into the current resource script. Select the **Dialog** item and click the **New** button. Our new dialog box has been created and should display on the screen as shown in Figure 1.6.

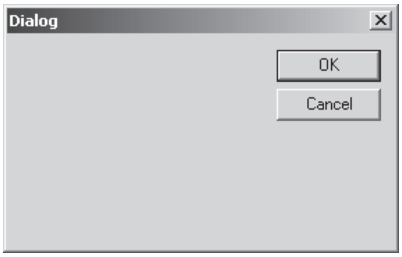


Figure 1.6: Our first dialog box

The new dialog box has a default name and ID. Let's change the name of the dialog box to a more suitable name, something specific to the function it will do. To change the configuration of a dialog box, simply double-click on the dialog box itself, using the left mouse button. The Dialog Properties window will open and display the current configuration information. Each tab contains information relevant to its name. In this instance, we'll change the default ID to a more suitable name, **IDD_MAP_DETAILS**. The caption for the dialog will be **Map Details**, so when the dialog box pops up, it will have a window name of Map Details. Once both changes are made, click on the other tabs to see what types of configuration options are available.

If we accept the current dialog configuration, when our dialog box appears it will display at the top-left corner of the screen. Although this is fine, a dialog box looks more professional when it's centered. To center a window, select the **More Styles** tab in the Dialog Properties window and check the **Center** option. When you've done so, close the window and let's begin writing the dialog box message handling code.

To run our newly created dialog box we use the appropriately named DialogBox macro. The DialogBox macro has four parameters that are required for it to execute. The first parameter is the handle of our instance, which is a new global variable we must declare. The second parameter is an

LPCTSTR representation of the dialog box. As we've discussed previously, all resources have a unique numerical value associated with them. To get an LPCTSTR version of the numerical value, we use the command MAKEINT-RESOURCE, passing the ID of the dialog box. The third parameter is the handle to our parent window. This parameter is set to NULL since we don't need to specify an owner for the dialog box. The last parameter is the message handler for the dialog box. You'll notice that the data type is DLG-PROC rather than the WINPROC we have used thus far. The only difference between the two is DLGPROC only returns Boolean values whereas the WINPROC function returns an integer. For all our dialog-based message handlers we'll use the original WINPROC format and cast the value as a DLGPROC data type.

Before we can add any source code to our project we'll need to add a new global variable called GlobalInstance, which is a handle of an instance. This variable will contain the instance information passed when the program is executed in the WinMain. In the WinMain add the following line:

```
GlobalInstance = hInstance;
```

This global variable will come in handy later on in the project because we'll use it extensively when using dialog boxes.

Next we'll need to add a new menu pop-up section called **Map** with the menu item **Details** in the pop-up. If you are unsure how to do this, follow the directions discussed earlier in the chapter in the "Adding Menus" section. I've placed the new Map menu pop-up between our existing File and Drawing entries. As with all menu items, the unique values have been created, giving us a new ID, ID_MAP_DETAILS. This new ID will be the launching point for the new dialog box we've created.

In the WMCommand function, we need to add the DialogBox macro command. Toward the middle of the function, add a new if statement, checking for an item click of ID_MAP_DETAILS. Our if statement should look like the following:

```
else if (wParam == ID_MAP_DETAILS) DialogBox (GlobalInstance,
        MAKEINTRESOURCE(IDD_MAP_DETAILS), NULL, (DLGPROC)MapDetailsDlgProc);
```

The message handler for the dialog box will be named **MapDetailsDlgProc**. The name is derived from the basic functionality plus "DlgProc" at the end, allowing the programmer to quickly identify what type of function it is. As stated earlier, we declare our dialog message handlers the same way we declare our regular message handlers with one exception: Any dialog-based message handler will not have a return value created using the DefWindowProc function. Instead, we'll return a constant value of 0, since the original DLGPROC type is an integer. Other values return odd results and in some cases will not allow the dialog to even display. The controls themselves are fairly short topics since we'll be using very little of their entire functionality.

The first message we'll add to our dialog message handler is the WM_INITDIALOG message. This message is sent when the dialog box is first created by the DialogBox function. The message is generally used for setting resource control defaults within the dialog. We'll take advantage of this message later to set the default text for a text box control.

Buttons

Messages are handled in the same case statement as the msg variable in the main message handler of our map editor. The second message we'll add is WM_COMMAND, which as discussed earlier in the chapter, handles button clicks from a variety of sources. We've already implemented this message in our main message handler so we can skip the formalities of how to write the message source code and move directly to which items must be handled when clicked.

In this dialog we only have buttons to deal with. The first button, OK, has an ID of IDOK. The second button, Cancel, has an ID of IDCANCEL. There's no need to change the ID names of these controls because they have names that are simple enough to remember. In the message we'll check to see if the user has pressed either item. If it's true, we'll display a message box with the name, then we'll close the dialog box. We use the function EndDialog to exit from a dialog box. There are two parameters for the function. The first parameter is a handle to the window and the second is the return value of the dialog box. A typical call to EndDialog would look similar to the following:

```
if (wParam == IDOK)
{
   MessageBox (hWnd, "OK Button!", "OK", MB_OK);
   EndDialog (hWnd, 0);
}
```

The return value parameter in the EndDialog function can be very useful when we need to return the status of a specific dialog box. For instance, if we had a dialog that required the user to enter a license key to use the map editor, we could have different return values for both the accepted and not accepted keys. If the user clicked Cancel, a value of 0 could be returned as a failure. If the OK button were clicked, the software could run a quick validation check and return either 1 for success or 0 for failure. Alternatively, you could use more than two values by specifying 0 as cancel, 1 as success, and 2 as access denied. This is just one example of where this type of function would be useful. Of course our editor will not require a license key, but a real-world example is always helpful in understanding the usefulness of certain functions. We've now covered buttons and can move on to the edit box controls.

Edit Box Controls

The purpose of an edit box is to receive user-typed input. In certain cases, the input can be masked as asterisks for such functionality as typing a password. In our case, we'll use edit boxes to add user-input text about our levels. Let's add a new edit box control in our newly created dialog box (IDD_MAP_DETAILS) and call it **IDC_MAP_DETAILS_NAME**. There are many types of settings for the edit box controls. Since we're doing basic user-input text we don't need anything fancy, but there are other options available by simply checking and unchecking the settings. Once you've viewed the other optional settings, exit the Edit Properties window and let's begin writing code to interface with the edit box.

Setting Text

To set the text of an edit box control we use the function SetDlgItemText, which as the name suggests sets the text in a given dialog item. There are three parameters needed for the function. The first parameter is the handle of the current window (hWnd). The second parameter is the ID of the dialog to be set (IDC_MAP_DETAILS_NAME). The final parameter is the characters we want as our text string. When written in source code, the function looks like the following:

```
SetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, "Map Name");
```

Getting Text

Now that we can set the text of a dialog item, we need to know how to get the text from one. The function we use to get the text from a dialog item is GetDlgItemText. The function requires four parameters to work properly. The first parameter is the handle of the current window (hWnd), the second is the dialog item (IDC_MAP_DETAILS_NAME), the third is an address for the text, and the final parameter specifies the maximum length of the string. When the function is filled in properly, it looks like the following:

```
char temp[500];
GetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, temp, 500);
```

If the edit box control were set to number you would have to convert the value of temp to an integer using the sscanf or strtol function. Unfortunately, there isn't too much in the way of expanded functionality for edit box controls. We've covered the two main functions for this control, so it's time to move to the next type of control, the list box.

List Box Controls

The list box control displays a list of items from which the user can select. Depending on the configuration of the list box, a user can select single or multiple items in the control. There is no limit to the number of items you can have in a list box. All items in the list box are sorted alphabetically unless you unselect that feature in the List Box Properties window.

Open our existing dialog box and create a new list box named **IDC_MAP_DETAILS_LEVEL_RULES**. This list box will display the specific rules for ending the level. For instance, we could have a list of three different rules: The first would require that the player touch the exit area, the second would require all enemies to have been killed, and the third would require the user to have at least 30 health. This is just one example of the use of the list box control.

Adding Data to the List Box

To add data to the control we must send a message to the dialog item. The message we send is LB_ADDSTRING. As with many control messages, the first two letters of the message indicate what type it is. For instance, LB is the abbreviation for list box. To send a message to a specific dialog item we use the function SendDlgItemMessage, which requires five parameters. Like edit boxes, many controls require the use of SendDlgItemMessage to alter their properties. The first parameter in the function is the handle to the current window (hWnd). The second parameter is the ID of the dialog item to which you are sending the message (IDC_MAP_DETAILS_LEVEL_RULES). The third parameter is the message we are sending; in this case it will be LB_ADDSTRING, allowing us to add a string. The fourth parameter is of the WPARAM data type; for this specific message it is required to be 0. The final parameter is of the LPARAM data type; however, we use this parameter to pass an address of a buffer, then cast it as an LPARAM data type. A filled-in function should look like the following:

```
SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING, 0,
        (LPARAM)"Exit");
```

We would add this function to the dialog initialization message (WM_INITDIALOG) so the list box options could be filled in when the dialog is displayed on the screen.

Getting the Selected Item

One of the most crucial messages sent to the list box is the currently selected item. Without the message, the list box would have very little value to the programmer, since there are other controls available for displaying multiline data. We use the message LB_GETCURSEL to retrieve the currently selected option. If there is no option selected, the return value from

SendDlgItemMessage is –1. If an item is selected, the order starts at 0 and increments from there.

As with all the other list box messages, we use the SendDlgItemMessage function. The first two parameters are the same as all the other instances of the function, but the final three are different. The message in this scenario is LB_GETCURSEL, which tells the dialog item that we want the currently selected item. The fourth and fifth parameters must be 0. The function will then return the integer-based value of the selected item. The function would look similar to this:

```
long cursel = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES,
         LB_GETCURSEL, 0, 0);
```

Setting the Selected Item

In the last section, we discussed how to get the current selection in a list box. Next, we'll discuss setting the current selected item. This comes in handy when we want to select the default option in the list box when it's first created. In order for this to work, there must be at least one item in the list box. To set the selection we send the LB_SETCURSEL message to the item, specifying the item to select as the fifth parameter. The first two parameters in the SendDlgItemMessage are the same as in the previous sections. The third parameter is LB_SETCURSEL, the message to send. The fourth parameter will be set to 0, since it is not needed.

```
SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_SETCURSEL, 0, 1);
```

Resetting the Contents

Resetting the content of a list box is a very important function. Without this functionality we could not easily clear the box of data and fill it with new values. When a dialog box contains a list box control, as ours does, it's a good idea to reset the contents of the list box when the dialog box is created in the WM_INITDIALOG message. This will ensure that any data previously stored in the list box is erased and new data can be added.

To reset the contents of a list box control we use the message LB_RESETCONTENT with the SendDlgItemMessage function. The first two parameters are unchanged from the LB_ADDSTRING message. The third parameter must now be set to LB_RESETCONTENT, as this is the message. The final two parameters must be set to 0 for the message to be properly sent. Here is an example of resetting the list box content:

```
SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_RESETCONTENT, 0, 0);
```

The message structure for resetting the contents couldn't be any simpler! You've probably noticed throughout this book that many of the Win32 controls have a similar structure, which makes them easier to learn. With this in mind, let's continue to our next section on combo box controls.

Combo Box Controls

Combo boxes are similar in design to list boxes in that they both hold itemized data, allow for sorting the data, and have no physical limitation in size. The two main differences between list boxes and combo boxes are that they have different abbreviated message formats — list box is LB and combo box is CB — and they are drawn differently. The list box displays its contents in a predefined box on the screen with the appropriate scroll bars. The combo box only displays one item with a down arrow that allows you to select other items. The combo box can save a lot of room when designing applications; however, it is a good idea to choose which control you want wisely because each has its own benefits. The list box allows you to select multiple items but takes up more space; the combo box can display many items but you can only select one.

Let's create a combo box underneath the list box we created earlier in the chapter. The name of the combo box will be **IDC_MAP_DETAILS_LEVEL_TYPE**. As the name implies, this combo box will contain a set type of the level we're creating. Once the combo box is created, we'll need to adjust its vertical size to hold several items. To adjust the size, select the combo box and click the down arrow. A new bounding box will appear with the bottom vertical marker in blue, allowing you to resize the item area. Stretch the bounding box to about twice the size of the original combo box. When you have resized the combo box, go to the combo box properties section and unselect the Sort option. This will prevent our data from being sorted when we add new items. In the Tab as the sort check box, there is another option called Type. This will specify the type of combo box we will be creating. The default combo box does not suit our needs because it allows the user to alter the item names and does not provide a full-size vertical scroll bar. If you select the Type combo box, it will display a list of three options. Select the Drop List item to use a basic combo box.

You've now learned the basics of combo boxes, so let's learn how to use the functionality of the control.

Adding Data to the Combo Box

Adding data to a combo box is similar to adding data to a list box. Both the list box and combo box controls use the SendDlgItemMessage function to send the message, both have abbreviated messages, and both have one parameter of data required to be filled in to add an item.

Since this section deals specifically with the combo box IDC_MAP_DETAILS_LEVEL_TYPE, we'll discuss the last three parameters of the SendDlgItemMessage function. We know the first parameter is always going to be the handle of a window (hWnd) and the second parameter will always be IDC_MAP_DETAILS_LEVEL_TYPE, so we can focus more specifically on what each message and parameter does.

To add data to a combo box we use the message CB_ADDSTRING. The fourth parameter in the function is set to 0 since it is not needed, and the final parameter is an address of characters (a text string) cast as an LPARAM data type. A typical CB_ADDSTRING message would look like the following:

```
SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING, 0, (LPARAM)
        "Single Player");
```

Getting the Selected Item

It is a good idea to select a starting or detail item when there is a possibility of trying to get the currently selected item. To get a selected item from a combo box, we send the CB_GETCURSEL message. Both the third and fourth parameters for the CB_GETCURSEL message are set to 0 since they are not used. If there is an item selected, the return value will be 0 or higher. If the return value is –1, there is no selected item. A typical CB_GET-CURSEL message would look like the following, keeping in mind that the function would return the selected item:

```
SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_GETCURSEL, 0, 0);
```

Setting the Selected Item

Setting the default item is something you should always do when the dialog box owner is first created. This ensures that there will be a default option for each selection, regardless of whether the user has selected an option. To select an option, simply send the CB_SETCURSEL option to the combo box, supplying the fourth parameter as 0. The fifth parameter is the item number you would like to select. The lowest value you can pass is 1. The CB_SETCURSEL message looks like the following when sent:

```
SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_SETCURSEL, 0, 1);
```

Resetting the Combo Box

Like the list box, the combo box has a message that can erase all the items inside it. To reset the content of a combo box, we send the command CB_RESETCONTENT. Both the fourth and fifth parameters are set to 0 since they are not needed. It’s a good idea to call the reset content message when a dialog is first created to ensure no remnants of a previous item instance are still in the control. A properly sent function will look like the following:

```
SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_RESETCONTENT, 0, 0);
```

Static Controls

A static control is a simple mechanism for displaying text on a dialog box. By default, all static controls use a generic ID so you cannot change the text; however, you can specify an ID so you can change the text. We'll use static controls to clean up our interface so the average user can tell what each control is. Place three static controls in our dialog box and move one above each of the other controls we've already created. To change the caption of each static control, double-click using the left mouse button. The Static Properties window will then appear, giving you the option of changing the caption. Change the name of the top static to **Map Name**. The middle static should be named **Level Exit Rules**, and the bottom static should be named **Level Type**. Unfortunately, the static controls don't offer much functionality to programmers, due to their design intent.

Chapter Example

This next example covers all the information about dialogs we've discussed. This is also the final example in Chapter 1, and therefore the longest.

ex1_5.cpp:

```
#include <windows.h>
#include <winbase.h>
#include <stdio.h>

#include "resource.h"

#define DEFAULT_BUTTON_WIDTH   100
#define DEFAULT_BUTTON_HEIGHT  20

HINSTANCE GlobalInstance;
HMENU     Menu;
HMENU     PopupMenu;
HWND      Window;
HWND      bCreateWall;

LRESULT CALLBACK MapDetailsDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
          LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, "Map Name");

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                   0, (LPARAM)"Erase Me");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_RESETCONTENT,
                   0, 0);
```

```
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                      0, (LPARAM)"Exit");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                      0, (LPARAM)"Get Fragged");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_SETCURSEL,
                      0, 1);

            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                      0, (LPARAM)"Erase Me");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_RESETCONTENT,
                      0, 0);
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                      0, (LPARAM)"Single Player");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                      0, (LPARAM)"Multi Player");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_SETCURSEL,
                      0, 1);
         } break;

         case WM_COMMAND:
         {
            if (wParam == IDOK)
            {
               long level_rule = SendDlgItemMessage (hWnd,
                         IDC_MAP_DETAILS_LEVEL_RULES, LB_GETCURSEL, 0, 0);
               long level_type = SendDlgItemMessage (hWnd,
                         IDC_MAP_DETAILS_LEVEL_TYPE, CB_GETCURSEL, 0, 0);

               char temp[500];

               sprintf (temp, "Level Type: %i\r\nLevel Rule: %i\r\nOK Button!",
                         level_type, level_rule);
               MessageBox (hWnd, temp, "OK", MB_OK);

               EndDialog (hWnd, 0);
            }
            else if (wParam == IDCANCEL)
            {
               MessageBox (hWnd, "Cancel Button!", "Cancel", MB_OK);
               EndDialog (hWnd, 0);
            }
         } break;
      }
      return (0);
   }

   void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
   {
      if (lParam == (LPARAM)bCreateWall) MessageBox (Window,"You Pressed
                bCreateWall", "Congrats!", MB_OK);
      else if (wParam == ID_FILE_EXIT) PostQuitMessage(0);
      else if (wParam == ID_DRAWING_WIREFRAME)
      {
```

```
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);
   }
   else if (wParam == ID_DRAWING_SOLID)
   {
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_UNCHECKED);
   }
   else if (wParam == ID_MAP_DETAILS) DialogBox (GlobalInstance,
             MAKEINTRESOURCE(IDD_MAP_DETAILS), NULL, (DLGPROC)MapDetailsDlgProc);

   // Popup Menu Items
   else if (wParam == ID_POPUP_MOVE) MessageBox (Window, "Move", "Click", MB_OK);
   else if (wParam == ID_POPUP_DELETE) MessageBox (Window, "Delete", "Click",
             MB_OK);
   else if (wParam == ID_POPUP_TEXTURE) MessageBox (Window, "Texture", "Click",
             MB_OK);
   else if (wParam == ID_POPUP_DUPLICATE) MessageBox (Window, "Duplicate",
             "Click", MB_OK);
}

void DisplayPopupMenu(long x, long y)
{
   HMENU temp = GetSubMenu(PopupMenu, 0);
   TrackPopupMenu(temp, TPM_LEFTALIGN|TPM_RIGHTBUTTON, x, y, 0, Window, NULL);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
      case WM_RBUTTONUP: DisplayPopupMenu(LOWORD(lParam), HIWORD(lParam)); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
          int CmdShow)
{
   WNDCLASS  wc;
   MSG       msg;

   GlobalInstance   = hInstance;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "ME";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);
   }

   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
            0, 0, 640, 480, NULL, NULL, hInstance, NULL);

   if (Window == NULL)
   {
      MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
      return (0);
   }

   bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE,
                 0, 100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window,
                 NULL, hInstance, NULL);

   Menu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_MENU));
   SetMenu (Window, Menu);

   PopupMenu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_POPUP_MENU));


   while (1)
   {
      if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
      {
         if (msg.message == WM_QUIT) break;
         TranslateMessage(&msg);
         DispatchMessage (&msg);
      }
   }

   return (1);
}
```

Conclusion

In this chapter, we learned the basics of Win32 SDK programming. With the knowledge from this chapter, you should be able to create basic Windows applications with the functionality seen in other Windows programs. The final example gives us the basic shell of our map editor.

Chapter 2

Introduction to OpenGL

The History of OpenGL

In the late 1980s Silicon Graphics, Inc. (SGI) released a 3D application programming interface (API) called IRIS GL. The original intent for the API was to develop graphics applications for uses such as CAD and animation. IRIS GL allowed the programmer to draw 3D objects known as primitives on the screen on SGI's proprietary Unix-like operating system, IRIX. IRIS GL created an entire window system to draw and handle data on the screen. There are over 1,500 applications that use the IRIS GL platform to render graphics. The downside of IRIS GL was that it was very proprietary due to its windowing system constraints, which made porting code difficult even among SGI's own products.

In 1992, SGI released a new 3D API called the Open Graphics Library, more commonly known as OpenGL. Unlike the original IRIS GL system, OpenGL allowed the operating system to draw 3D objects without using a proprietary window system. The OpenGL API uses a unique naming convention in which all GL commands are preceded by the lowercase letters "gl" (e.g., glColor). Data types in OpenGL also use a unique naming convention in which they are preceded by the uppercase letters "GL" (e.g., GLfloat).

For several years, the graphics industry was using OpenGL, while the PC game industry was still using non-accelerated DOS hardware. SGI created a body to govern the progression of the API. This governing body, called the OpenGL Architecture Review Board (ARB), includes companies such as SGI, nVIDIA, ATI, Matrox, 3Dlabs, and more. The members conference several times a year on where they would like to take the API. In the mid-1990s hardware acceleration began dropping lower in price. Regular consumer-level video cards began shipping with special features that only the manufacturers' graphics libraries could access. Windows 95 and NT4 changed this by adding software rendering support for OpenGL. This feature was hidden away but could be exposed if the programmer linked his applications with the OpenGL libraries, ensuring complete compatibility with all machines running those operating systems.

In 1996, id Software released three patches to its blockbuster title Quake that altered the path of 3D graphics for the PC game industry. These three patches were simply known as Verite Quake, Rendition Quake, and GL Quake. Obviously, the last one used OpenGL. Quake was the first PC game to take full advantage of hardware acceleration. The only problem was there were very few graphics boards on the market that were affordable to the average PC gamer, since OpenGL was originally conceived for workstation/rendering purposes. The first consumer video cards to bring OpenGL into the spotlight were 3DFX Interactive's Voodoo/Voodoo Rush series. These cards only had 2 to 6 MB video RAM — very low by today's standards, but they transformed the pixelated graphics of Quake into a new form of hellish environment, in a good way!

With 3DFX's video cards being easily accessible to the mainstream consumer, their influence in the PC graphics industry was quickly noticed and other video card manufacturers began making cards that supported GL Quake as they tried to snag a small piece of this newfound market. 3DFX had also created its own 3D API called Glide, which if written for allowed all 3DFX cards to be highly tuned for games. Going too proprietary has its disadvantages at times though. Many developers opted to use Glide over OpenGL because of 3DFX's large market presence at the time; however, that proved to be a mistake, as the company's share of the market slowly dried up when others opted for a more open 3D platform.

By 1998, Microsoft had begun releasing its own 3D API called Direct3D, included in their multimedia component DirectX, and Imagination Technologies released a graphics chip called PowerVR, which had its own API. Because of the popularity of GL Quake, many engine licensees used the Quake engine to create their games, making OpenGL the de facto standard for PC gaming graphics. For several years, the evolution of OpenGL stagnated while the Microsoft API became increasingly popular. Although OpenGL is still by far the leader in CAD/workstation graphics, Direct3D is a common standard among many game developers. By this point you may be wondering why you should learn OpenGL. That will be explained in the next section.

Why Use OpenGL?

There are many reasons to use OpenGL as the foundation of graphics software and games. As discussed earlier, OpenGL has a consortium (ARB) of graphics manufacturers and vendors planning the future of the API. The consortium at times takes opinions from independent software vendors regarding issues dealing with the API. The OpenGL syntax is very simple and allows the programmer to write complicated graphics software in a simple and logical manner. The API allows the video card manufacturers to create customized features and take advantage of them before they become part of the standard. OpenGL code can also run on a host of platforms, which other APIs such as Direct3D cannot do.

If you are interested in writing games or applications for multiple platforms, i.e., Windows, GNU/Linux, and Macintosh, OpenGL gives you the freedom to use the same code for all the supported platforms. The only difference between each platform is the initialization of OpenGL. There aren't many graphics libraries that allow this sort of flexibility. Table 2.1 gives a comparison of operating systems supported by OpenGL, Direct3D, and Glide. Unfortunately, Glide is no longer a viable alternative to OpenGL and Direct3D because 3DFX no longer exists and its technology is far outdated by today's graphics standards. Also, the Glide graphics libraries were proprietary to 3DFX's Voodoo series cards. But for the sake of interest, I've included Glide in the table below.

Table 2.1: 3D API/OS compatibility

OpenGL/Mesa	Direct3D	3DFX's Glide**
Windows 95/98/Me/NT/2000/XP	Windows 95/98/Me/NT/2000/XP	Windows 95/98/Me/NT
Mac OS 9/Mac OS X	Xbox*	GNU/Linux
GNU/Linux		
UNIX		
AIX		
HP-UX		
FreeBSD		
NEXTstep		
OPENstep		
OS/2		
BeOS		
Playstation 2 (via PS2Linux add-on)		

* Modified version of Direct3D
** No longer in the 3D graphics business

In the OpenGL column, you'll notice the name Mesa. Mesa is an open-source implementation of the OpenGL API created by Brian Paul. Although not officially endorsed by SGI, Mesa provides an OpenGL-like API to platforms that do not have a true OpenGL library from SGI.

When Mesa development began, Brian Paul had been granted permission by SGI to use the command structure and syntax of OpenGL. Because Mesa is not officially affiliated with SGI, there is no guarantee that source code written for Mesa will compile and run properly. Also, the name of the API could not use "Open" or "GL." If you are wondering where the name Mesa came from, Brian has indicated on the official Mesa site that "the name Mesa just popped into my head one day."

Although the Mesa documentation states that Mesa does not guarantee compatibility with OpenGL, it has passed conformance tests. From my experience, I've found that regular OpenGL runs properly under Mesa. The latest version of Mesa, version 5.0, is compatible with the latest release of OpenGL, version 1.4. Each major release of Mesa follows a minor release for OpenGL. The following timeline displays OpenGL and Mesa release dates for their respective versions. This timeline can come in handy for tracking the progress of both APIs.

Table 2.2: OpenGL/Mesa release dates

OpenGL	Mesa
Version 1.0 – June 1992	Version 1.0 – February 1995 (conforms to OpenGL 1.0)
Version 1.1 – December 1995	Version 2.0 – October 1996 (conforms to OpenGL 1.1)
Version 1.2 – March 1998	Version 3.0 – September 1998 (conforms to OpenGL 1.2)
Version 1.3 – July 2001	Version 3.5 – June 2001 (conforms to OpenGL 1.2, passed compliance tests)
Version 1.4 – July 2002	Version 4.0 – October 2001 (conforms to OpenGL 1.3)
	Version 5.0 – November 2002 (conforms to OpenGL 1.4)

Both OpenGL and Mesa support hardware-accelerated rendering. This allows the video card to process the mathematical equations normally calculated by the CPU. In a simple example of hardware acceleration, the video card would manage the scaling and rotation of objects as the distance to the user changes. If there were no hardware acceleration, the CPU would have to rotate and then scale (or vice versa) the objects it's going to draw. This task can get very processor intensive depending on the scene you are trying to render. With the CPU wasting valuable cycles on drawing, other operations such as input, audio, and artificial intelligence (AI) will suffer because of the demands of the graphics. If the video card processed the scene very quickly, the other operations would be free to quickly do their tasks and give good frame rates to the user.

With this in mind, let's move to the next section, which discusses how the world works in three dimensions.

The World in Three Dimensions

Imagine if every person in the world were flat. One-size-fits-all clothes could actually apply to customers, buses could hold more people who sneeze on you, and extreme sports professionals could be used as human parts kits. Unfortunately, none of these things will ever happen (except for the sneezing) because we live in a world where everything is three-dimensional. Every person has a width, height, and depth. Just like many games and applications, the world has three measurements for all objects. Although things such as paper seem too thin to measure, it is still possible to measure the three axes of width, height, and depth. It's ironic that the term "paper-thin" is generally associated with being flat, yet paper can still be measured in all three directions, albeit in very low measurement units.

Working in three dimensions is actually quite easy once you've got a basic understanding of how two dimensions work. In two dimensions there are two axes, called x and y. The x-axis moves horizontally from left to right or right to left. The y-axis moves vertically from top to bottom or bottom to top. Many games in the mid-1980s and early 1990s were 2D in one form or another. The best examples of 2D graphics programming were seen in the blockbuster side-scrolling games Super Mario Bros., Donkey Kong, and Sonic the Hedgehog. When the main character moved to the right, it was moving along the x-axis. Every time he jumped he would be moving along the y-axis.

With three-dimensional programming we use both the x- and y-axes, along with a new axis called z. The z-axis controls the depth (or perspective) of a specific object. If an object moves along the z-axis, it will move closer to or farther away from you if you are directly in front of it. To put this into perspective, imagine if Super Mario Bros. allowed you to dodge Bowser's fireballs without having to jump or duck. By simply adding the z-axis the player could have avoided a fireball moving directly at him. Figure 2.1 displays all three axes pointing in their respective directions.

As you can see in the figure, the directions in which all three axes point are simple to grasp. Assuming we're looking directly at an object in front of us, to walk closer we could simply increment the z-axis. Obviously, if we wanted to backpedal we would simply decrement the z. If we wanted to simulate a jumping effect we could simply increment the y-axis at a timed rate of increase, allowing the objects to move up when required, then on the fall we would decrement the y-axis by a different timed amount for a proper descent to the ground. Finally, if we wanted to strafe (move sideways), we would move objects in the opposite direction, giving the illusion that we are moving in the desired direction. To strafe left, we would simply increment

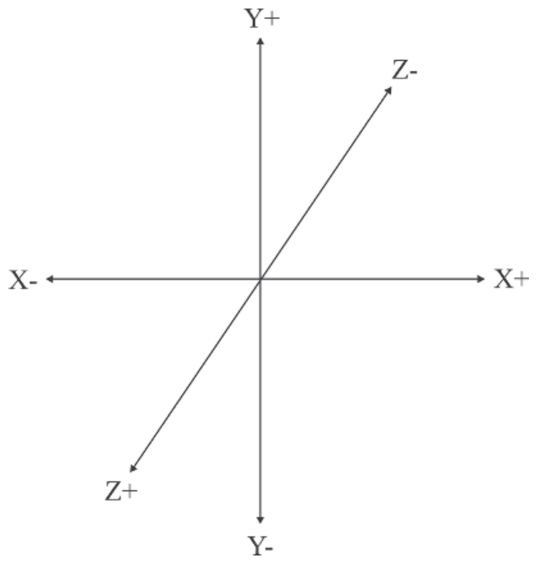


Figure 2.1: All three axes pointing in their proper directions. Note that X+ = Right, X– = Left, Y+ = Up, Y– = Down, Z+ = Nearer, and Z– = Farther.

the x-axis, whereas we would decrement the x-axis to strafe right. With this knowledge, you should have a basic understanding of which axis moves in each direction in 3D space. Later in the book we'll discuss the mathematical calculations that are required to walk around flat and sloped 3D worlds, but for now we'll discuss the many different buffers OpenGL has.

OpenGL Buffers

There are several key buffers that must be discussed so you can get an idea of what functionality is possible by accessing each of them. We will use several of the discussed buffers in our game to produce amazing special effects with the existing built-in functionality of OpenGL.

Accumulation Buffer

The accumulation buffer is different from other buffers in OpenGL because it does not render images directly into it. When images are rendered into any of the four color buffers, the contents are added to the accumulation buffer. Generally, the accumulation buffer is used to produce special effects like antialiasing and motion blurring. When creating cinematics, the accumulation buffer is a great asset because of its wide range of special effects capabilities. Unfortunately, full hardware acceleration for the accumulation buffer is not common among the latest three-dimensional graphics hardware.

If no hardware acceleration is available for the accumulation buffer, the processor must do all the calculations, which can drastically slow down the frames-per-second rate of the game.

Depth Buffer

In "the old days" of 3D programming, a programmer would have to run algorithms in real time that would sort the objects from back to front, preventing objects from overdrawing each other. Writing algorithms to sort the data on a per-frame basis can take away precious CPU cycles that could be dedicated to other tasks. Fortunately, the depth buffer provides a fast hardware-accelerated mechanism for rendering objects from back to front. If you supply 10 objects at varying distances from your current position, the depth buffer will sort the objects by their location and display them in the proper order. The depth buffer uses comparisons or rules to allow objects to display on the screen. By changing the comparisons we can allow objects to draw or not draw, depending on their distance from walls and whether they share a common distance. There is also a function that can specify the depth buffer range for clipping. Nowadays, the depth buffer is used almost religiously when writing 3D games. Since this buffer is an essential function for games, we'll learn how to enable and disable the depth buffer later in this book.

Frame Buffer

The frame buffer contains the final rendered result after all the other buffers and functionality has finished. The results of the frame buffer are generally the images displayed on the screen. The frame buffer has functions that allow you to get the current pixel values of a specific location or area, depending on your needs. Unfortunately, grabbing the pixel values is generally slow and not recommended when high rendering speeds are required.

Stencil Buffer

The stencil buffer provides a simple method to render images into a cookie-cutter-style shape. This functionality can be very useful when a programmer needs to render something like a rearview mirror in a car and leave the rest of the view normal. Simply put, the programmer would render the shape of the rearview mirror, setting the stencil mask to 1. This tells the stencil buffer that anywhere the rearview mirror is, there is a 1 in the stencil buffer. Then we would render the rear view and set the mask to only draw where the value in the stencil buffer is 1, thereby only drawing within the rearview mirror area and masking out the other geometry. The stencil buffer has the ability to customize what is drawn to the screen and where. This technique, called masking, allows the programmer to create elaborate special effects like mirroring, shadows, and much more without having to re-render entire scenes. Additionally, the stencil buffer can be useful for things such as constructive solid geometry, decals, and outlines.

Basic OpenGL Terminology

In this section we're going to discuss the basic terminology used in 3D and OpenGL programming. We'll discuss the terminology from the ground up, starting with the most basic element in a 3D object, a vertex.

Vertex

A *vertex* is a point in 3D space that is the building block for many objects. In OpenGL you can specify as few as two coordinates (X,Y) and as many as four (X,Y,Z,W). When a vertex is created without the w-axis specified, the default value is set to 1.0. If neither the z-axis nor the w-axis is specified, the default values are set to 0.0 and 1.0 respectively. In many of our examples we'll use the three main coordinates X, Y, and Z, since the W is generally used as a placeholder.

Triangle

Triangles are another of the primitive types a programmer can use when drawing objects. A *triangle* requires three points to be created. In OpenGL, we use three vertices to create a triangle. All objects, with a few exceptions, are drawn using vertices as their points, so unless otherwise stated we'll assume that a point will refer to a vertex.

Polygon

A *polygon* is an object with connected points that has a minimum of three points. When OpenGL draws the object, the video card will quickly break the object down into smaller triangles. Many people mistake polygon-per-second counters with triangle-per-second counters. Although a triangle can be considered a polygon, it's better to count triangles per second because one polygon could break down into 50,000 smaller triangles in a complex scene and there's virtually no way the programmer can easily find out how many triangles the polygon was broken down to from the graphics card. Alternatively, using triangles you could easily count how many triangles were sent to the video card for rendering, giving a very accurate frames-per-second counter.

Primitive

As discussed earlier in this chapter, a *primitive* is a three-dimensional object created using either triangles or polygons and a list of vertices. Assigning textures is done at a primitive level, meaning you assign a texture to the entire object and not to specific triangles. If you broke the triangles into separate objects to assign a different texture, you would have another primitive. I've always found it ironic that a detailed model with 50,000 triangles and a 1,000-triangle model are both called primitives. How could something with fifty times more triangles be considered primitive? In either case, we'll refer to all objects as primitives, unless otherwise stated.

With this new terminology, we have enough understanding to begin learning how to initialize OpenGL.

Initializing OpenGL

To initialize OpenGL we need to create a new global variable called RenderWindow, which is of the HWND data type. This new window will handle the drawing of OpenGL. Since we are building a map editor, we do not want the entire area to display the maps we are creating because we need room for our function buttons. We use the function GetClientRect to get the size of the specified window. The first parameter of the function is a handle of the window for which we want to retrieve the measurements. The second parameter returns the address of the RECT type containing the measurements of the window. In our WinMain we need to declare a new variable of the RECT type called rect. The function GetClientRect should be called after we've created our main window because if our window has not been initialized, we cannot get the measurements. The function GetClientRect should look similar to the following:

```
Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
          0, 0, 640, 480, NULL, NULL, hInstance, NULL);
if (Window == NULL)
{
   MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
   return (0);
}
GetClientRect (Window, &rect);
```

We'll use the values returned in the rect variable to create our rendering window. Rather than a text box or regular window we'll use a static box for the window style. I chose a static window over the other styles because it has the least amount of graphical overhead. Statics can have small borders around the bounding coordinates, which is what we want, and are quick to make. Also, users cannot accidentally interact with static boxes like they can with other window styles such as edit boxes. An accidental interaction could be something like a button click, which would focus all user input into the edit box rather than the main window, eliminating any keypress messages. This would destroy any sort of shortcut functionality we would try to make, which isn't a good thing!

There are several differences between creating our already existing window and our new static window. The first difference is the first parameter (lpClassName) in the function CreateWindow. We must change the class name from "BUTTON" to a more suitable name of "STATIC", which indicates we want to create a static. The second change is a minor modification to the third parameter (dwStyle), adding in WS_BORDER. When a static window is created, by default no bounding borders are drawn. For simplicity's sake, it's a good idea to specify the option to show a bounding box. This helps establish a visual workspace for the user to navigate through, and to the average user, it makes the software interface look nice.

The final differences are in the coordinates and dimensions of our new window. In Chapter 1 we created two constants that contained the width and height of a default button (DEFAULT_BUTTON_WIDTH and DEFAULT_BUTTON_HEIGHT). These two constants, along with our newly defined variable rect, will be used to set the dimensions of our rendering window. The fourth and fifth parameters of the CreateWindow function are the starting locations for the x- and y-axes, respectively. We'll set the starting X coordinate to DEFAULT_BUTTON_WIDTH, which ensures that the starting coordinate will always be to the right of our button bar on the left side of the screen. The starting Y location should be set to 0 so the window goes all the way to the top of the screen. The sixth and seventh parameters of the CreateWindow function must be altered to reflect the new changes in dimensions. We've set the starting locations to the top-left corner minus the default width of a button. Now we must compensate for the width and height variables using the rect variable we declared earlier to retrieve the size of the original window itself. To get the size of the window we simply subtract rect.left from rect.right, producing the window size, then subtract the DEFAULT_BUTTON_WIDTH constant. The final product will be the width of the render window we're about to create. Calculating the height of the window is much easier in that we only need to subtract rect.top from rect.bottom. At this point you may be wondering why I don't just hardcode the sizes and discuss something else of more relevance. By using the rect structure to create the proper sizes, we'll have a map editor that is able to resize if we set a different size for the main window. Furthermore, if we change the starting location of the window to something other than 1, this will compensate for the location change. This is certainly not technical but is of importance when testing for compatibility among different operating systems. Our newly created CreateWindow function should look like the following:

```
RenderWindow = CreateWindow("STATIC", NULL, WS_CHILD | WS_VISIBLE | WS_BORDER,
            DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-DEFAULT_BUTTON_WIDTH,
            rect.bottom-rect.top, Window, NULL, hInstance, NULL);
```

I'm sure you're tired of reading about Windows programming-related topics, so let's begin writing the initialization code for OpenGL. The first thing we'll need to do is create a new C++ class called RASTER. When creating C++ classes it's a good idea to not name the classes anything similar to the keywords OpenGL or GL. Not only will it be confusing in the source code, but SGI has already used the GL/gl namespace for all OpenGL functions and headers. This means that when we want to use OpenGL, we must include the gl.h file provided in the GL directory. (Note that the Macintosh platform is different in that it uses an OpenGL.H header rather than the original gl.h header found on most other platforms.) In addition to gl.h, we must include the glu.h header, which is located in the same directory as gl.h. The glu.h header contains all relevant information for the GL Utility library, which has many helper functions that come in handy during development. The final library we'll include in our header section of the class is the windows.h library, which allows us to use Windows data types within the class. The following code snippet displays the newly created header section of our RASTER class.

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <windows.h>
```

After including the header files, we'll use a tremendously handy pragma directive to specify which libraries the compiler should link when linking the file. This allows us to use one simple source line to specify the library as opposed to the conventional method of setting the libraries through the workspace settings options. In the event the workspace is corrupted, which is known to happen, there is a good chance that you'll lose all your library definitions, but by specifying the libraries in the source, we don't need to worry about this at all. When using the pragma, we specify comment as the directive. Next we'll need to use parentheses to specify the comment type that will link the appropriate data into the compiler. We'll use the lib comment type, and provide the library filename in double quotes as the second parameter. In the Windows operating system we have the choice of using either the original software library from SGI, opengl.lib, or the library provided by Microsoft, opengl32.lib.

The Microsoft library provides full hardware acceleration as well as software rendering. When a program is first executed, the appropriate OpenGL Installable Client Driver (ICD) is loaded from the video card settings in the system registry. If there are no OpenGL drivers specified in the registry, software rendering will be used by default. By loading the driver dynamically, all GL drivers should work without a problem. Vendor-specific functions can even be loaded by our program if we write the code for it, which we will do later in the book.

The GLU library must be loaded in the same fashion as our OpenGL libraries. Similar to the OpenGL headers, the GLU libraries have both an SGI (glu.lib) and Microsoft (glu32.lib) implementation. The appropriate GLU library should be loaded below its respective GL library. This means we would load the GL library first, then any other utility libraries second, such as GLU. Although it sounds strange, sometimes the compiler will require the GL library first because the utility libraries feed off the original APIs. The following code displays our newly acquired method of loading libraries into our objects.

```
#pragma comment(lib, "opengl32.lib")
#pragma comment(lib, "glu32.lib")
```

In case you're wondering, I named this class after the common graphics term "raster" because they both revolve around graphics, and it also sounds rather cool as a class name. The term "raster" was commonly used in two-dimensional programming as a way to represent the screen and images (raster images). The function of this class is to encapsulate all OpenGL initialization, releasing, and other important OpenGL-related functions revolving around configuration of the API and general-use functionality. By writing our OpenGL configuration code in this C++ class, we can reuse the object in the game we're going to create without having to worry about writing repetitive code.

Although we are using C++, we're not going to use strict object-oriented programming philosophies when writing the object methods. Although strict object-oriented programming is great, we're trying to keep this source code as simple and powerful as possible. It's easier to learn the basics and advance to more difficult topics if there is very little overhead in the code we are writing.

In our newly created class we'll need to create a default constructor and destructor. Neither the constructor nor the destructor will have any functionality in them at the moment because we must focus on initializing OpenGL. The first method we'll create in our object is called Init. The function of the Init method, as you may have guessed, is to initialize OpenGL. The return type of Init is of the bool data type. If the method succeeds, the return value is true. If any part of the method fails in initializing OpenGL, the method will display an appropriate error message and return false.

We need to pass three pieces of information through the Init method to initialize OpenGL in Windows. The first parameter is a handle to our rendering window. When OpenGL is fully initialized, the window passed will contain the images rendered in OpenGL. For this reason, this value cannot be NULL when we call the Init method; otherwise the Init function will fail. A NULL value will confuse the computer because it doesn't know what window to get the graphics context from. The second and third parameters are unsigned characters representing the color buffer bits (color_bits) and depth buffer bits (depth_bits). The color buffer bits specify for which color depth we'll render our images. This parameter has an overload value that is set to 24 by default. I'm not sure why you'd want to change this value, but users can easily change it to other modes to see the different results on their computers. Some video cards may benefit from changing this option, but we'll use the default option because there are features that won't work properly unless it is set to 24. We'll discuss what benefits 24 has over other values later in this book. The third and final parameter is the depth buffer bits, which specifies how many bits we want our depth buffer precision to be. As discussed earlier, the depth buffer sorts items in a back to front order. By specifying the depth buffer bits, you can change the precision of the testing algorithm, which may result in better clipping of items from the scene when items are very close to each other. By default the value for this parameter is set to 32, meaning 32-bit precision when doing the calculations. Although we can change this value simply by altering the overloads, we'll use the default value because it works well with most video cards and allows our objects to be rendered without any Z-buffer clipping issues. With our newly created Init method, our class should look like the following code snippet:

```
class RASTER
{
   public:

      RASTER();
      ~RASTER();

      bool Init(HWND hwnd, unsigned char color_bits=24,
                unsigned char depth_bits=32);
};
```

In our newly declared Init method we passed the handle of our rendering window. In order to initialize OpenGL we must get the device context of the window passed using the function GetDC. The GetDC function requires one parameter, which is a handle of a window. The return value is a handle of a device context, more commonly known as an HDC data type. If the function fails, the return value is NULL. In the event that part of the initialization fails, we will display the appropriate error message and return a false value to exit out of the Init method. The returned graphics context will be used in many different places throughout our program so we'll declare our variable, hDC, in the public section of the object. With hDC in the public section, we can use the variable from within the object or from outside if necessary. In the constructor of RASTER, set hDC to NULL as a default value. This gives us a quick method to see if OpenGL has been initialized when we are writing the drawing routines: If hDC is NULL, don't return false in the drawing routine! If properly written, the code for retrieving the device context should look similar to the following code.

```
hDC = GetDC(hWnd);
if (hDC == NULL)
{
   MessageBox(hWnd, "Error: Can't Get Device Context for Window", "ERROR",
              MB_OK | MB_ICONERROR);
   return (false);
}
```

After retrieving the device context, we set the pixel format descriptor. By setting the pixel format descriptor we're actually describing the type of pixel format we want to match with the device context. This is a very important step in initializing OpenGL because the pixel formats can affect everything from how the graphics are displayed and the precision of the depth buffer to the color bits when rendering and the bits for many different buffers like the accumulation buffer and stencil buffers. Unfortunately, setting the pixel format descriptor is highly proprietary to each operating system. Some operating systems, like Unix and GNU/Linux, share common initialization systems; however, others are fairly different in their process. In Windows we must declare a variable of the PIXELFORMATDESCRIPTOR data type and set its contents. There are more than 25 variables to set in the PIXEL-FORMATDESCRIPTOR structure, which may sound complicated, but many of the variables are actually junk or ignored in today's implementations of OpenGL.

There are many ways to fill the structure with the proper values. I'll discuss the long version first, then show two alternative methods that are much easier to type if you are going to write your own OpenGL source code. First, we must declare a new variable called pfd, which is of the PIXELFORMAT-DESCRIPTOR data type. This variable can be a local variable to our Init method since other methods will not require this variable after initialization has occurred. We'll fill in the structure after the source code we wrote on getting the device context earlier in the chapter.

The first variable in the structure is nSize, which is of the WORD data type. This variable contains the size of the structure using sizeof(PIXEL-FORMATDESCRIPTOR). By specifying the size of the structure, the OpenGL implementation has room to expand based on different sized structures. The second variable in the structure is nVersion, which is also of the WORD data type. This variable sets the version number of the data structure. Under the Win32 API, 1 is the only option available. The next parameter, dwFlags, is one of the most important in the entire structure because it specifies the properties of the pixel buffer. There are several flags we can set for this variable. The default flags we'll want for our graphics engine are PFD_DRAW_TO WINDOW, which tells the API to draw to the specified window, PFD_SUPPORT_OPENGL, which says the buffer supports OpenGL, and PFD_DOUBLEBUFFER, which tells the pixel buffer we want double buffering support. Double buffering is employed to reduce the flickering of graphics, which occurs when the screen is being updated in certain areas, by drawing all the contents of a scene to a buffer in the background and then quickly flipping the finished buffer to the foreground. Double buffering is a common standard among video card manufacturers these days, so you needn't be concerned that a card will not support it. In some cases, a video card can even support triple buffering, which offers two background buffers and one foreground buffer. We won't be taking advantage of triple buffering because double buffering suits our needs.

After the dwFlags variable comes the iPixelType variable, which is a byte. iPixelType specifies whether pixel data is stored in red, green, blue, and alpha (PFD_TYPE_RGBA) values or in a color index mode (PFD_TYPE_COLORINDEX), which is generally known as paletted mode. With today's graphics accelerators, there's no need to use paletted color modes, because most video cards support the RGBA pixel type. RGBA also allows us more flexibility when creating colors because we can easily manipulate the full amount of colors the video card allows without worrying about specifically defined limits. The next variable in the structure is cColorBits, which is also a byte. This variable uses the passed value of the color_bits variable to define the number of bit planes, or bits per pixel, for the rendered graphics excluding the alpha, which is meant for transparency. Accompanying the color buffer are four variables that allow you to set the appropriate color bit planes for their respective name.

The named color bits cRedBits, cGreenBits, cBlueBits, and cAlphaBits must be set to 0 when the pixel type has a value of PFD_TYPE_RGBA. The color bit planes can only have values set when iPixelType is set to PFD_TYPE_COLORINDEX (paletted mode). Unfortunately, using the PFD_TYPE_COLORINDEX pixel type is considered outdated by today's graphics standards because of the lush color modes modern video cards can handle. Paletted mode graphics were fairly common when video cards only had 2 and 4 MB of RAM and the high color depths weren't mainstream in gaming. Customized palettes were created to fulfill the lower color depths by creating the required colors in a list as opposed to combining R, G, B values together, which ended up saving vital memory. As games progressed from 8 bit to 16 bit, then to 32 bit, the need for palettes died off. Similar to the named color buffer bits, the named color shifts are used only when PFD_TYPE_COLORINDEX is specified for the pixel type. These variables indicate the appropriate spacing for each color bit. The variables cRedShift, cGreenShift, cBlueShift, and cAlphaShift will be set to 0 since we don't need them.

The cAccumBits byte specifies the total number of bit planes in the accumulation buffer. We aren't using the accumulation buffer in either our game or map editor, so we can set this value to 0. Much like the color bit planes, the accumulation buffer can set individual values for each accumulation bit plane. Since we don't want the accumulation buffer, we can set the variable bytes cAccumRedBits, cAccumGreenBits, cAccumBlueBits, and cAccum-AlphaBits to 0. It's unfortunate that all these bytes go unused, but we cannot lower our requirements to fit older and outdated rendering techniques when we are trying to write an advanced game engine. After the accumulation bit planes we specify the precision of the depth buffer (cDepthBits). The value we passed in the third parameter (depth_bits) will set the variable cDepthBits to the appropriate depth buffer bits. The depth buffer has an overloaded value of 32, so we don't have to specify a value when we call our Init method. By default, the value for the depth buffer bits should be 32, giving enough precision to clip things properly when sorting from back to front.

The cStencilBits byte specifies the bit depth of the stencil buffer. As mentioned previously, the stencil buffer is used to create cutouts and render special effects into special regions on the screen and certain areas in an object. We won't be using the stencil buffer in our game, so like many other variables in the PIXELFORMATDESCRIPTOR structure, we'll set this to 0. By now you're probably wondering how many more variables are left in this structure, and I can honestly say too many! Actually there are six variables left, but some are ignored by default and we'll spend less time discussing their functionality. After defining the depth of the stencil buffer, we specify the number of auxiliary buffers in the variable cAuxBuffers. The auxiliary buffer isn't a supported buffer through this implementation so we'll set this value to 0. The next variable in the structure is iLayerType, which specifies the type of plane the layer was on. This variable was used in earlier implementations, so we can set the value to 0 without worrying about compatibility issues.

The bReserved byte in the structure specifies how many overlay and underlay bit planes are to be created. We don't need any overlay or underlay planes, so we can set this variable to 0. Similar to iLayerType, the dwLayerMask variable was used in earlier implementations of OpenGL and is no longer used. For this reason we'll set the variable to 0. The dwVisibleMask byte specifies the transparent color or index of an underlay plane. Since we aren't using underlay planes at all, we can simply set this variable to 0, much like the other variables in the last portion of this structure. The final variable in this structure is dwDamageMask, which like many of the last variables in this structure is no longer used. If you haven't guessed by now, we'll set this variable to 0 and stop talking about each byte in this structure. A filled-in version of the PIXELFORMATDESCRIPTOR structure looks like the following:

```
pfd.nSize           = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion        = 1;
pfd.dwFlags         = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                      PFD_DOUBLEBUFFER;
pfd.iPixelType      = PFD_TYPE_RGBA;
pfd.cColorBits      = color_bits;
pfd.cRedBits        = 0;
pfd.cRedShift       = 0;
pfd.cGreenBits      = 0;
pfd.cGreenShift     = 0;
pfd.cBlueBits       = 0;
pfd.cBlueShift      = 0;
pfd.cAlphaBits      = 0;
pfd.cAlphaShift     = 0;
pfd.cAccumBits      = 0;
pfd.cAccumRedBits   = 0;
pfd.cAccumGreenBits = 0;
pfd.cAccumBlueBits  = 0;
pfd.cAccumAlphaBits = 0;
pfd.cDepthBits      = depth_bits;
pfd.cStencilBits    = 0;
pfd.cAuxBuffers     = 0;
pfd.iLayerType      = 0;
pfd.bReserved       = 0;
pfd.dwLayerMask     = 0;
pfd.dwVisibleMask   = 0;
pfd.dwDamageMask    = 0;
```

There are a lot of repetitive values in the code above. Many of the values could be set very quickly using different approaches to filling in the structure. I'm going to quickly discuss two other methods that can be used to fill in the same structure, eliminating the need to type all those variable names and values at the same time. The most common way to fill in the PIXELFORMATDESCRIPTOR structure variable is to set the values at the time of declaration of the variable, as seen in the snippet below:

```
PIXELFORMATDESCRIPTOR pfd =
{
sizeof(PIXELFORMATDESCRIPTOR),
1,
PFD_DRAW_TO_WINDOW | PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL,
PFD_TYPE_RGBA,
24,
0, 0, 0, 0, 0, 0,
0,
0,
0,
0, 0, 0, 0,
32,
0,
0,
        0,
        0,
        0, 0, 0
        };
```

The method above allows the programmer to quickly specify the values for the PIXELFORMATDESCRIPTOR, but only saves a handful of lines compared to the next method, which is my personal favorite. This next method is quite simple in its design — simply zero the structure and fill in the required bytes. I left this method until last because it's better to learn the proper way first, and then learn the alternatives. This final alternative is the simplest form of filling the PIXELFORMATDESCRIPTOR structure.

```
memset (&pfd, 0, sizeof(pfd));
pfd.nSize      = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion   = 1;
pfd.dwFlags    = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
pfd.cColorBits = color_bits;
pfd.cDepthBits = depth_bits;
```

This method is quick and to the point in its operations. When comparing the first example and this example, it's hard to imagine that they both accomplish the same goal. I apologize for putting you through the torture of discussing the pixel format structure, but it was important for you to understand how OpenGL deals with the pixel formats. Filling in the structure is the longest part of the initialization stage and we can now discuss the other functions that use this newly filled-in structure.

After filling in the PIXELFORMATDESCRIPTOR structure, we call the function ChoosePixelFormat, which as the name suggests chooses a pixel format. More specifically, the function will choose the appropriate pixel format based on the requirements we've set in the PIXELFORMATDESCRIPTOR structure. The function requires two variables to be passed. The first is the handle of our device context (hDC) and the second is our newly filled-in PIXELFORMATDESCRIPTOR structure (pfd). Create a local variable of the int data type called PixelFormat. This variable will retrieve the return value from the ChoosePixelFormat function call. If the returned value is 0, then an error occurred and we must display the appropriate error message, release the allocated device context, and exit from the initialization process, returning false. If the return value is not 0, then the function has succeeded and we can move to the next step in the initialization process. To release the device context we use the function ReleaseDC and supply both the handle of our window (hWnd) and our newly retrieved device context (hDC). By releasing the device context, we're freeing memory that would otherwise be allocated in the background. Although the size of the device context isn't huge, it is still a possible memory leak and we don't want those yucky things! Also, it is a good idea to nullify our device context to keep things consistent throughout our object. The following code snippet shows how to call the ChoosePixelFormat function and check for the appropriate return value.

```
PixelFormat = ChoosePixelFormat(hDC, &pfd);
if (PixelFormat == 0)
{
   MessageBox(hWnd, "Error: Can't Choose Pixel Format", "ERROR", MB_OK |
             MB_ICONERROR);
   ReleaseDC (hWnd, hDC);
   hDC = NULL;

   return (false);
}
```

If the previous function succeeds, we use the returned pixel format (PixelFormat) to set the pixel format that OpenGL will use when rendering images. This is a crucial step in our Init method because it signifies the end of the Windows initialization functions and the beginning of our Windows OpenGL initialization functions. The function SetPixelFormat is appropriately named for its function, which is to set the pixel format. When calling the function we pass our handle of a device context (hDC), the newly chosen pixel format (PixelFormat), and the address to our PIXELFORMAT-DESCRIPTOR structure (pfd). If the return value is non-zero, the function has succeeded. If the return value is 0, then the function has failed and, like all our other functions, we must display the appropriate error message and exit from the Init method. The following code snippet displays a working SetPixelFormat function with the appropriate error handling.

```
if (SetPixelFormat(hDC, PixelFormat, &pfd) == 0)
{
   MessageBox(hWnd, "Error: Can't Set The Pixel Format", "ERROR", MB_OK |
             MB_ICONERROR);
   ReleaseDC (hWnd, hDC);
   hDC = NULL;
   return (false);
}
```

With the pixel format set, we're almost finished initializing OpenGL; however, we still have several functions left. After setting the pixel format, we begin using the Windows OpenGL functions to create an OpenGL rendering device context. As stated before, OpenGL uses the gl namespace for all its functions. Windows-specific functions are different in that they use the wgl namespace. This means that all Windows-specific functions will have a "wgl" in front of their names. With this in mind, we'll need to create an OpenGL rendering context by using the function wglCreateContext and passing a handle to our device context (hDC). In order to finish the function call properly, we'll need to create a new public variable called glrc, which is of the HGLRC data type. This new variable will retrieve the return value from the function wglCreateContext and be used to create the OpenGL rendering context (to the handle of the device context). If the function call to wglCreateContext succeeds, the return value is the rendering context, which is passed to glrc. If the function fails, the return value is NULL and we should display the appropriate error message and exit from the method. A code snippet has been provided below to illustrate how the function works.

```
glrc = wglCreateContext(hDC);
if (glrc == NULL)
{
   MessageBox(hWnd, "Error: Can't Create GL Context", "ERROR", MB_OK |
            MB_ICONERROR);
   ReleaseDC (hWnd, hDC);
   hDC = NULL;
   return (false);
}
```

The final step in initializing OpenGL for rendering is to make the rendering context the current rendering context of the program or thread. Sounds confusing, but in plain English that means we're going to set the rendering context (glrc) as the default context for rendering images to the screen. A program can only have one rendering context unless it takes advantage of multithreading programming techniques. This topic is beyond the scope of this book, but I encourage you to research the topic further if you are interested in it. Thankfully, as mentioned earlier, this is the last step in the initialization process of OpenGL. The final function we'll use is wglMakeCurrent, which as discussed earlier makes the current rendering context for the thread or program. We'll pass our handle of a device context (hDC) and our newly created GL rendering context (glrc) to the function. If the return value is 0, the function has failed and we must display our final error message and exit the method. If the function returns a non-zero number, the function has succeeded and we are finally finished initializing OpenGL. The following code snippet illustrates how the wglMakeCurrent function should be executed with proper error checking.

```
if (!wglMakeCurrent(hDC, glrc))
{
   MessageBox(hWnd, "Error: Can't Make Current GL Context", "ERROR", MB_OK |
            MB_ICONERROR);
   wglDeleteContext(glrc);
   ReleaseDC (hWnd, hDC);
   glrc = NULL;
   hDC  = NULL;
   return (false);
}
```

We are now finished initializing OpenGL and we can return a true value in our Init method. This signifies that our Init method has succeeded and we can continue the initialization of our map editor. Although we're not finished discussing how OpenGL works and other functionality associated with it, we've still covered significant ground.

The entire Init method is shown below to display the final result of this section. I've included both the long and short methods of filling in the PIXELFORMATDESCRIPTOR structure. The long form has been commented out, which allows you to play with the settings to see the results. In future versions of the RASTER class we'll only display the short format to conserve space. And without further ado, the source code:

```
bool RASTER::Init(HWND hWnd, unsigned char color_bits, unsigned char depth_bits)
{
   PIXELFORMATDESCRIPTOR pfd;
   int PixelFormat;


   hDC = GetDC(hWnd);
   if (hDC == NULL)
   {
      MessageBox(hWnd, "Error: Can't Get Device Context for Window", "ERROR",
                MB_OK | MB_ICONERROR);
      return (false);
   }

   /* Original Method
   pfd.nSize           = sizeof(PIXELFORMATDESCRIPTOR);
   pfd.nVersion        = 1;
   pfd.dwFlags         = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                         PFD_DOUBLEBUFFER;
   pfd.iPixelType      = PFD_TYPE_RGBA;
   pfd.cColorBits      = color_bits;
   pfd.cRedBits        = 0;
   pfd.cRedShift       = 0;
   pfd.cGreenBits      = 0;
   pfd.cGreenShift     = 0;
   pfd.cBlueBits       = 0;
   pfd.cBlueShift      = 0;
   pfd.cAlphaBits      = 0;
   pfd.cAlphaShift     = 0;
   pfd.cAccumBits      = 0;
   pfd.cAccumRedBits   = 0;
   pfd.cAccumGreenBits = 0;
   pfd.cAccumBlueBits  = 0;
   pfd.cAccumAlphaBits = 0;
   pfd.cDepthBits      = depth_bits;
   pfd.cStencilBits    = 0;
   pfd.cAuxBuffers     = 0;
   pfd.iLayerType      = 0;
   pfd.bReserved       = 0;
   pfd.dwLayerMask     = 0;
   pfd.dwVisibleMask   = 0;
   pfd.dwDamageMask    = 0;
```

```
    */

    memset (&pfd, 0, sizeof(pfd));
    pfd.nSize      = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion   = 1;
    pfd.dwFlags    = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.cColorBits = color_bits;
    pfd.cDepthBits = depth_bits;



    PixelFormat = ChoosePixelFormat(hDC, &pfd);
    if (PixelFormat == 0)
    {
       MessageBox(hWnd, "Error: Can't Choose Pixel Format", "ERROR", MB_OK |
                  MB_ICONERROR);
       ReleaseDC (hWnd, hDC);
       hDC = NULL;
       return (false);
    }

    if (SetPixelFormat(hDC, PixelFormat, &pfd) == 0)
    {
       MessageBox(hWnd, "Error: Can't Set The Pixel Format", "ERROR", MB_OK |
                  MB_ICONERROR);
       ReleaseDC (hWnd, hDC);
       hDC = NULL;
       return (false);
    }

    glrc = wglCreateContext(hDC);
    if (glrc == NULL)
    {
       MessageBox(hWnd, "Error: Can't Create GL Context", "ERROR", MB_OK |
                  MB_ICONERROR);
       ReleaseDC (hWnd, hDC);
       hDC = NULL;
       return (false);
    }

    if (!wglMakeCurrent(hDC, glrc))
    {
       MessageBox(hWnd, "Error: Can't Make Current GL Context", "ERROR", MB_OK |
                  MB_ICONERROR);
       wglDeleteContext(glrc);
       ReleaseDC (hWnd, hDC);
       glrc = NULL;
       hDC  = NULL;
       return (false);
    }

    return (true);
 }
```

Now that we've created our Init method, we'll put it to good use by creating a new global variable called raster, which is a RASTER object. We'll call the raster Init method in our main body of source code after we've loaded our pop-up menus. Unlike our other function calls, we're going to check the return value for 0. If 0 is returned from our Init method, then we must exit from our program and return 0 again. By checking for 0, we are ensuring that OpenGL is initializing properly. If it doesn't, then the 0 return value is returned and the program cannot continue. The following source code displays the appropriate way of calling the Init method in the RASTER object.

```
PopupMenu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_POPUP_MENU));

if (!raster.Init(RenderWindow)) return (0);
```

Releasing OpenGL

Releasing OpenGL is a much easier process than initializing because we are releasing all the memory contexts as opposed to configuring them to run. Before writing the releasing source code, we must create a new method called Release in our RASTER class. The new method will have a handle to a window (hWnd) passed, allowing us to release our handle of a device context (hDC). The Release method has a return type of Boolean, which will return a default value of true if the method succeeded in releasing, and false if any of the functions failed.

When releasing OpenGL, the first thing we must do is check to see if our variables, hDC and glrc, are NULL. If either variable is NULL, then we must return a false value, indicating the release of OpenGL failed. By checking for NULL, we are making sure that OpenGL was actually initialized, because neither variable would be NULL if the API was properly initialized. There is no need to display an error message because this type of error is not significant enough to require one. After checking for NULL, we must make the rendering context no longer current. This will release the handle of a device context (hDC) from the GL rendering context (glrc) into their separate entities. To accomplish this we must once again call the function wglMakeCurrent, supplying NULL in both parameters. If the function succeeds, the return value is true. If the function fails, we must display an error message and return false. The source code below displays a proper call to wglMakeCurrent, with the proper error checking.

```
if (!wglMakeCurrent(NULL, NULL))
{
   MessageBox(hWnd, "Error: Release Of DC And RC Failed.", "Release Error",
            MB_OK | MB_ICONERROR);
   return (false);
}
```

After making the rendering context no longer current we can delete the rendering context by calling the function wglDeleteContext and passing our GL rendering context (glrc). The return type of the function is similar to all other functions in that it returns false if it fails and true if it succeeds. With this in mind, we'll display the proper error message and exit if an error exists. If the function succeeds, then we'll set our GL device context (glrc) to NULL, indicating that the variable is not allocated. The following code snippet displays the proper way to call the wglDeleteContext function.

```
if (wglDeleteContext(glrc) == false)
{
   MessageBox(hWnd, "Error: Release Rendering Context Failed.", "Release Error",
             MB_OK | MB_ICONERROR);
   return (false);
}
glrc = NULL;
```

The final function in the Release method is ReleaseDC. This function will release the handle of a device context from the window, freeing the allocated memory. When calling this function you must pass the window (hWnd) from which you would like to release the device context, and the handle of a device context (hDC) you want to free. If the function fails, like all the other functions, the return value is false and we must display the appropriate error message and exit from the method. If the function succeeds, we skip displaying the error message and set the hDC variable to NULL, indicating the variable has not been allocated or used. The following source code shows how the ReleaseDC function works.

```
if (ReleaseDC(hWnd, hDC) == false)
{
   MessageBox(hWnd, "Error: Release Device Context Failed.", "Release Error",
             MB_OK | MB_ICONERROR);
   return (false);
}
hDC = NULL;
```

If everything succeeds, a return value of true is executed and we exit the Release method. As if you haven't noticed, the Release method is fairly simple compared to its bigger brother Init. Thankfully, we're almost finished with the configuration source code of OpenGL and soon we'll be writing rendering source code. In the meantime, the source code for the Release method is shown here.

```
bool RASTER::Release(HWND hWnd)
{
   if (hDC == NULL || glrc == NULL) return (false);

   if (wglMakeCurrent(NULL, NULL) == false)
   {
      MessageBox(hWnd, "Error: Release Of DC And RC Failed.", "Release Error",
                 MB_OK | MB_ICONERROR);
      return (false);
   }

   if (wglDeleteContext(glrc) == false)
   {
      MessageBox(hWnd, "Error: Release Rendering Context Failed.", "Release
                 Error", MB_OK | MB_ICONERROR);
      return (false);
   }
   glrc = NULL;

   if (ReleaseDC(hWnd, hDC) == false)
   {
      MessageBox(hWnd, "Error: Release Device Context Failed.", "Release Error",
                 MB_OK | MB_ICONERROR);
      return (false);
   }
   hDC = NULL;

   return (true);
}
```

Similar to the Init method, the Release method must be added to the main body of our source code. Since this method unloads OpenGL, it must be added to the bottom of the source code to ensure the program has finished executing before it's called. The following source code displays the appropriate location for the RASTER object's Release method.

```
while (1)
{
   if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
   {
      if (msg.message == WM_QUIT) break;
      TranslateMessage(&msg);
      DispatchMessage (&msg);
   }
}
raster.Release(RenderWindow);
```

The Different Matrices

OpenGL has three matrices that control how the images are rendered. All three of the matrices (projection, model view, and texture) are 4x4 matrices and are stored slightly differently than conventional matrices used in calculations. Figure 2.2 displays the conventional storage methods of both OpenGL and math.

$$\begin{array}{cc}
\text{Normal} & \text{OpenGL} \\
\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} &
\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}
\end{array}$$

Figure 2.2: Matrix storage

OpenGL uses a transposed matrix system as opposed to the "normal" matrix storage system. All the matrices within OpenGL are stored this way, unless otherwise specified in the documentation, so there shouldn't be any sort of matrix compatibility problem within standard OpenGL code. When you supply your own matrices to manipulate a GL matrix, you'll have to change the way you store the data to OpenGL's method. To resolve the transposing issue in OpenGL, some video card manufacturers ship with GL extensions that allow you to quickly transpose the given matrix.

Each matrix has its own unique function that can drastically alter the way images are rendered in OpenGL. The description and functionality of each matrix have been provided below.

Projection

The projection matrix describes how the vertex coordinates are displayed on the screen, based on the field of view and near and far planes. The field of view (FOV) value is generally anywhere from 45 to 90. The projection matrix only needs to be set up once unless the size of the rendering window changes, in which case we would resize the projection to the newly sized window. The near and far planes of the projection matrix allow all objects to be cut out of the picture if they are closer than the near plane or farther than the far plane. If primitives are cut out of the rendered image, hopefully the result would be better performance.

Model View

The model view matrix is applied to each vertex coordinate and it modifies their locations. In other words, when we send the vertex coordinates to build our primitives, the model view matrix is then applied to each coordinate, which then determines its final destination. A good majority of our rendering source code will be sent directly to the model view matrix and, for that reason, it's probably the most important of the three matrices.

Texture

The texture matrix is used to manipulate how textures are drawn on specific objects. By default, the texture matrix is set to identity (default) matrix. When rendering primitives in OpenGL, it is not necessary to modify this matrix because of its default value. When modified, the texture matrix is fairly slow because you must continually save, manipulate, and finally restore the matrices for each texture matrix operation on a per-primitive, per-frame basis. If we don't continually restore to our original matrices, the next primitive to be rendered will use the existing texture matrix, causing issues when trying to properly texture items. We will take advantage of the texture matrix for our lighting system later in the book, so it's a good idea to remember this!

Accompanying the matrices are numerous functions for rotation, translation, multiplication, and more that allow you to manipulate them without the need to type matrix calculations manually. In many cases, the matrix manipulation is processed by the video card, so writing your own matrix manipulation routines could end up being slower to compute than allowing OpenGL to process them. If you were to use processor-specific enhancements like AMD's 3DNow! and Intel's SSE technologies, the performance of the calculations could be drastically improved over conventional math functions; however, this is beyond the scope of this book.

Matrix Functions

As discussed in the previous section, OpenGL has many functions for manipulating the different matrices. In this section we'll discuss several commonly used functions that will enable us to rotate the view, translate movements, save/restore each matrix, and reset each matrix.

Selecting a Matrix

Throughout the rendering process, you'll probably be faced with having to switch between the model view matrix and the texture matrix for placing objects and texturing them, respectively. OpenGL has a built-in function that allows you to switch between the model view, projection, and texture matrices with a simple function call to glMatrixMode. With the function call we pass the matrix name we want to use, i.e., GL_PROJECTION, GL_MODELVIEW, or GL_TEXTURE, for the respective matrix.

Setting the Identity Matrix

When OpenGL is initialized, all three matrices initially have a value of the identity matrix. In case you're wondering what the identity matrix is, it's the default value matrix. Figure 2.3 displays a proper 4x4 identity matrix used in OpenGL.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.3: Identity matrix

The values of the identity matrix aren't terribly important, since we won't have direct access to them, but it's something to keep in mind when programming. At times we'll need to reset the current matrix to the default values of the identity matrix. This can be accomplished by using the OpenGL command glLoadIdentity.

Rotating a Matrix

Rotating matrices in OpenGL is a very common practice, especially when writing games/applications that have odd camera views. For this reason, OpenGL provides a simple interface to rotate the current matrix around the specified axis by the angle specified. This great functionality allows us to customize the X, Y, and Z rotations differently without worrying about them conflicting with each other. A good example of multiple matrix rotations is the mouse-look functionality many first-person shooters have. The player can look left and right, which is a rotation around the x-axis, and can also look up and down, which would be rotating around the y-axis.

The function glRotate has four parameters — the angle of rotation and three flags that identify the axis (x, y, z) to be rotated. Conveniently, glRotate supports both floats and doubles as parameters using the first letter of the data type as the ending letter of the function. As an example, we'll rotate the x-axis by 45 degrees using the float version of the function.

```
glRotatef (45.0f, 1.0f, 0.0f, 0.0f);
```

Alternatively, we could rotate the same axis by 45 degrees again using the double version of the function as seen in the following line.

```
glRotated (45.0, 1.0, 0.0, 0.0);
```

As you can see, rotating the current matrix is simple! If we wanted to rotate the x- and y-axes in different amounts, we would simply call the function twice, changing the parameters accordingly. When the glRotate function is called, each axis generates its own 4x4 matrix. Figure 2.4 displays the matrices for their respective axis.

$$\begin{array}{ccc}
\text{X-axis} & \text{Y-axis} & \text{Z-axis} \\
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} &
\begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} &
\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{array}$$

Figure 2.4: Rotation matrices

And you probably laughed when your high school math teacher said to you, “One day you’ll need to know this stuff!”

Translating a Matrix

Translating the current matrix moves the current coordinates to the X, Y, Z locations specified by the user. This function is used when we want to move primitives throughout our 3D world without having to manually modify the current matrices or individual x, y, z variables of the primitive. Later in the book when we write our 3D engine, we’ll continually translate the model view matrix to fake walking through the 3D world.

To translate the current matrix to a new coordinate system, we use the function glTranslate and pass the new X, Y, and Z locations. Similar to the glRotate function, glTranslate supports both float and double parameter data types, with their respective name as the final character in the function. If we wanted to move across the X coordinate using floats as the parameter data type, we would type the following:

```
glTranslatef (128.0f, 0.0f, 0.0f);
```

In the event we wanted to use doubles as our parameter data type, obviously we would type the following:

```
glTranslated (128.0, 0.0, 0.0);
```

As you can see by the function call, the matrix manipulations in OpenGL are very simple! In many cases we'll pass integers rather than use doubles because we don't need the precision of doubles. The translation matrix is stored as a conventional 4x4 matrix and looks like the following figure.

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.5: Translation matrix

Scaling a Matrix

Scaling is performed when you multiply the current matrix by a scaling matrix. When making games, scaling becomes a great asset for certain types of special effects that rely on primitives growing and shrinking in size. OpenGL has a built-in function for scaling called glScale, which accepts three scaling values for the x-, y- and z-axes. Similar to glTranslate and glRotate, the glScale function has both float and double parameter versions. By default, the scale of the model view matrix (the drawing matrix) is set to 1.0 in all three axis. This means that primitives will be drawn in their normal size.

If we wanted to scale the current matrix to 50% of the original size, we could type the following:

```
glScalef (0.5f, 0.5f, 0.5f);
```

Alternatively, we could use the double version of this function by typing the following:

```
glScaled (0.5, 0.5, 0.5);
```

If you haven't noticed by now, all the matrix manipulation functions seem to have the three axes (x, y, and z) as parameters plus any extra parameters that are relevant to the function. This brings a fair amount of consistency throughout the API. After the current matrices are multiplied with the scaled matrices, the results are passed into the matrix below.

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.6: Scaling matrix

Saving and Restoring the Matrix

Throughout the rendering process we'll need to save and restore the current matrices. OpenGL has functions for both saving and restoring the matrices named glPushMatrix and glPopMatrix, respectively. When the user calls the glPushMatrix function, the current matrix is saved in the matrix stack. When the user calls the glPopMatrix function, the last matrix saved in the matrix stack is restored as the current matrix. This allows a great deal of flexibility when programming because we can set the default view, save the matrix into the stack, then modify the matrix further through rotation, translation, or scaling, and restore the original matrix once we are finished rendering. Neither function requires any parameters.

We can now put our knowledge of matrices to work and begin using the different matrices of OpenGL.

Resizing a GL Window

The final step in the initialization process of OpenGL is to set the viewport of the rendering window. By setting the rendering viewport we're describing an area of our window to which we want to render. This allows the programmer to specify to render different scenes in the same window by specifying their X, Y locations and their width and height. We'll set the viewport to the entire size of the rendering window, allowing the greatest visibility for creating maps. To begin, let's create a new function in our main program called ResizeGLWindow. There will be two parameters supplied to the function: width and height. Both parameters are of the long data type and will be used to resize the viewport.

To resize the viewport in OpenGL, we use the function glViewport and supply four parameters, which specify the X and Y coordinates and the width and height of the viewport. The x and y parameters are of the GLint data type, which is OpenGL's own integer data type and can be cast from a regular integer. We'll set both x and y to 0, indicating we want the viewport to begin at the corner of the screen. After the X and Y coordinates, we specify the width and height of the viewport. The width and height are of the GLsizei data type, which is another form of OpenGL integer. We'll use the width and height variables passed into the function and cast them as GLsizei data type. The glViewport function call would look similar to the following:

```
glViewport(0, 0, (GLsizei) width, (GLsizei) height);
```

If you're interested in the math behind the glViewport function, Figure 2.7 displays the calculations behind it. One of the benefits to using high-level 3D programming APIs is that the math calculations in many cases are hidden from the program. As with OpenGL, although hidden from the program, they are widely available if the programmer needs them.

$$xw = (xnd + 1)\left(\frac{width}{2}\right) + x$$

$$yw = (ynd + 1)\left(\frac{height}{2}\right) + y$$

Figure 2.7. Viewport calculations

After setting the viewport we must set the orthographic projection in the projection matrix. By setting the orthographic projection, we are setting the rendering boundaries of the screen for the left, right, top, bottom, and near and far planes. Without setting this variable or the perspective, the primitives you try to render will not draw properly, if they even draw. To set the orthographic projection we call a simple series of commands that involve switching to the projection matrix, loading the identity matrix, setting the orthographic projection, and switching back to the model view matrix. To begin, we'll switch to the projection matrix by running the glMatrixMode command and supplying the GL matrix name, GL_PROJECTION, as the only parameter.

After switching to the projection matrix, we set its values to those of the identity matrix (default) by calling the function glLoadIdentity. The glLoadIdentity function does not have any parameters associated with it. The next step is to set the orthographic projection by calling the OpenGL function glOrtho and supplying the clipping plane information in this specific order: left, right, bottom, top, near, and far. Unlike the viewport, we can set the plane boundary values to non-positive numbers. For this reason, I've set both the left and bottom planes to –200, the right and top to 200, the nearest plane to –2000, and the farthest plane to 2000. These values allow OpenGL to render primitives to the screen that are between the positions –200 and 200. This also gives us a good render width of 400 points in the OpenGL coordinate space. In case you are wondering where the 200 value came from, I randomly picked the number! Normally it is recommended to set the near and far planes to values larger than their counterparts, because they represent the distances from the user to the object and how far in the distance they can be seen before they are clipped out. Since we are writing a map editor, I chose the near/far values to be 10 times the original size of the clipping boundaries, giving us plenty of distance for the objects.

After setting the orthographic projection we must switch back to the model view matrix by calling the function glMatrixMode and supplying the GL matrix name, GL_MODELVIEW. This completes the code for the ResizeGLWindow function. The source code for the function follows.

```
void ResizeGLWindow(long width, long height)
{
   glViewport(0, 0, (GLsizei) width, (GLsizei) height);
   glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(-200,200, -200,-200, -2000,2000);
   glMatrixMode(GL_MODELVIEW);
}
```

After OpenGL is initialized we must set the appropriate projections in both our map editor and game engine by getting the appropriate window coordinates from the command GetClientRect and supplying the width and height values when calling the ResizeGLWindow function. To begin, we must call the function GetClientRect and specify the RenderWindow variable as the first parameter, then the address of the rect variable, which will retrieve the returned window coordinates. Next, we call the ResizeGLWindow function and subtract the leftmost coordinate from the rightmost coordinate to calculate the width parameter. The height parameter is calculated by subtracting the topmost coordinate from the bottommost coordinate. The following source code displays the proper window calculations after the initialization of OpenGL.

```
if (!raster.Init(RenderWindow)) return (0);

GetClientRect (RenderWindow, &rect);
ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);
```

Unfortunately, both our game engine and map editor use different calculations for the projection matrix. To keep things consistent between both projects, we'll use the same function name of ResizeGLWindow and simply change the projection matrix calculations.

You may be wondering what the differences between each project are, and I'll explain that right now. The game engine will be full screen and won't have to worry about window sizes changing. Also, the game engine will use a field-of-view rendering system, which displays primitives within the user-specified degrees of view. Each primitive is also drawn using the appropriate perspective as opposed to being flat. Our map editor is different in that all primitives drawn are flat and the window size can change randomly and must be recalculated. The rendering window size must be recalculated because otherwise the original viewport will be used even if the sizes of the rendering windows are different. For this reason, we must write the source code to move the rendering window and call our newly created function, ResizeGLWindow, when a resize occurs.

When the size of a window changes, the window message WM_SIZE is sent by the program, specifying the new window width, height, and modification flags. Just like the WMCommand function, we'll need to create a function that will handle all resizing messages sent. To avoid typing the same parameters again, simply copy and paste the source code from the WMCommand function and change the name of the new function to WMSize. This function will handle all window sizing messages (WM_SIZE) that are sent to our message handler (WinProc). In order for this function to be called, we must add the new message (WM_SIZE) into the message handler case statement and supply the four required parameters, as seen in the following code snippet.

```
switch (msg)
{
   case WM_DESTROY: PostQuitMessage(0); break;
   case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
   case WM_SIZE: WMSize (hWnd, msg, wParam, lParam); break;
   case WM_RBUTTONUP: DisplayPopupMenu(LOWORD(lParam), HIWORD(lParam)); break;
}
```

In the WMSize function we declare a new variable called rect, which is of the RECT data type. We'll use this variable to get the new window coordinates and set our child window sizes accordingly. First we call the GetClientRect function and supply our main handle of a window (Window) and the address of the rect structure. After the function is called the coordinates of the window are returned in the rect structure. Based on the returned window values we'll move the rendering window to its new location by running the function MoveWindow and supplying the handle of the window we want to move, the X and Y coordinates of the window, the width and height of the window, and finally a Boolean flat, which determines whether the window should be redrawn or not.

Since our main window is a parent window, it will always be updated and sized appropriately. For this reason we'll set the handle of the window to RenderWindow, which is our rendering window for OpenGL. The new X coordinate of the window will be set to DEFAULT_BUTTON_WIDTH, the constant we created for the button width. By setting the X coordinate of the rendering window to this value, we are able to ensure the window will begin at the end of the button width and nowhere else. The new button Y coordinate will be at the top of the screen, which is 0. The width and height variables are quite different in that they actually need to be computed mathematically. The width is calculated by subtracting the farthest left coordinate (rect.left) from the farthest right coordinate (rect.right) and subtracting the width of the default button. By subtracting the left from the right we get the width of the window, then we subtract the default button width because the rendering window starts at the value of DEFAULT_BUTTON_WIDTH. The height of the window is calculated by subtracting the topmost coordinate (rect.top) from the bottommost coordinate (rect.bottom). The final parameter is the repaint flag, which can be either true or false. With each window resizing we want the computer to automatically draw the updates, so we'll set this parameter to true. Since our buttons are located on the left side of the screen, we don't need to adjust their sizes because the right side of the application will always be altered.

After resizing all our windows, we must retrieve the new coordinates of the rendering window and resize the OpenGL rendering context using the newly retrieved window values. To begin, we call the GetClientRect function and supply the RenderWindow variable as our window and the address of the rect structure to which the window coordinates are returned. After calling GetClientRect, we run our newly created ResizeGLWindow function and supply the new width and height of our rendering window. The new width is calculated by subtracting the leftmost (rect.left) coordinate from the rightmost coordinate (rect.right) to produce the width. The new height is calculated by subtracting the topmost coordinate (rect.top) from the bottommost coordinate (rect.bottom) to produce the height of the window. As we add more windows to our map editor, we'll have to add the windows to our WMSize function to allow them to resize appropriately, but for the moment we are finished writing the resizing source code. The source code for the WMSize function has been provided below for study.

```
void WMSize(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   RECT rect;

   GetClientRect (Window, &rect);
   MoveWindow (RenderWindow, DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-
            DEFAULT_BUTTON_WIDTH, rect.bottom-rect.top, true);

   GetClientRect (RenderWindow, &rect);
   ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);
}
```

This code snippet represents the end of the "Resizing a GL Window" section. In the next section we'll discuss which default rendering options we'll set active.

Setting the Rendering Defaults

After initializing OpenGL and sizing the projection matrix, we set the default rendering options. This allows our map editor to begin rendering any data without any further configuration required. Features such as backface culling, blending, texturing, depth buffering, and more can all be controlled using OpenGL's simple enable/disable functionality. To begin we'll need to create a new function called SetGLDefaults. This function does not require any passed parameters, since all the functions we're enabling/disabling are defined within OpenGL.

As this is the first example of using OpenGL, it's got a fairly limited amount of functionality that we can expose at this point. Two major functions that should have their flags set are the depth buffer, which should be enabled, and backface culling, which we should disable. As discussed earlier, the depth buffer sorts all 3D primitives drawn from back to front and clips out all data lying behind other data. This function is a necessity since the manual calculations are too expensive on the processor. To enable the depth buffer, we use the OpenGL command glEnable and specify the GL constant for the depth buffer, GL_DEPTH_TEST. If we wanted to disable the depth buffer we could simply run the glDisable command and once again specify the GL constant for the depth buffer, GL_DEPTH_TEST.

Backface culling, sometimes referred to as *polygon culling*, is the process of eliminating primitives that aren't drawn in the user-specified direction (clockwise or counterclockwise). This is just one optimization technique programmers can use when trying to increase performance in their games. Since our map editor does not rely on the highest speeds for operation and requires all primitives to be drawn, we'll disable the function by calling the glDisable command, and supply the GL constant for polygon culling, GL_CULL_FACE. The source code below displays our newly created SetGLDefaults function, with the appropriate default rendering values set.

```
void SetGLDefaults()
{
   glEnable (GL_DEPTH_TEST);
   glDisable (GL_CULL_FACE);
}
```

As mentioned earlier, the call to the SetGLDefaults function should take place after the initialization of OpenGL has happened and the rendering window has been properly sized.

Drawing "Things" in OpenGL

In this section, we'll discuss how to draw primitives in OpenGL and finish our chapter example by drawing several primitives in our map editor. To begin, we need to create a new function called Render, which, as the name implies, will render our drawn primitives to the screen. When beginning the rendering process, the first thing we do is clear the rendering buffers used; in this case we would clear the depth buffer and the color buffers. The GL function glClear will clear the buffers specified by the user. The glClear function can be called numerous times to clear each buffer specified by the user, or we can use a bitwise-inclusive OR operator and specify multiple buffers at once. Both methods accomplish the same function; however, one seems slightly more efficient than the other, depending on your rendering needs. In our examples we'll call the glClear function once and include both relevant buffers.

As discussed earlier in the chapter, the depth buffer controls the rendering order of our primitives. We clear the depth buffer in each frame so the data we are currently rendering doesn't conflict with the depth values of primitives drawn in previous frames. When using glClear, we use the constant GL_DEPTH_BUFFER_BIT to specify that we want the depth buffer to be cleared. The other buffer we must clear is the color buffer, which stores all the relevant color values from the previously rendered frame. The GL constant for the color buffer is GL_COLOR_BUFFER_BIT.

After clearing the buffers it's a good idea to set the color buffer clearing color. Although it sounds confusing, we're simply filling the color buffer with a default value before we render our primitives. To set the clearing color we use the OpenGL command glClearColor and supply four unique values (red, green, blue, alpha channels) that create the color. Each parameter is of the GLclampf data type, which is actually a regular float data type. All four parameters have a range between 0.0 (the lowest intensity) and 1.0 (the highest intensity) of their respective component. In our example we'll set the values to 0.6f, 0.6f, 0.6f, and 1.0f, which would give us a nice shade of gray. If we were to change the values to 1.0f, 0.0f, 0.0f, 1.0f we would have a solid red color as the background color. Similarly, by changing the second or third parameters to 1.0f, their respective colors would be at full intensity as well. In case you are wondering, the fourth parameter, the alpha channel, controls the transparency of the entire color. If we were to specify 1.0f, 0.0f, 0.0f, 0.5f, our color would be a full-intensity red with 50% transparency. If the alpha were to change to 0.25f, then the color would be less visible because the transparency is lower. I encourage you to play with the glClearColor function and see what options are available to you for customization.

After setting the clear color, we can start writing the rendering code. To begin, we must reset the model view matrix (our default matrix) by calling glLoadIdentity. This ensures that our map editor has a default matrix before we start doing our rotations and translations of the matrix. Next we must save the current matrix using the OpenGL command glPushMatrix. By saving the current matrix we are keeping the default matrix in the matrix stack for when we are done with all our calculations. Rotating the camera isn't a big concern at the moment, but normally camera rotations would happen right after glPushMatrix when we want a first-person view of the screen. After the rotations are set, we set the camera's X, Y, Z position using the GL function glTranslatef and supplying the X, Y, Z location parameters in float format. We'll use the values of 0,0,0 for our location since this is our first example. These values will be changed later as we modify the code to allow for movement through the 3D world.

Drawing primitives in OpenGL is nothing more than a rendering hamburger! Although this sounds ridiculous, there are only three steps required when drawing any type of primitive. To begin drawing primitives we must call the function glBegin and supply the OpenGL primitive mode constant. Then we add the appropriate contents, such as vertex coordinates to shape the primitive using the command glVertex, and finally we finish the drawing process by calling the function glEnd, which stops the drawing process.

The primitive mode constants define how each primitive is created. For instance, if we specify GL_POLYGON as the parameter for the glBegin function, each vertex we supply will be a side of our polygon we're creating. The primitive modes include points, lines, triangles, quads (squares), and polygons. Figure 2.8 displays each primitive mode and respective shape.

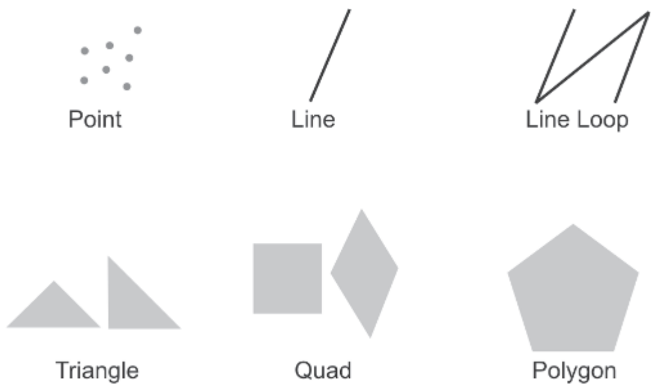


Figure 2.8: OpenGL primitive modes

Although these shapes don't look impressive, they are the building blocks to creating a 3D game. For that reason, we'll discuss what the different primitive modes draw in the following sections.

Points

A point is a circle drawn at a specified vertex coordinate pair. The size of the point can be changed using the OpenGL command glPointSize. When using glBegin, use the GL_POINTS constant to specify the points primitive mode. There is no physical limit to the number of points you can specify using the glVertex function.

Lines

Unlike traditional 2D graphics programming, lines can be drawn in three dimensions, provided they meet the basic requirement of having a pair of vertex coordinates for each line. The programmer can draw as many lines as needed using the glVertex function. The line width can be changed by calling the OpenGL function glLineWidth and specifying the width as the only parameter. To create a line, we use the GL_LINES constant in glBegin.

Triangles

A triangle is one of the fundamental shapes used when programming any type of three-dimensional application. Obviously, each triangle needs three vertex coordinates for it to be created. We can also continually send vertex coordinates in pairs of three to draw our primitives. To create a triangle we use the constant GL_TRIANGLES as the parameter in the glBegin function.

Quads

A quad is short for quadrilateral, which is a four-sided primitive. When creating this shape, we specify the vertex coordinates in groups of four. If there are fewer than four vertex coordinates given, the object is automatically discarded. To create a quad, we use the constant GL_QUADS in glBegin.

Polygons

The final primitive we'll discuss is a polygon, or multisided primitive. There are no physical limits to the number of points a polygon can have; however, some video card vendors may put extremely high limits on their cards that are not generally noticed. Due to the flexibility of creating the polygons by sending vertex coordinates, we cannot create multiple polygons within one glBegin/glEnd pair. Each polygon must be created by itself. To create a polygon we use the GL constant GL_POLYGON.

These primitive modes are the five main types of modes used when rendering. There are several offshoots of each that optimize the speed and the way each primitive is created; however, we'll discuss this later in the book. For our example we'll be creating a simple triangle that begins at the middle of the screen and reaches to the outer limit of the right side. In our Render function, we must first describe what type of primitive we are going to create by calling the command glBegin and supplying the constant GL_TRIANGLES. As discussed earlier, we must supply the vertex coordinates for each point in our triangle using the command glVertex. We'll use the float version of this command and specify coordinates for all three axes, which means we'll use the command glVertex3f. Once we've set the triangle's three points, we must call the OpenGL function glEnd to finalize the drawing process of the specified triangle.

The triangle we are going to draw will start in the middle of the screen, which is located at 0.0, 0.0, 0.0. If we specify 1.0 in the x- or y-axis (the first and second parameter), the point will be located at the outer edge of the axis on our rendering window. Similarly, if we want to go in the opposite direction we simply use a value of –1.0. The triangle we're creating starts in the middle and ends in the top right corner. The following code snippet displays the source code required to draw a triangle in OpenGL.

```
glBegin (GL_TRIANGLES);
   glVertex3f (0.0f, 0.0f, 0.0f);
   glVertex3f (0.0f, 1.0f, 0.0f);
   glVertex3f (1.0f, 1.0f, 0.0f);
glEnd();
```

Chapter Example

In the example source code that follows I've added several other primitives such as quads, polygons, and lines to give you a better idea of how primitives are created.

After we've created all our primitives we must call the glPopMatrix command to restore the previous matrix. This will ensure the state of our matrix will be perfect when the rendering is complete and we won't have any holes where matrices are being stored into the stack and not released afterward.

The final step in rendering scenes in OpenGL is to swap the back buffer with the front buffer. When each frame is rendered, all the content is drawn to the back buffer and not drawn on screen until the back buffer and front buffer are swapped. To swap the buffers we use the function SwapBuffers and specify our handle of a device context (hDC) in our raster class as the parameter. By supplying the handle we're providing the information necessary to flip between the front and back buffers in that specific context. If we didn't swap the buffers, nothing would be displayed on screen and this entire example would be pretty useless! Thankfully, I won't torture you further. The following source code shows examples of everything we've learned so far.

ex2_1.cpp:

```
#include <windows.h>
#include <winbase.h>
#include <stdio.h>

#include "resource.h"

#include "raster.h"

#define DEFAULT_BUTTON_WIDTH   100
#define DEFAULT_BUTTON_HEIGHT  20

HINSTANCE GlobalInstance;
HMENU     Menu;
HMENU     PopupMenu;
HWND      Window;
HWND      RenderWindow;
HWND      bCreateWall;
```

```
RASTER   raster;


void ResizeGLWindow(long width, long height)
{
   glViewport(0, 0, (GLsizei) width, (GLsizei) height);
   glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(-200,200, -200,-200, -2000,2000);
   glMatrixMode(GL_MODELVIEW);
}


void SetGLDefaults()
{
   glEnable (GL_DEPTH_TEST);
   glDisable (GL_CULL_FACE);
}


void Render()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glClearColor (0.6f, 0.6f, 0.6f, 1.0f);


   glLoadIdentity();
   glPushMatrix();
      glTranslatef (0.0f, 0.0f, 0.0f);


      glBegin (GL_TRIANGLES);
         glVertex3f (0.0f, 0.0f, 0.0f);
         glVertex3f (0.0f, 1.0f, 0.0f);
         glVertex3f (1.0f, 1.0f, 0.0f);
      glEnd();


      glBegin (GL_QUADS);
         glVertex3f (0.05f, -0.05f, 0.0f);
         glVertex3f (0.95f, -0.05f, 0.0f);
         glVertex3f (0.95f, -0.95f, 0.0f);
         glVertex3f (0.05f, -0.95f, 0.0f);
      glEnd();


      glBegin (GL_POLYGON);
         glVertex3f (-0.25f, -0.25f, 0.0f);
         glVertex3f (-0.50f, -0.125f, 0.0f);
         glVertex3f (-0.75f, -0.25f, 0.0f);
         glVertex3f (-0.875, -0.5f, 0.0f);
         glVertex3f (-0.75f, -0.75f, 0.0f);
```

```
        glVertex3f (-0.50f, -0.875f, 0.0f);
        glVertex3f (-0.25f, -0.75f, 0.0f);
        glVertex3f (-0.125f, -0.5f, 0.0f);
      glEnd();


      glBegin (GL_LINES);
        glVertex3f (-0.25f, 0.25f, 0.0f);
        glVertex3f (-0.75f, 0.75f, 0.0f);
      glEnd();


   glPopMatrix();


   SwapBuffers (raster.hDC);
}


LRESULT CALLBACK MapDetailsDlgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM
                    lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, "Map Name");

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                    0, (LPARAM)"Erase Me");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_RESETCONTENT,
                    0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                    0, (LPARAM)"Exit");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                    0, (LPARAM)"Get Fragged");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_SETCURSEL,
                    0, 1);

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                    0, (LPARAM)"Erase Me");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_RESETCONTENT,
                    0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                    0, (LPARAM)"Single Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                    0, (LPARAM)"Multi Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_SETCURSEL,
                    0, 1);
      } break;
```

```
        case WM_COMMAND:
        {
           if (wParam == IDOK)
           {
              long level_rule = SendDlgItemMessage (hWnd,
                        IDC_MAP_DETAILS_LEVEL_RULES, LB_GETCURSEL, 0, 0);
              long level_type = SendDlgItemMessage (hWnd,
                        IDC_MAP_DETAILS_LEVEL_TYPE, CB_GETCURSEL, 0, 0);

              char temp[500];

              sprintf (temp, "Level Type: %i\r\nLevel Rule: %i\r\nOK Button!",
                        level_type, level_rule);
              MessageBox (hWnd, temp, "OK", MB_OK);

              EndDialog (hWnd, 0);
           }
           else if (wParam == IDCANCEL)
           {
              MessageBox (hWnd, "Cancel Button!", "Cancel", MB_OK);
              EndDialog (hWnd, 0);
           }
        } break;
     }
     return (0);
  }

  void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
  {
     if (lParam == (LPARAM)bCreateWall) MessageBox (Window, "You Pressed
               bCreateWall", "Congrats!", MB_OK);
     else if (wParam == ID_FILE_EXIT) PostQuitMessage(0);
     else if (wParam == ID_DRAWING_WIREFRAME)
     {
        CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
        CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);
     }
     else if (wParam == ID_DRAWING_SOLID)
     {
        CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_CHECKED);
        CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_UNCHECKED);
     }
     else if (wParam == ID_MAP_DETAILS) DialogBox (GlobalInstance,
              MAKEINTRESOURCE(IDD_MAP_DETAILS), NULL, (DLGPROC)MapDetailsDlgProc);


     // Pop-up Menu Items
     else if (wParam == ID_POPUP_MOVE) MessageBox (Window, "Move", "Click", MB_OK);
     else if (wParam == ID_POPUP_DELETE) MessageBox (Window, "Delete", "Click",
               MB_OK);
     else if (wParam == ID_POPUP_TEXTURE) MessageBox (Window, "Texture", "Click",
               MB_OK);
```

```
   else if (wParam == ID_POPUP_DUPLICATE) MessageBox (Window, "Duplicate",
            "Click", MB_OK);
}


void DisplayPopupMenu(long x, long y)
{
   HMENU temp = GetSubMenu(PopupMenu, 0);
   TrackPopupMenu(temp, TPM_LEFTALIGN|TPM_RIGHTBUTTON, x, y, 0, Window, NULL);
}


void WMSize(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   RECT rect;

   GetClientRect (Window, &rect);
   MoveWindow (RenderWindow, DEFAULT_BUTTON_WIDTH, 0,
           rect.right-rect.left-DEFAULT_BUTTON_WIDTH, rect.bottom-rect.top, true);

   GetClientRect (RenderWindow, &rect);
   ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);
}


LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
      case WM_SIZE: WMSize (hWnd, msg, wParam, lParam); break;
      case WM_RBUTTONUP: DisplayPopupMenu(LOWORD(lParam), HIWORD(lParam)); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}


int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
           int CmdShow)
{
   WNDCLASS   wc;
   MSG        msg;
   RECT       rect;

   GlobalInstance   = hInstance;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "ME";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL,"Error: Cannot Register Class","ERROR!",MB_OK);
      return (0);
   }

   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
               0, 0, 640, 480, NULL, NULL, hInstance, NULL);

   if (Window == NULL)
   {
      MessageBox (NULL,"Error: Failed to Create Window","ERROR!",MB_OK);
      return (0);
   }
   GetClientRect (Window, &rect);


   bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE,
               0, 100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window,
               NULL, hInstance, NULL);


   RenderWindow = CreateWindow("STATIC", NULL, WS_CHILD | WS_VISIBLE | WS_BORDER,
               DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-DEFAULT_BUTTON_WIDTH,
               rect.bottom-rect.top, Window, NULL, hInstance, NULL);


   Menu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_MENU));
   SetMenu (Window, Menu);

   PopupMenu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_POPUP_MENU));


   if (!raster.Init(RenderWindow)) return (0);

   GetClientRect (RenderWindow, &rect);
   ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);

   SetGLDefaults();

   while (1)
   {
      Render();

      if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
      {
         if (msg.message == WM_QUIT) break;
         TranslateMessage(&msg);
         DispatchMessage (&msg);
```

```
        }
    }

    raster.Release(RenderWindow);


    return (1);
}
```

Conclusion

In this chapter we learned about OpenGL and how we initialize, render, and release the rendering API. Unfortunately, much of this code was a necessary evil for the rest of the book, which will discuss the more interesting parts of the game development process and how to make OpenGL work in your favor! With the information you learned in this chapter, you should be able to create simple OpenGL primitives on the screen.

In the next chapter we'll discuss our map format in detail and begin writing our map editor functionality.

Chapter 3

Defining the Map Format

In Chapter 3 we'll discuss the process of creating the map format for our game. When creating our map format, there are many features to consider such as object creation, lighting, cameras, texturing, starting positions, and much more. Because each game is different in design, we must customize our map format to our specific needs. When designing a new game, it's a good idea to brainstorm the rendering technologies you want to use within your map. Obviously when designing a map format there will be functions added later that were not originally in the specification. For this reason the first thing we'll discuss in the map format is version control within each map.

Version Control

Throughout the development process you will probably want to add new features to your map format. For this reason we use versioning control within each map, allowing us to continually update the format and add new functionality where needed without having to rewrite the structures of the format for each new version.

The first structure we'll create in our map format is the version structure, conveniently named MAP_VERSION. We'll use the MAP_ naming convention. By using this convention we get into the habit of properly naming variables and structures as to their appropriate tasks. There will be two variables in the structure: version and revision. The version variable is the major version number, and the revision is the current revision of the major version. Although it sounds confusing, it's very simple. Since this is the first version of the map format, it should be version 1.0. To express this in our MAP_VERSION structure, the version would be 1 and the revision number would be 0. If later we decide to make changes to the map format, the version number would still be 1, but the revision number would now be set to 1.

Both variables in the MAP_VERSION structure are of the GLint data type, which is an OpenGL integer. For the map format, we'll use OpenGL data types because this ensures a high degree of compatibility among many operating systems such as Windows, Linux, and others. Although we are not creating a Linux version of the game, it's nice to keep the option open if you ever want to add another platform choice for your customers. Using OpenGL will allow the developer to port the main rendering code to other platforms such as Linux with fewer porting issues than with other APIs. The final version of the MAP_VERSION structure should look like the following:

```
typedef struct
{
   GLint     version;
   GLint     revision;
} MAP_VERSION;
```

The Map Header

The next structure we'll design is the header, which we'll name MAP_HEADER. This structure is one of the most important within our map format because it controls the data sizes within the map. You may be wondering why we don't put the version information within the header. Well, if we decide to completely overhaul the header during development of our game, the version information may become useless because the header structure cannot load properly. If we separate the version structure from the header, we can completely change the format without worrying that the data within each structure may become corrupt or improperly stored.

The first variable in our header controls the maximum number of objects in the map. The appropriately named max_objects is of the GLint data type. All the variables in the MAP_HEADER structure are of the GLint data type unless otherwise written. After defining the maximum objects (max_objects) variable, we define the maximum number of lights (max_lights) in our map. We use lighting to create the "mood" within the game and provide a sense of realism. Without a lighting system, all objects in our game would be drawn using flat coloring, which doesn't look great and doesn't show what our engine is capable of rendering. After the lighting comes the definition of max_cameras, which describes the number of cameras in the map. Although this is not used within our game, it's a great asset when we want to view a specific room in first-person view while we are creating it.

The next set of variables in the MAP_HEADER structure are the nonessential variables: max_entities, max_items, and max_sounds. The max_entities variable controls the maximum number of entities in the map. In the case of our game engine, an entity represents a computer-controlled player or simply a "bad guy." Other games will classify an entity as anything that moves within the game, such as players, enemies, and non-player characters (NPCs). For the sake of simplicity we'll use entities only to describe enemies. After the max_entities variable comes the max_items variable, which controls the number of items in a level. If you're unfamiliar with what items are, I suggest you do some "research" in any recent video game to understand what they are and how they work! All kidding aside, an *item* is an object in 3D space that gives the player some sort of bonus for touching it. Sounds vague, right? In the classic video game Sonic the Hedgehog, the main character would collect rings. Every time Sonic happened to touch a ring, it was automatically collected and his total was incremented accordingly. This is the perfect example of an item. The user touches the item and benefits from it. Obviously when you're designing your own game, it's not written in stone that you must reward the user, but most designers do because it's common practice.

The next variable in the MAP_HEADER structure is the max_sounds variable. Just as the name implies, this variable controls the maximum number of sounds in the map. This will allow us to add "environmental" sound effects to the map, creating the illusion that the world is alive. A good example of environmental sound effects would be a fountain that sprays water particles. By simply adding a sound effect of water splashing, you can create the illusion that the water is actually producing those sounds. We'll discuss sound effects in greater detail later in the book.

After the max_sounds variable, we've got two variables of the GLboolean data type. Both the use_skybox and use_fog variables are used as control flags to enable or disable certain rendering features. In the case of these two variables, the use_skybox variable controls the rendering of a skybox. A *skybox*, in case you're wondering, is a huge cube that is textured to look like a three-dimensional sky. Normally, six unique pictures of the front, back, left, right, top, and bottom are required to give the illusion of the sky. If this variable is set to true, we must read the appropriate skybox structure into the map so we can display the sky textures. If the value in use_skybox is false, then we don't need to concern ourselves with loading the skybox data because it is not present in the map. We'll discuss this topic in further detail later, but for the time being we'll only concern ourselves with whether this value is true or false (that is, whether or not skyboxes are used).

The final variable in the MAP_HEADER structure is use_fog, which is appropriately named after its intended control flag. When this variable is true, the map must load a separate structure from the map that contains the appropriate fog information. This new structure will allow our game engine to initialize and set the default fog values each time we render a frame. If the value in use_fog is false, then obviously we aren't using any special fog technologies and therefore do not need to load the fog structure. Depending on the type of game or level you want to design, fog can play a significant part. We'll discuss fog features and functions later in the book.

The final version of our MAP_HEADER structure is shown in the following code snippet.

```
typedef struct
{
   GLint           max_objects;
   GLint           max_lights;
   GLint           max_cameras;

   GLint           max_entities;
   GLint           max_items;
   GLint           max_sounds;

   GLboolean       use_skybox;
   GLboolean       use_fog;
} MAP_HEADER;
```

Extra Map Features

After defining the header for our map format, we'll define the flagged fog and skybox structures. Since both structures rely on their Boolean flag counterpart, it's a good idea to define them just below the original header. In all honesty, it really doesn't matter where you define them, but it's my personal preference to keep things simple and, since these Boolean variables are technically header extensions, they would logically go after the header itself. To begin we'll define the MAP_SKYBOX structure, which as the name implies is the structure for the skybox data.

As discussed earlier, a skybox is a huge cube that displays six different textures that simulate a three-dimensional view of the world. Obviously if the skybox is a cube, the six sides of it are front, back, left, right, top, and bottom. Within each side there are two variables: filename and texid.

The first variable is an array of characters that contains the texture filename. The size of the array is determined by the constant MAX_STRING_SIZE, which we'll create and use as the default size for all our strings within the map format. The value for our newly created constant MAX_STRING_SIZE is 500. I chose the value 500 because it allows each filename to store the full path with long filenames. In theory we could use the standard 8.3 file naming convention and save several kilobytes of memory, but it makes life difficult when we want more descriptive names.

The second variable in the MAP_SKYBOX_SIDE structure, texid, is of the GLint data type. This variable will contain a computer-generated value when our game engine or map editor loads and binds the filenames as textures. When the structure is read from the file, the value can be ignored because it will be regenerated later in the code. The following code snippet displays the MAP_SKYBOX_SIDE structure definition.

```
typedef struct
{
   char      filename[MAX_STRING_SIZE];
   GLint     texid;
} MAP_SKYBOX_SIDE;
```

After defining the MAP_SKYBOX_SIDE structure, we can define our skybox structure, MAP_SKYBOX. The structure would have been defined earlier, but we didn't have the structure to hold the skybox side. Here, we simply declare six variables of the MAP_SKYBOX_SIDE type and name them appropriately (front, back, left, right, top, and bottom). The following source code displays our completed skybox with all six sides defined.

```
typedef struct
{
   MAP_SKYBOX_SIDE front;
   MAP_SKYBOX_SIDE back;
   MAP_SKYBOX_SIDE left;
   MAP_SKYBOX_SIDE right;
   MAP_SKYBOX_SIDE top;
   MAP_SKYBOX_SIDE bottom;
} MAP_SKYBOX;
```

After defining the MAP_SKYBOX structure we must define the FOG structure, which was the final GLboolean flag in our map header. Fog, just as the name implies, will display fog on the screen. In order to display fog, we must define a new structure called MAP_FOG, which specifically handles all the settings for our fog. Unfortunately, at this time there is no way of adding multiple fog samples so we are stuck using one fog source throughout our entire map. With this in mind, let's begin defining the fog structure.

The first variable in the fog structure is mode, which is of the GLint data type. This variable controls the calculation that the fog performs when rendering the data. Each mode has a slightly different style, allowing you to customize the type of fog you want. The next variable is a GLfloat and is the starting position of the fog when using the linear mode fog calculation. If the fog mode isn't using the linear equation, then this variable can be ignored. If you decide to use the linear equation for the calculation, then the start variable should be set to 0.0, allowing fog to be as close as possible. After defining the starting location for the fog, we must define the end location of the fog. This variable, like the start position, is a GLfloat data type and requires that the mode be using the linear equation for the value to be used. If the linear equation is being used, then the end value is the far distance in the fog coordinate.

After the end variable in the MAP_FOG structure is fog density, which is of the GLfloat data type. This value will store the density, or thickness, of our fog. The default density for our game engine will be 1.0, which is the recommended default value for fog density. Normally the density is incremented in small amounts, two decimal places to be exact, to get the perfect graphical result. The final variable in the fog structure is the fog color in red, green, blue, and alpha (RGBA) values. We'll use an array of four GLfloat variables to store the RGBA values of the color. Normally, color is stored in non-floating data types such as ints, longs, and unsigned characters (bytes); however, OpenGL has a wonderful color system that allows GLfloats to be used as well. Rather than using the usual range of 0 to 255 in a float, the color has a range from 0.0 to 1.0, where 0.0 is the lowest and 1.0 is the highest intensity (or 100%) of that specific pigment. The following source code displays the MAP_FOG structure.

```
typedef struct
{
   GLint          mode;
   GLfloat        start;
   GLfloat        end;
   GLfloat        density;
   GLfloat        rgba[4];
} MAP_FOG;
```

Fog will be covered in greater depth later in the book, but for the moment we'll continue discussing the map format structure. The next structure in the map format contains the starting position information for each user in both single player and multiplayer (deathmatch) levels. We name this next structure MAP_STARTING_POSITION because of the data it contains.

The first variable in the structure is an array of GLdouble data types named xyz. This array, as you can tell, contains the starting X, Y, and Z positions for the user. We use the GLdouble data type because of the precision that GLdouble has in decimal places and because it allows us to have huge (or gi-normus) levels. It may not seem terribly important to have huge levels when just beginning to write OpenGL games, but it's nice to keep the doors open if you want to create some sort of huge three-dimensional world like a massively multiplayer online role-playing game (MMORPG) such as EverQuest or Asheron's Call.

After defining the xyz array in the structure we'll define the starting angles (directions) of the user. Although setting the initial direction of each user doesn't sound important, by allowing the mapmaker to set those values, the beginning of the map can have totally different looks. Imagine installing the newest first-person shooter, then starting the first level only to find yourself staring at a wall. Doesn't sound too exciting for a game that touted its awesome graphics and map design! This type of design issue can be completely avoided by simply setting the initial direction of the user to a specific direction that best suits the beginning of the level. So instead of looking at that ugly interior wall, we'll start the first level looking at some of the marvels the game offers graphically.

The final variable in the structure is another array of the GLubyte data type. The variable, select_rgb, will be a commonly used array throughout our map structure when the structure has the capability of being selected. In the case of the MAP_STARTING_POSITION structure, the user can move the starting positions of single- and multiplayer levels by simply clicking and dragging the starting position with the mouse. To accommodate this, we must have write the source code to select the starting position in three-dimensional space, which isn't easy. The method we'll use is a bit complicated in that we must generate a unique red, green, and blue color for the selectable position, which is stored in select_rgb, then, when the user presses the select button, we take the red, green, and blue values of the pixel underneath the mouse and check against our list of selectable colors. If the value is present in any of the objects, lights, entities, starting positions, etc., then we've selected a specific "thing," which can be manipulated accordingly. This topic will be discussed in further detail later in the book, but for now we've finished the MAP_STARTING_POSITION structure and can display the code of the structure.

```
typedef struct
{
   GLdouble        xyz[3];
   GLfloat         angle[3];

   GLubyte         select_rgb[3];
} MAP_STARTING_POSITION;
```

After the MAP_STARTING_POSITION structure is the MAP_DETAILS structure, which is a mandatory structure for each map. Just as the name implies, this structure contains the details of each map file. Within the structure you'll find various pieces of useful map information such as the map name, map type (single player or deathmatch), starting locations, and more. The first variable in the structure is an array of characters named map_name. The size of the string is set with the constant MAX_STRING_SIZE, which defaults to 500. The map-name variable, as the name implies, stores the name of the map. By storing the map name inside the map format, we can later display the name when we are loading the level. This provides the users with a short description of the level before they begin playing. As a side note to this topic, when naming levels, it's better to use names that describe the setting, theme, or function of the level, i.e., "The Cemetery" or "The Sleeping Village," rather than a convention like Map_1 or Level_1.

The second variable within the MAP_DETAILS structure is map_type, which is of the GLint data type. This variable controls the type of map we are playing, whether it is single player or deathmatch. In the event we wanted to design a match with dual use (both types of gameplay) capabilities, we would simply use a bitwise inclusive OR operation to check for the type of game requested within the map. If the map type is found, then the settings for that style of game could be used rather than the default single-player settings.

After the map_type variable in the MAP_DETAILS structure is the map_exit_rules variable, which is of the GLint data type. Obviously within each map we must have some sort of mechanism to exit the level, whether it be a simple door to open, or by walking onto an elevator or killing all the monsters within a level. This variable fills this requirement by allowing the level designer (you) to set the exit rules for the map. Generally, an exiting rule system is used in single-player levels because it would be an annoyance to continually change levels when one player in a multiplayer game finishes the level in a short period of time. In contrast, deathmatch levels are normally controlled by how many frags (kills) the players have and by an elapsed amount of time.

The next three variables are the starting positions for both single- and multiplayer positions within the map. Naturally, since they are the starting positions, they must be in the MAP_STARTING_POSITION data structure we defined earlier. The first variable, single_player, holds the initial starting position for single-player mode in the game. The other two variables are in an array named deathmatch because they're obviously meant for deathmatch gameplay. In theory, we could have added just one deathmatch starting position and used the single-player starting position for the other deathmatch starting position; however, we could run into map design issues later. To be more specific, deathmatch levels generally have a roundabout flow to their design. This allows the player to continually run around a main circle, collect items, and frag his opponents. If a level was designed intentionally for a single player and deathmatch was added as an afterthought, the deathmatch flow may be nonexistent, which could make the level less fun to play in deathmatch. Similarly, if a map was designed for deathmatch and the designer added a single-player starting position for some kind of simple story, the user may find the level boring. We'll discuss the basics of level design later in the book, but for the moment let's look at our finalized MAP_DETAILS structure source code and then move on to rendering structures.

```
typedef struct
{
   char                   map_name[MAX_STRING_SIZE];
   GLint                  map_type;
   GLint                  map_exit_rules;

   MAP_STARTING_POSITION  single_player;
   MAP_STARTING_POSITION  deathmatch[2];
} MAP_DETAILS;
```

Main Rendering Structures

Now that we've covered the initial structures for the map format, we'll look at the next section of structures, which covers the core of the format. In order to understand why we need each of the following structures, it's a good idea to understand how the basic rendering process works. In Chapter 2, we learned that all objects in OpenGL are drawn using vertex coordinates. In the chapter example we displayed several objects on the screen, including a triangle and a quad. Although it sounds silly, rendering a triangle on the screen is the first step to rendering a huge three-dimensional world. Rather than simply hardcode the values of each vertex, we'll store them in a vertex structure. In any given shape there could literally be hundreds or even thousands of vertices, which is a good reason to always dynamically allocate the required amount of memory and not hardcode a maximum amount allowed.

In Chapter 2 we also discussed the different object types we could create. Normally, games use a combination of triangles, polygons, and quads, depending on the needs of the designers. Although there are many options available, we'll only use triangles because they are easy to build the format with, and they're the fastest type of object other than object strip equivalents. With each shape being created with triangles, we must store the three-vertex indices into a triangle structure. By storing the specific vertex numbers in the triangle structure, we can move the individual vertices without having to recalculate the connecting triangles themselves, since each triangle uses the vertex number to draw the points within the vertex array. Similar to vertices, there should be no limit to the number of triangles any shape can have. As you may have noticed, dynamic allocation of memory in our map format plays a key role in the design. It's a good thing Windows runs in protected mode, otherwise we'd probably break the 640 KB memory limit imposed by the shackles of 16-bit DOS programming with the size of these structures. All too often I've seen simple maps that are 500 KB turn into massive 5 to 10 MB maps by adding simple details such as curved walls, imported 3D objects, and more.

Each object within the map must have both vertices and triangles in order to be drawn. Other pieces of information such as the texture details, visibility, collision flags, and more can also be included within each object, making the structure fairly complex, even when drawing simple things like cubes. Each object in our map format has separate vertex, triangle, and texturing information because each object is different. Due to the threat of impending doom, no objects should share data. Okay, the world may not come to a standstill, but they should be separate because we may want to add special functionality to each item and, if the values are shared, other objects may end up drawing oddly because of previous object settings.

The final core variable in the map structure is an array of objects, which holds the data (vertices, triangles, textures) for all the objects. This array, like the rest of the core map structures, is a dynamically allocated array, which is specified by the variable in the header. To get an idea of what the structure looks like in illustration form, take a look at Figure 3.1.

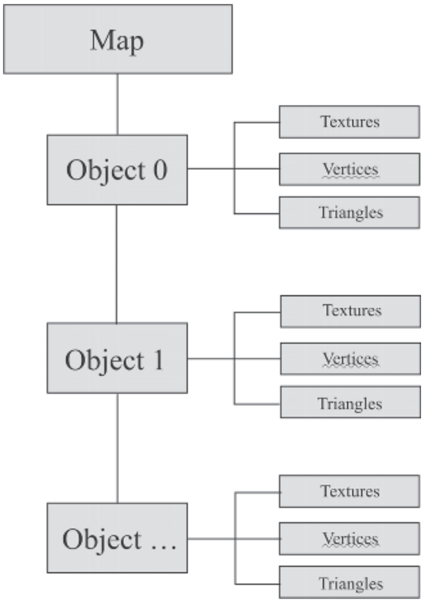


Figure 3.1: Core rendering structure tree

Now that we've covered the basics of how the core structure works, we can write the structures. We'll begin by defining the vertex structure, MAP_ VERTEX. The first variable in the structure is an array of three that is of the GLdouble data type. This variable contains the X, Y, and Z locations and is appropriately named xyz. We store the location data in an array because OpenGL has a vertex function that allows us to specify an array of locations rather than writing the individual X, Y, and Z locations. It's more of a convenience than a design decision.

The next variable in the structure is another array that is of the GLfloat data type. This array controls the red, green, blue, and alpha (transparency) colors of the vertex being drawn and is appropriately named rgba. The color data is important because we can create amazing special effects like semi-transparent water by simply changing the alpha (transparency). We could also change the look of the texture by manipulating the red, green, and blue values, which would change the shading for the object. We'll discuss vertex coloring in further detail later in the book.

After defining the color values, we define a new array of three that is of the GLfloat data type. The array, named normal, contains normal data for dynamic lighting. You may be asking yourself just what the heck are normals. Normals control the direction of light that is bounced off the object.

These values are only required when dynamic lighting is required for drawing, and should be disregarded if there is no lighting required or if the lighting system is based on lightmapping.

The final variable in the MAP_VERTEX structure that has useful information for drawing objects in our map is fog_depth. This variable contains a decimal value of the GLfloat data type that specifies the depth of the fog at that specific vertex location. Fog depth is normally used when creating special effects such as volumetric fog. In case you're wondering, volumetric fog allows you to specify a specific location to where the fog will be thickest. As you move away from that location, the fog will become lighter. In the video game Quake III Arena by id Software, a form of volumetric fog was used on the ground to produce amazing effects seen throughout the game. That is one example of the effects possible with fog depth. Because volumetric fog is not built into the core API of OpenGL, only video cards that have the GL_EXT_FOG_COORD OpenGL extension will be able to display this functionality. If the user doesn't have the extension, he or she won't be able to experience the game at its fullest.

The final variable in the MAP_VERTEX structure is an array of the GLubyte data type named select_rgb. This variable, as indicated in the discussion of the MAP_STARTING_POSITION structure, is the unique selectable color of the vertex. When we read the vertices from the structure, the select_rgb values aren't important because we'll generate new values once it's finished loading. The selectable colors within the vertices are different from those of the starting positions and regular objects because they are encapsulated within the main object itself. The selectable RGB values can be copied as long as they are unique within the vertex list. This differs from global unique colors like those of objects, starting positions, items, etc., because their values cannot be copied at all. The difference between them is simple. If we want to select and move the vertices within the object, we should be allowed to do so. Obviously, the points of each object won't be visible, so therefore copies are allowed. In the case of the global selectable RGB values, the user must always be allowed to select them and move them accordingly. The following source code displays the MAP_VERTEX structure.

```
typedef struct
{
   GLdouble        xyz[3];
   GLfloat         rgba[4];
   GLfloat         normal[3];
   GLfloat         fog_depth;

   GLubyte         select_rgb[3];
} MAP_VERTEX;
```

Before defining the triangle structure, we must first define the UV coordinate structure, MAP_UV_COORDS, which stores the texture wrapping information for the triangle we are going to draw. This structure is quite simple because it only has three variables in it, all of which are of the GLfloat data type. The first variable, named uv1, is an array of two that contains the UV coordinates for the first point in our triangle. In case you are wondering, UV coordinates specify how an object is textured. The U coordinate normally refers to the horizontal direction, and the V refers to the vertical direction. In the variable uv1, the first index of the array is the U, and the second index would be the V.

The second and third variables in the MAP_UV_COORDS structure are the same as the uv1 variable except the number is changed to 2 and 3 respectively. By supplying all three points, we have all the information we need to texture the triangle we've just rendered. We've now finished defining the MAP_UV_COORDS structure and can display the final source code of the structure.

```
typedef struct
{
   GLfloat          uv1[2];
   GLfloat          uv2[2];
   GLfloat          uv3[2];
} MAP_UV_COORDS;
```

Now that we've created the MAP_UV_COORDS structure, we can create the triangle structure, MAP_TRIANGLE. The first variable in the MAP_TRIANGLE structure is an array of three that is of the GLint data type. Appropriately named point, this variable contains the indexes of all three vertex points required to draw the triangle. In addition to storing the vertex point data, the MAP_TRIANGLE structure stores the UV coordinate information for multiple texturing layers. By using multiple texture layers, if the hardware supports the feature, the programmer can render the data once and have the computer draw both textures at once. This process is called single-pass rendering and requires support for GL_ARB_MULTITEXTURE, which is part of the OpenGL 1.3 standard. Because each vendor's video card is different, the number of layers the card can have differs between vendors. In some cases, the maximum number of layers can be anywhere from two all the way up to sixteen, depending on the card. For this reason, we'll create a constant called MAX_TEXTURE_LAYERS, which will store the default number of layers available in the engine. For compatibility reasons, we'll set the default value to 2, giving us two different texture layers available. We've completed the structure definition for the MAP_TRIANGLE structure, so we can display the final structure below.

```
typedef struct
{
   GLint            point[3];
```

```
  MAP_UV_COORDS    uv[MAX_TEXTURE_LAYERS];
} MAP_TRIANGLE;
```

The next structure in the map format controls the texture layers for each object. Appropriately named, MAP_TEXTURE specifies key data required for each texture, including filename, texture number, blending values, and type of texture being rendered. All this data is important because it provides us with a great deal of power when rendering an object with multiple textures. Even with a single texture layer being used, the functions provided allow you to customize exactly how you want the texture to look when blended with other objects on the screen. With this in mind, let's begin defining this new structure.

The first variable in the MAP_TEXTURE structure is a variable called filename. Stored as an array of characters, with the maximum size defined in MAX_STRING_SIZE, this variable is only required when referencing textures from their original filename to their respective texture number. In case you're wondering, we use texture numbers to easily reference texture data from the video card. Texture IDs usually begin at 1 and continue until all textures in the game are loaded. The table below displays a list of textures being loaded and the respective texture assigned to it.

Filename	Texture ID
Brick.bmp	1
Ceiling.bmp	2
Floor.bmp	3
Etc....	

The second variable in the MAP_TEXTURE structure is id, which is of the GLint type. This variable will store the numeric texture ID of the texture stored in filename. When reading the MAP_TEXTURE structure from a file, this variable can be ignored because it will be recalculated each time the map is loaded. By default, this variable will have a value of 0, which in OpenGL means there is no texture assigned to the object.

The next variable in the MAP_TEXTURE structure is style, which is of the GLint type. The style variable controls the style of texture we want to display. This is a customizable variable in that it allows us to program special texturing effects for the specific texture layer. There are four texturing styles, including the not-so-special disabled and default, in addition to scrolling and rotating textures. Each texturing style has its own benefits, which we'll discuss later in the book.

After defining the style variable we have a pair of variables that control the blending functionality: blend_src and blend_dst. Both variables are of the GLint type and are used in conjunction with the other. The purpose of blending is to take the original texture and apply specific rules to be added, changing the final output. Pretty vague, huh? In plain English, we use blending to control the final output of a rendered object's texture. There are numerous calculations that can be performed to manipulate the colors in all sorts of ways. The blend_src variable controls the source color of the texture, and the blend_dst variable controls the texture destination color. Both variables are used in the calculations, which is why I've coupled them together in this discussion. We'll discuss blending and the calculations behind blending in further depth later in the book, but for the time being we'll display the MAP_TEXTURE structure source and move on to the next structure.

```
typedef struct
{
   char            filename[MAX_STRING_SIZE];
   GLint           id;
   GLint           style;
   GLint           blend_src;
   GLint           blend_dst;
} MAP_TEXTURE;
```

After the MAP_TEXTURE structure comes the big Kahuna of structures, MAP_OBJECT. This is the structure that pulls our MAP_TEXTURE, MAP_TRIANGLE, and MAP_VERTEX structures together to create an object. There are also several customization variables in the structure that can change the way the object is handled in the game. The first variable in the structure is an array of characters called name. Like all other strings in the map format, this variable uses this MAX_STRING_SIZE constant as the maximum size. As the name implies, the MAP_OBJECT variable contains the name of the given object. For organization, we can name each object in a given room to easily distinguish between them.

The next variable in the structure is type, and it's of the GLint data type. This variable controls what type of object we've created. Although each object is drawn using exactly the same vertex/triangle combination, you can create premade surfaces and give them special types. Also, when creating our maps, the design process for the floors/ceilings is different from that of the walls. Although the creation differences are small, it's much easier to keep the object type stored in a variable because it may be useful later on.

Prior to finishing the format, I realized we would most likely want to extend the objects past the basic wall/floor/ceiling purpose. The next variable addresses this by allowing special values to be set on each object. Appropriately named special, this variable can control whether an object is lava, which kills the player upon touch, an exit point from the level, a one-way teleporter, or just a regular object. This variable is of the GLint data type.

Following the special variable there are two flag variables of the GLboolean data type. The first flag is called is_collidable, which stores whether or not the object in question is collidable. By default this value is true, which means a user walking into a wall will stop because it's been hit. If the variable is false, the user can walk through the wall as if it doesn't exist. The idea behind this may sound silly at first, but it can be rather useful when you want to add secret areas to a map. If the user walks through the correct wall, a secret area is accessed; otherwise the user cannot continue to move.

The second GLboolean variable is named is_visible because it controls the visibility of the object. By default, the value for this variable is true, which means the object will be drawn. If the value is set to false, then the entire object will not be drawn. This function is useful when designing mazes, puzzles, or linear games like racing games that only allow the user to move so much in each direction. Many racing games employ the invisible wall trick to limit where the player can drive. By adding invisible walls to the sides of the course, the player cannot get off course.

After the two GLboolean variables come three control variables of the GLint data type. These variables control the maximum number of textures, triangles, and vertices. These variables are a very important part of the MAP_OBJECT structure because they control the sizes of the arrays of vertices, triangles, and textures. Without these variables, we wouldn't know what size our arrays are and would surely crash our software. The first variable controls the maximum number of textures used to draw the object, hence the variable's name, max_textures. Although we can add more than one texture to the object, the maximum number of textures we can add is defined in the MAX_TEXTURE_LAYERS constant, which has a value of 2.

The next variable, max_triangles, controls the maximum number of triangles in the object. There is no hardcoded limit to the number of triangles we can have in any given object; however, the max_triangles variable cannot be less than 0 and cannot exceed the highest value of GLint, which in Windows is roughly two billion. At the time of this writing, no video card can handle two billion triangles per second at over 30 frames per second. I doubt there will be any video card released to the mass market within the next three years that will be able to handle such a large number either. Two billion triangles is a heck of a lot of triangles, so we need not concern ourselves with this limit. To store two billion GLint data types we would need to have well over 8 gigabytes of memory available, which at this time is pretty rare for desktop computers.

After defining the max_triangles variable we must declare the max_vertices variable. This variable, just like max_triangles, is of the GLint data type. Once again, this allows us to store up to two billion different vertices, but most likely will never happen because with the amount of data store in each vertex, we would steal too much RAM from the system.

Following the control variables are the arrays of data themselves. The first array in the structure is of the MAP_TEXTURE structure and is named texture. Obviously, this array stores the values of the object's textures. Although not required, it's a good idea to have at least one texture in each object so the speed of the levels don't dramatically change when textures are assigned after the initial design is finished. We'll discuss the process of map design later in the book, but for the time being, we'll discuss the next array in the structure, which is of the MAP_TRIANGLE type.

Although it seems silly to discuss, the triangle variable is of the MAP_TRIANGLE structure type. This array, as you probably figured out, contains all the triangle data for the entire object. When rendering the object, the process will immediately end if there are no triangles in the object. This is because each object is drawn based on triangles, which aren't present in the object. If there are triangles in the object, then the rendering process can continue.

The final array of data in the MAP_OBJECT structure is vertex, which is of the MAP_VERTEX type. If there are triangles in the object, there must be at least one vertex in the object. This does not ensure a crash-free program; however, if there is one triangle, we know there must be at least one vertex in the object. If for some reason there are triangles in the object but no vertices, then we should exit from the rendering procedure immediately, because chances are a crash is inevitable.

The last variable in the structure is an array of three that is of the GLubyte data type. This variable is the selectable red, green, blue array, which is used to create the selected object. The source code for the structure is shown below.

```
typedef struct
{
   char            name[MAX_STRING_SIZE];
   GLint           type;
   GLint           special;

   GLboolean       is_collidable;
   GLboolean       is_visible;

   GLint           max_textures;
   GLint           max_triangles;
   GLint           max_vertices;

   MAP_TEXTURE     *texture;
   MAP_TRIANGLE    *triangle;
   MAP_VERTEX      *vertex;

   GLubyte         select_rgb[3];
} MAP_OBJECT;
```

The next structure in the map format is MAP_CAMERA. As the name implies, this structure contains the data needed for a camera. The map editor will take advantage of the cameras, allowing the level designer to view the map through the eyes of the player. This allows the designer to customize how the level looks without having to run the game.

The first variable in the MAP_CAMERA structure is an array of characters with the constant MAX_STRING_SIZE as the maximum. This variable holds the name of the camera, hence the variable is named name. Technically, we could have skipped the customized camera names and placed the cameras in a sequential name system like “Camera 1,” “Camera 2,” etc. Unfortunately, using a sequential naming system doesn’t describe the camera very well. By allowing the cameras to be named, you can describe the room/area the camera is looking at, thereby allowing the camera to be easily selected.

The next variable in the structure is an array of three GLdoubles named xyz. This array stores the X, Y, and Z locations of the camera. When placing a camera in three-dimensional space, we want the position to be as precise as possible to ensure the camera location isn’t changed when the data is saved and loaded from the file. Throughout our map format we’ll be representing X, Y, and Z locations in arrays of GLdouble data type.

After we define the xyz variable, we’ve got another array of three. Unlike the previous array, the data type for this array is a GLfloat. The function of this array is to hold the angle data for the x-, y-, and z-axes. This data is essential because it controls the direction of the camera. Without the specific angles, we would be limited to one-way cameras, which wouldn’t be very useful. With the data stored in this array, we have the freedom to move the camera to achieve the perfect viewing angle. The GLfloat type was chosen because GLubyte only holds around three-quarters of the data we need, so the next logical choices were GLint or GLfloat. The rotation calculations will require floating-point values, so it’s best to use the GLfloat data type.

The final variable is that pesky selectable red, green, blue GLubyte array. Since we’ve covered this variable several times and the format hasn’t changed, we can skip the discussion of this variable. The source code for the MAP_CAMERA structure has been provided for you below.

```
typedef struct
{
   char           name[MAX_STRING_SIZE];
   GLdouble       xyz[3];
   GLfloat        angle[3];

   GLubyte        select_rgb[3];
} MAP_CAMERA;
```

Thankfully we're almost finished discussing the structures for the map format. Once it's completed, we'll have all the knowledge we'll need to begin writing the underlying object and map source code.

Without further ado, let's begin discussing the next structure, MAP_LIGHT. The MAP_LIGHT structure contains the data required to draw a light in our game engine. Lighting is a very important part of our maps because it can mean the different between creating a solid color, vibrant-looking dungeon or a dark and mysterious dungeon. The lighting and accompanying shading can set the mood of almost any game. In some cases, this can make or break the overall look of the game. With all this in mind, let's look at the MAP_LIGHT structure.

The first variable in the MAP_LIGHT structure is an array of characters that defines the name of the light. Like all the other strings in our map format, the maximum size of the variable is set by the constant MAX_STRING_SIZE. We set the name of the light in a similar fashion to cameras, allowing us to maintain some order rather than fall into a chaotic mess, which some levels can become. Whenever possible, it's always a good idea to write descriptive names for each light so you can easily point out each light in the map. The MAX_STRING_SIZE can be up to 500, which should be more than enough characters to work with when describing names.

The next variable in the structure contains the type of light value. Being of the GLint data type, the value can only be in integer form. By setting the type of light, we can differentiate between the different styles of light, giving us the freedom to add more styles as development continues. Also, each type of light may require special features such as animation, random color changing, variable blinking, or other crazy effects that are possible by specifying a new light type. This provides almost endless possibilities for lights.

Following the type variable we have an array of GLdouble with a size of 3. The variable, called xyz, stores the X, Y, and Z locations of the light. This information is very important because a light cannot be set without some sort of initial location. Another important value is the angle of the light, which is represented in the next array of three GLfloat called angle. The angle of the light is important because it points toward the direction it will shine. This allows the user to place a light down, then angle it to his or her desired location to achieve the originally intended look. If we didn't allow the user to alter the angle of the light, then we would only have lights that shine in one direction. Obviously, having lights shine in a single direction isn't very useful, hence the reason we store the angles of the light.

After the angle array in the MAP_LIGHT structure comes a new array of four GLfloats called rgba. Just as the name implies, this variable stores the red, green, blue, and alpha values for the light. This information is important because it allows us to style the light colors beyond the typical white used in most games. By simply changing the light color we can have any color in the rainbow! This allows us to dramatically change the environment from having a simple one-color lighting system, which doesn't show any depth, to a realistic lighting system. The difference between using colored light and multicolored lights is subtle but important. Imagine you are in a subway. The fluorescent lights in a subway tunnel are normally never the same color, but shades of several colors. If we created a small subway station and used one light of a single color, it may look great, but by using several different colors, we can achieve a much more realistic look.

To do this, simply exchange every second or third light with one of a bluish or greenish color instead of the original white. The topic of colored lights will be covered in depth later in the book, so for the time being, we must move to the next topic.

The next variable in the MAP_LIGHT structure is an array of characters called texture_filename. Like all our strings, this variable has a maximum size defined by the MAX_STRING_SIZE constant. This variable contains the filename of the light we are trying to render. Although we could generate the texture ourselves, by setting the texture filename we're giving a lot more freedom to the designer of the level. This allows the designer to use non-standard shaped textures as lights. Since not all lights are created as circles, we must address this capability in our game engine. By selecting a different filename for the texture, we can customize each light to fit the appropriate needs of the light. Of course when specifying a texture filename, a texture number must be associated with it, which brings us to the next variable, texture. The texture variable contains the texture_filename variable's associated texture number, which is required for rendering the light. The texture number is generated each time the game is started, which means any values loaded from the file can be ignored. The texture variable is of the GLint data type, keeping with the texture number data type consistency.

The next variable in the structure is of the GLint data type. This variable, named max_inclusions, controls the list of includable objects. When the value is set to 0, all objects that pass through our light, regardless of walls that are in front of the object, will have the light rendered on top of them. By setting this value to a non-zero number and selecting which walls are affected, we are essentially creating a list of objects that are affected by the light. This is tremendously useful when the programmer wants to shine a light on one wall and not have it visible in the room right next to it. The next variable is an array of GLint type called inclusions, which is the list of objects affected by the light. This variable is a dynamic array and is allocated using the value from max_inclusions. Each index in the array specifies the object number, thereby allowing us to draw the object with the specified light. When the variable max_inclusions is 0, which is the default, the value of inclusions is set to NULL.

The final variable in the MAP_LIGHT structure is the selectable red, green, blue array called select_rgb. Obviously we'll need to be able to select the light to adjust its settings, hence the need to have a selectable RGB value.

We've finally finished defining the MAP_LIGHT structure! The source code for this newly defined structure is provided below.

```
typedef struct
{
   char             name[MAX_STRING_SIZE];
   GLint            type;
   GLdouble         xyz[3];
   GLfloat          angle[3];
   GLfloat          rgba[4];

   char             texture_filename[MAX_STRING_SIZE];
   GLint            texture;

   GLint            max_inclusions;
   GLint            *inclusions;

   GLubyte          select_rgb[3];
} MAP_LIGHT;
```

Miscellaneous Map Structures

The next structure in the map format is MAP_ENTITY, which describes the basic starting information for entities in our game. *Entities* are non-player characters (NPCs) and enemies. A game isn't complete without enemies to fight or NPCs walking around doing mindless jobs! With that in mind, let's begin defining the MAP_ENTITY structure.

The first variable in the new structure is of the GLint data type. This variable, type, describes the type of entity we're creating. This would be the place where we could define whether the entity is an NPC or an enemy. Later, if you find the ambition, you may want to add other entity types such as passive and aggressive NPCs. This would allow certain NPCs to be more aggressive when doing their job, starting fights with you if you annoy them, and other NPCs to be, passive, humbly walking around you and not saying a thing.

The next variable in the structure is an array of three GLdoubles called xyz. This variable contains the X, Y, and Z starting positions for the entity. Without the starting positions, the entity wouldn't have a place to start from and it would throw off the entire level in terms of entities in the game. Obviously, when we define the starting position of the entity we'll also have to define the starting direction, which brings us to the next variable in the structure, angle. The angle array is an array of three GLfloats that stores starting angles for the specific entity. In most games, the x-axis (angle[0]) is the only index that has a non-zero number. Although the other two variables are very important, they aren't used much when defining the starting angle of the user in first-person 3D shooters.

Following the angle variable are three variables of the GLint data type. These variables, health, strength, and armour, are used to customize each entity. The health variable controls the health of each entity. The number ranges from 0 (the entity is dead) to the maximum size of the GLint data type. When assigning the health of each entity, it's a good idea to think of the entity in terms of percentage against the player's health. Assuming the player health is 100, a monster that is twice the size of the user should realistically have twice the health (200). If the enemy happens to be half the size of the hero, then obviously it should have a health that is 50% of the player's. Of course this does not bring armour or strength into consideration, which could change the entire balance. Even if the enemy is half the size of the user, if its strength is twice as much as the player's, it could be impossible to defeat. The strength variable works the same as the health variable in that it's based on the default user value. The armour variable works the same way, except in our game it will absorb percentages of the wounds inflicted from the enemy. That is, if the enemy shoots a rocket at a player with the best armor, then the armor will absorb about 90% of the hit, assuming it doesn't break apart.

One final note on the topic of the variables, though. If we were to hard-code these variables instead of having them user adjustable, we would ultimately hurt the designer because he or she may not be able to create levels that meet the specific requirements. Although it doesn't seem like a huge issue, the little details can make a difference between a good game and a bad one. These details are geared more toward game design than programming of the structure, so we'll discuss those variables later in the book.

The final variable in the MAP_ENTITY structure is the array of three GLubytes for selectable RGB.

We've finished defining the MAP_ENTITY structure and can display the source code now.

```
typedef struct
{
   GLint          type;
   GLdouble       xyz[3];
   GLfloat        angle[3];

   GLint          health;
   GLint          strength;
   GLint          armour;

   GLubyte        select_rgb[3];
} MAP_ENTITY;
```

After defining the MAP_ENTITY structure we must define one of the final structures in our map format — MAP_ITEM. This structure contains the information required to place items (weapons, health, armor, etc.) on the ground. The first variable in the structure is of the GLint data type and is named type. This variable defines the type of item, whether it is a weapon, health, armor, etc. Depending on the depth of the game you are creating, the number of items to pick up and use can vary greatly. Since our game will be fairly simple, we'll probably only have a handful of items you can use.

The next variable in the structure is of the GLint data type. This variable, called respawn_wait, controls the respawning of items. When an item is picked up, it is not available to be picked up anymore. By allowing the item to be respawned, the items are put back into the level after a period of time that is defined by the user. When the variable respawn_wait is greater than 0, the item will be respawned using the respawn_wait variable as the number of milliseconds before respawning.

The next variable in the structure, respawn_time, is the final respawn time that, when passed, allows the item to be picked up again. This variable is also of the GLint data type. To put this in perspective, if the level designer sets respawn_wait to 10,000, the user would have to wait 10 seconds before the item would respawn in that place. If the user picks up the item, the current time is stored in respawn_time with the value of respawn_wait added to it. When the current time exceeds the respawn_time, the item will once again be visible and available to be picked up.

The next variable in the structure is an array of three GLdoubles that specifies the starting X, Y, and Z locations of the item. Obviously, without defining the starting positions there would be no way to know if we've picked an item up.

The final variable in the MAP_ITEM structure is the typical select_rgb array of three GLubytes. This variable is a must because we need the capability to move items throughout our map easily.

We've completed defining the MAP_ITEM structure and can display the source code below.

```
typedef struct
{
   GLint          type;
   GLint          respawn_wait;
   GLint          respawn_time;
   GLdouble       xyz[3];

   GLubyte        select_rgb[3];
} MAP_ITEM;
```

Thankfully, we only have one more structure to define in the map format. This last structure is MAP_SOUND, which stores the data required to play environmental sounds in a level. Environmental sounds are used throughout the level to create a sense of reality. Some basic examples of environmental sounds could be things like water drips, which would play near a leaky faucet, or the sound of a refrigerator if we were talking near a kitchen. This small feature can make a game much better because it adds such a sense of realism. Let's begin defining this new structure.

The first variable in the MAP_SOUND structure is an array of characters with a maximum size set by the constant MAX_STRING_SIZE. The variable, filename, stores the filename for the sound. This information is obviously important because we won't know which sounds to load for them to play properly. The next variable in the structure is of the GLint data type and is named id. This variable stores the associated sound ID of the filename. When the level is first loaded, the sounds are loaded and bound to numbers in a similar fashion to textures. Unlike textures, sounds cannot be reused and therefore must be loaded for each sound structure we put into our game. We use the id variable as a simple mechanism to distinguish between all the different sounds once we've loaded the level.

After the id variable, we've got an array of three GLdoubles named xyz. This variable stores the X, Y, and Z positions of the sound. This information is important because we can plot our sounds in three-dimensional space, in a similar fashion to the way we draw our graphics. Following the xyz variable is an array of GLfloats that stores the angle information of the sound. Setting the angles of the sound is important because sound is directional. This allows us to create the effect of playing each sound as if it is projected behind the player. Another example of using sound angles would be having a player in a room that has a surround sound system able to hear a particular speaker's sound more clearly when he nears that speaker.

The final variable in the structure is our selectable RGB value. Like many other objects within our map format, we must be able to select the sound source to be able to move and delete it. We've now completed defining the map structure, and the source code is shown below.

```
typedef struct
{
   char          filename[MAX_STRING_SIZE];
   GLint         id;
   GLdouble      xyz[3];
   GLfloat       angle[3];

   GLubyte       select_rgb[3];
} MAP_SOUND;
```

Creating the MAP Class

At the bottom of the file map.h file, which contains our structures, we'll create a new class called MAP. In the public section of the class we'll need to add all the structures we created in the previous sections. To begin, we'll add a variable called version, which is of the MAP_VERSION type. As mentioned before, this is the structure that contains the map versioning information. The next variable in the class is called header, which is of the MAP_HEADER type. This is the header for the map format and controls the sizes of each array. After the header, we must add the variable skybox, of the MAP_SKYBOX data type, and fog, which is of the MAP_FOG data type. Both variables are controlled by their respective Boolean flags in the header.

After the fog variable, we must declare the details variable, which is of the MAP_DETAILS type. This variable contains the special details of the map, such as the map name and map type. The next variable in the MAP class is a pointer to the MAP_OBJECT type and is called object. Once the header is loaded, this variable will be dynamically allocated memory for all the objects in the map. After declaring the object variable, we must declare a new pointer of the MAP_ENTITY data type called entity. This array will store the values of each entity specified in our map format. This variable will be dynamically allocated memory once the header is loaded.

After declaring the light variable, we have two variables left to declare. The first variable is a pointer to the MAP_SOUND structure and is called sound. The final variable we must declare in the class is another pointer to a MAP_ITEM structure named item. Like all the other pointers in the MAP class, both variables will be dynamically allocated when the header is loaded from the file. In the class definition we'll add both the constructor and destructor to the definition, completing the basic class definition. The class definition for the MAP class has now been completed, and the source code for it is shown below.

```
class MAP {

   public:

      MAP_VERSION         version;
      MAP_HEADER          header;
      MAP_SKYBOX          skybox;
      MAP_FOG             fog;
      MAP_DETAILS         details;

      MAP_OBJECT          *object;
      MAP_ENTITY          *entity;
      MAP_LIGHT           *light;
      MAP_SOUND           *sound;
      MAP_ITEM            *item;
```

```
    MAP();
    ~MAP();
};
```

Chapter Example

Now that we've created the class definition, we can write the code for the default constructor. The default constructor will set all the public pointers to NULL and set the header to the default value of 0. When the class is created, the defaults will be applied to every variable and structure, helping to eliminate any pointer problems we may experience. There is no guarantee that we won't have any pointer problems, but this will minimize them.

The first two lines of the constructor must always specify the version information. By setting the version information in the constructor, we will ensure the map is set to the proper version right from the start. The first line in the constructor should set the version variable within the version structure to the constant VERSION. The second line of the constructor should set the revision variable within the version structure to the constant REVISION.

After setting the version information, we must set the rest of the structures to the default value of 0. To accomplish this, we'll use the memset function, specifying the address of the respective variable (header, details, skybox, and fog), the character we'd like to fill the structure/array with, which in our case is 0, and the size of the structure using the sizeof function and supplying the variable we're trying to fill. We'll use this function to set the header, details, skybox, and fog structures to 0.

Then we must set our pointers to NULL to finish the initial constructor code. The pointer variables that must be set to NULL are object, entity, light, sound, and item. By setting these to NULL, we are signaling that there is no data in the arrays. After setting the variables to NULL, we have finished writing the constructor. At the moment, there is no source code required in the destructor because we have nothing to check for just yet. In Chapter 4, we'll write the array destruction code for the destructor. But for the moment, let's display the source code for the example in this chapter.

ex3_1.cpp

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <windows.h>

#define MAX_TEXTURE_LAYERS      2
#define MAX_STRING_SIZE         500

#define VERSION                 1
#define REVISION                0
```

```
typedef struct
{
   GLint           version;
   GLint           revision;
} MAP_VERSION;


typedef struct
{
   GLint           max_objects;
   GLint           max_lights;
   GLint           max_lightmaps;
   GLint           max_cameras;

   GLint           max_entities;
   GLint           max_items;
   GLint           max_sounds;

   GLboolean       use_skybox;
   GLboolean       use_fog;
} MAP_HEADER;


typedef struct
{
   char            filename[MAX_STRING_SIZE];
   GLint           texid;
} MAP_SKYBOX_SIDE;


typedef struct
{
   MAP_SKYBOX_SIDE front;
   MAP_SKYBOX_SIDE back;
   MAP_SKYBOX_SIDE left;
   MAP_SKYBOX_SIDE right;
   MAP_SKYBOX_SIDE top;
   MAP_SKYBOX_SIDE bottom;
} MAP_SKYBOX;


typedef struct
{
   GLint           mode;
   GLfloat         start;
   GLfloat         end;
   GLfloat         density;
   GLfloat         rgba[4];
} MAP_FOG;


typedef struct
{
   GLdouble        xyz[3];
   GLfloat         angle[3];
   GLint           model;

   GLubyte         select_rgb[3];
} MAP_STARTING_POSITION;


typedef struct
{
   char            map_name[MAX_STRING_SIZE];
   GLint           map_type;
   GLint           map_exit_rules;

   MAP_STARTING_POSITION  single_player;
   MAP_STARTING_POSITION  deathmatch[2];
} MAP_DETAILS;


typedef struct
{
   GLdouble        xyz[3];
   GLfloat         rgba[4];
   GLfloat         normal[3];
   GLfloat         fog_depth;

   GLubyte         select_rgb[3];
} MAP_VERTEX;


typedef struct
{
   GLfloat         uv1[2];
   GLfloat         uv2[2];
   GLfloat         uv3[2];
} MAP_UV_COORDS;


typedef struct
{
   GLint           point[3];
   MAP_UV_COORDS   uv[MAX_TEXTURE_LAYERS];
} MAP_TRIANGLE;


typedef struct
{
   char            filename[MAX_STRING_SIZE];
   GLint           id;
   GLint           style;
   GLint           blend_src;
```

```
    GLint            blend_dst;
} MAP_TEXTURE;


typedef struct
{
    char             name[MAX_STRING_SIZE];
    GLint            type;
    GLint            special;

    GLboolean        is_collidable;
    GLboolean        is_visible;

    GLint            max_textures;
    GLint            max_triangles;
    GLint            max_vertices;

    MAP_TEXTURE      *texture;
    MAP_TRIANGLE     *triangle;
    MAP_VERTEX       *vertex;


    GLubyte          select_rgb[3];
} MAP_OBJECT;


typedef struct
{
    char             name[MAX_STRING_SIZE];
    GLdouble         xyz[3];
    GLfloat          angle[3];

    GLubyte          select_rgb[3];
} MAP_CAMERA;


typedef struct
{
    char             name[MAX_STRING_SIZE];
    GLint            type;
    GLdouble         xyz[3];
    GLfloat          angle[3];
    GLfloat          rgba[4];

    char             texture_filename[MAX_STRING_SIZE];
    GLint            texture;

    GLint            max_inclusions;
    GLint            *inclusions;
```

```
   GLubyte          select_rgb[3];
} MAP_LIGHT;


typedef struct
{
   GLint            type;
   GLdouble         xyz[3];
   GLfloat          angle[3];

   GLint            health;
   GLint            strength;
   GLint            armour;

   GLubyte          select_rgb[3];
} MAP_ENTITY;


typedef struct
{
   GLint            type;
   GLint            respawn_wait;
   GLint            respawn_time;
   GLdouble         xyz[3];

   GLubyte          select_rgb[3];
} MAP_ITEM;


typedef struct
{
   char             filename[MAX_STRING_SIZE];
   GLint            id;
   GLdouble         xyz[3];
   GLfloat          angle[3];

   GLubyte          select_rgb[3];
} MAP_SOUND;


class MAP {

   public:

      MAP_VERSION   version;
      MAP_HEADER    header;
      MAP_SKYBOX    skybox;
      MAP_FOG       fog;
      MAP_DETAILS   details;
      MAP_OBJECT    *object;
      MAP_ENTITY    *entity;
      MAP_LIGHT     *light;
      MAP_SOUND     *sound;
      MAP_ITEM      *item;

   MAP();
   ~MAP();
};


MAP::MAP()
{
   version.version  = VERSION;
   version.revision = REVISION;

   memset (&header, 0, sizeof(header));
   memset (&details, 0, sizeof(details));
   memset (&skybox, 0, sizeof(skybox));
   memset (&fog, 0, sizeof(fog));

   object           = NULL;
   entity           = NULL;
   light            = NULL;
   sound            = NULL;
   item             = NULL;
}


MAP::~MAP()
{
}
```

Conclusion

In this chapter we discussed the entire map format for our game. Although the example wasn't terribly exciting, we've gained all the knowledge we need to create the underlying object creation code without worrying about which features or structures may be required later in development. This chapter was a major milestone because it sets the tone for the rest of the book. We literally just have to write the interfacing source code for the map editor and then we're almost finished with its creation.

In Chapter 4, we'll learn how to create objects and an interface for our map editor. We'll also tie them together to create a drag-and-drop interface, allowing the user to create objects with simple points and clicks and making map design a simple process. This is where the real fun begins, because we can finally put all the information we learned in Chapters 1 to 3 to good use.

Chapter 4

Creating the Low-Level Map Code

In this chapter we'll write the underlying code to insert objects, vertices, and triangles into our map. Of course when inserting objects and vertices into any map, we must generate unique colors for each object to ensure that we can easily select each object later. This chapter is a major milestone in the development of our game/editor because once we finish this chapter, the editor will give us the capability of creating primitives using a click-and-drag interface similar to other map editors, which we'll do in the next chapter. Because this chapter deals with writing the low-level code for maps, there won't be a major eye-pleasing example for this chapter. I assure you, however, that the next chapter will have a nifty demo! Without further ado, let's begin!

Generating Unique Colors

With the creation of any new light, camera, object, etc., we must generate a new unique corresponding color to identify the item in question. This will allow us to easily identify the object we've selected because no two items will have the same unique color. In this chapter, we'll write the function to generate unique colors for objects, cameras, lights, sounds, starting positions, item positions, entities, and more! This process is necessary if we want to add the capability of selecting objects later in our map editor.

To generate the unique color, we'll create a new function in our MAP class called GenerateColor, which will have a return type of long. This function will continually randomize colors until it creates a unique set of red, green, and blue values and then it will return the final RGB value in long form, which will be a 24-bit representation of the number.

Because we generate the colors from individual RGB values, there are up to 2^{24} (16,777,216) possible colors. This gives us great flexibility within our software to easily generate a color without immediately worrying about whether it has already been used. This also provides us with an almost infinite number of selection options that we can create in each level. The only limiting factor at the moment is the amount of physical RAM the user must have for the software to run properly. At the time of publication, the average personal computer was shipping with 512 MB of RAM, which should hold thousands of objects without a problem; however, it won't hold 16 million. This won't be an issue for us since our levels will be fairly small, but it's a good idea to consider your game's average user base when designing levels. Your users may or may not have the required RAM to run the software adequately.

The GenerateColor function requires three new variables of the GLubyte data type to be declared. The new variables, r, g, and b, will contain the randomized red, green, and blue values needed to create our final color. After defining the new variables, we must set each of the three variables to a random number using the rand function, with 255 being the maximum value allowed as specified using the modulus operator. Since there are 16.7 million different colors, the shades are created by bit-shifting the red, green, and blue color components together to create the final color. When we add the bits of all three variables together, we get 24, and 2^{24} is 16.7 million.

After storing the randomized return values in the three variables, we must check to see if the values already exist somewhere in our map. To search the map for an existing color, we'll create a new function called ColorExists, which will return a Boolean data type specifying whether or not the three shades provided as function parameters exist within the function. We'll create a while loop that will infinitely loop until ColorExists returns false. Although we could have written the source code within the GenerateColor function, it would definitely clutter the source code because it must check through every different array we've got for the colors we've created.

Within this infinite while loop, we must once again randomize new r, g, and b values with maximum values of 255. This will allow us to continually generate new shades if the ColorExists variable finds that the colors we've created already exist. The final step in this function is to return the 24-bit representation of our RGB values using the RGB macro. Simply use the r, g, and b variables as the parameters for the RGB macro and return the final result. This will return our final RGB values without having to worry if they are duplicates. We've completed writing the code for the GenerateColor function, and can display the source code below.

```
long MAP::GenerateColor()
{
   GLubyte r, g, b;

   r = rand() % 256;
   g = rand() % 256;
   b = rand() % 256;
   while (!ColorExists(r, g, b))
   {
      r = rand() % 256;
```

```
        g = rand() % 256;
        b = rand() % 256;
    }

    return (RGB(r,g,b));
}
```

Checking to See If the Color Exists

In the previous section we discussed how to generate a unique 24-bit color based on the individual red, green, and blue values. In this section, we'll write the underlying code to check if the color values we've created have already been used in a different item within our map format and return a Boolean result. We mentioned this new function (ColorExists), but failed to define it. As stated previously, we know that the function has a bool return type and requires three parameters, all of which are of the GLubyte data type. The parameters, r, g, and b, contain the passed red, green, and blue values we created using the rand statements as shown in the previous section. This function simply checks several predefined colors and every array to see if there is a match between the variables we passed into the function and the ones currently stored in any of the arrays. If there is a match, then we simply return a true value. If there is no match in any of the arrays, we will return a value of false, which indicates that the ColorExists function has failed or the color doesn't exist.

When our function is initially called, we must check to see if the passed RGB values are equal to any of the following colors: red (255, 0, 0), green (0, 255, 0), blue (0, 0, 255), or white (255, 255, 255). If the passed RGB values equal any of these four colors, we must immediately exit the function with a return value of true, meaning the color exists and we should generate new RGB values. We must ensure these colors are reserved because we'll use them for selecting objects in our map editor. If an object is selected and another object already has the color, the user may become confused about which object is actually selected. For this reason we make those colors unavailable for the actual object color. Of course a red color could be drawn using the RGB values 254, 0, 0 and most likely you'd never know the difference, but we'll assume life is simple and this type of scenario won't happen!

Before we perform the loop for each array, we must ensure that there is at least one entry in the array. This will counter any problems we may experience by having unallocated arrays. Granted the maximum value for each item could be wrong, but the code we've written so far will not alter the contents of the arrays without incrementing the values appropriately. The code we've written should hopefully provide us with some sense of security that the data is correct. In either case, we'll presume it is, and if the maximum value is greater than zero, we'll start the loop.

The loop will start at 0 (like many loops do) and continue until the maximum value of the item. With each cycle of the loop, we'll check the values in the select_rgb array against the values we passed into the function. If select_rgb[0] (red) equals the passed r variable, select_rgb[1] (green) equals the passed g variable, and select_rgb[2] (blue) equals the passed b variable, then we've found a match and can immediately return true. If there is no match in this loop, we proceed to the next array in our list and try again. In the event there is no match at the end of the function, the default return value is set to false, indicating that the function has failed.

Although I've used the term "item" fairly loosely, I'm actually describing all the arrays in our map format and not the item array, which holds the data needed for each item in our map. We'll be checking the select_rgb array values against the following arrays in the map format: camera, entity, item, light, object, and sound. Unfortunately, there is no simple way to search through every array without writing a lot of tedious code, but it has to be done! The following source code displays the ColorExists method from the MAP class.

```
bool MAP::ColorExists(GLubyte r, GLubyte g, GLubyte b)
{
   if ((r == 255 && g == 0 && b == 0) ||
   ® == 0 && g == 255 && b == 0) ||
   ® == 0 && g == 0 && b == 255) ||
   ® == 255 && g == 255 && b == 255)) return (true);

   if (header.max_cameras > 0 )
   {
      for (long i = 0; i < header.max_cameras; i++)
      {
         if (camera[i].select_rgb[0] == r && camera[i].select_rgb[1] == g &&
             camera[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_entities > 0)
   {
      for (long i = 0; i < header.max_entities; i++)
      {
         if (entity[i].select_rgb[0] == r && entity[i].select_rgb[1] == g &&
             entity[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_items > 0)
   {
      for (long i = 0; i < header.max_items; i++)
      {
         if (item[i].select_rgb[0] == r && item[i].select_rgb[1] == g &&
             item[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_lights > 0)
```

```
    {
       for (long i = 0; i < header.max_lightmaps; i++)
       {
          if (light[i].select_rgb[0] == r && light[i].select_rgb[1] == g &&
              light[i].select_rgb[2] == b) return (true);
       }
    }
    if (header.max_objects > 0)
    {
       for (long i = 0; i < header.max_objects; i++)
       {
          if (object[i].select_rgb[0] == r && object[i].select_rgb[1] == g &&
              object[i].select_rgb[2] == b) return (true);
       }
    }
    if (header.max_sounds > 0)
    {
       for (long i = 0; i < header.max_sounds; i++)
       {
          if (sound[i].select_rgb[0] == r && sound[i].select_rgb[1] == g &&
              sound[i].select_rgb[2] == b) return (true);
       }
    }

    return (false);
 }
```

Inserting Objects

Before adding the new click-and-drag functionality to our map editor, we must write the underlying source code to insert objects, vertices, and triangles. This functionality is extremely important because it's the turning point in the development of the map editor. We'll be crossing from the land of learning how to write a simple OpenGL application to actually writing a functional map editor. Obviously this is very important because once the editor is finished, we can begin writing our game engine and discuss the ins and outs of making a basic 3D first-person shooter. The downside is that we have much ground to cover between now and then! Without further ado, let's begin writing the underlying source code for the map editor.

In Chapter 3 we defined all the structures we need for the map format and began writing the functionality in the MAP class. Here we'll be writing the underlying code in the MAP class to insert the objects, vertices, and triangles. This will allow us to write the user interface toward the end of the chapter and take advantage of the new creation methods in the MAP class.

The first and probably most important method we'll add to our class controls the insertion of objects. Unlike the other types of methods we'll be creating, this method requires a lot of code for it to work properly. This is because we have several embedded arrays within each index of the

MAP_OBJECT type. For this reason we must be careful when writing the code because we could inadvertently cause data corruption or crashing if the code is not written properly to begin with. The new method will be called InsertObject and has five parameters, three of which will have default parameters set for them. The first parameter is a pointer to a character called name, which will store the name of the object we are adding. This information is vital because it will allow us to easily identity different objects within our levels. In the event this variable is NULL, we'll set the internal object name to "Unknown", which will alleviate any potential crashes.

The next parameter is called type and is of the GLint data type. This variable will be used to distinguish between the different types of objects we'll be creating. Although all the information is saved in the same format, a wall and a floor could have different types and therefore could be drawn in completely different ways. We could also use the type variable to help with collision detection because we can use specifically generated formulas depending on the type of object we are creating. In either case, this variable is quite important when creating objects.

After the type variable, we have another GLint variable called special. This variable contains any special constants we may want to define for an object. A special constant for an object could be used for all sorts of things, such as to decrease the player's health by 10% or flat-out kill the player if he touches the object. This sort of special value would be very useful in the event we wanted to make a level with lava and the player had to follow a specific path. If the player fell off the patch and touched the lava, the special constant would be checked and he'd be flaming mad that he died! By default, this variable will have a value of 0, which indicates there are no special parameters for the variable. Later in the book we'll add special constants that can be assigned to each object in the level.

The next parameter in the function is called is_collidable and is of the GLboolean data type. This variable contains a Boolean representation of whether the object is collidable or not. If the value is true, which is the default value, the user can bump into the object when we write the collision detection. If the is_collidable variable is false, then the user can simply walk through the object without having to worry about getting hurt. This feature is rather useful when writing levels with secret passageways. By allowing certain walls to be not collidable, the player can simply walk through the wall to get the treasure he so badly desires!

Following the is_collidable variable we have another variable of the GLboolean data type called is_visible. This variable, like the previous one, has a default value of true and controls whether the object is visible to the user. You may be wondering why someone would want to make an invisible wall, and the answer is simple. When designing a map, we may want to show off an open area, but limit the users' movements to a narrow path. By setting the object to not visible (false value), the wall wouldn't be drawn, but it would still be checked for a collision provided the is_collidable variable is true as well. There are all sorts of uses for these variables, and hopefully you can put them to good use when designing your maps! We've now completed the definition of the InsertObject method in our MAP class and can begin writing the source code for the object.

The first thing we must do in our newly created method is to declare a new local variable called new_object, which is of the MAP_OBJECT data type. This structure will store the variables that we passed into the structure as well as set internal variables that aren't accessible through the parameters. After declaring the new_object variable, we must declare another variable of the long data type. The variable, rgb, will call the GenerateColor function and store the return value as the default value. This will store our unique RGB value in the rgb variable without having to worry about the mess associated with generating the number in the InsertObject function.

After declaring the RGB value, we will begin filling in the new_object structure with data. The first variable in the structure we'll set is name. Rather than simply do a string copy, it is a good idea to check to see if the name variable passed into the function is NULL. In the event the name variable is NULL, we'll set the new_object.name variable to "Unknown". If the name variable is not NULL, we'll string copy the value from name to new_object.name. Although it seems overly prudent to check for this, some implementations of strcpy have difficulties or downright crash when a NULL string is passed. This could really annoy a level designer who inserts a new object and loses an entire level because the editor crashes. For this reason we'll always try to test the string for NULL, and set a default value so we'll have a smaller chance of causing an issue. Although a value of unknown is rather vague for a name, we've at least named the object and can identify that the object was named automatically.

The next variable we must set in the new_object structure is the type variable. This variable sets the type of object we're creating and is passed a value as a parameter from the function. Later in the chapter we'll discuss how we are going to create a wall and different objects, but for the time being we'll continue to the next variable, which is special. The new_object.special variable will use the special value provided as a parameter to the function. By default this variable will probably be 0, but nonetheless we should still set it to the passed value.

Following the new_object.special variable, we must set the value of the new_object.is_collidable variable to the is_collidable parameter passed into the function. Remember this variable has a default value of true, so unless otherwise specified the object will always be visible. After is_collidable, we must set the variable new_object.is_visible to the is_visible variable passed in by the function. Thankfully, we've finished with the parameters of the function and can fast-track through the rest of the structure.

After the new_object.is_visible variable, we have three variables of the long data type. These variables, max_vertices, max_triangles, and max_textures, control the maximum values for their respective types. Each of these variables will be set to 0 by default, allowing the object to start with no data in the arrays. After we set the maximum values to 0, we must nullify their respective arrays, vertex, triangle, and texture, to prevent them from obtaining weird initial values and to follow common programming rules.

The last variable we'll set is the select_rgb array, which contains the unique selectable red, green, and blue values for the object. Since we've already generated a new color and stored the values in the variable rgb, we can use the macros GetRValue, GetGValue, and GetBValue to retrieve the desired color data. Each macro requires only one parameter, which is the RGB value and in our case is the rgb variable, and the data will be returned. The first index in the select_rgb array will contain the red value, which we'll use the GetRValue macro to obtain. Similarly we'll store the green color data in the second index of select_rgb and obtain the data using GetGValue. The third index will contain the blue color data, which we'll retrieve using GetBValue.

After filling out the MAP_OBJECT structure we can begin writing the bulk of the code for the InsertObject function. Although adding an object sounds fairly simple, there are many steps we must follow for the process to go smoothly. Obviously we'll have to allocate memory for the new object if there are no objects already in the level. In the event there are already objects in the level, we must back up each object, including vertices, triangles, and textures, into a temporary object list, destroy the array, reallocate the memory by adding another index to the array, rebuild the entire object array, destroy the temporary array data, and finally add the newly created object. This process can also be very time consuming if the computer running the software isn't very fast or there isn't much RAM. This process is very RAM intensive because it continually doubles in size when rebuilding the array. Although I'm painting a fairly bad picture of the method we'll be using to create the map editor, this method of object creation is relatively simple to understand and with the amount of RAM shipping with new computers (usually 512 MB), it would take a tremendous amount of data, even when doubled, to fill the RAM.

Following the basic description of how we'll add an object, we'll first check to see how many objects are in the level. If there are no objects in the level (equal to or less than 0), then all we must do is allocate memory for the object pointer for the amount of header.max_objects (0) +1. If the value of the header.max_objects variable is greater than 0, then we must go through the process of backing up the array as discussed in the previous section. We'll add one extra record into the maximum number of objects when we allocate memory, thereby giving us a buffer of one record. This will provide a small amount of protection against calculations that may go one record beyond the maximum objects specified. Hopefully you'll never need this protection, but in the event a calculation returns the wrong index, your editor won't immediately crash. After allocating memory for the object, we set the first index in the object array to the contents of the new_object variable and increment the header.max_objects variable to reflect the addition of the object.

Adding a single object to our level is simple; however, our levels aren't going to contain just one wall! For this reason we have an else clause in the original if statement that is executed when the header.max_objects variable is greater than 0. This is where the bulk of the source code for inserting objects is located because we must back up the data while destroying the allocated arrays. The first thing we must do in the else clause is declare a new pointer of the MAP_OBJECT type. The new variable, temp, will be allocated memory of the size of header.max_objects +1 upon declaration of the variable. This will be our backup array, which will store all the necessary data.

After declaring our temp array, we must copy the entire contents of our current map to the array. Using a simple for loop from 0 to the value of header.max_objects, we can copy the individual variables within each index. Variables such as type, special, is_collidable, is_visible, the maximum array values, and the selectable color data must be set for each index in the temp array. The only variables that are treated differently are name, which must be string copied using strcpy, and the vertex, triangle, and texture arrays, to which we must also allocate memory and copy. With the default values for the temp object set, we must check the max_vertices, max_triangles, and max_textures variables to see if they are greater than 0. If any of the values are greater than 0, we must allocate memory in the temp array for the respective variable and copy the contents of the array. Unlike copying data for the default object values, we can simply do a straight copy of each index instead of setting the values for the individual variables.

When finished copying the data to the temp array, we must destroy each array in our object array containing data and set the array to NULL. In some cases we may not have vertices, triangles, or even textures in an object so we must always check each array. In essence, there are several rules you must follow when creating a map. Obviously if you want to add a triangle you must have at least three different vertices; otherwise the object may not draw properly. If you don't have any vertices, then you'll no doubt run into an error as soon as the object is created. Textures are different from vertices and triangles in that textures aren't required to be attached to the object. We don't need any textures but we have a limit of MAX_TEXTURE_LAYERS if we want to add them. Although it may seem useless to have an object without a texture, it may suit a specific purpose later on when you're designing your levels.

Once we've finished the loop, we must destroy the object array and set it to NULL. It may be overkill to set the array to NULL when in the next line we're going to reallocate the memory to header.max_objects+2, but it will instill some C++ values in you! When allocating the memory for the object array, we set the size of the array to header.max_objects+2, allowing our new record to be added and still give us the one extra record of buffer. After allocating the memory for the object array, we must copy all the data from the temp array back to our newly allocated object array. Although this process seems rather repetitive, which it is, it's a necessary evil when we want the capability of dynamically inserting objects without having a hardcoded maximum number of objects.

Following the steps earlier in the chapter for copying data, we must have a for loop, which loops from 0 to the value in the variable header.max_objects. With each repetition through the loop we'll string copy the name variable from temp object array to the object array. Also we must copy the variables type, special, is_collidable, and is_visible, the maximum values, and the selectable RGB values back to the current object array index. As we discussed earlier in the chapter, these variables must be copied because they contain critical information about the object.

After setting the variables in the object array we must once again copy the vertex, triangle, and texture arrays. Following the previous instructions on this process, we must check each array to ensure that the maximum value is greater than zero. If the maximum value for the respective array is greater than zero, we will allocate memory for the array and copy the contents from the temp array to our newly allocated array. After we copy the contents of the array, we can destroy the temp equivalent of the array. By deleting the arrays as we copy the data back and forth we can simplify the destruction process of the temporary arrays without having to loop through the data at the end of the function. In the event the maximum size is less than or equal to 0, we must set the array to NULL to indicate the array is empty.

Once the main loop has finished, we can destroy the contents of the temp array since all other memory within the array has already been destroyed. Although it's not necessary, we'll also set the temp variable to NULL to maintain consistency throughout our software and end the insertion source code when there is more than one object in the map. In the final two lines of the source code, the InsertObject function set the last object in the list to the value in the new_object variable. Since we've allocated enough space in the object array, we simply use the header.max_objects variable to specify the final object in the array. After the new object value has been set, we complete the InsertObject function by simply incrementing the header.max_object variable to allow the new object to be accounted for in the array. These last two lines work regardless of how many objects are in the list because they use the value in header.max_object, which is set to 0 in the constructor and incremented as new objects are inserted.

Generating Unique Vertex Colors

Before we can insert new vertices into our map we must write the underlying code to generate the unique selectable RGB values. Following the naming convention we used when generating a selectable RGB value, we'll need to create two new functions. The first member function we'll create is called VertexColorExists, which has a bool return type. This new member function requires four parameters for it to work properly. The first parameter is of the long data type and is called obj. This variable contains the object number we need to reference the data from. This is important because it allows certain colors to be generated twice but contained in different objects, which ultimately allows us to have more vertices per object. The next three parameters, r, g, and b, are all of the GLubyte data type. The variables contain separate r, g, b values, which are the randomized color values of the potential color we want to create.

Keeping with tradition, we will not allow any full-intensity color (red, green, blue) or white to be passed into the function. In the event that the specific color values are passed into the VertexColorExists function, we'll simply return a value of true, which indicates the color already exists. This will keep us from using any colors that are deemed important. (Like many software packages, our map editor will use specific colors to indicate different pieces of information in the map.) Depending on how complex your project is, you may want to add more colors to the key palette, in which case I would recommend doing a mix of full intensities to create magenta, cyan, and yellow. Of course you can use whatever colors you like; these are just suggestions because they are easy to remember and understand. After checking the key color information we simply loop through the vertex array from 0 to the maximum value (object[obj].max_vertices), checking to see if the selectable RGB values are equal to the r, g, and b values we passed into the function. In the event the values are equal, we must immediately exit the function with a return value of true, indicating the RGB values exist. In the event the function doesn't find the passed RGB values in the loop or in the key colors check, we simply return a false value to indicate that the color doesn't exist and can be used when inserting a new vertex. The source code for the VertexColorExists member function is provided here:

```
bool MAP::VertexColorExists(long obj, GLubyte r, GLubyte g, GLubyte b)
{
   if ((r == 255 && g == 0 && b == 0) ||
      ® == 0 && g == 255 && b == 0) ||
      ® == 0 && g == 0 && b == 255) ||
      ® == 255 && g == 255 && b == 255)) return (true);

   for (long i = 0; i < object[obj].max_vertices; i++)
   {
      if (object[obj].select_rgb[0] == r && object[obj].select_rgb[1] == g &&
         object[obj].select_rgb[2] == b) return (true);
   }

   return (false);
}
```

The second member function we'll need to create to generate vertex colors is appropriately named GenerateVertexColor. This function has one parameter of the long data type, which stores the object number we are generating colors for. The return value for the function is of the long data type because we want to return the 24-bit RGB representation of the color as we did in the GenerateColor function. This is the function we'll use to generate vertex colors for each vertex being inserted into an object. The first thing we must do in the function is declare three new variables of the GLubyte data type. The variables r, g, and b will store our randomly generated values for their respective shade. After declaring the new variables, we must randomize values for the three variables, setting a maximum value of 255 (because a color cannot exceed that value). With the initial RGB values set, we must perform a while loop and ensure the color we randomized does not exist using our newly created function VertexColorExists. Obviously, if the color exists, we must set the r, g, and b variables to the newly randomized color components and run the process again until we arrive at a color that doesn't exist. Once the while loop finishes generating the RGB values, we simply return the value returned from the macro RGB, supplying the variables r, g, and b as the respective parameters for the macro. The source code for the GenerateVertexColor member function is provided below:

```
long MAP::GenerateVertexColor(long obj)
{
   GLubyte r, g, b;

   r = rand()%256;
   g = rand()%256;
   b = rand()%256;
   while (VertexColorExists(obj, r, g, b))
   {
      r = rand()%256;
      g = rand()%256;
      b = rand()%256;
   }
   return (RGB(r,g,b));
}
```

Inserting Vertices

Earlier in this chapter we discussed the process of inserting objects into the object array. In this section we'll write the code to insert a vertex into any object while generating the selectable RGB values for each new vertex. Fortunately, inserting vertices is a much faster process because we only have to concern ourselves with the object specified as the parameter and not the entire object array as with the InsertObject function.

To begin, we'll create a new method in our MAP class called InsertVertex, which unlike our InsertObject method will have a return type of bool. We want to specify a return type for this method because there is one parameter, which must be accurate data; otherwise the method will fail (return false). The InsertVertex function has a total of 12 parameters, four of which are required when calling the method; the rest are all overloaded. The first parameter in the InsertVertex function is of the long data type. The variable obj specifies the object into which we want to insert this vertex. The next three variables are all of the GLdouble data type. These variables are named x, y, and z because they store the respective coordinates for the vertex. After the x, y, and z variables we have four variables of the GLfloat data type. The r, g, b, and a variables contain their respective red, green, blue, and alpha intensities for the specified vertex. This information is necessary because it will allow us to easily change the color of an object without having to load a different texture. We'll use these variables to change the darkness of the objects we create, allowing us to set the mood of the level. Unlike the previous parameters, all four of these variables have default values of 1.0, which gives the programmer the option to specify different values or use the default values. In many cases we'll want to set the default color values to 1.0, and then change the colors appropriately later on. With default values of 1.0, the initial color of the object will be white, which will make the object look like it's textured using normal colors.

After specifying the r, g, b, and a parameters for the InsertVertex function we have another three variables of the GLfloat data type. The variables nx, ny, and nz specify the direction in which the vertex points. Each axis (x, y, and z) has its own individual values, which is why we have the nx (normal x-axis), ny (normal y-axis), and nz (normal z-axis) variables. Unlike the previous four parameters, the normal parameters will have default values of 0.0. If we use 1.0 for the default value, we'll actually be setting the normals to point in a specific direction. By setting each normal axis to 0.0, we won't make our vertex point in any specific direction. The normal information is important when doing the calculations for certain special effects like lighting and bumpmapping. We'll discuss normals and lighting later in the book, but for the time being let's move on to the final parameter of the InsertVertex method.

The final parameter in the InsertVertex method is fogdepth, which is of the GLfloat data type. This variable contains the information needed to generate volumetric fog on the vertex. In case you're wondering, volumetric fog is different from regular fog because we can specify the depth on a per-vertex basis, as opposed to the depth being automatically calculated by the video card. This allows us to create a wide variety of special effects like mist, steam, and fog that cover the floor but is clear near the ceiling. The default value for this variable will be 0.0, which will indicate there is no fog on the vertex. Keep in mind that this variable will only be used if the user has the specific OpenGL extension installed in his video card drivers. If the extension isn't installed, the volumetric fog will not be drawn. We'll discuss fog in further detail later in the book, but in the meantime we'll discuss the code required to insert a vertex into the current vertex list.

After defining the InsertVertex method, we must declare a new variable called new_vertex, which is of the MAP_VERTEX data type. This variable will store the parameters in InsertVertex in a proper vertex structure, which we'll set to the new vertex value once we've allocated the appropriate memory. After declaring the new_vertex variable we must declare a new variable called rgb, which is of the long data type. This new variable will have a default value, which is returned from the function GenerateVertexColor. The variable, like the one in the InsertObject method, will contain the RGB value, in long form, for the selectable RGB values. This is a very important set of variables because we will no doubt want to select and move vertices in our map editor.

Before we can allocate memory, we must ensure that the value specified in the obj parameter is not greater than the maximum number of objects (header.max_objects) in the map and not less than 0. This will ensure we don't try to access an object that doesn't exist in the map, which would eventually cause a crash. Obviously we want to safeguard our software from potential crashing issues when possible. This small check will hopefully help with stability in the event bad data is passed. In the event the value in the obj variable is greater than the header.max_objects variable or less than 0, we'll simply exit the function with a return value of false, indicating that the function has failed. In all other instances, the function will continue with normal operations.

Following the obj value validity check, we'll store parameters of the function in our newly created MAP_VERTEX variable, new_vertex. This will allow us to easily copy the data into the new index we'll be allocating in a few moments. Using the same ordering as our parameters, we'll store the parameters x, y, and z in the new_vertex.xyz array. The next parameters we'll store in the new_vertex variable are the RGBA values for the vertex. The array rgba in the new_vertex variable will store the parameters r, g, b, and a in consecutive order. After setting the RGBA values, we'll set the nx, ny, and nz normal values in the new_vertex.normal array. Then we'll set the new_vertex.fog_depth to the value of the parameter fogdepth.

After setting the variables in the new_vertex structure to parameters passed into the function, we must set the selectable red (new_vertex.select_rgb[0]), green (new_vertex.select_rgb[1]), and blue (new_vertex.select_rgb[2]) colors, which are returned using the GetRValue, GetGValue, and GetBValue macros, and supply our rgb variable as the single parameter required for the macros. We've now completely filled in the structure and can begin writing the actual code to insert the vertex. After filling in the new_vertex structure with data we must check the object[obj].max_vertices variable for 0. If the value of object[obj].max_vertices is 0, we'll allocate the object[obj].vertex array with one record. In the event the value of object[obj].max_vertices is not equal to 0, we'll declare a new pointer variable called temp, which is of the MAP_VERTEX data type. We'll dynamically allocate memory for this variable using the value of object[obj].max_vertices plus an extra record.

After allocating the memory for the object[obj].vertex array, we'll loop from 0 to the value of object[obj].max_vertices and copy each index from object[obj].vertex to the temp array equivalent. After copying the data from the vertex array, we'll destroy the array using the delete operator and reallocate memory on the next line to the value of object[obj].max_vertices plus two, giving us one new record with a buffer of one extra record. With the allocation of the object[obj].vertex array complete we can simply copy the data back from the temp array to the newly allocated object[obj].vertex array, looping from 0 to the value of object[obj].max_vertices. Once the loop is finished, we simply use the delete operator on the temp array to free the allocated memory from the array and set the temp array to NULL, which completes the non-zero value of the if statement.

Following the if statement we simply set the object[obj].vertex[object[obj].max_vertices] value to that of new_vertex and increment the value of object[obj].max_vertices by one. Since the value of object[obj].max_vertices is always equal to or greater than 0 and we set the contents of object[obj].vertex using the index of object[obj].max_vertices, it will always point to the next record we are inserting into the list. The final line of source code in the InsertVertex member function is to return a value of true, indicating the function succeeded in adding the vertex. The return value will always be true unless the user specifies an object that does not exist. We've now completed the source code for inserting a vertex into any object in the map. Simple, huh? The source code is provided below:

```
bool MAP::InsertVertex(long obj, GLdouble x, GLdouble y, GLdouble z, GLfloat r,
        GLfloat g, GLfloat b, GLfloat a, GLfloat nx, GLfloat ny, GLfloat nz,
        GLfloat fogdepth)
```

```
{
   MAP_VERTEX       new_vertex;
   long             rgb = GenerateVertexColor(obj);

   if (obj > header.max_objects || obj < 0) return (false);

   new_vertex.xyz[0]            = x;
   new_vertex.xyz[1]            = y;
   new_vertex.xyz[2]            = z;
   new_vertex.rgba[0]           = r;
   new_vertex.rgba[1]           = g;
   new_vertex.rgba[2]           = b;
   new_vertex.rgba[3]           = a;
   new_vertex.normal[0]         = nx;
   new_vertex.normal[1]         = ny;
   new_vertex.normal[2]         = nz;
   new_vertex.fog_depth         = fogdepth;
   new_vertex.select_rgb[0]     = GetRValue(rgb);
   new_vertex.select_rgb[1]     = GetGValue(rgb);
   new_vertex.select_rgb[2]     = GetBValue(rgb);

   if (object[obj].max_vertices == 0) object[obj].vertex = new MAP_VERTEX
       [object[obj].max_vertices+1];
   else
   {
      MAP_VERTEX    *temp = new MAP_VERTEX[object[obj].max_vertices+1];
      for (long i = 0; i < object[obj].max_vertices; i++) temp[i] =
           object[obj].vertex[i];

      delete [] object[obj].vertex;
      object[obj].vertex = new MAP_VERTEX[object[obj].max_vertices+2];

      for (i = 0; i < object[obj].max_vertices; i++) object[obj].vertex[i] =
           temp[i];
      delete [] temp;
      temp = NULL;
   }
   object[obj].vertex[object[obj].max_vertices] = new_vertex;
   object[obj].max_vertices++;

   return (true);
}
```

Inserting Triangles

After writing the code for inserting vertices, we must write the code to insert triangles into an object. Once the function is finished, we will be able to create fully three-dimensional objects by linking the indexes of vertices in our vertex array as the three points in each triangle. As mentioned before, all objects in our map format are broken down into triangles because many video cards can accelerate raw triangle information better than raw polygon data. Of course there are always exceptions to this rule, but for the most part, many video cards will draw a scene better using triangles than conventional polygons.

To begin we'll need to create a new member function called InsertTriangle, which as the name implies inserts triangles into a specified object. This new member function will have a bool return type to indicate whether the function has succeeded or failed. Although each triangle can hold numerous pieces of data, depending on the number of texture layers, we'll require 10 parameters to insert each triangle into the map format.

The first parameter is called obj, and is of the long data type. This variable will contain the index of the object into which we want to insert this triangle. The next three variables are all of the GLint data type. The variables p1, p2, and p3 contain the indexes of the vertices needed to draw a specified triangle. We use the GLint data type because there could literally be hundreds, thousands, or even millions of vertices in one specific object. We won't take advantage of millions of vertices, but it's nice to keep the possibility! The final six parameters are all of the GLfloat data type, and control the number of times the texture will be tiled on the triangle. This will be explained in more detail later in the book. After defining the new member function, we must follow our new inserting procedures and declare a new variable based on the data type we are inserting. In this case, we'll declare a new variable called new_triangle, which is of the MAP_TRIANGLE data type. This variable will store all the variables passed into the function as well as default values for the variables that were not passed into the function.

After declaring the new variable, we must ensure the obj variable does not exceed the maximum number of objects in our map to ensure our software doesn't try accessing memory that has not been allocated. In the event the obj variable is greater than the value of header.max_objects, we'll return a false value, which indicates that the function has failed. In the event the value of the obj variable is less than header.max_objects, we'll set each new_triangle.point index to the appropriate parameter (p1, p2, p3). Next we'll loop through the texture layers and set the texture tiling information for each triangle point to the parameter's input into the function. Finally we'll dynamically add this triangle into the object's triangle array, increase the object triangle count by one, then exit the function with a return value of true to indicate success.

```
bool MAP::InsertTriangle(long obj, GLint p1, GLint p2, GLint p3, GLfloat u1,
          GLfloat v1, GLfloat u2, GLfloat v2, GLfloat u3, GLfloat v3)
{
   MAP_TRIANGLE new_triangle;
   if (obj > header.max_objects) return (false);

   new_triangle.point[0] = p1;
   new_triangle.point[1] = p2;
   new_triangle.point[2] = p3;
   for (long i = 0; i < MAX_TEXTURE_LAYERS; i++)
   {
      new_triangle.uv[i].uv1[0] = u1;
      new_triangle.uv[i].uv1[1] = v1;

      new_triangle.uv[i].uv2[0] = u2;
      new_triangle.uv[i].uv2[1] = v2;

      new_triangle.uv[i].uv3[0] = u3;
      new_triangle.uv[i].uv3[1] = v3;
   }

   if (object[obj].max_triangles <= 0) object[obj].triangle = new
       MAP_TRIANGLE[1];
   else
   {
      MAP_TRIANGLE *temp = new MAP_TRIANGLE[object[obj].max_triangles+1];
      for (long i = 0; i < object[obj].max_triangles; i++) temp[i] =
           object[obj].triangle[i];

      delete [] object[obj].triangle;
      object[obj].triangle = new MAP_TRIANGLE[object[obj].max_triangles+2];
      for (i = 0; i < object[obj].max_triangles; i++) object[obj].triangle[i] =
           temp[i];

      delete [] temp;
      temp = NULL;
   }

   object[obj].triangle[object[obj].max_triangles] = new_triangle;
   object[obj].max_triangles++;

   return (true);
}
```

Chapter Example

Although the chapter seems rather small, the source code isn't! The topics discussed in this chapter literally create the building blocks of our game engine. Without these core functions, we wouldn't be able to insert a vertex, object, or triangle without having to continually write the code manually in each function to which we want to add the type. This is one of the wonderful things of encapsulation with C++! We can simply write several members within the MAP class to handle all the functionality we need, and keep it separate from the rest of the program.

ex4_1.cpp

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <windows.h>

#define MAX_TEXTURE_LAYERS      2
#define MAX_STRING_SIZE         500

#define VERSION                 1
#define REVISION                0


typedef struct
{
   GLint            version;
   GLint            revision;
} MAP_VERSION;


typedef struct
{
   GLint            max_objects;
   GLint            max_lights;
   GLint            max_lightmaps;
   GLint            max_cameras;

   GLint            max_entities;
   GLint            max_items;
   GLint            max_sounds;

   GLboolean        use_skybox;
   GLboolean        use_fog;
} MAP_HEADER;


typedef struct
{
   char             filename[MAX_STRING_SIZE];
   GLint            texid;
} MAP_SKYBOX_SIDE;


typedef struct
{
   MAP_SKYBOX_SIDE front;
   MAP_SKYBOX_SIDE back;
   MAP_SKYBOX_SIDE left;
   MAP_SKYBOX_SIDE right;
   MAP_SKYBOX_SIDE top;
   MAP_SKYBOX_SIDE bottom;
} MAP_SKYBOX;


typedef struct
{
   GLint            mode;
   GLfloat          start;
   GLfloat          end;
   GLfloat          density;
   GLfloat          rgba[4];
} MAP_FOG;


typedef struct
{
   GLdouble         xyz[3];
   GLfloat          angle[3];
   GLint            model;

   GLubyte          select_rgb[3];
} MAP_STARTING_POSITION;


typedef struct
{
   char             map_name[MAX_STRING_SIZE];
   GLint            map_type;
   GLint            map_exit_rules;

   MAP_STARTING_POSITION  single_player;
   MAP_STARTING_POSITION  deathmatch[2];
} MAP_DETAILS;


typedef struct
{
   GLdouble         xyz[3];
   GLfloat          rgba[4];
   GLfloat          normal[3];
   GLfloat          fog_depth;
   GLubyte         select_rgb[3];
} MAP_VERTEX;


typedef struct
{
   GLfloat               uv1[2];
   GLfloat               uv2[2];
   GLfloat               uv3[2];
} MAP_UV_COORDS;


typedef struct
{
   GLint                 point[3];
   MAP_UV_COORDS         uv[MAX_TEXTURE_LAYERS];
} MAP_TRIANGLE;


typedef struct
{
   char                  filename[MAX_STRING_SIZE];
   GLint                 id;
   GLint                 style;
   GLint                 blend_src;
   GLint                 blend_dst;
} MAP_TEXTURE;


typedef struct
{
   char                  name[MAX_STRING_SIZE];
   GLint                 type;
   GLint                 special;

   GLboolean             is_collidable;
   GLboolean             is_visible;

   GLint                 max_textures;
   GLint                 max_triangles;
   GLint                 max_vertices;

   MAP_TEXTURE           *texture;
   MAP_TRIANGLE          *triangle;
   MAP_VERTEX            *vertex;


   GLubyte               select_rgb[3];
} MAP_OBJECT;
```

```
typedef struct
{
   char              name[MAX_STRING_SIZE];
   GLdouble          xyz[3];
   GLfloat           angle[3];

   GLubyte           select_rgb[3];
} MAP_CAMERA;


typedef struct
{
   char              name[MAX_STRING_SIZE];
   GLint             type;
   GLdouble          xyz[3];
   GLfloat           angle[3];
   GLfloat           rgba[4];

   char              texture_filename[MAX_STRING_SIZE];
   GLint             texture;

   GLint             max_inclusions;
   GLint             *inclusions;

   GLubyte           select_rgb[3];
} MAP_LIGHT;


typedef struct
{
   GLint             type;
   GLdouble          xyz[3];
   GLfloat           angle[3];

   GLint             health;
   GLint             strength;
   GLint             armour;

   GLubyte           select_rgb[3];
} MAP_ENTITY;


typedef struct
{
   GLint             type;
   GLint             respawn_wait;
   GLint             respawn_time;
   GLdouble          xyz[3];

   GLubyte           select_rgb[3];
} MAP_ITEM;
```

```
typedef struct
{
   char            filename[MAX_STRING_SIZE];
   GLint           id;
   GLdouble        xyz[3];
   GLfloat         angle[3];

   GLubyte         select_rgb[3];
} MAP_SOUND;


class MAP {

   public:
      MAP_VERSION         version;
      MAP_HEADER          header;
      MAP_SKYBOX          skybox;
      MAP_FOG             fog;
      MAP_DETAILS         details;

      MAP_OBJECT          *object;
      MAP_ENTITY          *entity;
      MAP_CAMERA          *camera;
      MAP_LIGHT           *light;
      MAP_SOUND           *sound;
      MAP_ITEM            *item;

   MAP();
   ~MAP();

   bool ColorExists(GLubyte r, GLubyte g, GLubyte b);
   long GenerateColor();
   long GenerateVertexColor(long obj);
   bool VertexColorExists(long obj, GLubyte r, GLubyte g, GLubyte b);

   void InsertObject(char *name, GLint type, GLint special=0, GLboolean
          is_collidable=true, GLboolean is_visible=true);
   bool InsertVertex(long obj, GLdouble x, GLdouble y, GLdouble z, GLfloat r=1.0,
          GLfloat g=1.0, GLfloat b=1.0, GLfloat a=1.0, GLfloat nx=0.0, GLfloat
          ny=0.0, GLfloat nz=0.0, GLfloat fogdepth=0.0);
   bool InsertTriangle(long obj, GLint p1, GLint p2, GLint p3);

};


MAP::MAP()
{
   version.version  = VERSION;
   version.revision = REVISION;

   memset (&header, 0, sizeof(header));
   memset (&details, 0, sizeof(details));
   memset (&skybox, 0, sizeof(skybox));
   memset (&fog, 0, sizeof(fog));

   object                 = NULL;
   entity                 = NULL;
   camera                 = NULL;
   light                  = NULL;
   sound                  = NULL;
   item                   = NULL;

}


MAP::~MAP()
{
   if (header.max_objects > 0)
   {
      for (long i = 0; i < header.max_objects; i++)
      {
         if (object[i].max_vertices > 0)
         {
            delete [] object[i].vertex;
            object[i].vertex          = NULL;
            object[i].max_vertices    = 0;
         }

         if (object[i].max_triangles > 0)
         {
            delete [] object[i].triangle;
            object[i].triangle        = NULL;
            object[i].max_triangles   = 0;
         }

         if (object[i].max_textures > 0)
         {
            delete [] object[i].texture;
            object[i].texture         = NULL;
            object[i].max_textures    = 0;
         }
      }

      delete [] object;
      object                          = NULL;
      header.max_objects              = 0;
   }


   if (header.max_cameras > 0)
   {
      delete [] camera;
      camera                          = NULL;
      header.max_cameras              = 0;
   }
```

```
    if (header.max_entities > 0)
    {
       delete [] entity;
       entity                    = NULL;
       header.max_entities       = 0;
    }


    if (header.max_items > 0)
    {
       delete [] item;
       item                      = NULL;
       header.max_items          = 0;
    }


    if (header.max_sounds > 0)
    {
       delete [] sound;
       sound                     = NULL;
       header.max_sounds         = 0;
    }


    if (header.max_lights > 0)
    {
       delete [] light;
       light                     = NULL;
       header.max_lights         = 0;
    }
 }


 void MAP::InsertObject(char *name, GLint type, GLint special, GLboolean
           is_collidable, GLboolean is_visible)
 {
    MAP_OBJECT       new_object;
    long             rgb = GenerateColor();


    if (name != NULL) strcpy (new_object.name, name);
    else strcpy (new_object.name, "Unknown");
    new_object.type              = type;
    new_object.special           = special;
    new_object.is_collidable     = is_collidable;
    new_object.is_visible        = is_visible;
    new_object.max_vertices      = 0;
    new_object.max_triangles     = 0;
    new_object.max_textures      = 0;
    new_object.vertex            = NULL;
```

```
new_object.triangle          = NULL;
new_object.texture           = NULL;
new_object.select_rgb[0]     = GetRValue (rgb);
new_object.select_rgb[1]     = GetGValue (rgb);
new_object.select_rgb[2]     = GetBValue (rgb);


if (header.max_objects == 0) object = new MAP_OBJECT[header.max_objects+1];
else
{
   MAP_OBJECT *temp = new MAP_OBJECT[header.max_objects+1];
   for (long i = 0; i < header.max_objects; i++)
   {
      strcpy (temp[i].name, object[i].name);
      temp[i].type           = object[i].type;
      temp[i].special        = object[i].special;
      temp[i].is_collidable  = object[i].is_collidable;
      temp[i].is_visible     = object[i].is_visible;
      temp[i].max_vertices   = object[i].max_vertices;
      temp[i].max_triangles  = object[i].max_triangles;
      temp[i].max_textures   = object[i].max_textures;
      temp[i].select_rgb[0]  = object[i].select_rgb[0];
      temp[i].select_rgb[1]  = object[i].select_rgb[1];
      temp[i].select_rgb[2]  = object[i].select_rgb[2];

      if (temp[i].max_vertices > 0)
      {
         temp[i].vertex = new MAP_VERTEX[temp[i].max_vertices+1];
         for (long i2 = 0; i2 < temp[i].max_vertices; i2++) temp[i].vertex[i2]
                      = object[i].vertex[i2];

         delete [] object[i].vertex;
         object[i].vertex = NULL;
      }
      else temp[i].vertex = NULL;


      if (temp[i].max_triangles > 0)
      {
         temp[i].triangle = new MAP_TRIANGLE[temp[i].max_triangles+1];
         for (long i2 = 0; i2 < temp[i].max_triangles; i2++)
              temp[i].triangle[i2] = object[i].triangle[i2];

         delete [] object[i].triangle;
         object[i].triangle = NULL;
      }
      else temp[i].triangle = NULL;


      if (temp[i].max_textures > 0)
      {
         temp[i].texture = new MAP_TEXTURE[temp[i].max_textures+1];
```

```
            for (long i2 = 0; i2 < temp[i].max_textures; i2++) temp[i].texture[i2]
                         = object[i].texture[i2];

            delete [] object[i].texture;
            object[i].texture = NULL;
         }
         else temp[i].texture = NULL;

      }
      delete [] object;
      object = NULL;

      object = new MAP_OBJECT[header.max_objects+2];
      for (i = 0; i < header.max_objects; i++)
      {
         strcpy (object[i].name, temp[i].name);
         object[i].type         = temp[i].type;
         object[i].special      = temp[i].special;
         object[i].is_collidable= temp[i].is_collidable;
         object[i].is_visible   = temp[i].is_visible;
         object[i].max_vertices = temp[i].max_vertices;
         object[i].max_triangles= temp[i].max_triangles;
         object[i].max_textures = temp[i].max_textures;
         object[i].select_rgb[0]= temp[i].select_rgb[0];
         object[i].select_rgb[1]= temp[i].select_rgb[1];
         object[i].select_rgb[2]= temp[i].select_rgb[2];

         if (object[i].max_vertices > 0)
         {
            object[i].vertex = new MAP_VERTEX[object[i].max_vertices+1];
            for (long i2 = 0; i2 < object[i].max_vertices; i2++)
                 object[i].vertex[i2] = temp[i].vertex[i2];

            delete [] temp[i].vertex;
            temp[i].vertex = NULL;
         }
         else object[i].vertex = NULL;


         if (object[i].max_triangles > 0)
         {
            object[i].triangle = new MAP_TRIANGLE[object[i].max_triangles+1];
            for (long i2 = 0; i2 < object[i].max_triangles; i2++)
                 object[i].triangle[i2] = temp[i].triangle[i2];

            delete [] temp[i].triangle;
            temp[i].triangle = NULL;
         }
         else object[i].triangle = NULL;


         if (object[i].max_textures > 0)
         {
            object[i].texture = new MAP_TEXTURE[temp[i].max_textures+1];
            for (long i2 = 0; i2 < temp[i].max_textures; i2++)
                 object[i].texture[i2] = temp[i].texture[i2];

            delete [] temp[i].texture;
            temp[i].texture = NULL;
         }
         else object[i].texture = NULL;

      }

      delete [] temp;
      temp = NULL;
   }

   object[header.max_objects] = new_object;
   header.max_objects++;
}


long MAP::GenerateVertexColor(long obj)
{
   GLubyte r, g, b;

   r = rand()%256;
   g = rand()%256;
   b = rand()%256;
   while (VertexColorExists(obj, r, g, b))
   {
      r = rand()%256;
      g = rand()%256;
      b = rand()%256;
   }
   return (RGB(r,g,b));
}


bool MAP::InsertVertex(long obj, GLdouble x, GLdouble y, GLdouble z, GLfloat r,
          GLfloat g, GLfloat b, GLfloat a, GLfloat nx, GLfloat ny, GLfloat nz,
          GLfloat fogdepth)
{
   MAP_VERTEX       new_vertex;
   long             rgb = GenerateVertexColor(obj);

   if (obj > header.max_objects || obj < 0) return (false);

   new_vertex.xyz[0]           = x;
   new_vertex.xyz[1]           = y;
   new_vertex.xyz[2]           = z;
   new_vertex.rgba[0]          = r;
   new_vertex.rgba[1]          = g;
   new_vertex.rgba[2]          = b;
   new_vertex.rgba[3]          = a;
   new_vertex.normal[0]          = nx;
   new_vertex.normal[1]          = ny;
   new_vertex.normal[2]          = nz;
   new_vertex.fog_depth          = fogdepth;
   new_vertex.select_rgb[0]      = GetRValue(rgb);
   new_vertex.select_rgb[1]      = GetGValue(rgb);
   new_vertex.select_rgb[2]      = GetBValue(rgb);

   if (object[obj].max_vertices == 0) object[obj].vertex = new MAP_VERTEX
       [object[obj].max_vertices+1];
   else
   {
      MAP_VERTEX    *temp = new MAP_VERTEX[object[obj].max_vertices+1];
      for (long i = 0; i < object[obj].max_vertices; i++) temp[i] =
           object[obj].vertex[i];

      delete [] object[obj].vertex;
      object[obj].vertex = new MAP_VERTEX[object[obj].max_vertices+2];

      for (i = 0; i < object[obj].max_vertices; i++) object[obj].vertex[i] =
           temp[i];
      delete [] temp;
      temp = NULL;
   }
   object[obj].vertex[object[obj].max_vertices] = new_vertex;
   object[obj].max_vertices++;

   return (true);
}


bool MAP::ColorExists(GLubyte r, GLubyte g, GLubyte b)
{
   if ((r == 255 && g == 0 && b == 0) ||
      ® == 0 && g == 255 && b == 0) ||
      ® == 0 && g == 0 && b == 255) ||
      ® == 255 && g == 255 && b == 255)) return (true);

   if (header.max_cameras > 0 )
   {
      for (long i = 0; i < header.max_cameras; i++)
      {
         if (camera[i].select_rgb[0] == r && camera[i].select_rgb[1] == g &&
             camera[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_entities > 0)
   {
      for (long i = 0; i < header.max_entities; i++)
      {
         if (entity[i].select_rgb[0] == r && entity[i].select_rgb[1] == g &&
             entity[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_items > 0)
   {
      for (long i = 0; i < header.max_items; i++)
      {
         if (item[i].select_rgb[0] == r && item[i].select_rgb[1] == g &&
             item[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_lights > 0)
   {
      for (long i = 0; i < header.max_lightmaps; i++)
      {
         if (light[i].select_rgb[0] == r && light[i].select_rgb[1] == g &&
             light[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_objects > 0)
   {
      for (long i = 0; i < header.max_objects; i++)
      {
         if (object[i].select_rgb[0] == r && object[i].select_rgb[1] == g &&
             object[i].select_rgb[2] == b) return (true);
      }
   }
   if (header.max_sounds > 0)
   {
      for (long i = 0; i < header.max_sounds; i++)
      {
         if (sound[i].select_rgb[0] == r && sound[i].select_rgb[1] == g &&
             sound[i].select_rgb[2] == b) return (true);
      }
   }

   return (false);
}


long MAP::GenerateColor()
{
   GLubyte r, g, b;


   r = rand() % 256;
   g = rand() % 256;
   b = rand() % 256;
   while (ColorExists(r, g, b))
   {
      r = rand() % 256;
      g = rand() % 256;
      b = rand() % 256;
   }
   return (RGB(r,g,b));
}


bool MAP::InsertTriangle(long obj, GLint p1, GLint p2, GLint p3, GLfloat u1,
         GLfloat v1, GLfloat u2, GLfloat v2, GLfloat u3, GLfloat v3)
{
   MAP_TRIANGLE new_triangle;
   if (obj > header.max_objects) return (false);

   new_triangle.point[0] = p1;
   new_triangle.point[1] = p2;
   new_triangle.point[2] = p3;
   for (long i = 0; i < MAX_TEXTURE_LAYERS; i++)
   {
      new_triangle.uv[i].uv1[0] = u1;
      new_triangle.uv[i].uv1[1] = v1;

      new_triangle.uv[i].uv2[0] = u2;
      new_triangle.uv[i].uv2[1] = v2;

      new_triangle.uv[i].uv3[0] = u3;
      new_triangle.uv[i].uv3[1] = v3;
   }

   if (object[obj].max_triangles <= 0) object[obj].triangle = new
          MAP_TRIANGLE[1];
   else
   {
      MAP_TRIANGLE *temp = new MAP_TRIANGLE[object[obj].max_triangles+1];
      for (long i = 0; i < object[obj].max_triangles; i++) temp[i] =
          object[obj].triangle[i];

      delete [] object[obj].triangle;
      object[obj].triangle = new MAP_TRIANGLE[object[obj].max_triangles+2];
      for (i = 0; i < object[obj].max_triangles; i++) object[obj].triangle[i] =
             temp[i];

      delete [] temp;
      temp = NULL;
   }

   object[obj].triangle[object[obj].max_triangles] = new_triangle;
   object[obj].max_triangles++;

   return (true);
}


bool MAP::VertexColorExists(long obj, GLubyte r, GLubyte g, GLubyte b)
{
   if ((r == 255 && g == 0 && b == 0) ||
      ® == 0 && g == 255 && b == 0) ||
      ® == 0 && g == 0 && b == 255) ||
      ® == 255 && g == 255 && b == 255)) return (true);

   for (long i = 0; i < object[obj].max_vertices; i++)
   {
      if (object[obj].select_rgb[0] == r && object[obj].select_rgb[1] == g &&
                object[obj].select_rgb[2] == b) return (true);
   }

   return (false);
}
```

Conclusion

As mentioned before, the example for this chapter isn't terribly exciting. Actually it doesn't do anything at all, as planned, because it's a precursor to Chapter 5 where we get to the meat and potatoes of creating the map editor and its core functionality. Much of the work involved in creating objects can be done by simply calling the member functions we've created in this chapter. This will become very useful later in the book when we want to load levels and insert irregular-shaped objects into the maps.

Chapter 5

Creating Map Objects

By the end of this chapter, we'll have written all the code necessary to create floors, ceilings, and walls in our maps. In case you're wondering, we treat floors and ceilings differently to allow us to customize collision detection within the game. Each object type would have a different type of collision detection algorithm to determine whether the user has collided with another object. Ceilings are different because a user cannot walk directly into one; he or she would have to jump to hit it. We can easily track which types of movements are being performed, e.g., walking, jumping, or falling, and use the appropriate collision detection for specific objects.

The Basic Interface

In Chapter 1 we discussed the principles of event-driven programming. When the user executes a specific action, a message is sent to the event handler detailing which message was sent. Our map editor will use a four-step process to create primitives. The first step requires the user to select the primitive creation type he wants, e.g., wall, floor, or ceiling. After setting the primitive creation type, we must wait for the user to press the left mouse button and record the starting X/Y mouse positions. The next step is to let the user size the primitive as desired and record the end X/Y positions of the mouse as the ending coordinate. The start and end positions for the primitive will be used in the creation process to see the actual size and angle of the walls we are creating. The user will have a difficult time creating levels for your game if the editor isn't easy to use or doesn't provide adequate functionality for his needs. For that specific reason we'll create the simple click-and-drag interface for drawing primitives. If the user can create a line in Microsoft Paint, he'll be able to create a level in our map editor!

The final step in the primitive creation process is to wait for the left mouse button to be released and then record the final end X/Y mouse positions. Although we've continually recorded the positions during the mouse movement event, there may have been some odd rarity that another process may have had a higher priority and therefore lost the final mouse movement position. This will eliminate any mouse X/Y position problems by recording the position at which the button was released. Once the left button is released, we can add the primitive to the map and restart the entire process. The one thing not mentioned in the above process is a small step that converts each mouse X/Y position to its respective world location value. We want to convert the mouse X/Y positions into a format that will work in our map format, regardless of whether the primitive has been added. Each X/Y position must be converted to its world location equivalent because we want to display the drawn lines in real time. It's very difficult to display the drawn lines in real time if the lines are not properly calculated.

To begin this process, we must first create an enumerated type that has four values: CREATE_MODE_NULL (default value of 0), CREATE_MODE_START, CREATE_MODE_SIZE, and CREATE_MODE_FINISH. This type will control the different modes of create we are currently in. You may be wondering why I chose an enumerated type over using constants with numbers. Simply put, so we can add different modes into the mode creation process without having to renumber all the types. This will be a huge time-saver if you choose to upgrade or alter the engine in some way. Also I've found that most code nowadays doesn't include this type and thought it would be a good idea to show some of the possibilities available in the C language.

The CREATE_MODE_NULL value has a default value of 0, which indicates that we are not creating any primitives at the moment. The next value, CREATE_MODE_START, is the start value for the primitive creation stage. We would use this value when we are waiting for the user to press the left mouse button down. The CREATE_MODE_SIZE value would be set when the dimensions of the primitive are being measured using the movements of the mouse as input. The final enumerated value is CREATE_MODE_FINISH, which waits for the user to release the left mouse button to finish the creation process. Once the left mouse button is released, the mode is reset to CREATE_MODE_NULL and the process can begin again.

After creating the enumerated type, we must create two new structures to store our coordinate data. The first structure, called COORDS, contains the position data for the mouse and its relative world coordinates. We'll use this structure to store both the start and finish position information. The first variable in the structure is called mouse_x, and is of the long data type. This variable obviously stores the value of the mouse x-axis. As we discussed earlier in the book, the x-axis moves horizontally on the screen, assuming we're looking directly at the x-axis. The next variable in the structure is another long data type called mouse_y. This variable stores the y-axis value of the mouse. The y-axis, as discussed earlier, is drawn in a vertical direction, provided we haven't rotated the view at all and are looking directly at the x- or z-axes. Both mouse_x and mouse_y are important pieces of information because they contain the original coordinate set for the mouse. From these two variables we'll be able to compute the world XYZ values to transform our two-dimensional screen into a three-dimensional canvas. Although technically we could dump the variables once we've created the associated primitive, these values may come in handy for other uses, and therefore must be stored in the COORDS structure in the event we want to use them later on.

After defining the mouse X and Y positions in the COORDS structure we have three variables, all of the double data type. These variables, world_x, world_y, and world_z, will store the computed world positions for the mouse positions. In plain English, we'll record the mouse X and Y positions, calculate their world position equivalents, and store them in these three variables. These variables have to be doubles because regular integer-based variables do not have the precision needed for the coordinates to be created. Of course we could have chosen the float data type, except it's only 32 bits in size, which doesn't allow us to create huge maps. The double gives us slightly more precision than we need, but I'd rather have more precision and flexibility than less.

Each world variable represents a specific axis, e.g., world_x represents the x-axis world coordinate. Similarly, world_y identifies the y-axis and world_z the z-axis. With the definition of the world variables, we've completed defining the COORDS structure and can display the code for it here:

```
typedef struct
{
   long      mouse_x;
   long      mouse_y;

   double    world_x;
   double    world_y;
   double    world_z;
} COORDS;
```

After defining the COORDS structure, we have one final structure that we must create, called CREATION_COORDS. This structure will store both the start and finish coordinates of the mouse axis as well as the current creation mode. Thankfully, we've done most of the work in the previous structure and only need to define three variables within this new structure. The first variable in CREATION_COORDS is called mode and is of the long data type. This variable contains one of the four CREATE_MODE enumerated types we defined earlier in the chapter. As the creation mode changes, the value of mode will as well.

The final two variables in the structure are both of the COORDS structure type. The structures, start and finish, store the information about the mouse coordinates necessary to generate the primitive. Each structure will be filled in as the creation mode progresses. The start structure will be filled in first with the starting coordinates, and the finish structure will be filled in when the mouse is moved or when the left mouse button is released. We've now completed the structure definition process for this chapter and can continue on our merry little way toward creating the user interface (UI). The source code for the structure is provided below for you to study.

```
typedef struct
{
   long      mode;

   COORDS    start;
   COORDS    finish;
} CREATION_COORDS;
```

Moving right along, the next thing we must do is declare a new global variable called creation_coords, which is of the CREATION_COORDS structure type. This variable will contain all the values for the creation process. As it will be needed throughout our application, we must make it a global. Since this variable is very important, we must ensure the data is always set to the default values when the program is loaded. We'll set the default values for the variable in WinMain after the function call to SetGLDefaults, using the function memset.

Using the memset function, we'll supply the address to the creation_ coords variable as the first parameter, 0 as the second parameter, and the size of the CREATION_COORDS structure (sizeof(creation_coords)) as the third. This will set the entire contents of creation_coords to 0 for sizeof(creation_coords) amount of bytes. This function really helps in the initialization of structures because of its simplification of each variable/structure. Rather than set each variable manually to 0, we can simply set the entire contents at once. The code snippet below displays the code required to set the creation_coords variable to 0.

```
memset (&creation_coords, 0, sizeof(creation_coords));
```

After declaring the creation_coords variable and setting the default values within the structure to 0, we're ready to write the code to convert the two-dimensional mouse coordinates to three-dimensional coordinates used for rendering lines to the screen. We will call the new function ComputeMouseCoords and pass the X and Y positions of the current mouse position using the long data type for each position. The return value for the function will be of the COORDS structure type, which will store all the data we need for both 2D and 3D coordinates. To perform the calculation we'll need to declare several new variables within the function. The first variable in the function we must declare is of the COORDS structure type. This variable, called coords, will contain the mouse coordinates and the final computed world coordinates that are returned at the end of the function.

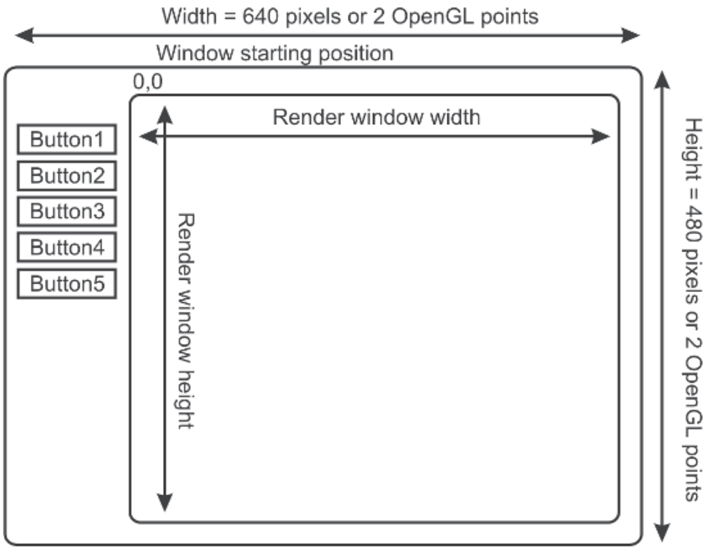
After declaring the coords variable we must declare a new variable called rect, which is of the RECT data type. This variable will store the dimensions (left, right, top, and bottom) of a window, which in our case will be the render window. We need the render window dimensions to calculate the position of the mouse based on the size of the window. The calculation will be discussed in further detail later in the book.

The next variable we'll declare is called window_width; it is of the float data type. This variable contains the floating-point representation of the render window width. We could use any data type to store the width of the render window; however, to eliminate Visual C++ warnings and data type casting throughout the calculation, we'll use floats. Following the window_width variable we have another float variable called window_height. This new variable will store the height of the render window using the floating-point representation of the value. The next two variables are both of the float data type. The variables window_start_x and window_start_y contain the window starting positions for their respective axis. The window starting positions contain the mouse positions in relation to the window when the beginning dimension is 0. In plain English, this means we'll shift the starting position of both the X and Y to 0 instead of being the value specified by the Left/Top variables within the RECT structure.

After declaring the variables in the ComputeMouseCoords function we must set the default mouse position for both the x- and y-axes. Since these two values are passed into the function, we'll simply use their original passed values and set the mouse_x and mouse_y variables within the COORDS structure to their respective positions. After setting the mouse positions, we must get the dimensions of the window by using the function GetWindowRect and supplying the handle of a window (RenderWindow) as the first parameter and the address of the RECT structure (rect), which will retrieve the values. The next step in the ComputeMouseCoords function is to calculate the width and height of the window and cast them both as floats. Using simple math, we'll subtract rect.left from rect.right to get the width, and rect.top from rect.bottom to get the height. Both result values will be cast to a float data type and stored in the appropriate variable (window_width or window_height). As stated before, this calculation will give us the width and height of the rendering window, which we'll need for the final calculation. Without those values, we wouldn't be able to plot the mouse in the appropriate place in three-dimensional space.

Following the width and height calculations, we'll calculate the starting positions for the X and Y positions of the mouse in relation to the render window. This calculation will only work with the two-dimensional coordinates. Converting the coordinates to three dimensions requires a separate calculation. To calculate the window starting position we simply subtract the rect.left variable from the coords.mouse_x variable. This will produce the window starting position for the x-axis, which must be cast as a float data type. The calculation is simple. Since the leftmost position (rect.left) of the rendering window may not be 0, we subtract that value from the current mouse position, giving us the starting location of the mouse. The y-axis is different in that we will use the straight value of coords.mouse_y instead of a calculation to figure out the starting y-axis position. When a menu is used within the program, the y-axis seems to get shifted down within the rect.top value, and therefore does not provide an accurate result. For this reason, we'll stick to regular mouse coordinates for the y-axis and move to the main calculation.

The final calculations in the ComputeMouseCoords function do the bulk of the work for positioning two-dimensional points in three-dimensional space. Once explained, the calculation is very simple to understand. The first part of the calculation is to divide window_start_x by window_width, which provides the position of the mouse as a percentage of the width. We then multiply the width percentage by the visible width of the screen in OpenGL points, which is 2, and which changes our percentage into a number based on 2.0 being the highest value or 100% of the width. The final step in the calculation is to subtract 1.0 from the value because our screen maximum is 1.0 in each direction. This will center the point if it happens to be in the middle of the screen. The final result is stored in the coords.world_x variable, which will be returned at the end of the function. The second calculation is nearly identical to the previous; however, we'll use window_start_y and window_height instead of the previous variables, and we'll also invert the final result because OpenGL draws things in the opposite direction vertically. If we didn't invert the final value, we would end up drawing the items in the proper x-axis, but the y-axis would be exactly the opposite. The following illustration shows a monitor's screen dimensions and the equation to give you an idea of how this calculation works.



```
coords.world_x   = (window_start_x / window_width) * 2.0 – 1.0;
coords.world_z   = –((window_start_y / window_height) * 2.0 –1.0);
```

Figure 5.1: Computer with render window dimensions and calculation below it

The final operation we must perform in the ComputeMouseCoords function is to return the variable coords and let the newly calculated values be of use to the program. We'll revisit this calculation throughout the development of our map editor, but for the time being we've finished writing the function! The source code for this newly created function is shown here:

```
COORDS ComputeMouseCoords(long xPos, long yPos)
{
   COORDS           coords;
   RECT             rect;

   float            window_width;
   float            window_height;
   float            window_start_x;
   float            window_start_y;

   coords.mouse_x   = xPos;
   coords.mouse_y   = yPos;

   GetWindowRect (RenderWindow, &rect);
   window_width     = (float)(rect.right - rect.left);
   window_height    = (float)(rect.bottom - rect.top);
   window_start_x   = (float)(coords.mouse_x - rect.left);
   window_start_y   = (float)coords.mouse_y;


   coords.world_x   = (window_start_x / window_width) * 2.0 - 1.0;
   coords.world_z   = -((window_start_y / window_height) * 2.0 - 1.0);

   return (coords);
}
```

Handling the WM_LBUTTONDOWN Message

Now that we've created the function to calculate the world coordinates based on the mouse coordinates, we can begin writing the underlying interface code to allow the user to click and drag the size ruler to determine the size of the primitive to be created. This process will require three new Windows messages to be handled within the message handler. The first message we'll add to the message handler is WM_LBUTTONDOWN, which is sent when the user presses the left mouse button down. Keeping with tradition, we'll create a new function called WMLButtonDown and pass all the parameters of the WndProc function to it. This will keep our message handler very simple to understand, and allow us to easily debug our applications because each event is handled in a different chunk of code and therefore can be easily targeted when problems arise. Moving right along, the new function should be placed before the WndProc definition. Although it's normally considered lazy or bad programming, we can copy and paste the parameters from the WndProc function to our new function, WMLButtonDown. There's no sense in rewriting the same parameters, unless you want to improve your typing skills, in which case you're more than welcome to retype it!

As discussed before, the principles behind our user interface are simple. When the user presses the left mouse button down, we record the coordinates and wait for the mouse to move. As the mouse moves around the screen and the left mouse button is down, we record the current mouse positions while our creation line is being drawn on the screen. Once the left mouse button is released, we record the current position of the mouse and display our newly created primitive. The figure below displays the process of creating a primitive.

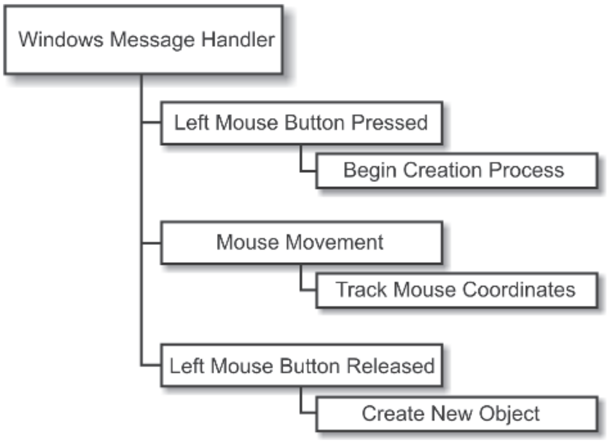


Figure 5.2: Primitive creation process

Following our simple rules, the first line of source code we'll add to the WMLButtonDown function will set the creation_coords.mode variable to CREATE_MODE_START. This will set the variable to the first stage in the creation process (the start stage). If this variable weren't set, our mouse movements wouldn't be recorded and our creation line wouldn't be drawn because the default value of creation_coords.mode is CREATE_MODE_NULL, which indicates that we are not creating any type of primitive.

After setting the creation mode, we set the creation_coords.start variable to retrieve the returned value from the ComputeMouseCoords function we created earlier. The parameters for the function will be LOWORD(lParam) (x-axis) and HIWORD(lParam) (y-axis). These functions will extract the X and Y positions of the mouse from the lParam variable that is passed to our WMLButtonDown function. The ComputeMouseCoords function is then supplied the mouse coordinates and the magic happens and returns the final result to the creation_coords.start variable. After setting the new starting position for the primitive, we must set the finish position to the starting position values. As odd as this sounds, we set the creation_coords.finish variable to the creation_coords.start value. By doing so, we'll be able to have a nice-looking click-and-drag interface that has the second point initially position itself at the start position and move from there. We've finished creating the WMLButtonDown function and can display the source code below.

```
void WMLButtonDown(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   creation_coords.mode   = CREATE_MODE_START;
   creation_coords.start  = ComputeMouseCoords(LOWORD(lParam), HIWORD(lParam));
   creation_coords.finish = creation_coords.start;
}
```

Handling the WM_LBUTTONUP Message

After writing the function to handle the WM_LBUTTONDOWN Windows message, we must create the source code to handle the WM_LBUTTONUP Windows message. The WM_LBUTTONUP message is sent when the user releases the left mouse button. We'll use this message to set the final X and Y mouse positions and compute the final world coordinates for the creation line. Similar to the WM_LBUTTONDOWN message, we'll create a new function called WMLButtonUp, which will handle the functionality of the left button release.

WMLButtonUp will contain the same parameters as our previously declared function WMLButtonDown, so we can simply copy and paste the function declaration code from the previous function to this new one. Since we've already written the source code to convert mouse coordinates to world coordinates, our example is fairly brief. Unlike the previous message, however, where we only had to worry about a couple lines, this Windows function will contain the bulk of the work for creating the objects in our map. Since there are three different types of objects in our map, we must include the map creation code for each one within this function, which will make the function fairly large.

To begin the code, the first thing we'll do when the WMLButtonUp function is called is ensure the value in the variable creation_coords.mode is not CREATE_MODE_NULL, so we know the user is creating an object and is not sitting idly. In the event the creation_coords.mode value is not equal to the value of CREATE_MODE_NULL, then we will run the source code within the if statement, which begins by setting the creation_coords.mode variable to CREATE_MODE_NULL, indicating there is no object currently being created. This will ensure that the next time the user presses the left mouse button and then releases it, we're not creating another object by mistake. After setting the variable creation mode status, we call the function ComputeMouseCoords, supplying the LOWORD(lParam) and HIWORD(lParam) macros of lParam as the parameters. This will convert the mouse coordinates, which are supplied in lParam. As mentioned in the previous section, both the X and Y coordinates for the mouse are in the one lParam variable, so we use the LOWORD/HIWORD macros to extract the X (LOWORD) and Y (HIWORD) positions. The return value for the ComputeMouseCoords function call will be stored in the creation_coords.finish variable, giving us all the information we need to create the objects. The fun part begins in actually creating the objects!

Create a Wall

Following the call to ComputeMouseCoords we'll check the value of the creation_coords.type and see if the value is equal to OBJECTTYPE_WALL. If the value is equal to OBJECTTYPE_WALL, then the current creation type is a wall and we'll perform the code within the if statement. Creating walls, or any other type of object in our map, is really simple since we wrote all the low-level code in the previous chapter! To create the new object we simply call the method map->InsertObject, supplying a NULL-terminated string as the first parameter for the name and the creation_coords.type variable as the second parameter to define the type of object we're creating.

After creating the object, we must insert the vertices for each point in the object. There will be a total of four vertices inserted, one for each point in the square wall. We'll use the method map->InsertVertex, supplying map->header.max_objects–1 as the object number to which we want to insert the vertices. Because we are inserting the values to the last object in the object list, the value of map->header.max_objects–1 is appropriate to use. The next three variables in the map->InsertVertex method are the X, Y, and Z coordinates for the vertex position. Depending on the current placement of the vertex, the X, Y, and Z coordinates will change. For this chapter, we're going to hardcode all the y-axis values to simplify the design process. In future chapters, we'll discuss how to change the height in a simple user-defined method.

The following figure displays the four points in a wall and their respective variables and values to help you understand the process of inserting vertices into an object.

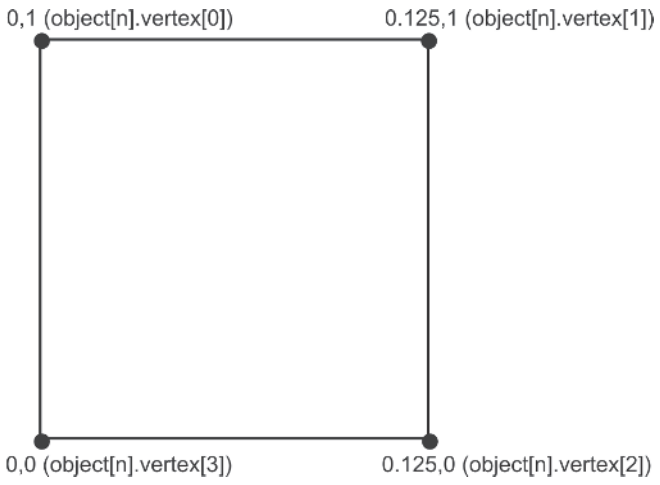

0,1 (object[n].vertex[0])
0.125,1 (object[n].vertex[1])
0,0 (object[n].vertex[3])
0.125,0 (object[n].vertex[2])

Figure 5.3: Wall with variable names and values

As you can see in Figure 5.3, we've hardcoded the starting and ending Y coordinates for our walls. The lowest point in the wall is 0.0, and the highest point is 1.0. Later in the book we'll create a function to edit the starting Y and height of each wall/floor, but for the time being we'll leave these values as the defaults and use the X/Z coordinates taken from the drag-and-drop coordinates we computed earlier.

After inserting the four user-defined vertices into the object, we must insert two triangles into the object to link the vertices we've just installed into the actual object itself. To insert triangles into the map, we'll simply use the method map->InsertTriangle. The first parameter for the method is the object number (map->header.max_objects–1) into which we want to insert the triangle. The next three parameters are the individual vertex indexes for the triangle. Since all three of our object types are square/rectangular-type objects, we can use the same formula for all three objects. The following figure displays the vertex indexes for the two triangles we're going to insert.

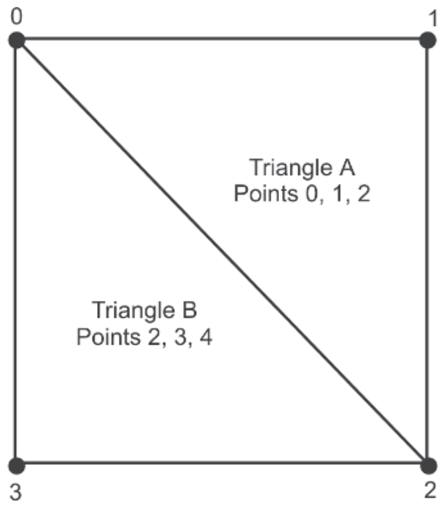


Figure 5.4: Index points

As you can see in the figure, we start at the top-left (vertex 0) and move to the top-right (vertex 1), then to the bottom-right (vertex 2). The second triangle starts at the bottom-left (vertex 3), then moves to the top-left (vertex 0), and finally moves to the bottom-right (vertex 2). The final texture tiling coordinates, more commonly known as UV coordinates in OpenGL, are represented by the number of times the texture should be tiled. In the case of a default wall, floor, or ceiling, we'll specify for the texture to be tiled once. So if the UV coordinates start at 0, the outer point would be 1. Each triangle must have its own set of UVs specified for it to draw the texture properly. The UV coordinates will follow the regular triangle points to texture the object, so our texture coordinates for Triangle A will be 0,0, 1,0, 1,1 and our UV coordinates for Triangle B will be 1,1, 0,1, and 0,0. UV coordinates and how they work will be discussed in further detail in Chapters 11 and 12. This completes the wall creation process and, more importantly, a third of the work in the WMLButtonUp function.

Create a Floor

In the event the value of creation_coords.type is not equal to OBJECTTYPE_WALL but instead is equal to OBJECTTYPE_FLOOR, then we'll obviously want to create a floor instead of a wall. Creating the floor is nearly identical to the wall creation process, with the exception that the object name will be different and the Y positions for each vertex will be at the lower point as opposed to being varied. With this in mind, we can simply copy and paste the source code from the wall creation section and modify it to implement the new functionality. In our floor creation section, we must first change the string parameter in the map->InsertObject member to "Floor" rather than the current "Wall", indicating we are creating a floor instead of a wall. After changing the string, we must change the height of each vertex to be 0 as opposed to the two values 0 and 1. Once the floor is created it will use the 0 Y coordinate, making it appropriate for the floor.

Create a Ceiling

In the event the creation_coords.type doesn't equal OBJECTTYPE_WALL or OBJECTTYPE_FLOOR, we can check to see if the variable equals OBJECTTYPE_CEILING. If the variable does equal OBJECTTYPE_ CEILING, then the code to create a ceiling must be executed. There are only two adjustments from the floor creation code to create ceilings. Rather than write the code again, simply copy and paste the source code from the floor creation section. There's no sense in rewriting the code numerous times to adjust one little parameter!

The first adjustment we'll make to the code for the ceiling creation process is to change the name of the object from "Floor" to "Ceiling", which obviously indicates that we want to create a ceiling and not a floor. The second adjustment we must make is to change the height of the vertices from 0 to 1, which is the maximum height of the walls. This adjustment should be made to all the vertices and will finalize the creation process of the ceiling.

After we've finished the code for creating the three different objects, we must reset the creation_coords.start and creation_coords.finish variables to ensure our data always begins with 0s when first called. Rather than set each variable in the two structures, we'll simply use the memset function and specify the address of the structures for the parameter in each function call, followed by the type of character we want to fill it with (0 in our case), and finally the size that we want to fill (sizeof(creation_coords.start) or sizeof(creation_coords.finish)). This completes the code for the WMLButtonUp function call, and we can display the source code below.

```
void WMLButtonUp(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (creation_coords.mode != CREATE_MODE_NULL)
   {
      creation_coords.mode      = CREATE_MODE_NULL;
      creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                          HIWORD(lParam));

      if (creation_coords.type == OBJECTTYPE_WALL)
      {
         map->InsertObject ("Wall", creation_coords.type);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.start.world_x, 1, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.finish.world_x, 1, creation_coords.finish.world_z);
```

```
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.finish.world_x, 0, creation_coords.finish.world_z);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.start.world_x, 0, creation_coords.start.world_z);
            map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
                 1.0f,0.0f, 1.0f,1.0f);
            map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
                 0.0f,1.0f, 0.0f,0.0f);
         }
         else if (creation_coords.type == OBJECTTYPE_FLOOR)
         {
            map->InsertObject ("Floor", creation_coords.type);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.start.world_x, 0, creation_coords.start.world_z);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.finish.world_x, 0, creation_coords.start.world_z);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.finish.world_x, 0, creation_coords.finish.world_z);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.start.world_x, 0, creation_coords.finish.world_z);
            map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
                 1.0f,0.0f, 1.0f,1.0f);
            map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
                 0.0f,1.0f, 0.0f,0.0f);
         }
         else if (creation_coords.type == OBJECTTYPE_CEILING)
         {
            map->InsertObject ("Ceiling", creation_coords.type);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.start.world_x, 1, creation_coords.start.world_z);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.finish.world_x, 1, creation_coords.start.world_z);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.finish.world_x, 1, creation_coords.finish.world_z);
            map->InsertVertex (map->header.max_objects-1,
                 creation_coords.start.world_x, 1, creation_coords.finish.world_z);
            map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
                 1.0f,0.0f, 1.0f,1.0f);
            map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
                 0.0f,1.0f, 0.0f,0.0f);
         }

         memset (&creation_coords.start, 0, sizeof(creation_coords.start));
         memset (&creation_coords.finish, 0, sizeof(creation_coords.finish));
      }
   }
```

Handling the WM_MOUSEMOVE Message

After writing the function to handle the WM_LBUTTONUP Windows message, we have one final message to write the source code for. The final message is called WM_MOUSEMOVE, and is sent by the program when the mouse is moved. This event will be used to record the current mouse position, convert it into world coordinates, and set the creation_coords.finish variable to the final result of the ComputeMouseCoords. This, of course, assumes that the creation_coords.mode variable is not CREATE_MODE_NULL, in which case we won't do anything.

Just like the previous two messages we've handled, WM_MOUSEMOVE will have a new function created to handle the details of the message. This will keep the main Windows message handler relatively small and the code clean. The new function, WMMouseMove, has exactly the same parameters as the previous two functions we've created so we can simply copy and paste the function declarations and alter them accordingly. This function will be different in that we'll write code to handle the event properly and display output for both the creation process and for standard messaging events.

To begin, we'll need to declare a new variable in the function called temp, which is an array of 500 characters. This variable will hold the unique output string we'll create and display at the end of the function call. After declaring the new variable, we must ensure our application is in the proper creation mode by checking the variable creation_coords.mode to see whether it is CREATE_MODE_NULL. If the value is not CREATE_MODE_NULL, then obviously we will execute the proper code. This function is especially dependent on this check because it will be the most commonly handled function since every mouse move we make will ultimately be handled here. For this reason alone, we must be very careful about writing this code because it could cause an unforeseeable error.

After checking the creation mode, we set the creation_coords.mode variable to CREATE_MODE_SIZE, indicating that we are in the sizing step of the creation process. Although this carries no significance in terms of code differences, it provides us with some structure within our application. It also allows us to add future functionality without having to rewrite huge blocks of code to accommodate the different stages of primitive creation. Similar to WMLButtonUp, we'll call the ComputeMouseCoords function, passing the LOWORD/HIWORD macro values of lParam, and store the result in the creation_coords.finish variable. This will store the current mouse coordinates in the variable, allowing us to continuously draw the creation line with no blocky movements. Keep in mind that the video card must be fast enough to draw the lines; otherwise, the creation process may become too slow due to hardware limitations. In our case, we shouldn't have any problems because we won't be designing tremendously complex levels in our map editor. A complete discussion of the map creation process could literally take up several chapters, if not entire books. We'll discuss the basics of making maps, but they won't be very intricate compared to retail games available on the market.

Unlike our previous two examples where we've cut the functionality fairly short, we'll be adding slightly more functionality to this function by placing the values of the mouse and the world coordinate equivalents into a string and drawing them in the application's title bar. We'll use the sprintf function to write the formatted location data to the string. The first two variables we'll be formatting into the string are the mouse position coordinates creation_coords.finish.mouse_x and creation_coords.finish.mouse_y. Both variables are of the int data type and can be easily formatted using the %i type field character. The final two variables we'll add to the sprintf formatting section are the world coordinates creation_coords.finish.world_x and creation_coords.finish_world_z. Since all world coordinates in the COORDS structure are doubles, we'll display up to four decimal places on the screen and truncate the rest. Although we want the highest amount of precision in our calculations, displaying more than four decimal places won't be terribly useful for the user.

In the event the creation_coords.mode variable has the value of CREATE_MODE_NULL, we'll fill the temp variable with a new string displaying the mouse X/Y positions. At the bottom of the if statement for WMMouseMove we'll simply use an else clause to build the default string. When building the default string, we'll once again use the sprintf function, supplying the temp variable as the string in which to store the data and specifying two %i type field characters. Since we aren't creating an object, the creation_coords variable does not have updated mouse coordinates and therefore cannot be used. Instead, we'll use the LOWORD/HIWORD macros to extract the proper X/Y mouse coordinates from the lParam variable.

Although it may not seem important, displaying the mouse position and world coordinates is an important function. When initially drawing a level on grid paper, we can convert the coordinates of each box into proper world coordinates, thereby giving us a good grid ratio. If we were unable to display the world/mouse coordinates, the process could take longer because we would have to eyeball the distances of walls, which may not be as accurate.

The final line in the WMMouseMove function sets the text of the title bar to the value of temp. The function SetWindowText requires two parameters for it to work properly. The first parameter is the window (Window) of which we want to set the value. This function isn't specific to just windows. It will work with all sorts of different structures like edit boxes and static windows. The second parameter of the function is a NULL-terminated string to which we want to change the window. In our case, we'll set the second parameter to the variable temp, which will set the title bar of our main window to the value of temp. This completes the discussion of the WMMouseMove function, and the source code is shown here:

```
void WMMouseMove(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
char temp[500];

   if (creation_coords.mode != CREATE_MODE_NULL)
   {
      creation_coords.mode      = CREATE_MODE_SIZE;
      creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                      HIWORD(lParam));
      sprintf (temp, "Map Editor, Mx=%i My=%i, X=%0.4f Z=%0.4f",
               creation_coords.finish.mouse_x, creation_coords.finish.mouse_y,
               creation_coords.finish.world_x, creation_coords.finish.world_z);
   }
   else sprintf (temp, "Map Editor, Mx=%i My=%i", LOWORD(lParam), HIWORD(lParam));

   SetWindowText (Window, temp);
}
```

After writing the code for the WMMouseMove function, we are ready to add the three new Windows messages into the event handler. By adding these messages into the event handler, we'll have hit a milestone in the development of the map editor, as we'll be able to actually use the map editor for something beyond a glorified shape viewer. The first event we'll add to our event handler is the WM_LBUTTONDOWN event, which is sent when the user presses the left mouse button down. When this event occurs we'll call the WMLButtonDown function and pass the hWnd, msg, wParam, and lParam variables with it, then break out of the case statement. Although it may seem rather useless to pass the msg variable, we give ourselves room for a slight amount of expansion when our application begins to take shape.

The second event we'll add to the event handler is WM_LBUTTONUP, which as you know is sent when the user releases the left mouse button. Following our similar naming pattern, we'll call the function WMLButtonUp and pass the hWnd, msg, wParam, and lParam variables, then break out of the current case statement. The final event we'll add to the event handler is the WM_MOUSEMOVE event, which handles all the mouse movements. This event will call the WMMouseMove function and once again pass the WndProc callback variables, then break out of the case statement. Unfortunately, writing the event handler code isn't terribly exciting, but it has to be done! We've now finished adding the new messages to the WndProc function and can display the source code for our WndProc message handler below.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
      case WM_SIZE: WMSize (hWnd, msg, wParam, lParam); break;
```

```
        case WM_RBUTTONUP: DisplayPopupMenu(); break;
        case WM_LBUTTONDOWN: WMLButtonDown (hWnd, msg, wParam, lParam); break;
        case WM_LBUTTONUP: WMLButtonUp (hWnd, msg, wParam, lParam); break;
        case WM_MOUSEMOVE: WMMouseMove (hWnd, msg, wParam, lParam); break;
    }
    return (DefWindowProc(hWnd, msg, wParam, lParam));
}
```

The New Globals

In addition to declaring the creation_coords global variable, we must declare two new windows called bCreateFloor and bCreateCeiling. As you may have guessed, these new windows (HWND) will be created using the BUTTON class, giving us two new buttons to create floors and ceilings. We will be writing the functionality to create floors, ceilings, and walls in this section, which is a major milestone in itself.

This is an exciting section because we actually begin writing fun (the interface) source code.

Modifying the Rendering Function

Unlike the previous section where we merely drew simple shapes, here we're writing the interface for our map editor and therefore should display the objects we're creating. For this simple reason, we're going to add two new sections to our Render function; the first section will draw the finished objects in our map, and the second section will draw the object that is currently being created. Both sections will draw using a basic wireframe drawing style. In case you're not familiar with wireframe rendering, the term *wireframe* refers to drawing an object's connecting vertices using lines instead of using solid color. This will make the object appear as a mesh as opposed to a filled-in object. Although drawing objects in solid color is useful, we only need a bare-bones mechanism for drawing the objects in wireframe, since it doesn't include the bells and whistles. Because wireframe rendering is generally considered the simplest form of rendering data, we'll implement it first, then proceed to writing other rendering types later. Please note as well that wireframe rendering of our maps will be specific to our map editor and will not be available as a rendering feature in our game engine itself. Although I've probably just sold you on using the wireframe rendering in our map editor, in most cases we'll never need to draw the wireframe of a map in our game because the code for drawing in solid/textured modes will look the same in the map editor and the game itself.

The first addition to the Render function will render the map in wireframe mode, which is the default rendering mode in the map editor. When drawing our map, regardless of whether it is wireframe, solid, or textured, we must

always ensure that there is at least one object in the map. A simple if statement checking the value of map->header.max_objects will determine how many objects are currently in the map. If the value is greater than 0, we'll continue through the rendering process. If the value is less than or equal to 0, we won't bother to render the map since there isn't anything there! After ensuring there are objects in the map, we must loop through the entire list of objects from 0 to the value less than map->header.max_objects. With each iteration of the loop we'll draw the wireframe object itself. As discussed in Chapter 2, when drawing objects in OpenGL we use the glBegin/glEnd pair and supply the object type as the single parameter in glBegin before supplying the vertex data. This will tell the video card how to assemble the vertex data, and ultimately how to draw the object itself. Since we are drawing the objects in wireframe mode, the GL_QUADS/GL_
POLYGON modes will not be suitable choices for glBegin because they both create filled objects. Instead we'll use the GL_LINE_LOOP mode, which will connect each vertex together to form a huge multipoint line.

Between the glBegin/glEnd pair we must loop through the entire list of vertices (from 0 to the value of map->object[i].max_vertices), supplying the vertex data for the object using the function glVertex2d. As discussed in Chapter 2, glVertex has many different syntaxes available to the programmer, giving great flexibility when inputting vertices. Since our view of the map is limited to a top view, we only need to display the x- and z-axes because the y-axis deals specifically with the height of an object, which is not important in this case. For this reason we are going to use the glVertex2 syntax rather than glVertex3 because there is no need for the third parameter and the glVertex2 syntax will put a default value of 1.0 in its position. The "d" in glVertex2d represents the data type of the parameters in the function, which in our case is a double.

When we input the vertex data, we use the x-axis (map->object[i]
.vertex[i2].xyz[0]) as the first parameter and the z-axis (map->object[i]
.vertex[i2].xyz[2]) as the second parameter, which will give us the proper wireframe rendering.

Once the loop is finished inputting the vertex coordinates, glEnd is called, which will finish the creation of the object. The following source code will ensure there are objects in the map, then render everything using the GL_LINE_LOOP drawing mode.

```
if (map->header.max_objects > 0)
 {
 for (long i = 0; i < map->header.max_objects; i++)
    {
    glBegin (GL_LINE_LOOP);

    for (long i2 = 0; i2 < map->object
    [i].max_vertices; i2++) glVertex2d (map->object[i].vertex[i2].xyz[0],
          map->object[i].vertex[i2].xyz[2]);

    glEnd();
    }
}
```

This new source code will be added to the bottom of the program, just before the call to glPopMatrix, to ensure this is the last thing being rendered before we restore the previous matrices. The first thing we must write is an if statement to check the value of the variable creation_coords.type and ensure it is equal to the value of OBJECTTYPE_WALL. If the variable is equal to OBJECTTYPE_WALL, we'll simply draw a line from the first start coordinate to the finish coordinate. Since we've already established that we want to draw a line from one point to another we'll use the constant GL_LINES as the parameter for glBegin. When we specify the GL_LINES parameter in the glBegin function, each group of two vertices will be drawn as a line. This actually allows us to draw multiple lines at once, instead of specifying multiple glBegin/glEnd pairs. Of course in this example we only need one line so we don't need to worry about these other details, but they are nice to know, just in case.

When specifying the vertices for our line, we'll once again use the glVertex2d function. This time, however, we'll use the values stored in the creation_coords.start and creation_coords.finish variables. Keeping with our "top down" view, we'll specify the X and Z coordinates for both the start and finish points. As discussed earlier, we don't need to display the height, and so we can easily map out the levels by simply drawing the z-axis as the depth of the level and the x-axis as the width. This will produce a top view without having to use any complicated math, which is good! Getting back to the line drawing, the first call to glVertex2d will use the parameters creation_coords.start.world_x and creation_coords.start.world_z to create the starting vertex. The second vertex (finish vertex) will use the parameters creation_coords.finish.world_x and creation_coords.finish.world_z. Notice that the only differences between the two calls to glVertex2d are the "start" and "finish" keywords, which contain the appropriate values for the creation of the line (wall).

If the event creation_coords.type has a value of OBJECTTYPE_FLOOR or OBJECTTYPE_CEILING, we'll draw a specific square using the values in the creation_coords.start and creation_coords.finish structures. Unlike drawing the wall, where we could use GL_LINES as the parameter for glBegin, it's easier to use the mode GL_LINE_LOOP, which will draw a line with multiple points. Obviously this is to our advantage because we only need to specify the four glVertex2d calls rather than the eight calls that would be needed for the equivalent functionality using GL_LINES. Using the coordinates in the creation_coords.start and creation_coords.finish structures, we'll simply use the x- and z-axis values to specify the boundary box. We'll start by using the creation_coords.start.world_x and creation_coords.start.world_z variables, then move to the X boundary by using the creation_coords.finish.world_x variable, move to the Z boundary by using creation_coords.finish.world_z, and finally use the boundary finish z variable with the initial X starting position creation_coords.start.world_x to finish the loop through the bounding coordinates. The following illustration displays the process of drawing the boundary area of the object.

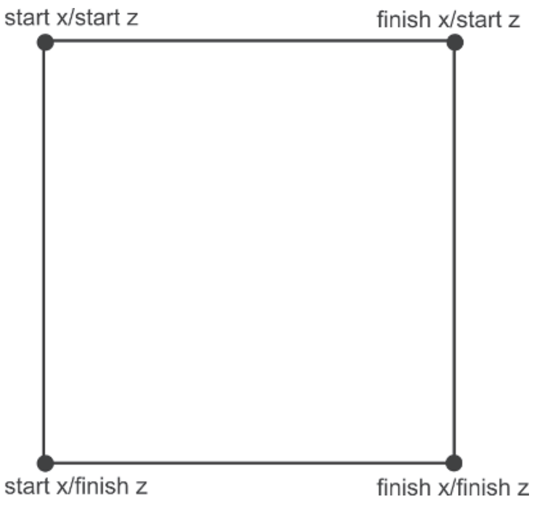


Figure 5.5: Boundary area for floors/ceilings

As you can see in Figure 5.5, the process of rendering the boundary area is really simple. In the event the creation_coords.type doesn't equal any of the three types we've defined, then there is no object being created and we can skip over this section completely! The structures discussed in this section will be continually modified when the user is dragging the mouse to the appropriate sizes for their map designs. There shouldn't be any speed concerns when dragging the items unless the user has a slow video card or the video card doesn't have OpenGL drivers. There will be a slowdown when dragging the items around if there are tens of thousands of objects, but in that sort of instance an optimized map editor would be recommended and not a simplistic one made with less than 1,500 lines!

The source code for the object creation rendering is shown here:

```
if (creation_coords.type == OBJECTTYPE_WALL)
{
   glBegin (GL_LINES);
      glVertex2d (creation_coords.start.world_x, creation_coords.start.world_z);
      glVertex2d (creation_coords.finish.world_x, creation_coords.finish.world_z);
   glEnd();
}
```

```
else if (creation_coords.type == OBJECTTYPE_FLOOR || creation_coords.type ==
         OBJECTTYPE_CEILING)
{
   glBegin (GL_LINE_LOOP);
      glVertex2d (creation_coords.start.world_x, creation_coords.start.world_z);
      glVertex2d (creation_coords.finish.world_x, creation_coords.start.world_z);
      glVertex2d (creation_coords.finish.world_x, creation_coords.finish.world_z);
      glVertex2d (creation_coords.start.world_x, creation_coords.finish.world_z);
   glEnd();
}
```

Creating the New Buttons

In order to write the interface for our map editor we'll need to have the capability to create more than one type of object, whether it is a wall, floor, or ceiling. For this reason we declare two new windows called bCreateFloor and bCreateCeiling, both of which will be buttons creating their respective object type. Before we can write the code to create the objects, we must write the code to create the windows themselves. Since there is little difference in the settings between our two new buttons and our already created bCreateWall, we can simply copy and paste the CreateWindow function call in WinMain and modify it according to the specifications needed for two new variables.

Rather than return the value of each CreateWindow function call to the bCreateWall variable, we'll pass the returned data to the two newly created variables. Obviously, when we create buttons we'll change the second parameter of the CreateWindow function, which is the name of the window. We don't want three windows that all read "Create Wall," since this would be a bad design decision and just plain stupid! The two new titles will be "Create Floor" and "Create Ceiling."

The other parameter in the CreateWindow function calls that we'll change is the fifth parameter, the Y position of the button. If we left this parameter the same for each button, they would all overlap, which would make it very difficult for the user to select a specific type of object to create. We'll simply use the base starting position of 100, then add the value of DEFAULT_BUTTON_HEIGHT multiplied by 2 and by 4 to spread the button Y positions apart. The source code for the button creation function is provided below for you to study.

```
bCreateCeiling = CreateWindow("BUTTON", "Create Ceiling", WS_CHILD |
           WS_VISIBLE, 0, 100+(DEFAULT_BUTTON_HEIGHT*2), DEFAULT_BUTTON_WIDTH,
           DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);

bCreateFloor = CreateWindow("BUTTON", "Create Floor", WS_CHILD |
           WS_VISIBLE, 0, 100+(DEFAULT_BUTTON_HEIGHT*4), DEFAULT_BUTTON_WIDTH,
           DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);
```

Checking for Button Clicks

After creating the two new buttons, it's time to spice them up with actual functionality. As discussed in Chapter 1, the Windows message WM_COMMAND handles all button presses and menu item clicks. In Chapter 1 we created a function called WMCommand, which WM_COMMAND pointed to for handling all the messages it received. As I mentioned before, WM_COMMAND handles a large majority of the work within a Windows program. It's better to have a separate function to handle these messages because the code can become very large and unmanageable at times when directly in the WM_COMMAND case clause. This section will show how easily the WMCommand function can fill up when adding simple functionality to a Windows program.

Since we've already written the handling code for the bCreateWall button, this is logically (at least in my opinion) the best place to start. Replacing the MessageBox we display when the button is pressed, the first thing we'll do is set the type of object we are going to be creating, which in our case is OBJECTTYPE_WALL. After setting the type of object we are going to create, we're going to change the names of the three buttons to indicate that the wall is selected and the other two buttons are not. Of course there are many ways in which this could be accomplished, but we'll simply use the SetWindowText function, supplying one of the three windows (bCreateWall, bCreateFloor, or bCreateCeiling) and a string. In the case of selecting a wall we'll set the text of bCreateWall window to "*Wall*" to indicate this object is selected, while we set the other two windows, bCreateCeiling and bCreateFloor, to their normal names ("Create Ceiling" and "Create Floor"). The code for selecting bCreateFloor and bCreateCeiling uses the same concept as bCreateWall except we'll substitute the creation type to either OBJECTTYPE_FLOOR or OBJECTTYPE_CEILING and change which window has the selected title appropriately. This process was the easiest way I could think of to select a creation object. I was toying with the idea of having a box that would surround the object but gave up on that idea because that would involve declaring/creating a new window, then strategically placing the window when the user clicks a different button. This would ultimately make things unnecessarily complicated.

After adding the code to select an object creation type, our source code in WMCommand has increased by about 20 more lines. See how the WMCommand code can increase in size quickly! This source code sets the object creation type when selecting the different buttons:

```
if (lParam == (LPARAM)bCreateWall)
{
creation_coords.type = OBJECTTYPE_WALL;
SetWindowText (bCreateWall, "*Wall*");
SetWindowText (bCreateCeiling, "Create Ceiling");
SetWindowText (bCreateFloor, "Create Floor");
}
else if (lParam == (LPARAM)bCreateFloor)
{
creation_coords.type = OBJECTTYPE_FLOOR;
SetWindowText (bCreateFloor, "*Floor*");
SetWindowText (bCreateCeiling, "Create Ceiling");
SetWindowText (bCreateWall, "Create Wall");
}
else if (lParam == (LPARAM)bCreateCeiling)
{
creation_coords.type = OBJECTTYPE_CEILING;
SetWindowText (bCreateCeiling, "*Ceiling*");
SetWindowText (bCreateFloor, "Create Floor");
SetWindowText (bCreateWall, "Create Wall");
}
```

Chapter Example

The source code for the chapter has been provided below for you to study.

ex5_1.cpp

```
/* Headers
*************************************************************************/
#include <windows.h>
#include <winbase.h>
#include <stdio.h>

#include "resource.h"

#include "raster.h"
#include "map.h"


/* Constants
*************************************************************************/
#define DEFAULT_BUTTON_WIDTH    100
#define DEFAULT_BUTTON_HEIGHT   20


/* Enumerated Type
*************************************************************************/
enum { CREATE_MODE_NULL = 0, CREATE_MODE_START, CREATE_MODE_SIZE,
       CREATE_MODE_FINISH };


/* Structures
*************************************************************************/
typedef struct
{
   long      mouse_x;
   long      mouse_y;
   double    world_x;
   double    world_y;
   double    world_z;
} COORDS;


typedef struct
{
   long      mode;
   long      type;

   COORDS    start;
   COORDS    finish;
} CREATION_COORDS;


/* Global Variables
**********************************************************************/
HINSTANCE           GlobalInstance;
HMENU               Menu;
HMENU               PopupMenu;
HWND                Window;
HWND                RenderWindow;
HWND                bCreateWall;
HWND                bCreateFloor;
HWND                bCreateCeiling;
RASTER              raster;

CREATION_COORDS     creation_coords;
MAP                 *map = new MAP;


/* ResizeGLWindow
***********************************************************************/
void ResizeGLWindow(long width, long height)
{
   glViewport(0, 0, (GLsizei) width, (GLsizei) height);
   glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(-200,200, -200,-200, -2000,2000);
   glMatrixMode(GL_MODELVIEW);
}


/* SetGLDefaults
***********************************************************************/
void SetGLDefaults()
{
   glEnable (GL_DEPTH_TEST);
   glDisable (GL_CULL_FACE);

   glClearColor (0.6f, 0.6f, 0.6f, 1.0f);
```

```
}


/* Render ***********************************************************************/
void Render()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glLoadIdentity();
   glPushMatrix();
      glTranslatef (0.0f, 0.0f, 0.0f);


      glBegin (GL_TRIANGLES);
         glVertex3f (0.0f, 0.0f, 0.0f);
         glVertex3f (0.0f, 1.0f, 0.0f);
         glVertex3f (1.0f, 1.0f, 0.0f);
      glEnd();


      glBegin (GL_QUADS);
         glVertex3f (0.05f, -0.05f, 0.0f);
         glVertex3f (0.95f, -0.05f, 0.0f);
         glVertex3f (0.95f, -0.95f, 0.0f);
         glVertex3f (0.05f, -0.95f, 0.0f);
      glEnd();


      glBegin (GL_POLYGON);
         glVertex3f (-0.25f, -0.25f, 0.0f);
         glVertex3f (-0.50f, -0.125f, 0.0f);
         glVertex3f (-0.75f, -0.25f, 0.0f);
         glVertex3f (-0.875, -0.5f, 0.0f);
         glVertex3f (-0.75f, -0.75f, 0.0f);
         glVertex3f (-0.50f, -0.875f, 0.0f);
         glVertex3f (-0.25f, -0.75f, 0.0f);
         glVertex3f (-0.125f, -0.5f, 0.0f);
      glEnd();


      glBegin (GL_LINES);
         glVertex3f (-0.25f, 0.25f, 0.0f);
         glVertex3f (-0.75f, 0.75f, 0.0f);
      glEnd();


      if (map->header.max_objects > 0)
      {
         for (long i = 0; i < map->header.max_objects; i++)
         {
            glBegin (GL_LINE_LOOP);
```

```
                for (long i2 = 0; i2 < map->object[i].max_vertices; i2++)
                    glVertex2d (map->object[i].vertex[i2].xyz[0],
                    map->object[i].vertex[i2].xyz[2]);
            glEnd();
          }
        }



        if (creation_coords.type == OBJECTTYPE_WALL)
        {
          glBegin (GL_LINES);
            glVertex2d (creation_coords.start.world_x,
                 creation_coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x,
                 creation_coords.finish.world_z);
          glEnd();
        }
        else if (creation_coords.type == OBJECTTYPE_FLOOR || creation_coords.type ==
                OBJECTTYPE_CEILING)
        {
          glBegin (GL_LINE_LOOP);
            glVertex2d (creation_coords.start.world_x,
                 creation_coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x,
                 creation_coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x,
                 creation_coords.finish.world_z);
            glVertex2d (creation_coords.start.world_x,
                 creation_coords.finish.world_z);
          glEnd();
        }



    glPopMatrix();


    SwapBuffers (raster.hDC);
  }


  /* MapDetailsDlgProc ***********************************************************/
  LRESULT CALLBACK MapDetailsDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
                                     LPARAM lParam)
  {
    switch (msg)
    {
      case WM_INITDIALOG:
      {
        SetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, "Map Name");
```

```
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING, 0,
                                (LPARAM)"Erase Me");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_RESETCONTENT,
                                 0, 0);
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING, 0,
                                (LPARAM)"Exit");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING, 0,
                                (LPARAM)"Get Fragged");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_SETCURSEL,
                                 0, 1);

            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING, 0,
                                (LPARAM)"Erase Me");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_RESETCONTENT,
                                 0, 0);
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING, 0,
                                (LPARAM)"Single Player");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING, 0,
                                (LPARAM)"Multi Player");
            SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_SETCURSEL,
                                 0, 1);
         } break;

         case WM_COMMAND:
         {
            if (wParam == IDOK)
            {
               long level_rule = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_
                         RULES, LB_GETCURSEL, 0, 0);
               long level_type = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_
                         TYPE, CB_GETCURSEL, 0, 0);

               char temp[500];

               sprintf (temp, "Level Type: %i\r\nLevel Rule: %i\r\nOk Button!",
                        level_type, level_rule);
               MessageBox (hWnd, temp, "Ok", MB_OK);

               EndDialog (hWnd, 0);
            }
            else if (wParam == IDCANCEL)
            {
               MessageBox (hWnd, "Cancel Button!", "Cancel", MB_OK);
               EndDialog (hWnd, 0);
            }
         } break;
      }
      return (0);
   }
```

```
/* WMCommand ********************************************************************/
void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (lParam == (LPARAM)bCreateWall)
   {
      creation_coords.type = OBJECTTYPE_WALL;
      SetWindowText (bCreateWall, "*Wall*");
      SetWindowText (bCreateCeiling, "Create Ceiling");
      SetWindowText (bCreateFloor, "Create Floor");
   }
   else if (lParam == (LPARAM)bCreateFloor)
   {
      creation_coords.type = OBJECTTYPE_FLOOR;
      SetWindowText (bCreateFloor, "*Floor*");
      SetWindowText (bCreateCeiling, "Create Ceiling");
      SetWindowText (bCreateWall, "Create Wall");
   }
   else if (lParam == (LPARAM)bCreateCeiling)
   {
      creation_coords.type = OBJECTTYPE_CEILING;
      SetWindowText (bCreateCeiling, "*Ceiling*");
      SetWindowText (bCreateFloor, "Create Floor");
      SetWindowText (bCreateWall, "Create Wall");
   }
   else if (wParam == ID_FILE_EXIT) PostQuitMessage(0);
   else if (wParam == ID_DRAWING_WIREFRAME)
   {
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);
   }
   else if (wParam == ID_DRAWING_SOLID)
   {
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_UNCHECKED);
   }
   else if (wParam == ID_MAP_DETAILS) DialogBox (GlobalInstance,
            MAKEINTRESOURCE(IDD_MAP_DETAILS), NULL, (DLGPROC)MapDetailsDlgProc);


   // Popup Menu Items
   else if (wParam == ID_POPUP_MOVE) MessageBox (Window, "Move", "Click", MB_OK);
   else if (wParam == ID_POPUP_DELETE) MessageBox (Window, "Delete", "Click",
            MB_OK);
   else if (wParam == ID_POPUP_TEXTURE) MessageBox (Window, "Texture", "Click",
            MB_OK);
   else if (wParam == ID_POPUP_DUPLICATE) MessageBox (Window, "Duplicate",
            "Click", MB_OK);
}


/* DisplayPopupMenu *************************************************************/
void DisplayPopupMenu()
{
   HMENU temp = GetSubMenu(PopupMenu, 0);
   POINT point;
   GetCursorPos (&point);
   TrackPopupMenu(temp, TPM_LEFTALIGN|TPM_RIGHTBUTTON, point.x, point.y, 0,
            Window, NULL);
}


/* WMSize ************************************************************************/
void WMSize(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   RECT rect;

   GetClientRect (Window, &rect);
   MoveWindow (RenderWindow, DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-
            DEFAULT_BUTTON_WIDTH, rect.bottom-rect.top, true);

   GetClientRect (RenderWindow, &rect);
   ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);
}


/* ComputeMouseCoords ************************************************************/
COORDS              ComputeMouseCoords(long xPos, long yPos)
{
   COORDS           coords;
   RECT             rect;

   float            window_width;
   float            window_height;
   float            window_start_x;
   float            window_start_y;


   coords.mouse_x   = xPos;
   coords.mouse_y   = yPos;


   GetWindowRect (RenderWindow, &rect);
   window_width     = (float)(rect.right - rect.left);
   window_height    = (float)(rect.bottom - rect.top);
   window_start_x   = (float)(coords.mouse_x - rect.left);
   window_start_y   = (float)coords.mouse_y;


   coords.world_x   = (window_start_x / window_width) * 2.0 - 1.0;
   coords.world_z   = -((window_start_y / window_height) * 2.0 - 1.0);

   return (coords);
}
```

```
/* WMLButtonDown ***********************************************************/
void WMLButtonDown(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   creation_coords.mode   = CREATE_MODE_START;
   creation_coords.start  = ComputeMouseCoords(LOWORD(lParam), HIWORD(lParam));
   creation_coords.finish = creation_coords.start;
}


/* WMLButtonUp *************************************************************/
void WMLButtonUp(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (creation_coords.mode != CREATE_MODE_NULL)
   {
      creation_coords.mode      = CREATE_MODE_NULL;
      creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                         HIWORD(lParam));

      if (creation_coords.type == OBJECTTYPE_WALL)
      {
         map->InsertObject ("Wall", creation_coords.type);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.start.world_x, 1, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.finish.world_x, 1, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.finish.world_x, 0, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.start.world_x, 0, creation_coords.start.world_z);
         map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
              1.0f,0.0f, 1.0f,1.0f);
         map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
              0.0f,1.0f, 0.0f,0.0f);
      }
      else if (creation_coords.type == OBJECTTYPE_FLOOR)
      {
         map->InsertObject ("Floor", creation_coords.type);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.start.world_x, 0, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.finish.world_x, 0, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.finish.world_x, 0, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1,
              creation_coords.start.world_x, 0, creation_coords.finish.world_z);
         map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
              1.0f,0.0f, 1.0f,1.0f);
         map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
              0.0f,1.0f, 0.0f,0.0f);
      }
      else if (creation_coords.type == OBJECTTYPE_CEILING)
      {
         map->InsertObject ("Ceiling", creation_coords.type);
```

```
          map->InsertVertex (map->header.max_objects-1,
               creation_coords.start.world_x, 1, creation_coords.start.world_z);
          map->InsertVertex (map->header.max_objects-1,
               creation_coords.finish.world_x, 1, creation_coords.start.world_z);
          map->InsertVertex (map->header.max_objects-1,
               creation_coords.finish.world_x, 1, creation_coords.finish.world_z);
          map->InsertVertex (map->header.max_objects-1,
               creation_coords.start.world_x, 1, creation_coords.finish.world_z);
          map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
               1.0f,0.0f, 1.0f,1.0f);
          map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
               0.0f,1.0f, 0.0f,0.0f);
       }

      memset (&creation_coords.start, 0, sizeof(creation_coords.start));
      memset (&creation_coords.finish, 0, sizeof(creation_coords.finish));
   }
}


/* WMMouseMove ***************************************************************/
void WMMouseMove(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   char temp[500];

   if (creation_coords.mode != CREATE_MODE_NULL)
   {
      creation_coords.mode      = CREATE_MODE_SIZE;
      creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                          HIWORD(lParam));
      sprintf (temp, "Map Editor, Mx=%i My=%i, X=%0.4f Z=%0.4f",
               creation_coords.finish.mouse_x, creation_coords.finish.mouse_y,
               creation_coords.finish.world_x, creation_coords.finish.world_z);
   }
   else sprintf (temp, "Map Editor, Mx=%i My=%i", LOWORD(lParam), HIWORD(lParam));

   SetWindowText (Window, temp);
}


/* WndProc *******************************************************************/
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
      case WM_SIZE: WMSize (hWnd, msg, wParam, lParam); break;
      case WM_RBUTTONUP: DisplayPopupMenu(LOWORD(lParam), HIWORD(lParam)); break;
      case WM_LBUTTONDOWN: WMLButtonDown (hWnd, msg, wParam, lParam); break;
      case WM_LBUTTONUP: WMLButtonUp (hWnd, msg, wParam, lParam); break;
      case WM_MOUSEMOVE: WMMouseMove (hWnd, msg, wParam, lParam); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}


/* WinMain ******************************************************************/
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
           int CmdShow)
{
   WNDCLASS               wc;
   MSG                    msg;
   RECT                   rect;

   GlobalInstance         = hInstance;

   wc.cbClsExtra          = 0;
   wc.cbWndExtra          = 0;
   wc.hbrBackground       = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor             = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon               = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance           = hInstance;
   wc.lpfnWndProc         = WndProc;
   wc.lpszClassName       = "ME";
   wc.lpszMenuName        = NULL;
   wc.style               = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL,"Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);
   }

   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
         0, 0, 640, 480, NULL, NULL, hInstance, NULL);

   if (Window == NULL)
   {
      MessageBox (NULL,"Error: Failed to Create Window","ERROR!",MB_OK);
      return (0);
   }
   GetClientRect (Window, &rect);


   bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE,
         0, 100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window,
         NULL, hInstance, NULL);

   bCreateCeiling = CreateWindow("BUTTON", "Create Ceiling", WS_CHILD |
         WS_VISIBLE, 0, 100+(DEFAULT_BUTTON_HEIGHT*2), DEFAULT_BUTTON_WIDTH,
         DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);
```

```
    bCreateFloor = CreateWindow("BUTTON", "Create Floor", WS_CHILD | WS_VISIBLE,
              0, 100+(DEFAULT_BUTTON_HEIGHT*4), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);


    RenderWindow= CreateWindow("STATIC", NULL, WS_CHILD | WS_VISIBLE | WS_BORDER,
              DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-DEFAULT_BUTTON_WIDTH,
              rect.bottom-rect.top, Window, NULL, hInstance, NULL);


    Menu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_MENU));
    SetMenu (Window, Menu);

    PopupMenu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_POPUP_MENU));


    if (!raster.Init(RenderWindow)) return (0);

    GetClientRect (RenderWindow, &rect);
    ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);

    SetGLDefaults();

    memset (&creation_coords, 0, sizeof(creation_coords));

    while (1)
    {
       Render();

       if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
       {
          if (msg.message == WM_QUIT) break;
          TranslateMessage(&msg);
          DispatchMessage (&msg);
       }
    }

    raster.Release(RenderWindow);
    delete map;


    return (1);
 }
```

Conclusion

In this chapter, we've finally begun to see the fruits of our labor. Of course things will be much sweeter once we add more functionality into the map editor, but we went from a do-nothing map editor in Chapter 4 to a functional map editor that allows you to insert walls, floors, and ceilings! This chapter moved through a lot of code, but it's finally over, and we can progress to some slightly easier topics!

Chapter 6

Drawing Primitives

In this chapter we'll be revising our previously written Render function to accommodate two types of rendering (wireframe and solid color) as well as adding support for layering. If you are unfamiliar with the term *layering*, many computer-aided design (CAD) and geographic information systems (GIS) software packages use layering to selectively draw objects. This chapter will be fairly short compared to previous chapters because we'll be focusing on some specific functionality instead of adding several generalized functions at once. Without further ado, let's begin!

Adding Layers

We'll be implementing the layering functionality to allow the user to select which type of objects to render, whether they are walls, floors, or ceilings. Because we draw both our ceilings and floors on a flat plane, we have no way of easily distinguishing which objects (floors, ceilings) are of which type. If we implement layering, we can simplify which object types we want drawn to aid the map design process.

To begin, the first thing we'll do is add a new menu to our menu resource. As discussed in earlier chapters, to add a menu to a resource script, simply double-click the script file in the project workspace, then double-click the menu resource. Once the menu resource is displayed on the screen, simply double-click one of the free spaces on the menu. When the properties dialog is displayed, we'll enter the name as **Layers** and make sure the Pop-Up option is checked. If you want to get really creative you can also put an ampersand before the L in Layers to make the name have an Alt+L hot key. When you've finished entering the name, click the "X" button to exit the dialog box.

Now that we've created the new Layers menu, we can insert three menu items. To insert a new menu item, simply click the left mouse button on the Layers menu. The menu should display with an empty space as the only menu item. Double-click the empty space to bring up the properties of this item. When the dialog box opens, type the name **Floor** in the caption section, and make sure that the Checked option is checked and the Pop-Up option is not. Once these values are set up properly, click the "X" button to exit the dialog box and finalize the settings. The menu item will be created with a default ID value of ID_LAYERS_FLOOR, indicating that the item is located in the LAYERS menu and the item is FLOOR.

We'll repeat this process two more times using the captions **Ceiling** and **Wall** for each. If they're all created properly, we'll have three menu items in the Layers menu, all of which have check marks next to their captions. The IDs for the three different layer types are ID_LAYERS_FLOOR (for floors), ID_LAYERS_CEILING (for ceilings), and ID_LAYERS_WALL (for walls).

Moving back to the code within Visual C++, we must define a new structure called LAYER, which, as if you couldn't guess, stores the different values for layering. The first variable we'll define within the structure is called draw_floor, and is of the bool data type. This variable will contain the Boolean (true/false) value for whether or not to draw the floor. Simple logic, right? The next two variables are both of the bool data type. The variables draw_ceiling and draw_wall control whether their given object types are drawn using the Boolean values. By default, these values will all be set to true, enabling the users to customize what they want once they've loaded the program and can see that they work properly. If we set these variables to false by default, they wouldn't draw when the user initially starts the map editor, which could be confusing to the user. The structure definition source code is provided below for you to view.

```
typedef struct
{
   bool draw_floor;
   bool draw_ceiling;
   bool draw_wall;
} LAYER;
```

After defining the LAYER structure we must declare a new variable called layer, which is of the LAYER data type. We'll use this structure to control which objects are drawn onto the screen, whether it's in wireframe or solid rendering mode. Although it may be useful to save the layering information to a configuration file of some sort, it's not required since we only have three layers at the moment. The time and effort in writing/testing the code to load the layering information from a file would outweigh the benefits of doing so. Other pieces of data would be much more beneficial to write to some sort of a configuration file.

Because the values in the LAYER structure correspond to real-world map editor functionality, we must set a default value for each member variable of LAYER in WinMain to ensure nothing out of the ordinary happens. Rather than specify each variable (e.g., layer.draw_floor = true), we'll simply use the standard C language function memset to set the common value for each member variable. We'll place the new call to memset right after our previous call to memset, using the address of the variable creation_coords. The first parameter of the memset function is the address of the layer variable. The next parameter is the value we want to fill the structure with, which in our case is true, but we'll use the value 1 since it's the same thing! The final parameter is the size we want to fill, for which we'll use the C language function sizeof, supplying the variable layer as the parameter. Every member variable within the layer variable will be automatically set to true, which means the three different layers will all draw. In the event any of the variables are false, we simply don't draw the object type! The code for filling the layer structure with a true value (1) is provided here:

```
memset (&layer, 1, sizeof(layer));
```

Now that we've created the appropriate menus and structures and initialized them for proper usage, we can begin implementing the functionality of the menu items. Unfortunately, we couldn't begin discussing the implementation of the menu functionality without creating the LAYER structure. The basic process for each button click is as follows: The user first clicks the menu item (for instance, Floor), we check the associated layer value (e.g., layer.draw_floor) to see if it's true, then we uncheck the appropriate Layers menu item (e.g., Floor) using the function CheckMenuItem. In the event the value of the layer member variable is not true, then we must check the menu item pressed using the function CheckMenuItem. After changing the menu item state, we simply invert the layer member variable and we are finished implementing the menu item click for an individual menu item.

To break down the process in plain English, when the user presses one of the three Layers menu items, the WMCommand is sent. Assuming we clicked the Floor menu item, the Floor item will be identified as ID_LAYER_FLOOR and we proceed to the code itself. The first line of code will be a separate if statement checking whether the value of layer.draw_floor is true. If the value is true, we use the function CheckMenuItem, supplying the menu variable we want to change (Menu) as the first parameter, the ID of the menu item (ID_LAYERS_FLOOR) as the second parameter, and the state of the item (MF_UNCHECKED) for the third parameter. By default, all three items are checked and their values are set to true, which would indicate they are checked as well. When the user presses the menu item when the item is checked, we do the opposite function for checking. For instance, if the item weren't checked we would check the item; otherwise we would uncheck the item.

In the event the value of layer.draw_floor is false, which would constitute the else portion of the if statement, we would write the same CheckMenuItem function call with the exception of the menu state, which would be changed to MF_CHECKED to indicate we want the item to be checked instead of unchecked. The final line of code will invert the current member layer to its opposite value (i.e., layer.draw_floor was true and now it's false). This completes the code necessary for one Layers menu item click. Rather than write the code for the other two menu item clicks, we can simply copy and paste the code and modify it to suit our purposes. The source code for the WMCommand modification is provided below.

```
else if (wParam == ID_LAYERS_FLOOR)
{
   if (layer.draw_floor) CheckMenuItem (Menu, ID_LAYERS_FLOOR, MF_UNCHECKED);
   else CheckMenuItem (Menu, ID_LAYERS_FLOOR, MF_CHECKED);
   layer.draw_floor = !layer.draw_floor;
}
else if (wParam == ID_LAYERS_CEILING)
{
   if (layer.draw_ceiling) CheckMenuItem (Menu, ID_LAYERS_CEILING, MF_UNCHECKED);
   else CheckMenuItem (Menu, ID_LAYERS_CEILING, MF_CHECKED);
   layer.draw_ceiling = !layer.draw_ceiling;
}
else if (wParam == ID_LAYERS_WALL)
{
   if (layer.draw_wall) CheckMenuItem (Menu, ID_LAYERS_WALL, MF_UNCHECKED);
   else CheckMenuItem (Menu, ID_LAYERS_WALL, MF_CHECKED);
   layer.draw_wall = !layer.draw_wall;
}
```

Updating Wireframe Rendering with Layers

Now that we've added the functionality for selecting the individual layers to render, we can modify our existing wireframe rendering code to take advantage of layers. Unlike the previous chapters where we displayed shapes on the screen, we're going to cut all the hardcoded shapes we've created out of the Render function. We're also going to add a new function called DrawWireframe, which will house the algorithm we wrote previously to render objects in our map using the wireframe style of rendering. Although we originally put wireframe rendering in the Render function, it should technically be in its own function because we'll be writing more advanced code in the Render function and don't want every rendering algorithm to clutter up one main function.

The main modifications we must make to the wireframe rendering algorithm are in the for loop itself. Rather than simply render each object, we'll first perform an if statement, which will check to see what type of object we're dealing with and whether the associated layer drawing value is true or false. This will happen with each iteration of the loop. If the drawing value is true, then we'll obviously render the object! If the object is false, we'll disregard drawing the object and continue through the for loop.

Since we have three different types of objects that are accessible through the Layers menu items, we must check all three individual values as discussed previously. If we neglect to put one of the three statements in the if statement, then the specified object most likely wouldn't display. Obviously this won't help our situation at all! As an example, if the value in layer.draw_floor is true and the value in map->object[i].type is equal to the value in OBJECTTYPE_FLOOR, then we would render the specified object. If either value were false, we would check the other two if statements to find a match. If all three of the if statements fail (don't produce a true condition), then objects wouldn't be rendered. In some cases this could be considered a pain, especially if the object type becomes corrupted through hard drive problems or through file hacking, but it's a chance we'll have to take. Now that we've discussed the new DrawWireframe function we can display the source code below.

```
void DrawWireframe()
{
   if (map->header.max_objects > 0)
   {
      for (long i = 0; i < map->header.max_objects; i++)
      {
      if ((layer.draw_floor && map->object[i].type == OBJECTTYPE_FLOOR) ||
          (layer.draw_ceiling && map->object[i].type == OBJECTTYPE_CEILING) ||
          (layer.draw_wall && map->object[i].type == OBJECTTYPE_WALL))
         {
            glBegin (GL_LINE_LOOP);
               for (long i2 = 0; i2 < map->object[i].max_vertices; i2++)
                    glVertex2d (map->object[i].vertex[i2].xyz[0],
                    map->object[i].vertex[i2].xyz[2]);
               glEnd();
         }
      }
   }
}
```

Drawing Solid Primitive

In previous chapters we dealt with wireframe rendering only. Because we're introducing multiple rendering styles (wireframe and solid) in this chapter, we must have a method to distinguish between each style. Rather than using integers to represent the intended rendering style, which could get confusing, we'll create a new enumeration to represent both wireframe and solid drawing modes. For each enumeration we'll use the DRAW_MODE_ naming convention followed by the type of drawing mode. Using this convention to specify the wireframe drawing mode, we would use the enumeration value DRAW_MODE_WIREFRAME. Similarly, if we wanted to specify the solid drawing mode, we'd specify DRAW_MODE_SOLID. When we define the new enumeration, the first value specified will be the wireframe value with the default value of 0, followed by the solid drawing value. The code for the new enumeration is shown here:

```
enum { DRAW_MODE_WIREFRAME = 0, DRAW_MODE_SOLID };
```

Earlier we defined the LAYER structure, but we haven't finished defining all the structures for the chapter. The next structure we'll define is called CONFIG. This structure appropriately contains the configuration data for the map editor. There aren't many variables in our map editor that should be stored in this new CONFIG structure. In this chapter we'll add one variable of the long data type called draw_mode. This variable will store the current drawing mode using the values in the DRAW_MODE enumeration we just defined. Now that we've finished defining the CONFIG structure we can display the source code for it below.

```
typedef struct
{
   long draw_mode;
} CONFIG;
```

After defining the CONFIG structure, we must declare a new global variable called config, which is of the CONFIG type. Because this variable contains more critical information, it is essential that we set a default value for each variable within the structure. For this reason, we'll call the function memset using the address of the config variable as the first parameter, 0 as the second parameter (indicating the default value), and the sizeof function on the config variable to return the third variable, which is the number of bytes to fill. We can fill the structure with 0s because at the moment there is only one variable, which has a default value of 0. When we need variables to have a default value other than 0, we can set that when needed, but for the time being, we can fill the structure with 0s. Here is the line of code:

```
memset (&config, 0, sizeof(config));
```

When our map editor initially starts up, the default style for the drawing mode will be DRAW_MODE_WIREFRAME (wireframe drawing), which is what we want. If the user wanted to change the current mode to solid rendering, he would have to select the Drawing menu and click the Solid menu item. Alas, if the user selects the Solid menu item nothing will happen because the menu item doesn't have any real functionality aside from switching the check mark from wireframe to solid and vice versa.

Our next step will be to modify the WMCommand function, more specifically the ID_DRAWING_WIREFRAME and ID_DRAWING_SOLID menu item handling source code to physically switch drawing modes. This will lead to adding our final DrawSolid function, which will complete the chapter. Moving to the WMCommand function, we'll simply add a line of code in the if statement that checks for the menu item ID_DRAWING_WIREFRAME and will set the config.draw_mode variable to DRAW_MODE_WIREFRAME. This means when the user clicks the Wireframe menu item, the current configuration drawing mode will be set to wireframe. Similarly, in the ID_DRAWING_SOLID if statement, we'll add a line of code that will set the config.draw_mode to DRAW_MODE_SOLID. Following the logic from before, when the user clicks the Solid (ID_DRAWING_SOLID) menu item, the current configuration drawing mode will be set to solid. The modification to the WMCommand function is now complete and was pretty simple. The source code for the modification is provided below.

```
else if (wParam == ID_DRAWING_WIREFRAME)
{
   CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
   CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);

   config.draw_mode = DRAW_MODE_WIREFRAME;
}
else if (wParam == ID_DRAWING_SOLID)
{
   CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_CHECKED);
   CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_UNCHECKED);

   config.draw_mode = DRAW_MODE_SOLID;
}
```

With all the configuration code in place, we can begin writing the DrawSolid function itself. The DrawSolid function will have no return type and will not accept any parameters. Although the function declaration seems fairly plain, it fulfills the requirements of the job.

After defining the function, we'll create a loop that will start at 0 and loop through all the objects in the map editor. With each iteration of the loop, we'll set the color of the object to the selectable RGB value by using the function glColor3ubv and specifying the selectable RGB variable (map->object[i].select_rgb) as the single parameter. Technically, we could have used the function glColor3ubv and specified each value in the selectable RGB array as individual parameters in the function call, but it's much easier to specify the 3ubv when you already have an array of values you want to specify! We'll take advantage of this when we write the main rendering engine for the game because it's much easier to specify each color, vertex, and other important data. We haven't used it for the rendering process in the map editor because we manipulate the data in the map editor to draw it in a top view as opposed to the proper first-person view.

After we set the color of the objects, we must have another if statement to check the individual layer value and object types to ensure the object is supposed to be rendered. This process must be done for each object type (wall, floors, and ceilings). The if statements are broken into two sections: floors and ceilings in one and walls in the other. We break them up because walls are rendered differently than floors and ceilings. With this in mind, the first if statement will check the value of layer.draw_floor to ensure it's true and the current object type is OBJECTTYPE_FLOOR. If one or both of the conditions fail within the if statement we have a conditional OR that will be executed. The OR condition will check to see if layer.draw_ceiling is true and the object type is OBJECTTYPE_CEILING. If the event conditions fail (floor/ceiling layers aren't checked, or object types aren't floor or ceiling), the object won't be drawn using the floor/ceiling rendering code and we'll proceed to the else-if statement to see if it's a wall.

In the event one of the two if statements is true, then we'll draw the object as a floor/ceiling. To begin the rendering process we'll need to tell OpenGL we want to create an object, more specifically triangles, so we'll call glBegin and specify GL_TRIANGLES as the single parameter. As we discussed in earlier chapters, when we specify glBegin we must specify glEnd to finish the object creation process. To keep things simple, we'll add the call to glEnd before actually writing the main rendering code to ensure we don't forget it later! This is a good habit to adopt because it will help you write code that will have fewer OpenGL matrix/creation errors.

After adding the glBegin/glEnd function calls, we must add another for loop that will loop from 0 to the value in object[i].max_triangles. Because we've already defined floors/ceilings as having two triangles, it's a good assumption that there will be triangles; however, if you choose to modify the engine code, you can literally have as many as you want, which is why we don't want to hardcode the value. This new loop will be positioned between the glBegin/glEnd pair, and is intended to be the main rendering loop for the object. Unlike drawing the objects in the wireframe drawing mode, each object when rendered in solid mode must be built using the individual triangle points, which are indexes to vertices. Rather than access each variable using the variable map->object[i].triangle[i2].point and specifying the index, we'll declare three new variables in the code portion of the loop. All three variables are of the long data type and will be used to store the indices for each triangle point. The variables vertex_0, vertex_1, and vertex_2 will store the appropriate triangle point, starting with map->object[i].triangle [i2].point[0], followed by map->object[i].triangle[i2].point[1], and ending with map->object[i].triangle[i2].point[2].

With each iteration of the loop the vertex indices will change, giving us the necessary indices to create the current triangle. To create the triangle we must call the OpenGL function glVertex2d, specifying the x-axis (map->object[i].vertex[vertex_0].xyz[0]) for the first parameter and the z-axis (map->object[i].vertex[vertex_0].xyz[2]) for the second parameter. For rendering objects within the game engine itself, we'll use the function glVertex3d and specify X, Y, Z, which is the normal standard, but since we're drawing things using a top view we have a slightly different parameter set than normal. Notice in the first call we used the vertex_0 variable for the vertex index. With the other two calls to glVertex2d, we'll specify vertex_1 and vertex_2 as the vertex index to finish building the triangle. After we place those two new calls, we've finished the code for rendering floors and ceilings.

Now we must write the code to render walls. Following the logic from the floors/ceilings section of the DrawSolid function, before we draw the wall we must ensure that the layer.draw_wall variable is true and that the object type is OBJECTTYPE_WALL. If both values are true, then we can draw the object. As you have probably noticed by now, I don't like making you write the same piece of source code over and over and over again! The rendering code for walls is nearly identical to that of the code for rendering floors/ceilings, with the exception of the parameter in glBegin. For this reason we can simply copy and paste the rendering code from the previous section to the wall rendering code and change the rendering style from GL_TRIANGLES to GL_LINE_LOOP. We use GL_LINE_LOOP as opposed to solid triangles because they don't render properly otherwise. Because our walls are paper thin, lines are the appropriate method of rendering them. Technically, we could change the thickness of the walls when rendering in a top view; however, that would be misrepresenting the data and could cause some major issues when the user tries to develop levels. Thankfully, we've finished the discussion of rendering solid objects and can display the source code below.

```
void DrawSolid()
{
   for (long i = 0; i < map->header.max_objects; i++)
   {
      glColor3ubv (map->object[i].select_rgb);

      if ((layer.draw_floor && map->object[i].type == OBJECTTYPE_FLOOR) ||
         (layer.draw_ceiling && map->object[i].type == OBJECTTYPE_CEILING))
      {
         glBegin (GL_TRIANGLES);
         for (long i2 = 0; i2 < map->object[i].max_triangles; i2++)
         {
            long vertex_0 = map->object[i].triangle[i2].point[0];
            long vertex_1 = map->object[i].triangle[i2].point[1];
            long vertex_2 = map->object[i].triangle[i2].point[2];

            glVertex2d (map->object[i].vertex[vertex_0].xyz[0],map->
                        object[i].vertex[vertex_0].xyz[2]);
            glVertex2d (map->object[i].vertex[vertex_1].xyz[0],map->
                        object[i].vertex[vertex_1].xyz[2]);
            glVertex2d (map->object[i].vertex[vertex_2].xyz[0],map->
                        object[i].vertex[vertex_2].xyz[2]);
         }
         glEnd();
      }
      else if (layer.draw_wall && map->object[i].type == OBJECTTYPE_WALL)
      {
         for (long i2 = 0; i2 < map->object[i].max_triangles; i2++)
         {
            long vertex_0 = map->object[i].triangle[i2].point[0];
            long vertex_1 = map->object[i].triangle[i2].point[1];
            long vertex_2 = map->object[i].triangle[i2].point[2];

            glBegin (GL_LINE_LOOP);
               glVertex2d (map->object[i].vertex[vertex_0].xyz[0],map->
                           object[i].vertex[vertex_0].xyz[2]);
               glVertex2d (map->object[i].vertex[vertex_1].xyz[0],map->
                           object[i].vertex[vertex_1].xyz[2]);
               glVertex2d (map->object[i].vertex[vertex_2].xyz[0],map->
                           object[i].vertex[vertex_2].xyz[2]);
            glEnd();
         }
      }
   }
}
```

To finish our work for the chapter, the last thing we must do is edit our Render function once again, and rip out all those ugly hardcoded shapes we had created in previous chapters. In their place we'll add two lines of code that will check to see if the value of config.draw_mode is equal to DRAW_MODE_WIREFRAME. If the values end up being equal, then we'll run the DrawWireframe function, indicating that the drawing mode is wireframe. If the function fails, then we'll test to see if the value in config.draw_mode is DRAW_MODE_SOLID. If those values are equal, then we'll run the DrawSolid function, which will draw the objects in solid colors. We don't need to specify any parameters for the DrawSolid function because it has overloaded values already set! The modifications to the Render function have been finished, so we can display the source code below.

```
void Render()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glLoadIdentity();
   glPushMatrix();
      glTranslatef (0.0f, 0.0f, 0.0f);

      if (config.draw_mode == DRAW_MODE_WIREFRAME) DrawWireframe();
      else if (config.draw_mode == DRAW_MODE_SOLID) DrawSolid();

      if (creation_coords.type == OBJECTTYPE_WALL)
      {
         glBegin (GL_LINES);
            glVertex2d (creation_coords.start.world_x, creation_
                        coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x, creation_
                        coords.finish.world_z);
         glEnd();
      }
      else if (creation_coords.type == OBJECTTYPE_FLOOR || creation_coords.type
                                    == OBJECTTYPE_CEILING)
      {
         glBegin (GL_LINE_LOOP);
            glVertex2d (creation_coords.start.world_x, creation_
                        coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x, creation_
                        coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x, creation_
                        coords.finish.world_z);
            glVertex2d (creation_coords.start.world_x, creation_
                        coords.finish.world_z);
         glEnd();
      }

   glPopMatrix();
   SwapBuffers (raster.hDC);
}
```

Chapter Example

Here is the complete code for this chapter's example:

ex6_1.cpp

```
#include <windows.h>
#include <winbase.h>
#include <stdio.h>

#include "resource.h"

#include "raster.h"
#include "map.h"

#define DEFAULT_BUTTON_WIDTH    100
#define DEFAULT_BUTTON_HEIGHT   20

enum { CREATE_MODE_NULL = 0, CREATE_MODE_START, CREATE_MODE_SIZE,
       CREATE_MODE_FINISH };
enum { DRAW_MODE_WIREFRAME = 0, DRAW_MODE_SOLID };

typedef struct
{
   long      mouse_x;
   long      mouse_y;

   double    world_x;
   double    world_y;
   double    world_z;
} COORDS;


typedef struct
{
   long             mode;
   long             type;

   COORDS           start;
   COORDS           finish;
} CREATION_COORDS;


typedef struct
{
   long             draw_mode;
} CONFIG;


typedef struct
{
   bool draw_floor;
   bool draw_ceiling;
   bool draw_wall;
} LAYER;

HINSTANCE           GlobalInstance;
HMENU               Menu;
HMENU               PopupMenu;
HWND                Window;
HWND                RenderWindow;
HWND                bCreateWall;
HWND                bCreateFloor;
HWND                bCreateCeiling;
RASTER              raster;

CREATION_COORDS     creation_coords;
MAP                 *map = new MAP;

CONFIG              config;
LAYER               layer;

void ResizeGLWindow(long width, long height)
{
   glViewport(0, 0, (GLsizei) width, (GLsizei) height);
   glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(-200,200, -200,-200, -2000,2000);
   glMatrixMode(GL_MODELVIEW);
}

void SetGLDefaults()
{
   glEnable (GL_DEPTH_TEST);
   glDisable (GL_CULL_FACE);
   glClearColor (0.6f, 0.6f, 0.6f, 1.0f);
}

void DrawWireframe()
{
   if (map->header.max_objects > 0)
   {
      for (long i = 0; i < map->header.max_objects; i++)
      {
         if ((layer.draw_floor && map->object[i].type == OBJECTTYPE_FLOOR) ||
            (layer.draw_ceiling && map->object[i].type == OBJECTTYPE_CEILING) ||
            (layer.draw_wall && map->object[i].type == OBJECTTYPE_WALL))
         {
            glBegin (GL_LINE_LOOP);
               for (long i2 = 0; i2 < map->object[i].max_vertices; i2++)
                    glVertex2d (map->object[i].vertex[i2].xyz[0],
                    map->object[i].vertex[i2].xyz[2]);
            glEnd();
         }
      }
   }
}

void DrawSolid()
{
   for (long i = 0; i < map->header.max_objects; i++)
   {
      glColor3ubv (map->object[i].select_rgb);

      if ((layer.draw_floor && map->object[i].type == OBJECTTYPE_FLOOR) ||
         (layer.draw_ceiling && map->object[i].type == OBJECTTYPE_CEILING))
      {
         glBegin (GL_TRIANGLES);
         for (long i2 = 0; i2 < map->object[i].max_triangles; i2++)
         {
            long vertex_0 = map->object[i].triangle[i2].point[0];
            long vertex_1 = map->object[i].triangle[i2].point[1];
            long vertex_2 = map->object[i].triangle[i2].point[2];

            glVertex2d (map->object[i].vertex[vertex_0].xyz[0],map->
                        object[i].vertex[vertex_0].xyz[2]);
            glVertex2d (map->object[i].vertex[vertex_1].xyz[0],map->
                        object[i].vertex[vertex_1].xyz[2]);
            glVertex2d (map->object[i].vertex[vertex_2].xyz[0],map->
                        object[i].vertex[vertex_2].xyz[2]);
         }
         glEnd();
      }
      else if (layer.draw_wall && map->object[i].type == OBJECTTYPE_WALL)
      {
         for (long i2 = 0; i2 < map->object[i].max_triangles; i2++)
         {
            long vertex_0 = map->object[i].triangle[i2].point[0];
            long vertex_1 = map->object[i].triangle[i2].point[1];
            long vertex_2 = map->object[i].triangle[i2].point[2];

            glBegin (GL_LINE_LOOP);
               glVertex2d (map->object[i].vertex[vertex_0].xyz[0],map->
                           object[i].vertex[vertex_0].xyz[2]);
               glVertex2d (map->object[i].vertex[vertex_1].xyz[0],map->
                           object[i].vertex[vertex_1].xyz[2]);
               glVertex2d (map->object[i].vertex[vertex_2].xyz[0],map->
                           object[i].vertex[vertex_2].xyz[2]);
            glEnd();
         }
      }
   }
}

void Render()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glLoadIdentity();
   glPushMatrix();
      glTranslatef (0.0f, 0.0f, 0.0f);

      if (config.draw_mode == DRAW_MODE_WIREFRAME) DrawWireframe();
      else if (config.draw_mode == DRAW_MODE_SOLID) DrawSolid();

      if (creation_coords.type == OBJECTTYPE_WALL)
      {
         glBegin (GL_LINES);
            glVertex2d (creation_coords.start.world_x, creation_
                        coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x, creation_
                        coords.finish.world_z);
         glEnd();
      }
      else if (creation_coords.type == OBJECTTYPE_FLOOR || creation_
               coords.type == OBJECTTYPE_CEILING)
      {
         glBegin (GL_LINE_LOOP);
            glVertex2d (creation_coords.start.world_x, creation_
                        coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x, creation_
                        coords.start.world_z);
            glVertex2d (creation_coords.finish.world_x, creation_
                        coords.finish.world_z);
            glVertex2d (creation_coords.start.world_x, creation_
                        coords.finish.world_z);
         glEnd();
      }

   glPopMatrix();
   SwapBuffers (raster.hDC);
```

```
}

LRESULT CALLBACK MapDetailsDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
          LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, "Map Name");

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING, 0,
                             (LPARAM)"Erase Me");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_RESETCONTENT,
                              0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING, 0,
                             (LPARAM)"Exit");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING, 0,
                             (LPARAM)"Get Fragged");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_SETCURSEL,
                              0, 1);

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Erase Me");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_RESETCONTENT,
                              0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Single Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Multi Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_SETCURSEL,
                              0, 1);
      } break;

      case WM_COMMAND:
      {
         if (wParam == IDOK)
         {
            long level_rule = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_
                              RULES, LB_GETCURSEL, 0, 0);
            long level_type = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_
                              TYPE, CB_GETCURSEL, 0, 0);

            char temp[500];

            sprintf (temp, "Level Type: %i\r\nLevel Rule: %i\r\nOk Button!",
                     level_type, level_rule);
            MessageBox (hWnd, temp, "Ok", MB_OK);

            EndDialog (hWnd, 0);
         }
         else if (wParam == IDCANCEL)
         {
            MessageBox (hWnd, "Cancel Button!", "Cancel", MB_OK);
            EndDialog (hWnd, 0);
         }
      } break;
   }
   return (0);
}

void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (lParam == (LPARAM)bCreateWall)
   {
      creation_coords.type = OBJECTTYPE_WALL;
      SetWindowText (bCreateWall, "*Wall*");
      SetWindowText (bCreateCeiling, "Create Ceiling");
      SetWindowText (bCreateFloor, "Create Floor");
   }
   else if (lParam == (LPARAM)bCreateFloor)
   {
      creation_coords.type = OBJECTTYPE_FLOOR;
      SetWindowText (bCreateFloor, "*Floor*");
      SetWindowText (bCreateCeiling, "Create Ceiling");
      SetWindowText (bCreateWall, "Create Wall");
   }
   else if (lParam == (LPARAM)bCreateCeiling)
   {
      creation_coords.type = OBJECTTYPE_CEILING;
      SetWindowText (bCreateCeiling, "*Ceiling*");
      SetWindowText (bCreateFloor, "Create Floor");
      SetWindowText (bCreateWall, "Create Wall");
   }
   else if (wParam == ID_FILE_EXIT) PostQuitMessage(0);
   else if (wParam == ID_DRAWING_WIREFRAME)
   {
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);

      config.draw_mode = DRAW_MODE_WIREFRAME;
   }
   else if (wParam == ID_DRAWING_SOLID)
   {
      CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_CHECKED);
      CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_UNCHECKED);

      config.draw_mode = DRAW_MODE_SOLID;
   }
   else if (wParam == ID_MAP_DETAILS) DialogBox (GlobalInstance,
            MAKEINTRESOURCE(IDD_MAP_DETAILS), NULL, (DLGPROC)MapDetailsDlgProc);
   else if (wParam == ID_LAYERS_FLOOR)
   {
      if (layer.draw_floor) CheckMenuItem (Menu, ID_LAYERS_FLOOR, MF_UNCHECKED);
      else CheckMenuItem (Menu, ID_LAYERS_FLOOR, MF_CHECKED);
      layer.draw_floor = !layer.draw_floor;
```

```
    }
    else if (wParam == ID_LAYERS_CEILING)
    {
       if (layer.draw_ceiling) CheckMenuItem (Menu, ID_LAYERS_CEILING,
           MF_UNCHECKED);
       else CheckMenuItem (Menu, ID_LAYERS_CEILING, MF_CHECKED);
       layer.draw_ceiling = !layer.draw_ceiling;
    }
    else if (wParam == ID_LAYERS_WALL)
    {
       if (layer.draw_wall) CheckMenuItem (Menu, ID_LAYERS_WALL, MF_UNCHECKED);
       else CheckMenuItem (Menu, ID_LAYERS_WALL, MF_CHECKED);
       layer.draw_wall = !layer.draw_wall;
    }

    // Popup Menu Items
    else if (wParam == ID_POPUP_MOVE) MessageBox (Window, "Move", "Click", MB_OK);
    else if (wParam == ID_POPUP_DELETE) MessageBox (Window, "Delete", "Click",
            MB_OK);
    else if (wParam == ID_POPUP_TEXTURE) MessageBox (Window, "Texture", "Click",
            MB_OK);
    else if (wParam == ID_POPUP_DUPLICATE) MessageBox (Window, "Duplicate",
            "Click", MB_OK);
}

void DisplayPopupMenu()
{
    HMENU temp = GetSubMenu(PopupMenu, 0);
    POINT point;
    GetCursorPos (&point);
    TrackPopupMenu(temp, TPM_LEFTALIGN|TPM_RIGHTBUTTON, point.x, point.y, 0,
                   Window, NULL);
}

void WMSize(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    RECT rect;

    GetClientRect (Window, &rect);
    MoveWindow (RenderWindow, DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-
                DEFAULT_BUTTON_WIDTH, rect.bottom-rect.top, true);

    GetClientRect (RenderWindow, &rect);
    ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);
}

COORDS        ComputeMouseCoords(long xPos, long yPos)
{
    COORDS    coords;
    RECT      rect;

    float     window_width;
    float     window_height;
```

```
   float            window_start_x;
   float            window_start_y;


   coords.mouse_x   = xPos;
   coords.mouse_y   = yPos;


   GetWindowRect (RenderWindow, &rect);
   window_width     = (float)(rect.right - rect.left);
   window_height    = (float)(rect.bottom - rect.top);
   window_start_x   = (float)(coords.mouse_x - rect.left);
   window_start_y   = (float)coords.mouse_y;


   coords.world_x   = (window_start_x / window_width) * 2.0 - 1.0;
   coords.world_z   = -((window_start_y / window_height) * 2.0 - 1.0);

   return (coords);
}

void WMLButtonDown(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   creation_coords.mode   = CREATE_MODE_START;
   creation_coords.start  = ComputeMouseCoords(LOWORD(lParam), HIWORD(lParam));
   creation_coords.finish = creation_coords.start;
}


void WMLButtonUp(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (creation_coords.mode != CREATE_MODE_NULL)
   {
      creation_coords.mode      = CREATE_MODE_NULL;
      creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                          HIWORD(lParam));

      if (creation_coords.type == OBJECTTYPE_WALL)
      {
         map->InsertObject ("Wall", creation_coords.type);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 1, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 1, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 0, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 0, creation_coords.start.world_z);
         map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
              1.0f,0.0f, 1.0f,1.0f);
         map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
              0.0f,1.0f, 0.0f,0.0f);
      }
```

```
        else if (creation_coords.type == OBJECTTYPE_FLOOR)
        {
           map->InsertObject ("Floor", creation_coords.type);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.start.world_x, 0, creation_coords.start.world_z);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.finish.world_x, 0, creation_coords.start.world_z);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.finish.world_x, 0, creation_coords.finish.world_z);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.start.world_x, 0, creation_coords.finish.world_z);
           map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
                1.0f,0.0f, 1.0f,1.0f);
           map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
                0.0f,1.0f, 0.0f,0.0f);
        }
        else if (creation_coords.type == OBJECTTYPE_CEILING)
        {
           map->InsertObject ("Ceiling", creation_coords.type);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.start.world_x, 1, creation_coords.start.world_z);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.finish.world_x, 1, creation_coords.start.world_z);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.finish.world_x, 1, creation_coords.finish.world_z);
           map->InsertVertex (map->header.max_objects-1, creation_
                coords.start.world_x, 1, creation_coords.finish.world_z);
           map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
                1.0f,0.0f, 1.0f,1.0f);
           map->InsertTriangle (map->header.max_objects-1, 2, 3, 0, 1.0f,1.0f,
                0.0f,1.0f, 0.0f,0.0f);
        }

        memset (&creation_coords.start, 0, sizeof(creation_coords.start));
        memset (&creation_coords.finish, 0, sizeof(creation_coords.finish));
     }
  }


  void WMMouseMove(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
  {
     char temp[500];

     if (creation_coords.mode != CREATE_MODE_NULL)
     {
        creation_coords.mode      = CREATE_MODE_SIZE;
        creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                           HIWORD(lParam));
        sprintf (temp, "Map Editor, Mx=%i My=%i, X=%0.4f Z=%0.4f", creation_
                 coords.finish.mouse_x, creation_coords.finish.mouse_y, creation_
                 coords.finish.world_x, creation_coords.finish.world_z);
     }
```

```
   else sprintf (temp, "Map Editor, Mx=%i My=%i", LOWORD(lParam), HIWORD(lParam));

   SetWindowText (Window, temp);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
      case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
      case WM_SIZE: WMSize (hWnd, msg, wParam, lParam); break;
      case WM_RBUTTONUP: DisplayPopupMenu(); break;
      case WM_LBUTTONDOWN: WMLButtonDown (hWnd, msg, wParam, lParam); break;
      case WM_LBUTTONUP: WMLButtonUp (hWnd, msg, wParam, lParam); break;
      case WM_MOUSEMOVE: WMMouseMove (hWnd, msg, wParam, lParam); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
           int CmdShow)
{
   WNDCLASS         wc;
   MSG              msg;
   RECT             rect;

   GlobalInstance   = hInstance;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "ME";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);
   }

   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                         0, 0, 640, 480, NULL, NULL, hInstance, NULL);
   if (Window == NULL)
   {
      MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
      return (0);
   }
```

```
GetClientRect (Window, &rect);


bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE,
                 0, 100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT,
                 Window, NULL, hInstance, NULL);

bCreateCeiling = CreateWindow("BUTTON", "Create Ceiling", WS_CHILD |
                 WS_VISIBLE, 0, 100+(DEFAULT_BUTTON_HEIGHT*2),
                 DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT,
                 Window, NULL, hInstance, NULL);

bCreateFloor = CreateWindow("BUTTON", "Create Floor", WS_CHILD | WS_VISIBLE,
                 0, 100+(DEFAULT_BUTTON_HEIGHT*4), DEFAULT_BUTTON_WIDTH,
                 DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);


RenderWindow= CreateWindow("STATIC", NULL, WS_CHILD | WS_VISIBLE | WS_BORDER,
                 DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-DEFAULT_
                 BUTTON_WIDTH, rect.bottom-rect.top, Window, NULL,
                 hInstance, NULL);


Menu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_MENU));
SetMenu (Window, Menu);

PopupMenu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_POPUP_MENU));


if (!raster.Init(RenderWindow)) return (0);

GetClientRect (RenderWindow, &rect);
ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);

SetGLDefaults();


memset (&creation_coords, 0, sizeof(creation_coords));
memset (&layer, 1, sizeof(layer));
memset (&config, 0, sizeof(config));



while (1)
{
   Render();

   if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
   {
      if (msg.message == WM_QUIT) break;
      TranslateMessage(&msg);
      DispatchMessage (&msg);
   }
```

```
    }

    raster.Release(RenderWindow);
    delete map;


    return (1);
}
```

Conclusion

Although this chapter was fairly short compared to previous chapters, it discussed topics that are equally important to the development of our map editor. When creating our maps, we can now selectively display data on the screen, eliminating the clutter we'll eventually have when we add more features/objects/stuff to the map. We've also cleaned up the Render function, moving the main pieces of rendering code to separate functions and allowing us to easily render our data without cluttering up our main Render function.

The solid rendering drawing mode was also added to our map editor, giving us the ability to see objects in their native colors and a semi-real look at how things will be displayed. Although it's not a fully textured rendering, it gives us a better idea of how things will be rendered in our game engine.

Chapter 7

Adding Game-Related Functionality

Placing Start Positions

Writing the code to place a start position in our map editor is relatively simple in comparison to the other features we've created. Essentially we must create a new button, write the button handling code that will set the creation information, wait until the user releases the left mouse button (WM_LBUTTONUP Windows message), and use the world coordinates calculated from the Windows message as the start coordinates for the single-player position. Now that process wasn't difficult, was it? Obviously placing the single-player start position is a necessary function to write; otherwise the player would always start at position 0,0,0. Although having the player start at position 0,0,0 doesn't sound bad, imagine creating an entire level from scratch and then realizing at the last minute that the start position 0,0,0 is nowhere near any walls. You'd be mighty annoyed if you spent hours making a level only to find out it would have to be completely reworked for such a simple task.

To begin this "daunting" task, we must open the header file map.h and add a new value of the object type to the enumeration we've created. At the bottom of the enumeration we'll add the value ITEM_START_POSITION. This is the only modification needed to map.h for adding start positions, so we can display the source code for the enumeration below.

```
enum {OBJECTTYPE_WALL = 1,
      OBJECTTYPE_FLOOR,
      OBJECTTYPE_CEILING,
      ITEM_START_POSITION
     };
```

After adding the new object type to the enumeration, we must close the current file and add a new global variable to the main source file. The name for the new global variable will be bPlaceStartPosition and it will be of the HWND data type. This will be the button the users will click when they want to set the single-player start position. Obviously when we declare a new global variable of the HWND data type we must create the actual window allowing it to be drawn to the screen. To do this, we'll move to the WinMain function at the bottom of the source file and place our new function call to CreateWindow after the function call to CreateWindow that creates the Create Floor button and returns it to bCreateFloor.

Since we're creating a standard button without any special features we can simply copy and paste the source code from the previous function call to CreateWindow and modify it accordingly. There are only three changes that must be performed. The first modification is to the return variable. Rather than returning our new button to bCreateFloor, we'll return the value to our newly declared variable bPlaceStartPosition. The second parameter of the function call must be changed to "Place Start", which will change the caption on the button. Obviously this is important so we don't have two buttons with the same caption. The third and final adjustment we must make is to the fifth parameter, which is the starting Y position of the button. We can adjust the height to 100+(DEFAULT_BUTTON_HEIGHT*6), which will give us a blank space between the previous button and our new one. We've now completed writing the code to create the button and can display the source code for it below.

```
bPlaceStartPosition = CreateWindow("BUTTON", "Place Start", WS_CHILD |
         WS_VISIBLE, 0, 100+(DEFAULT_BUTTON_HEIGHT*6), DEFAULT_BUTTON_WIDTH,
         DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);
```

Before we discuss the code involved in handling the button, we should discuss a new addition to our code. In the previous chapters when we added functionality to each button and had the button caption change to suit which button was selected, we didn't write efficient code for setting the captions. If you look closely you'll notice that in each if statement we set the other captions back to their defaults. Although this is a minor defect, it represents a big issue. While initially designing that portion of the code, it may have seemed like a good idea; however, as you add more functionality to the map editor, the process of setting each button caption to its default would become more and more tedious.

I decided to point this out now before we had too many functions, in which case you'd be really annoyed if you had to rewrite a ton of code to facilitate a slight efficiency increase. At any rate, we'll create a new function called ShowSelectedButton, which will set every button to its default caption, then set the caption of the selected button to its appropriate selected version. Rather than type each line of SetWindowText again, we'll simply move the SetWindowText function calls from the button-click handling code in WMCommand to our newly created function. In place of each function call to SetWindowText, we'll put a call to ShowSelectedButton. After we've copied all the SetWindowText calls, we can add another one to the function with the parameter bPlaceStartPosition and the text "Place Start". After we set the default values for each button, we can now set the selected button caption value. We'll create a new case statement that will use the value of creation_coords.type as the expression, and we'll use the different object type values in the enumeration as the labeled statements. In plain English, we'll use the variable creation_coords.type as the parameter in the switch statement, and then set the appropriate button name based on the appropriate enumeration value. Since we've already made the selected button captions, we can copy and paste those values once again from the WMCommand function.

Remember that we'll need to add a break after each call to SetWindowText; otherwise, we'll end up setting all the button captions instead of the specific one we want. At the bottom of our newly created case statement we'll add the ITEM_START_POSITION value and create a new function call to SetWindowText with bPlaceStartPosition as the first parameter and "*StartPos*" as the second. This will set our newly created button to "*Start Pos*" when the user selects the button. We've now completed the code for the ShowSelectedButton function and can display the source code below.

```
void ShowSelectedButton()
{
   SetWindowText (bCreateFloor, "Create Floor");
   SetWindowText (bCreateCeiling, "Create Ceiling");
   SetWindowText (bCreateWall, "Create Wall");
   SetWindowText (bPlaceStartPosition, "Place Start");

   switch (creation_coords.type)
   {
      case OBJECTTYPE_FLOOR: SetWindowText (bCreateFloor, "*Floor*"); break;
      case OBJECTTYPE_CEILING: SetWindowText (bCreateCeiling, "*Ceiling*"); break;
      case OBJECTTYPE_WALL: SetWindowText (bCreateWall, "*Wall*"); break;
      case ITEM_START_POSITION: SetWindowText (bPlaceStartPosition, "*StartPos*");
            break;
   }
}
```

Moving to the WMCommand function, we'll add a new else-if statement to check the value of lParam to see if it's equal to a cast LPARAM value of bPlaceStartPosition. If the values are equal, then the user has pressed the button and we can perform the code within the if statement. Within the if statement, the first thing we must do is set the creation_coords.type variable to our newly created enumeration value ITEM_START_POSITION, then we'll make another function call to ShowSelectedButton to update the button name and show the currently selected button. The source code for the initial button click of bPlaceStartPosition is provided below.

```
else if (lParam == (LPARAM)bPlaceStartPosition)
{
   creation_coords.type = ITEM_START_POSITION;
   ShowSelectedButton();
}
```

To finalize the placement of a single-player start position, we must add a new else-if clause to the code in the WMLButtonUp function in the object type identification section. When the user presses the bPlaceStartPosition button, the creation type is set to ITEM_START_POSITION. When the user presses and releases the left mouse button, the world coordinates are calculated and the object type identification is checked. The if statement we'll add will check the creation_coords.type variable for the value of ITEM_START_POSITION. If the values are equal, we'll proceed through the specified code.

When executing the code within the if statement, the first thing we'll do is declare a new variable called rgb, which is of the long data type. This variable will have a default set by the returned value specified in the map->GenerateColor method. This will generate a color for the start position, so we may select the position and move it around once we write the code for that. You may be wondering why we would want to move it when setting the position is so simple. Well, you may want to tweak the position slightly, where a slight movement may be the best solution.

After generating a new unique color and returning it to the newly declared variable rgb, we must set the single-player selectable rgb colors. Since we've already generated the unique color, we must extract the individual RGB values from the value in the rgb variable and set the individual color components of the map->details.single_player.select_rgb array. To retrieve the red color value, we simply call the function GetRValue and supply the rgb variable as the parameter. The variable map->details.single_player.select_rgb[0] will store the returned red component value from the GetRValue function call. Similarly, when we want to retrieve the green color component we specify the index 1 in the select_rgb array and use the GetGValue function. Retrieving the blue component works in the same fashion with the exception that we use index 2 of the select_rgb array and the function GetBValue.

The next step is to set the XYZ start positions using the mouse world coordinates, which have already been calculated. To set the X position we simply set the variable map->details.single_player.xyz[0] equal to the value of creation_coords.start.world_x. When setting the Y position we use index 1 of the xyz array and set it to the value of creation_coords.start.world_y. The Z position works in a similar fashion with the exception that we use index 2 of the xyz array and we set it to the value creation_coords.start.world_z. After setting the Z position, we've set all the variables needed for the start position. After writing much of the framework in the previous chapters, the higher-level functionality is fairly simple to understand! We now display the source code for setting the start position here.

```
else if (creation_coords.type == ITEM_START_POSITION)
{
   long rgb = map->GenerateColor();

   map->details.single_player.select_rgb[0] = GetRValue(rgb);
   map->details.single_player.select_rgb[1] = GetGValue(rgb);
   map->details.single_player.select_rgb[2] = GetBValue(rgb);

   map->details.single_player.xyz[0] = creation_coords.start.world_x;
   map->details.single_player.xyz[1] = creation_coords.start.world_y;
   map->details.single_player.xyz[2] = creation_coords.start.world_z;
}
```

The final step in setting the start positions is to write the code to draw the start position on the screen. To do this, we'll create a new function called DrawStartPosition, which will contain the code to draw the single-player start position marker. Before we draw the start position marker, we'll set the current object color to a full-intensity blue to ensure the object color is unique so we can easily distinguish the marker from an object. Of course if our color generation algorithm randomly chooses a color one shade below the full-intensity blue we'll have to hope the object isn't the same size; otherwise, the level designer would be screwed! The chances of that happening are very low (approximately 1 in 16,777,213). So you can see why I'm not terribly concerned about an object using a generated color close to that of our start position color.

To set the object color, we'll call the function glColor3f and specify 0.0f for the first two parameters and 1.0f for the third parameter, which will give us a full-intensity blue. Rather than draw some sort of really intricate object to represent the start position marker, we'll draw a small diamond (using the GL_QUADS primitive type) around the already computed start position. Of course when we make any type of object we must have a pair of glBegin/glEnd function calls and specify the vertices in the middle. When we call the function glBegin, we'll specify the single parameter of GL_QUADS.

Now that we've specified the begin/end pair for building the object, we must specify the vertices themselves. Much like the previous sections for rendering, we only need to use two of the three axes to draw the objects. Following the same pattern, we'll use the glVertex2d function to input each vertex into the object. Each vertex will use the vertex's X (map->details.single_player.xyz[0]) and Z (map->details.single_player.xyz[2]) coordinates. Along with each value we'll add or subtract 0.01 from each direction to create our diamond shape. In case you are wondering where I got the number 0.01 from, it is 0.05% of the actual screen size. I chose 0.05% because it represents a small object when drawn on the screen. Although 1% probably would have worked, this value will give us slightly more space to work with visually. Besides, it's just a simple diamond.

When we draw our diamond, we'll start at the base position and subtract 0.01 from the Y coordinate, which will position the first vertex at the bottom of the shape. The next vertex will add 0.01 to the x-axis and keep the standard Y position, which will move the next point to the right of the shape. After setting the right position, we'll set the top position by specifying the default x-axis and subtracting 0.01 from the Y position. The final position we'll draw is the left position; we'll simply subtract 0.01 from that x-axis and keep the Y position the same! This will create a diamond-shaped object (in actuality it's a box rotated at a 45° angle).

Now that we've finished inputting the vertex, the glEnd statement will be called. To finish the function, we'll simply call the glColor3f function, specifying 1.0f for each parameter and resetting the color to white. After setting the color, we've finished drawing the start position and can display the source code below.

```
void DrawStartPosition()
{
   glColor3f (0.0f, 0.0f, 1.0f);
   glBegin (GL_QUADS);
      glVertex2d (map->details.single_player.xyz[0], map->details.single_
                  player.xyz[2]-0.01);
      glVertex2d (map->details.single_player.xyz[0]+0.01, map->details.single_
                  player.xyz[2]);
      glVertex2d (map->details.single_player.xyz[0], map->details.single_
                  player.xyz[2]+0.01);
      glVertex2d (map->details.single_player.xyz[0]-0.01, map->details.single_
                  player.xyz[2]);
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}
```

We're finished writing the code to add a single-player start position, and can move to the next section, which discusses how to add two deathmatch start positions.

Placing Multiplayer Starting Positions

Adding multiplayer start positions follows the same steps as the previous section, in that we must create a new button for the user to press, set the creation type, wait for the left mouse button to be released, and calculate the mouse coordinates into world coordinates. Unlike the single-player section, we'll add a dialog box that will prompt the user whether the new position will be for player 1 or player 2. Since this is a multiplayer start position, we could be inserting a start position for either player. As you can see, the steps for inserting a deathmatch position are fairly simple since we've already done much of the work in the previous section.

To begin coding the deathmatch starting position code, the first thing we'll do is open the map.h file and add another object type to our enumeration. The new value we'll add is called ITEM_DM_POSITION, which will obviously be the deathmatch start position. To keep things in order, we'll add the value to the bottom of the enumerated value list. The code for the new enumeration list is provided below.

```
enum {OBJECTTYPE_WALL = 1,
      OBJECTTYPE_FLOOR,
      OBJECTTYPE_CEILING,
      ITEM_START_POSITION,
      ITEM_DM_POSITION,
     };
```

Moving to the main source file, we'll add a new global variable of the HWND data type. The new variable, bPlaceDMPosition, will be the new button for placing deathmatch start positions. Obviously when we declare a new variable we'll have to actually create the window for it as well. To do this, we'll move to the WinMain function at the bottom of the code, and place the new call to CreateWindow between the call that returns the button for bPlaceStartPosition and the call that returns the window for the variable RenderWindow.

To save coding time, we'll copy and paste the code from the bPlaceStartPosition button and modify it according to our needs. Obviously the returned value must be placed inside bPlaceDMPosition, and we'll need to change the window name from "Place Start" to "Place DM". The final change we must make in the function call is to the starting Y position of the button itself. Following the previous scheme, we'll add 2 to the current number, so we'll multiply the default button height by 8 to give the resulting Y position button coordinate. The modifications to the CreateWindow function have been finished and we can display the source code below.

```
bPlaceDMPosition = CreateWindow("BUTTON", "Place DM", WS_CHILD | WS_VISIBLE, 0,
       100+(DEFAULT_BUTTON_HEIGHT*8), DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT,
       Window, NULL, hInstance, NULL);
```

Now that we have a new button in our map editor, we'll need to write the actual button handling code to actually make use of the button. To do this we'll move to the WMCommand function and add a new if statement to check whether the lParam variable is equal to an LPARAM cast bPlaceDMPosition variable. If the variables are equal, then the button has been clicked and we must set the creation_coords.type variable to our newly created object type ITEM_DM_POSITION, indicating we are creating a deathmatch start position, then call the function ShowSelectedButton to update the button names to reflect which one has been selected. The code for the new if statement is provided on the following page.

```
else if (lParam == (LPARAM)bPlaceDMPosition)
{
   creation_coords.type = ITEM_DM_POSITION;
   ShowSelectedButton();
}
```

After adding the new if statement to the WMCommand, we should add several lines of code to the ShowSelectedButton function to allow the newly created button to show that it is selected when the button has been pressed. To update the function we must add a new call to the function SetWindow-Text, supplying the window we want to change (bPlaceDMPosition) and a NULL-terminated string for the window name "Place DM." The new call to the SetWindowText function will be placed below the previous call to SetWindowText, where we've set the text for the bPlaceStartPosition button. By adding this function call, the button will be set back to its original default when the user clicks any of the other buttons on the left-hand side of the screen.

The next thing we must do is add a new line to the switch statement, which will use the ITEM_DM_POSITION object type as the case parameter and set the window to "*DM Pos*". In the event the creation_coords.type variable is set to the object type ITEM_DM_POSITION, then our other SetWindowText function call will be run and it will change the button to look like it is selected. This is extremely important, because we want the user to know right away which button is selected. This modification is fairly straightforward compared to the rest, so we don't need to show the source code here.

After adding the two lines of code to the ShowSelectedButton function, we'll create a new dialog box called IDD_DM_POSITION. Inside the dialog box, we'll add a combo box control called IDC_DM_POSITION_TYPE. Although in the example I've set the dialog box to center itself by default, this is not required. The important thing here is for the dialog box to appear with two options in the combo box. As with all dialog boxes, we'll have to create a new message callback function that will handle all the messages the dialog box sends. We'll create a callback called DMPositionDlgProc, which will be located just above the WMLButtonUp function.

We'll add a switch statement to handle each message, even though there are only two messages to worry about. The first message we'll add to the switch statement is the WM_INITDIALOG message, which is normally used to initialize the controls within the dialog box and variables relating to the dialog box. In our case, we'll use the message to initialize the combo box we've added. There are a total of four messages to be sent to the control. Each message will be sent using the SendDlgItemMessage function, specify-ing the hWnd variable that is passed into the message handler as the first parameter, the resource ID of the control (IDC_DM_POSITION_TYPE), the message value, and the two accompanying parameters for it.

The first message we'll send to the control will reset the content. This will erase everything inside the control, then add all the values back into the variable.

We do this to ensure there are no remnants of data from previous usage of the function. The message value used as the third parameter in the SendDlgItemMessage function will be CB_RESETCONTENT, and the fourth and fifth parameters will be set to 0 since they are not used in this message.

The second and third messages will add text to the combo boxes using the message value CB_ADDSTRING. The fourth parameter must remain 0; however, we'll supply a NULL-terminated string as the fifth parameter and cast it to the LPARAM data type. Rather than actually declare a new variable for the two messages, we'll hardcode the text "Player 1" for the first message and "Player 2" for the second message.

The final message we'll send to the control will set the current selection for the player value, essentially making a default value. To set this value we'll use CB_SETCURSEL as the message value, followed by 0 for the fourth parameter and a number (starting at 1) to indicate which item should be selected. We've now finished writing the dialog box initialization message handler, so we can finish this section by adding the other message handling code (WM_COMMAND), which will deal with button clicks.

By default, a new dialog box comes with OK and Cancel buttons. If the user clicks the Cancel button, we'll exit the dialog box immediately using the function EndDialog. In coding terms, we'll create a new if statement in the WM_COMMAND, which will check to see if the value of wParam is equal to the value of IDCANCEL. If the values are equal, we'll exit from the dialog box using the function EndDialog, supplying the passed hWnd value as the first parameter and a value of 0 for the dialog return value.

If the value of wParam is equal to the value of IDOK, then we must declare a new variable called player, which is of the long data type. In the declaration of the variable, we'll call the SendDlgItemMessage with the message type CB_GETCURSEL to return the currently selected item in the combo box into our newly declared variable, player. We place the value of the currently selected item in the player variable to easily index the different deathmatch player X, Y, and Z values in the map->details.deathmatch array. The values within the map->details.deathmatch array that must be set to have proper deathmatch starting positions are the map->details.deathmatch [player].xyz[0] (x-axis), map->details.deathmatch[player].xyz[1] (y-axis), and map->details.deathmatch[player].xyz[2] (z-axis). The other values within the structure can be ignored since we're only focusing on the start positions. Each value in the deathmatch structure will be set to the corresponding creation_coords.start.world_ variable for the individual axes, creating the start position.

After we set the coordinates for the deathmatch start position, we simply exit the dialog once again using the EndDialog function and specifying the passed hWnd value and a return value of 1, which will signify that the dialog box succeeded. Now that we've finished the code for the new dialog box, we can display the source code below.

```
LRESULT CALLBACK DMPositionDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_RESETCONTENT, 0, 0);
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Player 1");
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Player 2");
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_SETCURSEL, 0, 1);
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            long player = SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE,
                          CB_GETCURSEL, 0, 0);

            map->details.deathmatch[player].xyz[0] = creation_
                 coords.start.world_x;
            map->details.deathmatch[player].xyz[1] = creation_
                 coords.start.world_y;
            map->details.deathmatch[player].xyz[2] = creation_
                 coords.start.world_z;

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

To have the dialog box display on the screen, we'll have to update the WMLButtonUp function, adding a new if statement (at the bottom of the code) that will check to see if the value of creation_coords.type is equal to that of the value of ITEM_DM_POSITION. If the values are equal (which means we're currently creating/setting a deathmatch position), then we'll run the dialog box to specify the player we're setting the position for by calling the DialogBox macro, specifying the GlobalInstance as the first parameter and the resource ID as the second parameter, from within a call to MAKEINTRESOURCE. The third parameter is the parent window, which will be NULL (since we don't need it), then finally we'll specify our newly created DMPositionDlgProc dialog callback function cast as the DLGPROC data type. We've now finished the source code for the dialog box, and the update to the WMLButtonUp function is shown here:

```
else if (creation_coords.type == ITEM_DM_POSITION) DialogBox (GlobalInstance,
        MAKEINTRESOURCE(IDD_DM_POSITION), NULL, (DLGPROC)DMPositionDlgProc);
```

After updating the WMLButtonUp function, we have one last addition to make before we can begin discussing the code needed to add entities in a map. The final function we'll need to add will display the newly placed deathmatch positions in our level. This will be a short section because we've already covered the code needed to draw a single-player start position, so we can simply modify the code to suit our needs for deathmatch start positions. To begin the drawing process we'll create a new function above the already created Render function called DrawDeathMatchPositions, which, as the name implies, will draw the two deathmatch start positions. No parameters are necessary for this function to work properly.

Since the bulk of the drawing code has already been written by the DrawStartPosition function, we'll simply copy and paste the source code and modify it to our needs. In the case of the DrawDeathMatchPositions function, we'll modify the first line of code, which changes the color from 0.0, 0.0, 1.0 (full-intensity blue) to another unique color, 1.0, 0.0, 1.0, which will produce the color purple. Although we could technically use the blue color from the previous section, it would make it very difficult to distinguish between the two different types of start positions.

We'll need to modify the drawing code to draw both player 1's (index 0) and player 2's (index 1) starting positions. To modify the drawing code we simply replace the string map->details.single_player. with map->details.deathmatch[0]. By replacing that simple piece of code, we'll draw the first deathmatch starting position, which is half the battle. To add the second starting position, we'll copy and paste the source code for drawing the first start position and modify the index from 0 to 1 for each axis. Obviously we could simplify this process by creating a for loop to loop through the two indexes, but there would be no benefit in code size and with only two indexes there would be little point in worrying about it now. The rest of the drawing function (all two lines of it) can stay the same, which will finish the code for drawing multiplayer start positions. The source code for the newly created function is shown here:

```
void DrawDeathMatchPositions()
{
   glColor3f (1.0f, 0.0f, 1.0f);
   glBegin (GL_QUADS);
      glVertex2d (map->details.deathmatch[0].xyz[0], map->
                details.deathmatch[0].xyz[2]-0.01);
```

```
        glVertex2d (map->details.deathmatch[0].xyz[0]+0.01, map->
                    details.deathmatch[0].xyz[2]);
        glVertex2d (map->details.deathmatch[0].xyz[0], map->
                    details.deathmatch[0].xyz[2]+0.01);
        glVertex2d (map->details.deathmatch[0].xyz[0]-0.01, map->
                    details.deathmatch[0].xyz[2]);

        glVertex2d (map->details.deathmatch[1].xyz[0], map->
                    details.deathmatch[1].xyz[2]-0.01);
        glVertex2d (map->details.deathmatch[1].xyz[0]+0.01, map->
                    details.deathmatch[1].xyz[2]);
        glVertex2d (map->details.deathmatch[1].xyz[0], map->
                    details.deathmatch[1].xyz[2]+0.01);
        glVertex2d (map->details.deathmatch[1].xyz[0]-0.01, map->
                    details.deathmatch[1].xyz[2]);
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}
```

To make the deathmatch start positions display on the screen, we'll need to update the Render function and add a call to our newly created function DrawDeathMatchPositions. At this point you may be wondering why I haven't added both single-player and deathmatch starting positions to different layers. Simple, since there are only three objects total, there is no reason to concern ourselves with the extra coding required to take advantage of the layers. If we were to have 8+ deathmatch starting positions, then it would make sense to make a specific layer for them; however, when there are two, there is no point in the extra code!

Placing Entity Positions

Before we can begin discussing the code needed to insert entities, we must discuss what exactly an entity is. Many games interpret entities differently, so I'll explain what entities will be in our game engine. An *entity* will be a character model that is present in the given map. The entity may not necessarily be an enemy or "bad guy," but could be a non-player character (NPC) who walks around saying something stupid like, "I've lost my glasses" or something similar. Essentially, an entity is a "thing" within your map/game that can have its own unique characteristics. In theory, a gun/weapon could be considered an NPC because it has its own unique characteristics in how it works and fires; however, a gun/weapon cannot be of use without an entity picking it up and using it. That's the base definition right there. A more in-depth definition is rather difficult to explain, but I'll give it a whirl: If an entity must use another entity, the entity being used is not an entity but rather an item, because it must be used by or have interaction with another entity. In either case, we'll be adding both entities and items to the map editor, so it doesn't matter!

As with all types of “things” we can put into our map editor, we must add the item type to the map object/item type numeration in the map.h file, allowing us to easily identify the item type. In the case of entities we’ll add a new item type to the enumeration called ITEM_ENTITY. The updated source code for the map enumeration is provided below.

```
enum {OBJECTTYPE_WALL = 1,
      OBJECTTYPE_FLOOR,
      OBJECTTYPE_CEILING,
      ITEM_START_POSITION,
      ITEM_DM_POSITION,
      ITEM_ENTITY
     };
```

Along with adding a new button to the object/item type enumeration, we must add a new button to the application that will allow us to insert an entity into the level. Obviously this is an important step because we would have no easy method of inserting entities otherwise. The button we’ll create will be called bInsertEntity and will be declared in the globals section of the main source file. Of course when we declare the button we must also create the button itself by making a call to the CreateWindow function and have the returned value stored in the bInsertEntity variable. The only changes that must be made to the CreateWindow function (when copying and pasting the code from the previous call to create the button bPlaceDMPosition) are to the second parameter, which is the name of the window (in this new case it will be "Insert Entity"), and the fifth parameter, which specifies the default height of the button itself.

If you haven’t noticed by now, most of the calls to CreateWindow (at least when dealing with buttons) have all been fairly simple to understand, and require only minimal modifications to add new buttons into the editor. The source code for the new CreateWindow function call is provided below:

```
bInsertEntity = CreateWindow("BUTTON", "Insert Entity", WS_CHILD | WS_VISIBLE, 0,
                100+(DEFAULT_BUTTON_HEIGHT*10), DEFAULT_BUTTON_WIDTH,
                DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);
```

After creating the new button, we’ll modify the WMCommand function to handle the button click and prepare the editor for the creation of the entity. Moving to the WMCommand function, we’ll add a new else-if statement that will check the value of the passed lParam variable against an LPARAM cast bInsertEntity variable. If the values are equal, then we’ll set the current creation type to entity by setting the creation_coords.type variable to the value of ITEM_ENTITY, and call the function ShowSelectedButton to update the current button selections. We’ve finished modifying the WMCommand function, and can display the newly added source code here:

```
else if (lParam == (LPARAM)bInsertEntity)
{
   creation_coords.type = ITEM_ENTITY;
   ShowSelectedButton();
}
```

Now that we've added the code to handle the user pressing the bInsertEntity button, we'll move to the WMLButtonUp function and add the code needed to handle the actual insertion of the entity. Like the previous section where we inserted the deathmatch start positions, we'll have a dialog box pop up that displays the properties of the entity we're about to add. We'll place the new else-if statement underneath the deathmatch dialog box else-if statement within the WMLButtonUp function. The new else-if statement we'll add will check the value of creation_coords.type against the value of ITEM_ ENTITY. In the event the two values are equal, we'll call the DialogBox macro to start the new dialog box we'll create.

As we've done before, we'll simply copy and paste the call to DialogBox from the previous else-if statement and modify it accordingly. Only two parameters in the macro must be changed. The first change we'll make is to the second parameter, which is the ID of the dialog box. Following our naming convention, we'll call the new dialog box ID IDD_INSERT_ENTITY, which indicates that the dialog box functionality is intended to supply new entity properties. The other parameter we'll change in the macro is the fourth parameter, which specifies the dialog message handler. Since the name of the dialog box will be "Insert Entity," we'll call the message handler InsertEntityDlgProc to clearly indicate that it's a dialog procedure (message handler). After specifying the new message handler for the dialog box, we display the line of source code below.

```
else if (creation_coords.type == ITEM_ENTITY) DialogBox (GlobalInstance,
         MAKEINTRESOURCE(IDD_INSERT_ENTITY), NULL, (DLGPROC)InsertEntityDlgProc);
```

Now that we've made the appropriate changes to the else-if statement, allowing the map editor interface to place an entity upon releasing the left mouse button, we must create the actual dialog box to ensure our code will compile. Simply open the resource editor and create a new dialog box called **IDD_INSERT_ENTITY**. It is a good idea to name the dialog box **New Entry** in the caption/title category of the dialog properties (to the right of the resource ID). Within the dialog box we'll add one combo box control and three edit controls. We'll set the resource ID of the combo box to **IDC_ INSERT_ENTITY_TYPE** because it will specify two different types of entities (NPCs and enemies). The edit controls will be set to only accept numerical input (change the properties of each one individually) to accept this. The names of the edit controls will be IDC_INSERT_ENTITY_ HEALTH, IDC_INSERT_ENTITY_STRENGTH, and IDC_INSERT_ ENTITY_ARMOUR. The controls, as if you couldn't tell, are for the health, strength, and armor of each entity we put into the level. Along with the three edit controls, although not required, it's a good idea to add some static controls to the dialog box and place them next to the edit and combo box controls. Each static control should be named appropriately for the control it's next to. In the case of the combo box, the static control should be named "Type:". By adding the static controls, we'll help the user understand each control and in turn make our map editor slightly easier to use. Figure 7.1 displays how the dialog box could look.

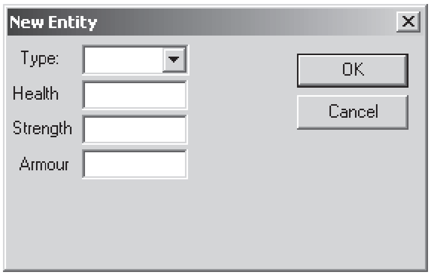


Figure 7.1: New Entity dialog box

We can now begin writing the code for the dialog box message handler (InsertEntityDlgProc). This callback function will be positioned above the WMLButtonUp function, where it will be called from. As always, we can copy the function definition from the function above (which happens to be DMPositionDlgProc) and change the name to save a few keystrokes! Because the information in this dialog is essential, we must take advantage of both the WM_INITDIALOG message to initialize the controls and the WM_COMMAND message to handle any button clicks that may happen.

The first message we'll handle in our newly created message handler is WM_INITDIALOG, which as mentioned before will initialize all the controls. Starting with the combo box, the first thing we'll do is send the CB_RESETCONTENT message (using SendDlgItemMessage) to reset the contents, ensuring there is no overlap of data there. Next we'll send the string CB_ADDSTRING to add one string named "Joe". In case you're wondering, Joe is the name of our enemy. I realize most games will have really awesome names, but this is just a simple introduction to item placement and, to a lesser extent, game design. Next we'll set the three default edit control values, which can all be done using the function SetDlgItemText. The initial value for IDC_INSERT_ENTITY_HEALTH will be 100, IDC_INSERT_ENTITY_STRENGTH will be 10, and IDC_INSERT_ENTITY_ARMOUR will be 0. The SendDlgItemMessage and SetDlgItemText functions have been discussed many times, so we'll skip them here. We've now finished the code for the WM_INITDIALOG message and can begin discussing the code required for the WM_COMMAND message.

The WM_COMMAND message will be somewhat larger because we must retrieve all the values from each control and convert them into integer-based values. To keep things simple we'll first check to see if the user pressed the default Cancel button for the dialog. This can be done by checking to see if the passed wParam parameter is equal to IDCANCEL. If the values are equal, then we'll exit the dialog box using the EndDialog function with the return value of 0. If the value of wParam is equal to the value of IDOK, then the user has pressed the OK button and we must process the information that's in each control. Before we can retrieve the information from each control we must declare several new variables, including a string of 500 characters called temp and four variables of the long data type (type, health, strength, and armour).

Following the new local variable declarations, we'll get the currently selected combo box item by calling the SendDlgItemMessage and specifying CB_GETCURSEL as the message type, then store the returned value in the newly declared type variable. After retrieving the currently selected entity type, we'll get the value the user entered in the health edit box. To get the value we'll call the GetDlgItemText function and specify the edit control resource ID (IDC_INSERT_ENTITY_HEALTH), which will return the string value to the third parameter (which is our temp variable). We'll specify the maximum size in bytes of the string to copy (max is 500) as the fourth parameter. Upon execution, the string version of the integer value will be copied to the temp variable. We'll use a simple call to sscanf, specifying the temp variable as the buffer we want to read the formatted data from and a standard integer format as the second parameter. We'll use the address of the health variable to store the integer value into the health variable.

To get the strength and armour values, we simply follow the same process as getting the health variable, and grab the string values from the IDC_INSERT_ENTITY_STRENGTH and IDC_INSERT_ENTITY_ARMOUR controls, format the integer values from the strings, and place the values into the appropriate variables. Once each value has been taken from the controls, we'll simply call the MAP method InsertEntity, supplying the type variable as the first parameter and the XYZ world coordinates of the entity located in the variables creation_coords.start.world_x, creation_coords.start.world_y, and creation_coords.start.world_z. The fifth, sixth, and seventh parameters are used for the angles of the entity. This feature isn't going to be used at the moment so we can simply use 0s for all three values. The last three variables of the InsertEntity function are the health, strength, and armour of the entity, which we can specify using the local variables we declared earlier. The final thing we must do to finish the InsertEntityDlgProc callback is simply exit the dialog by calling the EndDialog function and using a value of 1 to represent the second parameter (the first being the passed hWnd variable). In case you're wondering, the 1 value represents that the dialog box succeeded. The source code for the newly created InsertEntityDlgProc callback follows.

```
LRESULT CALLBACK InsertEntityDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE, CB_RESETCONTENT, 0, 0);
         SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Joe");
         SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE, CB_SETCURSEL, 0, 1);

         SetDlgItemText (hWnd, IDC_INSERT_ENTITY_HEALTH, "100");
         SetDlgItemText (hWnd, IDC_INSERT_ENTITY_STRENGTH, "10");
         SetDlgItemText (hWnd, IDC_INSERT_ENTITY_ARMOUR, "0");
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char temp[500];
            long type;
            long health;
            long strength;
            long armour;

            type = SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE,
                   CB_GETCURSEL, 0, 0);

            GetDlgItemText (hWnd, IDC_INSERT_ENTITY_HEALTH, temp, 500);
            sscanf (temp, "%i", &health);

            GetDlgItemText (hWnd, IDC_INSERT_ENTITY_STRENGTH, temp, 500);
            sscanf (temp, "%i", &strength);

            GetDlgItemText (hWnd, IDC_INSERT_ENTITY_ARMOUR, temp, 500);
            sscanf (temp, "%i", &armour);

            map->InsertEntity(type, creation_coords.start.world_x,
                 creation_coords.start.world_y, creation_coords.start.world_z,
                 0,0,0, health, strength, armour);
            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

Now that we've added the ability to insert new entities into our map, we'll need to write the code to draw them on the screen. Since we've already written the code for placing a single-player start position, we can simply copy and paste the rendering code into a new function called DrawEntities. Much like the previous types of "things" drawn within the game, all entities must be drawn using a distinct color to easily identify them as entities. For this reason, we'll change the color from 0,0,1 (full-intensity blue) to 0,1,1, which sets the color to cyan.

After changing the entity color to the unique color cyan, we'll add a new for loop that will loop from 0 to the value of map->header.max_entities. Within each iteration of the loop, we'll modify each glVertex function call to specify the current X, Y, and Z coordinates of the current entity (i.e., map->entity[i].xyz[0], map->entity[i].xyz[1], and map->entity[i].xyz[2]). The vertex data will continue to have the offsets present in the function calls to allow each entity to be drawn as a diamond shape as opposed to a simple dot. Depending on your screen resolution, the diamonds may end up looking like dots, but this can't be helped unless you get ambitious and decide to write a calculation to fix shape distortion. This is beyond the scope of the book, but it is something to definitely keep in mind if you want to expand beyond the current limitations of the game engine!

The source code for the newly created DrawEntities function is written below for you to examine.

```
void DrawEntities()
{
   glColor3f (0.0f, 1.0f, 1.0f);
   glBegin (GL_QUADS);
      for (long i = 0; i < map->header.max_entities; i++)
      {
         glVertex2d (map->entity[i].xyz[0], map->entity[i].xyz[2]-0.01);
         glVertex2d (map->entity[i].xyz[0]+0.01, map->entity[i].xyz[2]);
         glVertex2d (map->entity[i].xyz[0], map->entity[i].xyz[2]+0.01);
         glVertex2d (map->entity[i].xyz[0]-0.01, map->entity[i].xyz[2]);
      }
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}
```

Obviously the last operation we must perform is calling the DrawEntities function to draw the newly created entities. To do this we'll modify the Render function to add the function call to DrawEntities; however, since there can be multiple entities in our map we should add a new layer name for entities before adding the Render modification. To add the new layer item, we'll open the menu resource and add a new menu item with the caption "Entity" and a menu resource of ID_LAYER_ENTITY. The new menu item will be checked by default. Moving back to the source file, whenever we add new layering options, we must always add a new Boolean variable to the LAYER structure to control whether the specified option is checked or not. The new variable we'll add to the structure will be called draw_entity. There is no need to set a default value to this variable since it's automatically set when the program initially begins.

As with every menu item, we must add a new if statement to the WMCommand function to check/uncheck this Layers menu item. When the user selects the newly created Layers menu item (Entities), we'll set the checkmark status to the opposite of its current state and set the value of the layer.draw_entity variable to its opposite value. The source code for the new else-if statement is shown below.

```
else if (wParam == ID_LAYERS_ENTITY)
{
   if (layer.draw_entity) CheckMenuItem (Menu, ID_LAYERS_ENTITY, MF_UNCHECKED);
   else CheckMenuItem (Menu, ID_LAYERS_ENTITY, MF_CHECKED);
   layer.draw_entity = !layer.draw_entity;
}
```

Finishing this huge detour, we'll finally add the DrawEntities function call below the DrawDeathMatchPositions function call. To ensure the entities are only drawn when the layer is enabled, we'll check the value of layer.draw_entity. If the value is true, we'll call DrawEntities. This concludes the section on placing entities within the map.

Placing Items

This section will discuss inserting items into the map. An *item* will be anything the user can pick up or interact with that does not move on its own. For instance, a gun would be a good example of an item. The user can pick the gun up, but it cannot move on its own. Before we can write the code in the main source file, we must add the lower-level code in the MAP class to physically insert the item into the level. This is a simple process of dynamically allocating memory for the item pointer we've already declared within the class. The new method we'll add is called InsertItem, which has a bool return data type. There are six parameters in the method, two of which have default values assigned to them. The first three variables (x, y, z) are all of the GLdouble data type and specify the X, Y, and Z coordinates of the item we're placing. The next three variables are all of the GLint data type and specify the type of item we're placing (type), whether the item should be regenerated (respawn_wait), and how long the respawn time (respawn_time) should be.

As with all insertion methods within our MAP class, we'll create a new variable to store the passed data. In the case of items, we'll create a new variable called new_item, which is of the MAP_ITEM data type. We'll also declare a new variable called rgb, which will hold our selectable RGB value for the item, allowing us to select it later. After declaring the rgb variable, we'll call the GenerateColor function to generate the unique selectable RGB color and store the returned value to the newly declared rgb variable. After storing the new RGB value in the variable, we'll simply call the macros GetRValue, GetGValue, and GetBValue, supplying the rgb variable as the single parameter, and store the returned color components in the new_item.select_rgb array, starting with index 0 and finishing with index 2.

After copying the selectable RGB color components to the new_item.select_rgb variable we'll copy the passed x, y, and z variables to the appropriate new_item.xyz array index. The xyz array will store the X position in index 0, Y position in index 1, and Z position in index 2. After copying the XYZ positions, we'll set the new_item.type variable to the value of the passed type variable. To finish filling the structure we'll store the passed respawn_wait and respawn_time variables in their respective new_item.respawn_wait and new_item.respawn_time equivalents.

With the new_item structure filled in, we can follow the simple rules we've followed in the past when dynamically allocating memory for arrays. First, we check to see if the header.max_items variable is 0, which is the default, and allocate memory for the one record. In the event there are records in the array, we'll declare a new variable called temp, allocate memory to it, then dump the contents of the item array to it. Then we'll destroy the contents of the item array and reallocate memory to the variable with space for another record. We then copy the contents of the temp array back to the original item array, destroy the temp variable, and exit the else-if statement. We store the new_item variable in item[header.max_items], increment the value of header.max_items to indicate we've got a new item in the list, and return a true value indicating the function completed successfully. As this is the only return value for the function, it's not important to check its return value. In the event we actually wrote several return values, then the return value of true would be important, but that's not the case here! Now that we've finished discussing the InsertItem method of the MAP class, we can display the source code below.

```
bool MAP::InsertItem(GLdouble x, GLdouble y, GLdouble z, GLint type, GLint
                     respawn_wait, GLint respawn_time)
{
   MAP_ITEM   new_item;
   long       rgb;

   rgb                    = GenerateColor();
   new_item.select_rgb[0] = GetRValue(rgb);
   new_item.select_rgb[1] = GetGValue(rgb);
   new_item.select_rgb[2] = GetBValue(rgb);

   new_item.xyz[0]        = x;
   new_item.xyz[1]        = y;
   new_item.xyz[2]        = z;
   new_item.type          = type;
   new_item.respawn_wait  = respawn_wait;
   new_item.respawn_time  = respawn_time;

   if (header.max_items == 0) item = new MAP_ITEM[header.max_items+1];
   else
   {
      MAP_ITEM      *temp;

      temp = new MAP_ITEM[header.max_items+1];
      for (long i = 0; i < header.max_items; i++) temp[i] = item[i];

      delete [] item;
      item = new MAP_ITEM[header.max_items+2];
      for (i = 0; i < header.max_items; i++) item[i] = temp[i];

      delete temp;
   }
   item[header.max_items] = new_item;
   header.max_items++;

   return (true);
}
```

Before we finish with the map.h file, we'll add another object type to the type enumeration. The new value, ITEM, will be the default type used when declaring items. I guess if we were to follow the naming convention we've used so far, the appropriate name would be ITEM_ITEM; however, I don't see the logic in having something called ITEM_ITEM. It seems silly to me, so we'll simplify the name to ITEM. The source code for the new type enumeration is provided below.

```
enum {
      OBJECTTYPE_WALL = 1,
      OBJECTTYPE_FLOOR,
      OBJECTTYPE_CEILING,
      ITEM_START_POSITION,
      ITEM_DM_POSITION,
      ITEM_ENTITY,
      ITEM
     };
```

With the low-level code added to the MAP class, we can move back to the main source file and begin the usual process of inserting "things" into the map editor code. To begin writing code, the first thing we'll do is declare a new global variable of the HWND data type called bInsertItem. This variable will be the button used to insert items into our map. As with all buttons we declare, we must actually create it using the CreateWindow function to make the button visible. Much like the previous calls to CreateWindow, the only changes that must be performed are to the return variable, which must be bInsertItem, the second parameter, which specifies the caption of the button ("Insert Item"), and of course the fifth parameter, which is the starting

Y coordinate of the button. For simplicity we'll follow the current trend and add 2 to the current number to space the button farther down the screen. In case you're wondering, this map editor is being designed for a resolution of 800x600. Nowadays, the 640x480 resolution is too low when compared to the high-resolution (1280x1024 and greater) capabilities of most monitors. The source code for the new call to CreateWindow is provided below.

```
bInsertItem = CreateWindow("BUTTON", "Insert Item", WS_CHILD | WS_VISIBLE, 0,
              100+(DEFAULT_BUTTON_HEIGHT*12), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);
```

After creating the new button we must modify the WMCommand function to process any button clicks that may occur. To do this, we'll add another else-if clause to the bottom of our WMCommand function that will check to see if the value of lParam is equal to the value of bInsertItem cast as an LPARAM data type. In the event the values are equal, we'll set creation_ coords.type to ITEM and call the function ShowSelectedButton to update the button captions. The source code for the small WMCommand modification is shown below.

```
else if (lParam == (LPARAM)bInsertItem)
{
   creation_coords.type = ITEM;
   ShowSelectedButton();
}
```

If we call the ShowSelectedButton function in its current state, not much will happen when the user presses the Insert Item button. With this in mind, we'll modify the function and add the appropriate button functionality into it to handle the newly created function. The first modification we'll make to the function will be in the long list of function calls to SetWindowText. We'll add a new call to SetWindowText, specifying the button (bInsertItem) as the first parameter and the default caption, "Insert Item", as the second parameter. The other modification to the function we'll make is an addition to the case statement, adding the new ITEM type to the statement and making another call to SetWindowText, which will be the highlighted text used when the user clicks the button. The first parameter will once again be the same as that of the previous one, but the text will be simply "*Item*", indicating it's selected. This wraps up the changes to ShowSelectedButton and we can move on to the changes needed to handle the placement of the new item when the user lets go of the left mouse button.

In the WMLButtonUp function we'll add a new else-if statement to check to see if the value of creation_coords.type is equal to the value of ITEM. If the values are equal, then we'll launch a new dialog box called IDD_ INSERT_ITEM, which we'll create in just a moment. The message callback function for this new dialog box will be called InsertItemDlgProc because of the functionality it entails. The new addition to the WMLButtonUp function is given below.

```
else if (creation_coords.type == ITEM) DialogBox(GlobalInstance,
        MAKEINTRESOURCE(IDD_INSERT_ITEM), NULL, (DLGPROC)InsertItemDlgProc);
```

Before we write the callback function to handle the messages of the new dialog box, we must create the dialog box itself. There's no sense in writing the code before we know the names of the controls we are trying to write the code for! In any case, we'll create a new dialog box with a resource ID called IDD_INSERT_ITEM. The caption for the dialog box will be "Insert Item" to keep things simple. Although it's not necessary, it's a good idea to change the properties of the dialog box to center it in the screen. I prefer having the dialogs centered in the screen because it helps the user navigate easily and looks more professional than having dialogs appear randomly on the screen.

This new dialog box will require one combo box, called IDC_INSERT_ITEM_TYPE, and two edit boxes (with number formatting), called IDC_INSERT_ITEM_RESPAWN_TIME and IDC_INSERT_ITEM_RESPAWN_WAIT. Although it's not required, I've added several static controls with captions filled in to name the options appropriately. When making a map editor, these captions are not required, but it really does add to the user interface if you include these little details!

Moving to the code end of things, the first message we'll handle is the typical WM_INITDIALOG, which as we know is sent when the dialog box initializes. Since we've got these three controls, we must obviously initialize them with proper data to ensure that no remnants of previous dialog box calls appear. The first message we'll send will clear the data (CB_RESETCONTENT) from the combo box. The next message will add a string, in our case the item name "Gun", using the message CB_ADDSTRING. The final message we'll send will be CB_SETCURSEL to set the currently selected item to the first and only item.

After setting the combo box, we'll set the default values for both edit controls by using the function SetDlgItemText and specifying 0 as the text string for each parameter.

The WM_COMMAND message is slightly more involved in that it requires the programmer to get the current data and insert it into the current item list. Rather than process the OK button click first, we'll process the Cancel button first because it only requires one line of code, which checks the value of wParam against IDCANCEL. If they are equal, then the EndDialog function is called with a return value of 0 specified as the second parameter. In the event the wParam variable is equal to the value IDOK, then we'll declare several new variables and retrieve the data.

The first variable we'll declare is an array of 500 characters called temp. As we've discussed before, we use this variable to retrieve the strings from each edit control. The next three variables (type, respawn_wait, and respawn_time) are all of the long data type. After declaring the new local variables, the first value we'll get will be the currently selected item (there can only be one at this point). Using the CB_GETCURSEL message and supplying the IDC_INSERT_ITEM_TYPE resource ID, we'll return the value to the variable type. Both the respawn_time and respawn_wait variables will be retrieved using the GetDlgItemText function, specifying the specific resource ID and temporary string/buffer (temp). Once the strings are in the temp variable, we'll use the sscanf function to format the data into the appropriate long form equivalent.

With all the data retrieved from the dialog box, we can call the new map->InsertItem function, specifying the X, Y, and Z coordinates (creation_coords.finish.world_x, creation_coords.finish.world_y, and creation_coords.finish.world_z), the type of item to insert (type variable), the respawn time (respawn_time), and the respawn wait (respawn_wait). The new item will be inserted into the map and things are dandy! We can now exit the dialog box by calling the macro EndDialog and specifying the second parameter as 1 to indicate the function was successful. The source code for the dialog box is provided below.

```
LRESULT CALLBACK InsertItemDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE, CB_RESETCONTENT, 0, 0);
         SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Gun");
         SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE, CB_SETCURSEL, 0, 1);
         SetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_TIME, "0");
         SetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_WAIT, "0");
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char temp[500];
            long type;
            long respawn_wait;
            long respawn_time;

            type = SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE,
                   CB_GETCURSEL, 0, 0);
            GetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_WAIT, temp, 500);
            sscanf (temp, "%i", &respawn_wait);

            GetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_TIME, temp, 500);
            sscanf (temp, "%i", &respawn_time);

            map->InsertItem (creation_coords.finish.world_x, creation_
                coords.finish.world_y, creation_coords.finish.world_z, type,
                respawn_wait, respawn_time);
            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

As we did in the previous section, we must add a new layer option called Item, which will control whether or not all the items in the map will be drawn. To begin this process we'll add a new draw_item variable to the LAYER structure. Like all variables within the LAYER structure, this variable does not require initialization because it's set to true by default due to the memset call at the beginning of the program. In addition to adding a new variable to the LAYER structure, we'll add a new menu item called "Item" to the Layers menu in the resource script. The menu item will be checked by default to ensure that both draw_item and menu item are synchronized to the proper start settings. When we create the new menu item, we'll set the resource ID to ID_LAYERS_ITEM if the ID is different.

To allow the user to actively select whether or not the items in the map are drawn we must add a new else-if statement to the WMCommand that handles the appropriate button click for this newly created menu item. The new else-if statement will check to see if the user clicked the newly created Item menu item by determining if the value of wParam is equal to ID_LAYERS_ITEM. If the user did click the menu item, then we'll either check or uncheck the item based on the value of layer.draw_item, and make the value of layer.draw_item different from its previous value (change from 1 to 0 or 0 to 1). The source code for the new else-if statement is included below for you to study.

```
else if (wParam == ID_LAYERS_ITEM)
{
   if (layer.draw_item) CheckMenuItem (Menu, ID_LAYERS_ITEM, MF_UNCHECKED);
   else CheckMenuItem (Menu, ID_LAYERS_ITEM, MF_CHECKED);
   layer.draw_item = !layer.draw_item;
}
```

Now that we've added the layering capabilities for items we can update the Render function, adding a new line below the DrawEntities line that will check whether the value of layer.draw_item is true, then call the function DrawItems. If you were to compile the source code right now you'd probably notice there's no function called DrawItems. Obviously this is a necessary function because there wouldn't be any purpose in placing items without being able to see where they are. For this reason we'll create the new function DrawItems above the Render function.

The DrawItems function does not have a return value nor does it accept any parameters. Essentially it's a dull function that draws all the items within the map in the color yellow. To begin the code in the DrawItems function, we'll change the color to yellow by calling glColor3f and specifying 1,1,0 as the RGB values. Each object will be drawn using the QUADS type to produce our diamond-shaped object. We'll begin making the shape with a call to glBegin, specifying GL_QUADS as the primitive type. Next we'll loop from 0 to the value of map->header.max_items, drawing the diamond's four points using the values of map->item[i].xyz[0], map->item [i].xyz[1], and map->item[i].xyz[2] as reference points. This loop will draw all the items in the level using the diamond shape. Once the loop finishes we'll simply call glEnd to notify OpenGL we're finished building the object and reset the color to the default white (1,1,1) by calling glColor3f. The source code for the DrawItems function is provided below for you to study.

```
void DrawItems()
{
   glColor3f (1.0f, 1.0f, 0.0f);
   glBegin (GL_QUADS);
      for (long i = 0; i < map->header.max_items; i++)
      {
         glVertex2d (map->item[i].xyz[0], map->item[i].xyz[2]-0.01);
         glVertex2d (map->item[i].xyz[0]+0.01, map->item[i].xyz[2]);
         glVertex2d (map->item[i].xyz[0], map->item[i].xyz[2]+0.01);
         glVertex2d (map->item[i].xyz[0]-0.01, map->item[i].xyz[2]);
      }
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}
```

Inserting Sounds

Inserting sound effects into a level is not an absolute necessity, but it does add a great deal of realism to a game. Imagine you were playing a game in which you're walking through a sewer system and it is completely silent. Although it may seem fine at first, the game would be greatly enhanced by having sound effects play throughout the level. You could place sound effects at specific locations that would pan throughout the speakers, such as a dripping noise that moves as you turn left and right. Of course this is a simple example of how we would place sounds, but you get the idea.

Before we start writing code to the main source file we must update the map.h file to actually insert a sound effect. We'll create a new method within the MAP class called InsertSound (very appropriate, huh?) that will insert the sound data. This new method has a total of seven parameters in the declaration. The first three parameters, x, y, and z, are all of the GLdouble data type and store the position of the sound effect in 3D space. The next parameter is an array of characters called filename, which, as the name describes, stores the filename of the sound effect. The final three parameters, xa, ya, and za, are all of the GLfloat data type and store the sound direction in degrees. These three variables will have overloaded values of 0.0f, allowing you to specify the first four variables. The directional information can be input manually if you want to see how the direction can affect the playback of the sound. The new method will have a return type of bool.

After defining the new method, we can begin writing the code to interface with it. The first thing we'll do is declare two new local variables. The first variable will be of the MAP_SOUND data type and will be called new_sound. This variable will store the input parameter data in a tidy structure to insert into the sound structure. The other variable we'll declare is called rgb and is of the long data type. This variable stores the 24-bit selectable RGB value for the sound. This will allow us to select the object once the selection code has been written.

Now that the local variables have been declared we can begin the main code for the InsertSound method. The first thing we'll do is call the GenerateColor function and store the returned value in the rgb variable. Next we'll set the individual R, G, and B values using the macros GetRValue, GetGValue, and GetBValue and supplying the rgb variable as the value. The returned values will be stored in the new_sound.select_rgb array, starting at index 0. Now that the sound has a unique selectable RGB value, we can fill in the rest of the MAP_SOUND structure with the appropriate values. Starting with the first index (0) in the new_sound.angle array, we'll specify xa, then ya, and finally za. This will set the angles of the sound to the user-defined values. In the event we use the overloaded values, angle array will be 0.0f.

Following the new_sound.angle array, we'll use the strcpy function to copy the input filename into the new_sound.filename variable. This will copy the filename so we may load it later. After copying the filename string, we must set the new_sound.id variable to 0 as a default value. Finally, we'll set the new_sound.xyz array to the values of x, y, and z, starting at index 0. With the new_sound variable filled in we can begin the insertion process. As with all dynamic allocation we've done in this book, the first thing we'll do is check the value of the variable header.max_sounds. If the value is equal to 0, then we'll allocate memory in the sound pointer for one index. If the value of header.max_sounds is greater than 0, we'll declare another local variable called temp, which is a pointer of the MAP_SOUND data type. The temp variable will be allocated memory of the value of header.max_sounds plus 1 to store the data in the sound array. After allocating the memory we'll copy the data using a simple for loop, looping from 0 to the value header.max_ sounds. With each iteration of the loop we'll copy the contents of sound[i] to temp[i].

After copying the data to the temp array, we can release the memory from the sounds array and reallocate the memory for the array using the value header.max_sounds+2, adding one new index as well as the one extra index as an emergency buffer. Now that the sounds array has been allocated with one new index, we can copy the content back from the temp array to the sounds array. Following the same ideas as previously discussed, we'll simply use a for loop to loop from 0 to the value of header.max_sounds, copying the value of temp[i] back into sound[i]. With the data copied back to the sound array from the temp array we can free the temp array memory and set it to NULL. This will conclude the dynamically allocated section of the InsertSound method. The last thing we must do is set the last sound index (sound[header.max_sounds]) to the value of new_sound (the variable we declare at the beginning of the method). This will set the last index regardless of whether we allocated a single index or dynamically allocated a new one to the last index. Then we increment the header.max_sounds variable to update the current count of sounds. The final step is to return a value of true to finish the function! The source code for the InsertSound method is provided below.

```
bool MAP::InsertSound(GLdouble x, GLdouble y, GLdouble z, char *filename,
        GLfloat xa, GLfloat ya, GLfloat za)
{
   MAP_SOUND  new_sound;
   long       rgb;

   rgb = GenerateColor();
   new_sound.select_rgb[0] = GetRValue(rgb);
   new_sound.select_rgb[1] = GetGValue(rgb);
   new_sound.select_rgb[2] = GetBValue(rgb);

   new_sound.angle[0]     = xa;
   new_sound.angle[1]     = ya;
   new_sound.angle[2]     = za;
   strcpy (new_sound.filename, filename);
   new_sound.id           = 0;
   new_sound.xyz[0]       = x;
   new_sound.xyz[1]       = y;
   new_sound.xyz[2]       = z;

   if (header.max_sounds == 0) sound = new MAP_SOUND[header.max_sounds+1];
   else
   {
      MAP_SOUND *temp;
```

```
        temp = new MAP_SOUND[header.max_sounds+1];
        for (long i = 0; i < header.max_sounds; i++) temp[i] = sound[i];

        delete [] sound;
        sound = new MAP_SOUND[header.max_sounds+2];
        for (i = 0; i < header.max_sounds; i++) sound[i] = temp[i];

        delete [] temp;
        temp = NULL;
    }

    sound[header.max_sounds] = new_sound;
    header.max_sounds++;

    return (true);
}
```

Now that we've written the low-level code to insert a sound into the map, we can write the code to insert the sound in the map editor. To begin, we'll add a new type to the object type enumeration in the map.h file. The new enumeration will be called SOUND. Next we'll declare a new global variable in the main source file called bInsertSound, which is of the HWND data type. This bInsertSound variable will contain the Insert Sound button, which the user will have to press in order to insert a sound in the map. With the bInsertSound variable declared, we can update the main function with the new button creation source code. We'll use the function CreateWindow to create the new button as we've done in the past. To keep things in order, we'll add the new CreateWindow function call to the bottom of the function calls (just below the bInsertItem creation call).

To keep things simple, we'll simply copy and paste the button creation code from the bInsertItem creation code above and modify it accordingly. One of the few modifications we'll need to make to the function call is to the return variable. Obviously we'll change the return variable to bInsertSound from its original bInsertItem. Next, we'll change the button caption to "Insert Sound" for obvious reasons! The final modification we'll make to the CreateWindow function call is the starting Y coordinate. The spacing number (12) will be incremented by 2 to position the window appropriately following the standard we've been using throughout the map editor. The source code for the button creation function follows.

```
bInsertSound = CreateWindow("BUTTON", "Insert Sound", WS_CHILD | WS_VISIBLE, 0,
               100+(DEFAULT_BUTTON_HEIGHT*14), DEFAULT_BUTTON_WIDTH,
               DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);
```

With the bInsertSound button now created in the map editor we must update the ShowSelectedButton function to reflect the possibility of the user pressing the bInsertSound button. To update the function we'll simply add a line to the bottom of the SetWindowText group of calls, supplying the bInsert-Sound variable as the first parameter and the text "Insert Sound" as the second parameter. This will be the default value for the button when the user presses something other than the Insert Sound button. We must also edit the case statement to add the SOUND type as an option. Within the SOUND case we'll call the SetWindowText function, specifying bInsertSound as the first parameter and the text "*Sound*". When the user presses the bInsert-Sound button, the button caption will change to "*Sound*" instead of its default text.

Now that we've updated the ShowSelectedButton function to display the appropriate caption, we'll update the WMCommand function to handle the initial bInsertSound button click. To update the WMCommand function, we'll add a new else-if clause that checks to see if the value of lParam is equal to the cast LPARAM value of bInsertSound. In the event the values are equal, we'll set the creation_coords.type variable to the sound enumeration SOUND. Then we can call the ShowSelectedButton function to update the button captions. The updated source code is provided below.

```
else if (lParam == (LPARAM)bInsertSound)
{
   creation_coords.type = SOUND;
   ShowSelectedButton();
}
```

After adding the functionality to the bInsertSound button, we should implement the main functionality of inserting the sound. Before we write the code to insert the sound, we'll create a new dialog box that will be used to display the settings for the sound you want to insert. The only setting we'll have available to the user is the filename of the sound we want to play. Although we have the ability to use the XYZ angles of the sound, we'll save those values for other internal uses. The new dialog box we'll create will have a resource ID called IDD_INSERT_SOUND with the caption "Insert Sound." For consistency I recommend checking the Center option in the dialog properties to center the dialog box in the screen upon initial creation. Along with centering the dialog box, I resized it to give a more finished look.

Inside the dialog box we'll insert an edit box control with the resource ID IDC_INSERTSOUND_FILENAME. This control will be used to collect the filename of the sound file that is to be played at the specified location. I have also added an optional static text box with the caption "Sound File" to make the dialog slightly more user-friendly. Figure 7.2 shows just one way you could create this dialog box. The dialog box could be designed in many different ways, but this method at least gives you an idea of how I created it for the example.

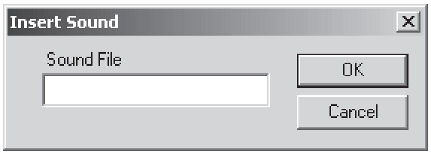


Figure 7.2: The dialog box

With the dialog box created we can begin writing the code to handle the messages sent to and from the dialog box. The message handler for the dialog box will follow our simple naming convention and be called InsertSoundDlgProc. The parameters will be the same as those for any other dialog box procedure or message handler, so we can simply copy and paste the declaration from another dialog procedure and change its name. By this point if you had written everything manually as opposed to copying and pasting the code you'd probably need a new keyboard or have very sore fingers!

To begin the code for the InsertSoundDlgProc function we'll create a case statement to handle the different messages being sent to the dialog box. The first message we'll handle is the WM_INITDIALOG message, which is sent when the dialog box is first initialized. Within the message we'll call the SetDlgItemText function to set a default filename for the sound. The first parameter of the SetDlgItemText function specifies the window to which we're sending the data, which in this case is hWnd. The second parameter is the resource ID of the control to which the text is to be sent, which is IDC_INSERTSOUND_FILENAME. The final parameter is the text "my_sound.wav", which will set the filename for the sound file. The only downside to having the user enter the filename manually is that it could be located in some odd path (e.g., C:\Documents and Settings\Chris\My Sounds\80s Beeps\Zaaapp.wav). Unfortunately this is a chance we'll have to take when giving the user the ability to enter the data. This concludes the code to handle the WM_INITDIALOG message.

The next message we'll handle is the WM_COMMAND message, which will allow us to handle button clicks within the dialog box. Since we used a default dialog box, there should be the two default buttons, OK and Cancel. The first button click we'll handle is the Cancel button, because it doesn't require any major code. Within the WM_COMMAND function, we'll simply check to see if the variables wParam and IDCANCEL are equal. If they are equal, then we'll simply exit the dialog by calling EndDialog and supplying the current window (hWnd) and a return value of 0 to indicate the dialog box failed. If the value of wParam is equal to IDOK, then the user has pressed the OK button and we begin the sound insertion process. The first thing we must do is declare a new array of 500 characters called filename, which will store the string input from the user as the sound filename. Next we'll grab the text from the IDC_INSERTSOUND_FILENAME text box control using the function GetDlgItemText and supplying the current window (hWnd) as the first parameter, the IDC_INSERTSOUND_FILENAME resource ID as the second parameter, the filename variable as the third parameter, and the maximum size allowed to be returned in characters (500). The input data will be retrieved from the specified ID to the specified string up to the maximum amount of characters allowed. Sounds simple enough, right?

After retrieving the string we simply call the map->InsertSound method and supply the current X coordinate (creation_coords.finish.world_x), the Y coordinate (0), the Z coordinate (creation_coords.finish.world_z), and the filename variable. We don't need to worry about the other variables in the method because they all have overloaded values. This will insert the sound at the current mouse position with the filename specified by the user. After inserting the sound file, simply call the EndDialog function, specifying the current window (hWnd) as the first parameter and 1 as the second parameter to return a successful return value. At the bottom of the dialog procedure, return a value of 0 regardless of which button was pressed to allow the dialog box to handle messages properly. We've finished entering the source code for the InsertSoundDlgProc dialog box and can display the source code below.

```
LRESULT CALLBACK InsertSoundDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG: SetDlgItemText (hWnd, IDC_INSERTSOUND_FILENAME,
                          "my_sound.wav"); break;

      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char filename[500];
            GetDlgItemText (hWnd, IDC_INSERTSOUND_FILENAME, filename, 500);

            map->InsertSound (creation_coords.finish.world_x, 0, creation_
                 coords.finish.world_z, filename);

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

With the dialog box now ready to insert sounds, we can update the WMLButtonUp function to physically start the dialog box. The modification to the

WMLButtonUp function is simple. We'll add a small else-if clause to the statement to check whether the value of the creation_coords.type variable is equal to the value of SOUND, then we'll call the DialogBox macro with the parameters of the program instance (GlobalInstance), the dialog resource we want to run (MAKEINTRESOURCE(IDD_INSERTSOUND)), NULL for the third parameter, and a DLGPROC cast InsertSoundDlgProc. This completes the majority of the sound insertion process; however, it doesn't resolve one final issue — drawing!

Drawing the inserted sounds is a key issue we should address before we move on to the final section. Obviously, this is something that cannot be ignored because there would be no point in placing sounds within a map if you weren't able to see where the heck you put them! For this reason we'll quickly write a function called DrawSounds to draw the locations of the sounds in the current map. Because of the way we draw the objects in our map, we must move the code to just above the Render function. The easiest solution to writing the drawing code is to simply copy and paste the code from the DrawItems function and modify it accordingly. Only three modifications need to be made to the code, including the color, which is designated as 0.0f, 1.0f, 0.0f, or green. The second modification we need is to change the maximum value in the loop from map->header.max_items to map->header.max_sounds to obviously loop through the values of the sounds and not the values of the items. The final modification is to the array name we use when specifying the coordinates of the vertices. In the DrawItems function we specify the map->item array; however, simple logic tells us we want to use the map->sound array for each of the vertex points. These simple modifications to the DrawItems function saved us a fair amount of coding time, which could be better spent doing other things. The source code for the DrawSounds function (or modified DrawItems function) is shown here:

```
void DrawSounds()
{
   glColor3f (0.0f, 1.0f, 0.0f);
   glBegin (GL_QUADS);
      for (long i = 0; i < map->header.max_sounds; i++)
      {
         glVertex2d (map->sound[i].xyz[0], map->sound[i].xyz[2]-0.01);
         glVertex2d (map->sound[i].xyz[0]+0.01, map->sound[i].xyz[2]);
         glVertex2d (map->sound[i].xyz[0], map->sound[i].xyz[2]+0.01);
         glVertex2d (map->sound[i].xyz[0]-0.01, map->sound[i].xyz[2]);
      }
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}
```

The final modification we must make to the code to finish dealing with sounds is to the Render function itself. Underneath the calls to DrawEntities and DrawItems we'll add a new if statement which, provided the value of layer.draw_sound is true, will call the DrawSounds function. This will provide us with a consistent way of drawing only the desired objects in the map.

Placing Lights

The last function we'll add in this chapter will insert light points into the map using the current world coordinates. As with every type of object or thing we add into the map editor, we must add a new type to our enumeration list in the map.h file. The name of the enumeration we'll add will be appropriately named LIGHT. In addition to inserting the LIGHT enumeration, we also must create a new method within the MAP class that will insert lights into the map. This function is needed before we can continue any further with the process of inserting lights into the map editor. The new method, InsertLight, will have a total of 11 parameters needed in the function call. The return value for the method is of the bool data type.

The first parameter in the method is a pointer to a character called name. This variable will be a NULL-terminated string and will contain the name of the light. The second parameter is another pointer to a character called filename, which is also a NULL-terminated string but will contain the filename of the bitmap used to project the light. This will be explained in further detail later in this section. After the filename variable come three variables, all of the GLdouble data type. The variables x, y, and z contain the world coordinates for the inserted light, which is an obvious necessity when trying to position a light!

The next three variables are all of the GLfloat data type. The variables xa, ya, and za store the input angles of the light and all have default values of 0.0f assigned to them in the class definition. This information is necessary in order to direct the light in the appropriate direction. The final three variables in the method declaration are yet another group of three GLfloat data types called r, g, and b. Each of the three variables has a default value of 1.0f. As if you couldn't tell by the names of the variables, they store the red, green, and blue color components for the light.

The insertion code itself is very much like the other insertion methods, so I won't repeat all of it here. I'm going to briefly discuss how the system works because of the number of variables being copied back and forth. Before any insertion can begin we declare a new variable called new_light, which is of the MAP_LIGHT data type. We'll also declare a new variable called rgb, which is of the long data type. Within the rgb declaration we call the GenerateColor function and store the returned value from the function in the rgb variable. The value returned from the function contains the unique selectable RGB value for the light we're about to create. Without the information, we won't be able to select it once we write the code for the functionality in Chapter 8.

After declaring both variables we must copy all the input parameters to their new_light structure equivalents. Some variables within the new_light structure are slightly different from the input parameters. For instance, the angles of the light are stored in an array called angle, where index 0 stores the xa variable, index 1 stores ya, etc. The inclusions pointers should be set to NULL and the max_inclusions variable should be set to 0 since there are no inclusions in the light by default. The name of the light must be string copied from the name parameter to the new_light.name variable.

The rgba variable within new_light works the same way as the angle variable (index 0 is set to r, index 1 is g), with the exception that we're going to set the third index to a hardcoded value of 1.0f. Although we don't have the option of inserting the alpha channel in the map editor, we may want it for certain special effects when doing the lighting, so we want to set a default value for it. After setting the color for the lighting, we must retrieve the individual color information for the selectable RGB values. Like the previous insertion functions, we have an array in the new_light variable called select_rgb, which will contain the three RGB values. Index 0 of the array will call the GetRValue macro specifying the rgb variable as the parameter. This will extract the red component from the generated color data. Index 1 will use the GetGValue macro with the rgb variable to extract the green component. Index 2 will use the GetBValue macro with the rgb variable as the parameter to extract the blue component.

Following the select_rgb variable array we must set the texture number for the filename we assigned to the light. Although I recommend setting the value to 0, any value will do since this variable will automatically be set when the game loads the map. The input filename parameter must be string copied to the new_light.texture_filename variable to store the texture that is assigned to the light. The variable new_light.type is used to specify what type of light it is. Since we've only got one style of light, we can set it to 0, the default! The final xyz array within the new_light structure works in the same manner. Index 0 stores the x, index 1 stores y, and index 2 stores z.

Now that the new_light structure has been filled we can begin the insertion process. Like every function we've created that has dynamic allocation in it, we must always check to see if the array in question already contains data. If the value in header.max_lights is equal to 0, then we'll allocate memory in the light array for one index. If the value in header.max_lights is greater than 0, we'll define a pointer variable called temp, which is of the MAP_LIGHT data type. The temp variable will then be allocated memory using the value map->header.max_lights+1, giving one extra index for a buffer. The contents of the light array will be copied from the light array to the temp array, looping through every light and even through the inclusion list if there is data in it as well. If there is data in the inclusion list, the temp[i].inclusions array will be allocated to the size of temp.max_inclusions, then have the data copied from the light[i].inclusions array.

With every iteration of the loop we'll destroy the light[i].inclusions array, provided the contents have been copied to the temp[i].inclusions array. Once all the copying from the light array to the temp array is complete, we'll delete the light array and allocate memory for it again except we'll use the value map->header.max_lights+2, giving us one new index plus the normal buffer index. After allocating the memory we'll simply copy the data back from the temp array, including the inclusions array, to the light array. With each iteration of the loop we'll delete the inclusions list for the temp variable, if there is data in it, after we've copied the data back. After all the data from the temp array is copied back to the light array, we delete the temp array and leave the dynamic allocation section of the InsertLight method.

Once all the memory concerns are addressed we simply set the light[header.max_lights] variable to the value of new_light, increase header.max_lights variable by one, and exit from the method by returning 1 to indicate it was successful. The source code for the InsertLight function is provided below.

```
bool MAP::InsertLight(char *name, char *filename, GLdouble x, GLdouble y,
          GLdouble z, GLfloat xa, GLfloat ya, GLfloat za, GLfloat r, GLfloat g,
          GLfloat b)
{
   MAP_LIGHT  new_light;
   long       rgb = GenerateColor();

   new_light.angle[0]           = xa;
   new_light.angle[1]           = ya;
   new_light.angle[2]           = za;
   new_light.inclusions         = NULL;
   new_light.max_inclusions     = 0;
   strcpy (new_light.name, name);
   new_light.rgba[0]            = r;
   new_light.rgba[1]            = g;
   new_light.rgba[2]            = b;
   new_light.rgba[3]            = 1.0;
   new_light.select_rgb[0]      = GetRValue(rgb);
   new_light.select_rgb[1]      = GetGValue(rgb);
   new_light.select_rgb[2]      = GetBValue(rgb);
   new_light.texture            = 0;
   strcpy (new_light.texture_filename, filename);
   new_light.type               = 0;
   new_light.xyz[0]             = x;
   new_light.xyz[1]             = y;
   new_light.xyz[2]             = z;

   if (header.max_lights == 0) light = new MAP_LIGHT[header.max_lights+1];
   else
   {
      MAP_LIGHT *temp;

      temp = new MAP_LIGHT[header.max_lights+1];
```

```
for (long i = 0; i < header.max_lights; i++)
{
   temp[i].angle[0] = light[i].angle[0];
   temp[i].angle[1] = light[i].angle[1];
   temp[i].angle[2] = light[i].angle[2];

   temp[i].max_inclusions= light[i].max_inclusions;
   if (temp[i].max_inclusions > 0)
   {
      temp[i].inclusions = new GLint[temp[i].max_inclusions+1];
      for (long i2 = 0; i2 < temp[i].max_inclusions; i2++)
           temp[i].inclusions[i2] = light[i].inclusions[i2];
      delete [] light[i].inclusions;
   }
   else temp[i].inclusions = NULL;
   strcpy (temp[i].name, light[i].name);
   temp[i].rgba[0]        = light[i].rgba[0];
   temp[i].rgba[1]        = light[i].rgba[1];
   temp[i].rgba[2]        = light[i].rgba[2];
   temp[i].rgba[3]        = light[i].rgba[3];
   temp[i].texture        = light[i].texture;
   strcpy (temp[i].texture_filename, light[i].texture_filename);
   temp[i].type           = light[i].type;
   temp[i].xyz[0]         = light[i].xyz[0];
   temp[i].xyz[1]         = light[i].xyz[1];
   temp[i].xyz[2]         = light[i].xyz[2];
   temp[i].select_rgb[0]  = light[i].select_rgb[0];
   temp[i].select_rgb[1]  = light[i].select_rgb[1];
   temp[i].select_rgb[2]  = light[i].select_rgb[2];
}

delete [] light;
light[i].inclusions = new GLint[header.max_lights+2];

for (i = 0; i < header.max_lights; i++)
{
   light[i].angle[0]      = temp[i].angle[0];
   light[i].angle[1]      = temp[i].angle[1];
   light[i].angle[2]      = temp[i].angle[2];

   light[i].max_inclusions= temp[i].max_inclusions;
   if (light[i].max_inclusions > 0)
   {
      light[i].inclusions = new GLint[light[i].max_inclusions+1];
      for (long i2 = 0; i2 < light[i].max_inclusions; i2++)
           light[i].inclusions[i2] = temp[i].inclusions[i2];
      delete [] temp[i].inclusions;
   }
   else light[i].inclusions = NULL;
   strcpy (light[i].name, temp[i].name);
   light[i].rgba[0]       = temp[i].rgba[0];
   light[i].rgba[1]       = temp[i].rgba[1];
   light[i].rgba[2]       = temp[i].rgba[2];
```

```
            light[i].rgba[3]       = temp[i].rgba[3];
            light[i].texture       = temp[i].texture;
            strcpy (light[i].texture_filename, temp[i].texture_filename);
            light[i].type          = temp[i].type;
            light[i].xyz[0]        = temp[i].xyz[0];
            light[i].xyz[1]        = temp[i].xyz[1];
            light[i].xyz[2]        = temp[i].xyz[2];
            light[i].select_rgb[0] = temp[i].select_rgb[0];
            light[i].select_rgb[1] = temp[i].select_rgb[1];
            light[i].select_rgb[2] = temp[i].select_rgb[2];
         }


         delete [] temp;
         temp = NULL;
      }

      light[header.max_lights] = new_light;
      header.max_lights++;

      return (true);
   }
```

Moving to the main source file, we need to add one new variable to the LAYER structure called draw_light. This new variable, which will be of the bool data type, controls whether or not the lighting data is shown on the screen. We also need to declare a new global variable called bInsertLight, which will be the new button used to insert a light. The declaration of the new variable will be placed at the bottom of the global declarations, just above the raster variable declaration.

After declaring the new variable we'll move to the WinMain function where we create the buttons for the map editor. We'll add a new call to the CreateWindow function underneath the call that creates the Insert Sound button (bInsertSound). As with all the previous CreateWindow function calls, we'll simply copy and paste the code from the previous function call and modify accordingly. In the case of the Insert Light button, we'll change the return value to bInsertLight so the returned button will be in the proper variable. The next modification we'll make is to the name of the window, which will be changed from the original "Insert Sound" to "Insert Light" for obvious reasons! The final modification we must make to the new function call is in the starting Y coordinate, in which we change the multiplier from 14 to 16. This concludes the modification of the CreateWindow function call to create the Insert Light button.

With the Insert Light button now created, we can move to the WMCommand function to handle the button clicks for the newly created button. Thankfully this is the last insertion button we'll have to deal with in the map editor since this process is becoming very tedious! The modification to the WMCommand function is relatively simple; we simply ensure the value of lParam is equal to the value of the LPARAM cast bInsertLight variable. In the event the variables are equal, we'll set the creation_coords.type to LIGHT, which indicates that we're inserting a light, and update the selected buttons by calling the ShowSelectedButton function. The source code for the button-click addition to the WMCommand function is provided below for you to study.

```
else if (lParam == (LPARAM)bInsertLight)
{
   creation_coords.type = LIGHT;
   ShowSelectedButton();
}
```

To improve the interface of the map editor we'll add a new option to the Layers menu in the main menu of the program. The new menu item we'll add will be called "Light," and will be checked by default. The resource ID for the new menu item will be ID_LAYERS_LIGHT. Besides adding the bInsertLight button-click functionality to the WMCommand function, we'll add a new else-if clause to check to see if Light (ID_LAYERS_LIGHT) was clicked. If the value of wParam is equal to ID_LAYERS_LIGHT, then we'll perform the code within the else-if statement. As with all layering options, we must check the appropriate layering variable (in this case layer.draw_light) to see if it's true. In the event the value is true, then we'll uncheck the ID_LAYERS_LIGHT menu resource. If the value of layer.draw_light is false, then we'll check the ID_LAYERS_LIGHT menu resource. After checking or unchecking the layering option, we'll simply switch the layer.draw_light variable to the opposite of its current value (i.e., true = false, false = true) to ensure the next button click will function in the opposite way. The source code for the layering addition to the WMCommand function is provided below.

```
else if (wParam == ID_LAYERS_LIGHT)
{
   if (layer.draw_light) CheckMenuItem (Menu, ID_LAYERS_LIGHT, MF_UNCHECKED);
   else CheckMenuItem (Menu, ID_LAYERS_LIGHT, MF_CHECKED);
   layer.draw_light = !layer.draw_light;
}
```

The next function to be updated is the ShowSelectedButton function to indicate when the Insert Light button is selected and when it is not. Like every other button, we'll set a default value to the button when the function is called using the SetWindowText function and specifying the button variable (bInsertLight) and the default button name (Insert Light). The case statement should be updated as well by adding a LIGHT clause to the list of possible options. Within the LIGHT clause we'll call the SetWindowText function to set the Insert Light button to "*Light*", which will be used as the selected indication. Because of the size of the update, there is no reason to display the source code here.

With the update to the ShowSelectedButton function complete, we can move to the WMLButtonUp function where we'll update the code to handle the insertion of lights. The modification to the WMLButtonUp function is a simple else-if statement added to the bottom of the if statement. Within the else-if statement we'll check to see if the value of the creation_coords.type variable is equal to the value of LIGHT. If the values are equal, we'll display the light insertion dialog box, which will be called IDD_INSERT_LIGHT. As with all dialog boxes, we'll create a custom dialog box procedure to handle any messages that would pass to or from the dialog box. The dialog box procedure will be named InsertLightDlgProc and we will call the dialog box using the DialogBox macro, supplying the GlobalInstance variable as the first parameter. The second integer resource version of the dialog box is (MAKEINTRESOURCE(IDD_INSERT_LIGHT)). The third parameter is parent window but it is not needed so we'll set the parameter to NULL. The final parameter in the macro is the dialog procedure where the messages will be sent. We'll cast our input value to the DLGPROC type and use Insert-LightDlgProc as the dialog procedure. The source code for the else-if statement is shown below.

```
else if (creation_coords.type == LIGHT) DialogBox (GlobalInstance,
        MAKEINTRESOURCE(IDD_INSERT_LIGHT), NULL, (DLGPROC)InsertLightDlgProc);
```

After creating the call to the IDD_INSERT_LIGHT dialog box, it would be a great idea to create it in the resource editor! We'll begin by inserting a new dialog box, then load the properties of the dialog box and change the static ID to IDD_INSERT_LIGHT and the caption to "Insert Light". Then we'll move to the More Styles tab and check the Center option. This will create our dialog box with the caption "Insert Light" and center it in the middle of the screen. With the dialog box now created, we can fill it with the proper controls to give the users the control they need to customize the lights they want to insert into their levels.

The first control we'll insert into the dialog box will be a single edit control, which will be used as a template for the other five controls we need in the dialog box. It's much easier to customize one control and then copy and paste it instead of trying to match the same size and other options after you've inserted the controls. Of course these are small interface tweaks and don't play any role in the speed or overall development of the map editor, but they are nice to have in your program to keep things looking clean! The first thing we'll do to the edit control is modify its size so it's about half its original width. The height will also be changed to be the smallest value, but this will only be a change of a few pixels, unlike the width, which can be changed very drastically.

In the Styles tab of the edit control we'll change the text alignment from Left to Center, which will center the input text in the edit control. The final modification to the edit control is to check the Number check box, allowing only numbers to be entered into the edit control. This check box is also located in the Styles tab, under the Text Align option. With all the modifications complete we can create eight more edit controls by copying and pasting the first one and changing the static ID name from IDC_INSERTLIGHT_R to IDC_INSERTLIGHT_G, IDC_INSERTLIGHT_B, IDC_INSERTLIGHT_XA, IDC_INSERTLIGHT_YA, IDC_INSERTLIGHT_ZA, IDC_INSERTLIGHT_X, IDC_INSERTLIGHT_Y, and IDC_INSERTLIGHT_Z.

The first three names are the inputs for the red, green, and blue values for the light and should be in a straight line. Unlike previous red, green, and blue values, we'll be using an unsigned character with a range of 0 to 255 to describe the individual RGB values. A value of 0 would be the dullest value and a value of 255 would be a full-intensity value of the color component. We use the unsigned characters in the edit controls because it's easy to convert color values from 2D graphics applications like Adobe Photoshop into values in the map editor. If we used floating-point values to represent the color components, most users wouldn't have a clue how to convert a value from an unsigned character to the floating-point equivalent and vice versa.

The next three edit controls will direct the angle of the light. At this point you may be wondering where the heck you get the light angle information from. The following table displays some commonly used angles for you to consider when creating a level.

Table 7.1: Commonly used light angles

X-Angle	Y-Angle	Z-Angle	Description
0-360	0	0	A light directly ahead of you, similar to a flashlight
0	0-360	0	A light that shines directly up and down
0	0	0-360	A light that controls rotation of the lightmap being shown

Table 7.1 provides a basic outline of what can be done with angles input into the edit controls. The values can be arranged to customize the look of the light. A light that shines down the y-axis will probably be the most commonly used type of light in our maps since it shines down on the floor, which makes it look like a typical overhead light or sunlight.

Before we continue talking about the controls needed to insert a light into our map, we'll discuss the basics of how our lighting system works. When the user inserts a light into the map the angle values are taken from the edit controls and stored in the map format. The game itself will load the data from the map data and eventually begin rendering the lights. Our lighting system uses per-pixel lighting by manipulating the texture settings to overlay a texture over a set of given surfaces to project a spotlight. This may sound complicated, but it's really very simple! With each light within each frame rendered, our rendering code will switch to the texture matrix to rotate the projection to the user-defined angles, and the light would move to the appropriate position in 3D space.

After positioning the light in the specified location, the light would draw all the objects listed in its inclusion list. The inclusion list is a list of object ID numbers to draw for the specified light. This allows the light to limit its rendering to a specified number of objects. If we didn't use the inclusion list, we would run into a problem with lights that shone through walls, which isn't very realistic.

Of course you could include all the objects if you wanted, but it's easier to only specify the walls you want affected by the light. In the next chapter we'll discuss how to add objects in the map to the inclusion list, but for the time being we'll finish the dialog box by adding some static controls to the edit controls to describe what each edit control is for and begin writing the functionality for the InsertLightDlgProc function. The static controls aren't a necessary feature of the dialog box and therefore we won't bother discussing what text to enter into each one. With the design of the dialog box out of the way, we can begin writing the code to handle the messages sent to the dialog box.

The definition of the InsertLightDlgProc function is no different than any other dialog box procedure we've defined in the past, so we can simply copy and paste the dialog box procedure definition from the InsertSoundDlgProc function and change the name to InsertLightDlgProc. Within the InsertLightDlgProc function we'll handle the two typical messages WM_INITDIALOG and WM_COMMAND.

As we've discussed previously, the WM_INITDIALOG message is sent when the dialog box is first initialized. When the message is sent we'll set the IDC_INSERTLIGHT_R, IDC_INSERTLIGHT_G, and IDC_INSERTLIGHT_B edit control values to 255, which is the full-intensity value. We set all three values to 255 to set white as the default color for the light. Of course the user can change the values at any time to produce a unique color. The next three edit controls, IDC_INSERTLIGHT_XA, IDC_INSERTLIGHT_YA, and IDC_INSERTLIGHT_ZA, will be set to 0 since we don't need any special rotations for a light. The final three edit controls are the XYZ coordinates of the light, which are stored in IDC_INSERTLIGHT_X, IDC_INSERTLIGHT_Y, and IDC_INSERTLIGHT_Z. Although there are three variables, we'll set the XZ coordinates using the creation_coords.finish.world_ (x/z) variables. The Y coordinate will be hardcoded to 0.0 because our mouse coordinate calculation does not support the calculation of the Y coordinate. If we set the edit control value to the value of creation_coords.finish.world_y, then we'll have random garbage from the memory address, which will definitely mess up our light location. To set each edit control we'll use the SetDlgItemText function and specify the hWnd value as the first parameter, the resource ID as the second parameter, and a NULL-terminated string containing the value we want to set as the third parameter. This concludes the code needed to handle the WM_INIT-DIALOG message.

The WM_COMMAND message will handle two different button clicks. The first button click handled is the default Cancel button click, which has a resource ID of IDCANCEL. If the value of the wParam variable is equal to IDCANCEL, then the user has clicked the Cancel button and we can exit the dialog using the macro EndDialog. The hWnd variable will be the first parameter and 0 will be used as the second parameter, which is the return value for the dialog box. If the user clicks the OK button (wParam equals IDOK), we must retrieve the values from each of the edit controls in string form, convert each string value to a local variable, insert the light, and finally exit the dialog box.

To begin the process we'll define a new variable called temp, which is an array of 500 characters. We'll use the temp variable to store the retrieved string values from the edit controls. Besides the temp variable we'll also declare nine GLfloat variables called r, g, b, xa, ya, za, x, y, and z to represent each of the edit controls we're getting data from. Once the data has been retrieved from the edit controls, we'll convert it to the appropriate data types and store the string converted variable data in each of the newly declared variables.

The first edit control we'll take the data from is the IDC_INSERT-LIGHT_R control, followed by IDC_INSERTLIGHT_G and finally IDC_INSERTLIGHT_B. Each edit control will follow the same system process for taking the data, so we can just discuss the first resource and simply substitute variables for the other two. To retrieve the data from IDC_INSERTLIGHT_R we'll call the function GetDlgItemText, specifying the current window (hWnd) as the first parameter, the resource ID (IDC_INSERTLIGHT_G) as the second parameter, the string we want the array to be placed in (temp), and the maximum size of the string returned (500).

With the data returned into the temp variable, we can simply call the sscanf function to format the string data into the appropriate floating-point variable. After formatting the string we'll convert the floating-point variable to a percentage of 255.0 to create the final color component value. To convert the input number into a percentage, we simply divide the retrieved value by 255.0 and store the returned value in the original floating-point variable (i.e., r = r / 255.0f;). We change each color component into a percentage so the values draw in OpenGL properly. If we kept the original value input by the user we probably wouldn't have the desired light color. As mentioned before, this process must occur for all three color component variables (r, g, and b).

The three angles used to project the light follow the same process for retrieving and formatting their input data as the color components. Unlike the color component data, the angles don't have their data altered in any way. When the data is formatted from the input text string (temp), it is stored in one of the three variables xa, ya, and za, which represent the axis angles. With the angles and color information safely stored in the proper variables, we can grab the final three x, y, and z variables from their appropriate controls and use the temp variable to store a string containing the name of the current light. Although we didn't add support in the dialog box to do this, you could add another edit control and retrieve the data instead of creating a default name. Of course this is just an idea for a feature you may want to incorporate into your own map editors.

To keep things simple we'll create our own default light name by placing the word "Light" followed by the current integer value of the map->header.max_lights variable in the string. This method works great unless you begin deleting the lights in the map editor, because there could be overlapping values set as a certain light. For instance, if the user deleted light number 10 and there were 20 lights, everything after light 10 would be shifted down one position in the light array, yet would still have its original light name assigned.

With all the local variables set to their appropriate values we can finally insert the light into the map. To do this, we simply call the newly created map method InsertLight and specify the name of the light as the first parameter, which in our case will be the temp variable. The second parameter sets the bitmap we'll be using to project light. The lighting system in our game is different from others in that we project a texture onto map objects to simulate them being lit. Because we're projecting a texture onto an object we must specify a bitmap we want to use to simulate that light. To keep things simple we'll use a hardcoded value string of "lightmap.bmp", which will be the bitmap file used to project our light. It is easier to call the projected bitmap texture a lightmap, because of its purpose. *Lightmapping* is the term used to describe using textures as lights instead of using dynamic lighting. We don't use dynamic lighting because it requires lots of polygons on the screen to resemble something similar to light and some video card manufacturers only allow a maximum of eight lights or only allow for eight lights to be accelerated. Lightmapping is a much faster process because we would render the scene normally, then bind the lighting texture to the objects and re-render the scene. No special tricks needed! Essentially any type of texture can be used as a lightmap, but I find the best results are obtained when using a black to white circular gradient, which is easily created in Adobe Photoshop or any 2D graphics software.

The next three parameters specify the X, Y, and Z coordinates of the light to be placed. Obviously this information is crucial since we don't want our light to be positioned in the wrong part of our world. Each of these parameters will use its corresponding x, y, or z variable to define the light's location. The next three parameters specify the projection angles of the light, which as we know specifies the direction that the light points. We use the xa, ya, and za variables to specify the direction of the light.

The last three variables for the method specify the R, G, and B color components of the light we want to insert. As stated earlier, I recommend using a black to white circular gradient to the lights. When the user specifies a certain color by entering the values into the IDC_INSERTLIGHT_R, IDC_INSERTLIGHT_G, and IDC_INSERTLIGHT_B color edit controls, the texture projected onto the map objects is colored using the variables r, g, and b, which are floating-point conversions of the originally input values. If the lightmap is drawn using only grayscale colors, then the light will be colored properly. If the lightmap uses regular coloring (i.e., reds, greens, and blues) instead of grayscale, it will affect the coloring and using the input values may distort the lightmap when it's projected onto the objects. In either event, we use the variables r, g, and b as the final three variables.

Now that the insertion is complete we can exit the dialog box by calling the EndDialog function, specifying the current window (hWnd) and the return value for the dialog box (1). At the bottom of the dialog box we'll set the default return value to 0, which will allow the dialog box to function properly. With the return value for the InsertLightDlgProc function set we can display the source code below.

```
LRESULT CALLBACK InsertLightDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
         {
            char temp[500];

            SetDlgItemText (hWnd, IDC_INSERTLIGHT_R, "255");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_G, "255");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_B, "255");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_XA, "0");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_YA, "0");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_ZA, "0");

            sprintf (temp, "%f", creation_coords.finish.world_x);
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_X, temp);

            SetDlgItemText (hWnd, IDC_INSERTLIGHT_Y, "0.0");

            sprintf (temp, "%f", creation_coords.finish.world_z);
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_Z, temp);
         }break;
      case WM_COMMAND:
```

```
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char temp[500];
            GLfloat r;
            GLfloat g;
            GLfloat b;
            GLfloat xa;
            GLfloat ya;
            GLfloat za;
            GLfloat x;
            GLfloat y;
            GLfloat z;

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_R, temp, 500);
            sscanf (temp, "%f", &r);
            r = r / 255.0f;

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_G, temp, 500);
            sscanf (temp, "%f", &g);
            g = g / 255.0f;

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_B, temp, 500);
            sscanf (temp, "%f", &b);
            b = b / 255.0f;

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_XA, temp, 500);
            sscanf (temp, "%f", &xa);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_YA, temp, 500);
            sscanf (temp, "%f", &ya);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_ZA, temp, 500);
            sscanf (temp, "%f", &za);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_X, temp, 500);
            sscanf (temp, "%f", &x);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_Y, temp, 500);
            sscanf (temp, "%f", &y);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_Z, temp, 500);
            sscanf (temp, "%f", &z);

            sprintf (temp, "Light #%i", map->header.max_lights);
            map->InsertLight (temp, "lightmap.bmp", x, y, z, xa, ya, za, r, g, b);

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

Drawing the lights in the map editor follows the same procedure as drawing sounds or any other style of item. For that reason we can save ourselves a bit of time and copy and paste the code from the DrawSounds function to create a new function called DrawLights. The first thing we'll have to change in the new function is the color in which the light points are drawn. Unfortunately we've run out of simple combinations of R, G, and B values, so we'll change the first glColor statement to 0.5f, 1.0f, and 0.0f. This will give us red at 50% intensity, green at 100%, and blue at 0%, resulting in a nice-looking forest green color, which is something we don't already have.

The next modification we'll make is to the for loop, where we'll change the maximum value from header->max_sounds to header->max_lights, which will allow us to loop through the maximum number of lights. The final modification we'll make to the function is to the glVertex2d function calls, where we'll change each call to the map->sound variable to the map->light variable. If we didn't do this, it would surely cause problems and wouldn't even display the lights, which was the entire purpose of creating this function! This completes the modifications to the DrawLights function, so we can display the source code below.

```
void DrawLights()
{
   glColor3f (0.5f, 1.0f, 0.0f);
   glBegin (GL_QUADS);
      for (long i = 0; i < map->header.max_lights; i++)
      {
         glVertex2d (map->light[i].xyz[0], map->light[i].xyz[2]-0.01);
         glVertex2d (map->light[i].xyz[0]+0.01, map->light[i].xyz[2]);
         glVertex2d (map->light[i].xyz[0], map->light[i].xyz[2]+0.01);
         glVertex2d (map->light[i].xyz[0]-0.01, map->light[i].xyz[2]);
      }
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}
```

Chapter Example

We only have one last source code addition to make, and the chapter is finished. The final addition is to the Render function, where we'll add an if statement to the bottom of the if statements already in place, checking if the value of layer.draw_light is true. In the event the value is true, then we'll call the DrawLights function to draw all the light positions in the level. This works on the principle that there are lights in the world and that the Light layers option is checked. If the value is unchecked at any time, the drawing of the light positions will cease because the value of layer.draw_light will be false.

The updated map editor source code is displayed on the following pages.

ex 7_1.cpp

```
#include <windows.h>
#include <winbase.h>
#include <stdio.h>

#include "resource.h"

#include "raster.h"
#include "map.h"

#define DEFAULT_BUTTON_WIDTH    100
#define DEFAULT_BUTTON_HEIGHT   20

enum { CREATE_MODE_NULL = 0, CREATE_MODE_START, CREATE_MODE_SIZE,
       CREATE_MODE_FINISH };
enum { DRAW_MODE_WIREFRAME = 0, DRAW_MODE_SOLID };

typedef struct
{
   long       mouse_x;
   long       mouse_y;

   double     world_x;
   double     world_y;
   double     world_z;
} COORDS;

typedef struct
{
   long             mode;
   long             type;

   COORDS           start;
   COORDS           finish;
} CREATION_COORDS;

typedef struct
{
   long             draw_mode;
} CONFIG;

typedef struct
{
   bool             draw_floor;
   bool             draw_ceiling;
   bool             draw_wall;
   bool             draw_entity;
   bool             draw_item;
   bool             draw_sound;
   bool             draw_light;
} LAYER;
```

```
HINSTANCE           GlobalInstance;
HMENU               Menu;
HMENU               PopupMenu;
HWND                Window;
HWND                RenderWindow;
HWND                bCreateWall;
HWND                bCreateFloor;
HWND                bCreateCeiling;
HWND                bPlaceStartPosition;
HWND                bPlaceDMPosition;
HWND                bInsertEntity;
HWND                bInsertItem;
HWND                bInsertSound;
HWND                bInsertLight;
RASTER              raster;

CREATION_COORDS     creation_coords;
MAP                 *map = new MAP;

CONFIG              config;
LAYER               layer;


void ResizeGLWindow(long width, long height)
{
   glViewport(0, 0, (GLsizei) width, (GLsizei) height);
   glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(-200,200, -200,-200, -2000,2000);
   glMatrixMode(GL_MODELVIEW);
}


void SetGLDefaults()
{
   glEnable (GL_DEPTH_TEST);
   glDisable (GL_CULL_FACE);

   glClearColor (0.6f, 0.6f, 0.6f, 1.0f);
}

void DrawWireframe()
{
   if (map->header.max_objects > 0)
   {
      for (long i = 0; i < map->header.max_objects; i++)
      {
         if ((layer.draw_floor && map->object[i].type == OBJECTTYPE_FLOOR) ||
            (layer.draw_ceiling && map->object[i].type == OBJECTTYPE_CEILING) ||
            (layer.draw_wall && map->object[i].type == OBJECTTYPE_WALL))
         {
            glBegin (GL_LINE_LOOP);
               for (long i2 = 0; i2 < map->object[i].max_vertices; i2++)
                    glVertex2d (map->object[i].vertex[i2].xyz[0],
                    map->object[i].vertex[i2].xyz[2]);
            glEnd();
         }
      }
   }
}


void DrawSolid()
{
   for (long i = 0; i < map->header.max_objects; i++)
   {
      glColor3ubv (map->object[i].select_rgb);

      if ((layer.draw_floor && map->object[i].type == OBJECTTYPE_FLOOR) ||
         (layer.draw_ceiling && map->object[i].type == OBJECTTYPE_CEILING))
      {
         glBegin (GL_TRIANGLES);
         for (long i2 = 0; i2 < map->object[i].max_triangles; i2++)
         {
            long vertex_0 = map->object[i].triangle[i2].point[0];
            long vertex_1 = map->object[i].triangle[i2].point[1];
            long vertex_2 = map->object[i].triangle[i2].point[2];

            glVertex2d (map->object[i].vertex[vertex_0].xyz[0],map->
                        object[i].vertex[vertex_0].xyz[2]);
            glVertex2d (map->object[i].vertex[vertex_1].xyz[0],map->
                        object[i].vertex[vertex_1].xyz[2]);
            glVertex2d (map->object[i].vertex[vertex_2].xyz[0],map->
                        object[i].vertex[vertex_2].xyz[2]);
         }
         glEnd();
      }
      else if (layer.draw_wall && map->object[i].type == OBJECTTYPE_WALL)
      {
         for (long i2 = 0; i2 < map->object[i].max_triangles; i2++)
         {
            long vertex_0 = map->object[i].triangle[i2].point[0];
            long vertex_1 = map->object[i].triangle[i2].point[1];
            long vertex_2 = map->object[i].triangle[i2].point[2];

            glBegin (GL_LINE_LOOP);
               glVertex2d (map->object[i].vertex[vertex_0].xyz[0],map->
                           object[i].vertex[vertex_0].xyz[2]);
               glVertex2d (map->object[i].vertex[vertex_1].xyz[0],map->
                           object[i].vertex[vertex_1].xyz[2]);
               glVertex2d (map->object[i].vertex[vertex_2].xyz[0],map->
                           object[i].vertex[vertex_2].xyz[2]);
            glEnd();
         }
```

```
            }
        }
    }


    void DrawStartPosition()
    {
        glColor3f (0.0f, 0.0f, 1.0f);
        glBegin (GL_QUADS);
            glVertex2d (map->details.single_player.xyz[0], map->details.single_
                            player.xyz[2]-0.01);
            glVertex2d (map->details.single_player.xyz[0]+0.01, map->details.single_
                            player.xyz[2]);
            glVertex2d (map->details.single_player.xyz[0], map->details.single_
                            player.xyz[2]+0.01);
            glVertex2d (map->details.single_player.xyz[0]-0.01, map->details.single_
                            player.xyz[2]);
        glEnd();
        glColor3f (1.0f, 1.0f, 1.0f);
    }


    void DrawDeathMatchPositions()
    {
        glColor3f (1.0f, 0.0f, 1.0f);
        glBegin (GL_QUADS);
            glVertex2d (map->details.deathmatch[0].xyz[0], map->
                            details.deathmatch[0].xyz[2]-0.01);
            glVertex2d (map->details.deathmatch[0].xyz[0]+0.01, map->
                            details.deathmatch[0].xyz[2]);
            glVertex2d (map->details.deathmatch[0].xyz[0], map->
                            details.deathmatch[0].xyz[2]+0.01);
            glVertex2d (map->details.deathmatch[0].xyz[0]-0.01, map->
                            details.deathmatch[0].xyz[2]);

            glVertex2d (map->details.deathmatch[1].xyz[0], map->
                            details.deathmatch[1].xyz[2]-0.01);
            glVertex2d (map->details.deathmatch[1].xyz[0]+0.01, map->
                            details.deathmatch[1].xyz[2]);
            glVertex2d (map->details.deathmatch[1].xyz[0], map->
                            details.deathmatch[1].xyz[2]+0.01);
            glVertex2d (map->details.deathmatch[1].xyz[0]-0.01, map->
                            details.deathmatch[1].xyz[2]);
        glEnd();
        glColor3f (1.0f, 1.0f, 1.0f);
    }


    void DrawEntities()
    {
        glColor3f (0.0f, 1.0f, 1.0f);
        glBegin (GL_QUADS);
            for (long i = 0; i < map->header.max_entities; i++)
```

```
      {
         glVertex2d (map->entity[i].xyz[0], map->entity[i].xyz[2]-0.01);
         glVertex2d (map->entity[i].xyz[0]+0.01, map->entity[i].xyz[2]);
         glVertex2d (map->entity[i].xyz[0], map->entity[i].xyz[2]+0.01);
         glVertex2d (map->entity[i].xyz[0]-0.01, map->entity[i].xyz[2]);
      }
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}


void DrawItems()
{
   glColor3f (1.0f, 1.0f, 0.0f);
   glBegin (GL_QUADS);
      for (long i = 0; i < map->header.max_items; i++)
      {
         glVertex2d (map->item[i].xyz[0], map->item[i].xyz[2]-0.01);
         glVertex2d (map->item[i].xyz[0]+0.01, map->item[i].xyz[2]);
         glVertex2d (map->item[i].xyz[0], map->item[i].xyz[2]+0.01);
         glVertex2d (map->item[i].xyz[0]-0.01, map->item[i].xyz[2]);
      }
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}


void DrawSounds()
{
   glColor3f (0.0f, 1.0f, 0.0f);
   glBegin (GL_QUADS);
      for (long i = 0; i < map->header.max_sounds; i++)
      {
         glVertex2d (map->sound[i].xyz[0], map->sound[i].xyz[2]-0.01);
         glVertex2d (map->sound[i].xyz[0]+0.01, map->sound[i].xyz[2]);
         glVertex2d (map->sound[i].xyz[0], map->sound[i].xyz[2]+0.01);
         glVertex2d (map->sound[i].xyz[0]-0.01, map->sound[i].xyz[2]);
      }
   glEnd();
   glColor3f (1.0f, 1.0f, 1.0f);
}


void DrawLights()
{
   glColor3f (0.5f, 1.0f, 0.0f);
   glBegin (GL_QUADS);
      for (long i = 0; i < map->header.max_lights; i++)
      {
         glVertex2d (map->light[i].xyz[0], map->light[i].xyz[2]-0.01);
         glVertex2d (map->light[i].xyz[0]+0.01, map->light[i].xyz[2]);
         glVertex2d (map->light[i].xyz[0], map->light[i].xyz[2]+0.01);
         glVertex2d (map->light[i].xyz[0]-0.01, map->light[i].xyz[2]);
```

```
        }
    glEnd();
    glColor3f (1.0f, 1.0f, 1.0f);
}


void Render()
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();
    glPushMatrix();
        glTranslatef (0.0f, 0.0f, 0.0f);

        if (config.draw_mode == DRAW_MODE_WIREFRAME) DrawWireframe();
        else if (config.draw_mode == DRAW_MODE_SOLID) DrawSolid();

        DrawStartPosition();
        DrawDeathMatchPositions();
        if (layer.draw_entity) DrawEntities();
        if (layer.draw_item) DrawItems();
        if (layer.draw_sound) DrawSounds();
        if (layer.draw_light) DrawLights();


        if (creation_coords.type == OBJECTTYPE_WALL)
        {
            glBegin (GL_LINES);
                glVertex2d (creation_coords.start.world_x, creation_
                                coords.start.world_z);
                glVertex2d (creation_coords.finish.world_x, creation_
                                coords.finish.world_z);
            glEnd();
        }
        else if (creation_coords.type == OBJECTTYPE_FLOOR || creation_coords.type
                == OBJECTTYPE_CEILING)
        {
            glBegin (GL_LINE_LOOP);
                glVertex2d (creation_coords.start.world_x, creation_
                                coords.start.world_z);
                glVertex2d (creation_coords.finish.world_x, creation_
                                coords.start.world_z);
                glVertex2d (creation_coords.finish.world_x, creation_
                                coords.finish.world_z);
                glVertex2d (creation_coords.start.world_x, creation_
                                coords.finish.world_z);
            glEnd();
        }

    glPopMatrix();
    SwapBuffers (raster.hDC);
}
```

```
LRESULT CALLBACK MapDetailsDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, "Map Name");

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                             0, (LPARAM)"Erase Me");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_RESETCONTENT,
                             0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                             0, (LPARAM)"Exit");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                             0, (LPARAM)"Get Fragged");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_SETCURSEL,
                             0, 1);

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                             0, (LPARAM)"Erase Me");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_RESETCONTENT,
                             0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                             0, (LPARAM)"Single Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                             0, (LPARAM)"Multi Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_SETCURSEL,
                             0, 1);
      } break;

      case WM_COMMAND:
      {
         if (wParam == IDOK)
         {
            long level_rule = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_
                              RULES, LB_GETCURSEL, 0, 0);
            long level_type = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_
                              TYPE, CB_GETCURSEL, 0, 0);

            char temp[500];

            sprintf (temp, "Level Type: %i\r\nLevel Rule: %i\r\nOk Button!",
                     level_type, level_rule);
            MessageBox (hWnd, temp, "Ok", MB_OK);

            EndDialog (hWnd, 0);
         }
         else if (wParam == IDCANCEL)
         {
            MessageBox (hWnd, "Cancel Button!", "Cancel", MB_OK);
            EndDialog (hWnd, 0);
         }
      } break;
   }
   return (0);
}


void ShowSelectedButton()
{
   SetWindowText (bCreateFloor, "Create Floor");
   SetWindowText (bCreateCeiling, "Create Ceiling");
   SetWindowText (bCreateWall, "Create Wall");
   SetWindowText (bPlaceStartPosition, "Place Start");
   SetWindowText (bPlaceDMPosition, "Place DM");
   SetWindowText (bInsertEntity, "Insert Entity");
   SetWindowText (bInsertItem, "Insert Item");
   SetWindowText (bInsertSound, "Insert Sound");
   SetWindowText (bInsertLight, "Insert Light");

   switch (creation_coords.type)
   {
      case OBJECTTYPE_FLOOR: SetWindowText (bCreateFloor, "*Floor*"); break;
      case OBJECTTYPE_CEILING: SetWindowText (bCreateCeiling, "*Ceiling*"); break;
      case OBJECTTYPE_WALL: SetWindowText (bCreateWall, "*Wall*"); break;
      case ITEM_START_POSITION: SetWindowText (bPlaceStartPosition,
                                 "*StartPos*"); break;
      case ITEM_DM_POSITION: SetWindowText (bPlaceDMPosition, "*DMPos*"); break;
      case ITEM_ENTITY: SetWindowText (bInsertEntity, "*Entity*"); break;
      case ITEM: SetWindowText (bInsertItem, "*Item*"); break;
      case SOUND: SetWindowText (bInsertSound, "*Sound*"); break;
      case LIGHT: SetWindowText (bInsertLight, "*Light*"); break;
   }
}


void WMCommand(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (lParam == (LPARAM)bCreateWall)
   {
      creation_coords.type = OBJECTTYPE_WALL;
      ShowSelectedButton();
   }
   else if (lParam == (LPARAM)bCreateFloor)
   {
      creation_coords.type = OBJECTTYPE_FLOOR;
      ShowSelectedButton();
   }
   else if (lParam == (LPARAM)bCreateCeiling)
   {
      creation_coords.type = OBJECTTYPE_CEILING;
      ShowSelectedButton();
   }
```

```
else if (lParam == (LPARAM)bPlaceStartPosition)
{
   creation_coords.type = ITEM_START_POSITION;
   ShowSelectedButton();
}
else if (lParam == (LPARAM)bPlaceDMPosition)
{
   creation_coords.type = ITEM_DM_POSITION;
   ShowSelectedButton();
}
else if (lParam == (LPARAM)bInsertEntity)
{
   creation_coords.type = ITEM_ENTITY;
   ShowSelectedButton();
}
else if (lParam == (LPARAM)bInsertItem)
{
   creation_coords.type = ITEM;
   ShowSelectedButton();
}
else if (lParam == (LPARAM)bInsertSound)
{
   creation_coords.type = SOUND;
   ShowSelectedButton();
}
else if (lParam == (LPARAM)bInsertLight)
{
   creation_coords.type = LIGHT;
   ShowSelectedButton();
}
else if (wParam == ID_FILE_EXIT) PostQuitMessage(0);
else if (wParam == ID_DRAWING_WIREFRAME)
{
   CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_CHECKED);
   CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_UNCHECKED);

   config.draw_mode = DRAW_MODE_WIREFRAME;
}
else if (wParam == ID_DRAWING_SOLID)
{
   CheckMenuItem (Menu, ID_DRAWING_SOLID, MF_CHECKED);
   CheckMenuItem (Menu, ID_DRAWING_WIREFRAME, MF_UNCHECKED);

   config.draw_mode = DRAW_MODE_SOLID;
}
else if (wParam == ID_MAP_DETAILS) DialogBox (GlobalInstance,
         MAKEINTRESOURCE(IDD_MAP_DETAILS), NULL, (DLGPROC)MapDetailsDlgProc);
else if (wParam == ID_LAYERS_FLOOR)
{
   if (layer.draw_floor) CheckMenuItem (Menu, ID_LAYERS_FLOOR, MF_UNCHECKED);
   else CheckMenuItem (Menu, ID_LAYERS_FLOOR, MF_CHECKED);
   layer.draw_floor = !layer.draw_floor;
}
```

```
   else if (wParam == ID_LAYERS_CEILING)
   {
      if (layer.draw_ceiling) CheckMenuItem (Menu, ID_LAYERS_CEILING,
               MF_UNCHECKED);
      else CheckMenuItem (Menu, ID_LAYERS_CEILING, MF_CHECKED);
      layer.draw_ceiling = !layer.draw_ceiling;
   }
   else if (wParam == ID_LAYERS_WALL)
   {
      if (layer.draw_wall) CheckMenuItem (Menu, ID_LAYERS_WALL, MF_UNCHECKED);
      else CheckMenuItem (Menu, ID_LAYERS_WALL, MF_CHECKED);
      layer.draw_wall = !layer.draw_wall;
   }
   else if (wParam == ID_LAYERS_ENTITY)
   {
      if (layer.draw_entity) CheckMenuItem (Menu, ID_LAYERS_ENTITY, MF_UNCHECKED);
      else CheckMenuItem (Menu, ID_LAYERS_ENTITY, MF_CHECKED);
      layer.draw_entity = !layer.draw_entity;
   }
   else if (wParam == ID_LAYERS_ITEM)
   {
      if (layer.draw_item) CheckMenuItem (Menu, ID_LAYERS_ITEM, MF_UNCHECKED);
      else CheckMenuItem (Menu, ID_LAYERS_ITEM, MF_CHECKED);
      layer.draw_item = !layer.draw_item;
   }
   else if (wParam == ID_LAYERS_SOUND)
   {
      if (layer.draw_sound) CheckMenuItem (Menu, ID_LAYERS_SOUND, MF_UNCHECKED);
      else CheckMenuItem (Menu, ID_LAYERS_SOUND, MF_CHECKED);
      layer.draw_sound = !layer.draw_sound;
   }
   else if (wParam == ID_LAYERS_LIGHT)
   {
      if (layer.draw_light) CheckMenuItem (Menu, ID_LAYERS_LIGHT, MF_UNCHECKED);
      else CheckMenuItem (Menu, ID_LAYERS_LIGHT, MF_CHECKED);
      layer.draw_light = !layer.draw_light;
   }


   // Popup Menu Items
   else if (wParam == ID_POPUP_MOVE) MessageBox (Window, "Move", "Click", MB_OK);
   else if (wParam == ID_POPUP_DELETE) MessageBox (Window, "Delete", "Click",
                      MB_OK);
   else if (wParam == ID_POPUP_TEXTURE) MessageBox (Window, "Texture", "Click",
                      MB_OK);
   else if (wParam == ID_POPUP_DUPLICATE) MessageBox (Window, "Duplicate",
                      "Click", MB_OK);
}

void DisplayPopupMenu()
{
   HMENU temp = GetSubMenu(PopupMenu, 0);
   POINT point;
   GetCursorPos (&point);
   TrackPopupMenu(temp, TPM_LEFTALIGN|TPM_RIGHTBUTTON, point.x, point.y, 0,
                  Window, NULL);
}


void WMSize(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   RECT rect;

   GetClientRect (Window, &rect);
   MoveWindow (RenderWindow, DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-
               DEFAULT_BUTTON_WIDTH, rect.bottom-rect.top, true);

   GetClientRect (RenderWindow, &rect);
   ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);
}


COORDS        ComputeMouseCoords(long xPos, long yPos)
{
   COORDS     coords;
   RECT       rect;

   float            window_width;
   float            window_height;
   float            window_start_x;
   float            window_start_y;


   coords.mouse_x   = xPos;
   coords.mouse_y   = yPos;


   GetWindowRect (RenderWindow, &rect);
   window_width     = (float)(rect.right - rect.left);
   window_height    = (float)(rect.bottom - rect.top);
   window_start_x   = (float)(coords.mouse_x - rect.left);
   window_start_y   = (float)coords.mouse_y;


   coords.world_x   = (window_start_x / window_width) * 2.0 - 1.0;
   coords.world_z   = -((window_start_y / window_height) * 2.0 - 1.0);

   return (coords);
}


void WMLButtonDown(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   creation_coords.mode   = CREATE_MODE_START;
   creation_coords.start  = ComputeMouseCoords(LOWORD(lParam), HIWORD(lParam));
   creation_coords.finish = creation_coords.start;
```

```
}


LRESULT CALLBACK DMPositionDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_RESETCONTENT, 0, 0);
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Player 1");
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Player 2");
         SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE, CB_SETCURSEL, 0, 1);
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            long player = SendDlgItemMessage (hWnd, IDC_DM_POSITION_TYPE,
                          CB_GETCURSEL, 0, 0);

            map->details.deathmatch[player].xyz[0] = creation_
                 coords.start.world_x;
            map->details.deathmatch[player].xyz[1] = creation_
                 coords.start.world_y;
            map->details.deathmatch[player].xyz[2] = creation_
                 coords.start.world_z;

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}


LRESULT CALLBACK InsertEntityDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE, CB_RESETCONTENT, 0, 0);
         SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE, CB_ADDSTRING, 0,
                              (LPARAM)"Joe");
         SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE, CB_SETCURSEL, 0, 1);

         SetDlgItemText (hWnd, IDC_INSERT_ENTITY_HEALTH, "100");
         SetDlgItemText (hWnd, IDC_INSERT_ENTITY_STRENGTH, "10");
         SetDlgItemText (hWnd, IDC_INSERT_ENTITY_ARMOUR, "0");
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char temp[500];
            long type;
            long health;
            long strength;
            long armour;

            type = SendDlgItemMessage (hWnd, IDC_INSERT_ENTITY_TYPE,
                   CB_GETCURSEL, 0, 0);

            GetDlgItemText (hWnd, IDC_INSERT_ENTITY_HEALTH, temp, 500);
            sscanf (temp, "%i", &health);

            GetDlgItemText (hWnd, IDC_INSERT_ENTITY_STRENGTH, temp, 500);
            sscanf (temp, "%i", &strength);

            GetDlgItemText (hWnd, IDC_INSERT_ENTITY_ARMOUR, temp, 500);
            sscanf (temp, "%i", &armour);

            map->InsertEntity(type, creation_coords.start.world_x, creation_
                 coords.start.world_y, creation_coords.start.world_z, 0,0,0,
                 health, strength, armour);
            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}


LRESULT CALLBACK InsertItemDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE, CB_RESETCONTENT, 0, 0);
         SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE, CB_ADDSTRING, 0,
                             (LPARAM)"Gun");
         SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE, CB_SETCURSEL, 0, 1);
         SetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_TIME, "0");
         SetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_WAIT, "0");
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char temp[500];
            long type;
            long respawn_wait;
            long respawn_time;

            type = SendDlgItemMessage (hWnd, IDC_INSERT_ITEM_TYPE, CB_GETCURSEL,
                                       0, 0);

            GetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_WAIT, temp, 500);
            sscanf (temp, "%i", &respawn_wait);

            GetDlgItemText (hWnd, IDC_INSERT_ITEM_RESPAWN_TIME, temp, 500);
            sscanf (temp, "%i", &respawn_time);

            map->InsertItem (creation_coords.finish.world_x, creation_
                 coords.finish.world_y, creation_coords.finish.world_z, type,
                 respawn_wait, respawn_time);
            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}


LRESULT CALLBACK InsertSoundDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG: SetDlgItemText (hWnd, IDC_INSERTSOUND_FILENAME,
                          "my_sound.wav"); break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char filename[500];
            GetDlgItemText (hWnd, IDC_INSERTSOUND_FILENAME, filename, 500);

            map->InsertSound (creation_coords.finish.world_x, 0, creation_
                              coords.finish.world_z, filename);

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

```
LRESULT CALLBACK InsertLightDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
         {
            char temp[500];

            SetDlgItemText (hWnd, IDC_INSERTLIGHT_R, "255");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_G, "255");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_B, "255");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_XA, "0");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_YA, "0");
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_ZA, "0");

            sprintf (temp, "%f", creation_coords.finish.world_x);
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_X, temp);

            SetDlgItemText (hWnd, IDC_INSERTLIGHT_Y, "0.0");

            sprintf (temp, "%f", creation_coords.finish.world_z);
            SetDlgItemText (hWnd, IDC_INSERTLIGHT_Z, temp);
         }break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char temp[500];
            GLfloat r;
            GLfloat g;
            GLfloat b;
            GLfloat xa;
            GLfloat ya;
            GLfloat za;
            GLfloat x;
            GLfloat y;
            GLfloat z;

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_R, temp, 500);
            sscanf (temp, "%f", &r);
            r = r / 255.0f;

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_G, temp, 500);
            sscanf (temp, "%f", &g);
            g = g / 255.0f;

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_B, temp, 500);
            sscanf (temp, "%f", &b);
            b = b / 255.0f;
            GetDlgItemText (hWnd, IDC_INSERTLIGHT_XA, temp, 500);
            sscanf (temp, "%f", &xa);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_YA, temp, 500);
            sscanf (temp, "%f", &ya);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_ZA, temp, 500);
            sscanf (temp, "%f", &za);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_X, temp, 500);
            sscanf (temp, "%f", &x);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_Y, temp, 500);
            sscanf (temp, "%f", &y);

            GetDlgItemText (hWnd, IDC_INSERTLIGHT_Z, temp, 500);
            sscanf (temp, "%f", &z);

            sprintf (temp, "Light #%i", map->header.max_lights);
            map->InsertLight (temp, "lightmap.bmp", x, y, z, xa, ya, za, r, g, b);

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}


void WMLButtonUp(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   if (creation_coords.mode != CREATE_MODE_NULL)
   {
      creation_coords.mode      = CREATE_MODE_NULL;
      creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                          HIWORD(lParam));

      if (creation_coords.type == OBJECTTYPE_WALL)
      {
         map->InsertObject ("Wall", creation_coords.type);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 1, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 1, creation_coords.finish.world_z);

         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 0, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 0, creation_coords.start.world_z);

         map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
              1.0f,0.0f, 1.0f,1.0f);
         map->InsertTriangle (map->header.max_objects-1, 3, 1, 2, 1.0f,1.0f,
              0.0f,1.0f, 0.0f,0.0f);
      }
      else if (creation_coords.type == OBJECTTYPE_FLOOR)
      {
         map->InsertObject ("Floor", creation_coords.type);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 0, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 0, creation_coords.start.world_z);

         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 0, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 0, creation_coords.finish.world_z);

         map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
              1.0f,0.0f, 1.0f,1.0f);
         map->InsertTriangle (map->header.max_objects-1, 3, 1, 2, 1.0f,1.0f,
              0.0f,1.0f, 0.0f,0.0f);
      }
      else if (creation_coords.type == OBJECTTYPE_CEILING)
      {
         map->InsertObject ("Ceiling", creation_coords.type);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 1, creation_coords.start.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 1, creation_coords.start.world_z);

         map->InsertVertex (map->header.max_objects-1, creation_
              coords.finish.world_x, 1, creation_coords.finish.world_z);
         map->InsertVertex (map->header.max_objects-1, creation_
              coords.start.world_x, 1, creation_coords.finish.world_z);

         map->InsertTriangle (map->header.max_objects-1, 0, 1, 2, 0.0f,0.0f,
              1.0f,0.0f, 1.0f,1.0f);
         map->InsertTriangle (map->header.max_objects-1, 3, 1, 2, 1.0f,1.0f,
              0.0f,1.0f, 0.0f,0.0f);
      }

      memset (&creation_coords.start, 0, sizeof(creation_coords.start));
      memset (&creation_coords.finish, 0, sizeof(creation_coords.finish));
   }
}


void WMMouseMove(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   char temp[500];

   if (creation_coords.mode != CREATE_MODE_NULL)
   {
      creation_coords.mode      = CREATE_MODE_SIZE;
```

```
        creation_coords.finish    = ComputeMouseCoords(LOWORD(lParam),
                                           HIWORD(lParam));
        sprintf (temp, "Map Editor, Mx=%i My=%i, X=%0.4f Z=%0.4f", creation_
                 coords.finish.mouse_x, creation_coords.finish.mouse_y, creation_
                 coords.finish.world_x, creation_coords.finish.world_z);
     }
     else sprintf (temp, "Map Editor, Mx=%i My=%i", LOWORD(lParam), HIWORD(lParam));

     SetWindowText (Window, temp);
  }

  LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
  {
     switch (msg)
     {
        case WM_DESTROY: PostQuitMessage(0); break;
        case WM_COMMAND: WMCommand (hWnd, msg, wParam, lParam); break;
        case WM_SIZE: WMSize (hWnd, msg, wParam, lParam); break;
        case WM_RBUTTONUP: DisplayPopupMenu(); break;
        case WM_LBUTTONDOWN: WMLButtonDown (hWnd, msg, wParam, lParam); break;
        case WM_LBUTTONUP: WMLButtonUp (hWnd, msg, wParam, lParam); break;
        case WM_MOUSEMOVE: WMMouseMove (hWnd, msg, wParam, lParam); break;
     }
     return (DefWindowProc(hWnd, msg, wParam, lParam));
  }


  int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR
             lpCmdString, int CmdShow)
  {
     WNDCLASS         wc;
     MSG              msg;
     RECT             rect;

     GlobalInstance   = hInstance;

     wc.cbClsExtra    = 0;
     wc.cbWndExtra    = 0;
     wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
     wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
     wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
     wc.hInstance     = hInstance;
     wc.lpfnWndProc   = WndProc;
     wc.lpszClassName = "ME";
     wc.lpszMenuName  = NULL;
     wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
     if (!RegisterClass(&wc))
     {
        MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
        return (0);
     }
```

```
Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
         0, 0, 640, 480, NULL, NULL, hInstance, NULL);
if (Window == NULL)
{
   MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
   return (0);
}
GetClientRect (Window, &rect);


bCreateWall = CreateWindow("BUTTON", "Create Wall", WS_CHILD | WS_VISIBLE, 0,
              100, DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window, NULL,
              hInstance, NULL);

bCreateCeiling = CreateWindow("BUTTON", "Create Ceiling", WS_CHILD |
              WS_VISIBLE, 0, 100+(DEFAULT_BUTTON_HEIGHT*2),
              DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window, NULL,
              hInstance, NULL);

bCreateFloor = CreateWindow("BUTTON", "Create Floor", WS_CHILD | WS_VISIBLE,
              0, 100+(DEFAULT_BUTTON_HEIGHT*4), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);

bPlaceStartPosition = CreateWindow("BUTTON", "Place Start", WS_CHILD |
              WS_VISIBLE 0, 100+(DEFAULT_BUTTON_HEIGHT*6),
              DEFAULT_BUTTON_WIDTH, DEFAULT_BUTTON_HEIGHT, Window, NULL,
              hInstance, NULL);

bPlaceDMPosition = CreateWindow("BUTTON", "Place DM", WS_CHILD | WS_VISIBLE,
              0, 100+(DEFAULT_BUTTON_HEIGHT*8), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);

bInsertEntity = CreateWindow("BUTTON", "Insert Entity", WS_CHILD | WS_VISIBLE,
              0, 100+(DEFAULT_BUTTON_HEIGHT*10), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);

bInsertItem = CreateWindow("BUTTON", "Insert Item", WS_CHILD | WS_VISIBLE,
              0, 100+(DEFAULT_BUTTON_HEIGHT*12), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);

bInsertSound = CreateWindow("BUTTON", "Insert Sound", WS_CHILD | WS_VISIBLE,
              0, 100+(DEFAULT_BUTTON_HEIGHT*14), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);

bInsertLight = CreateWindow("BUTTON", "Insert Light", WS_CHILD | WS_VISIBLE,
              0, 100+(DEFAULT_BUTTON_HEIGHT*16), DEFAULT_BUTTON_WIDTH,
              DEFAULT_BUTTON_HEIGHT, Window, NULL, hInstance, NULL);

RenderWindow = CreateWindow("STATIC", NULL, WS_CHILD | WS_VISIBLE | WS_BORDER,
              DEFAULT_BUTTON_WIDTH, 0, rect.right-rect.left-DEFAULT_BUTTON_
              WIDTH, rect.bottom-rect.top, Window, NULL, hInstance, NULL);
```

```
    Menu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_MENU));
    SetMenu (Window, Menu);

    PopupMenu = LoadMenu (hInstance, MAKEINTRESOURCE(IDR_POPUP_MENU));


    if (!raster.Init(RenderWindow)) return (0);

    GetClientRect (RenderWindow, &rect);
    ResizeGLWindow (rect.right-rect.left, rect.bottom-rect.top);

    SetGLDefaults();

    memset (&creation_coords, 0, sizeof(creation_coords));
    memset (&layer, 1, sizeof(layer));
    memset (&config, 0, sizeof(config));

    while (1)
    {
       Render();

       if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
       {
          if (msg.message == WM_QUIT) break;
          TranslateMessage(&msg);
          DispatchMessage (&msg);
       }
    }

    raster.Release(RenderWindow);
    delete map;


    return (1);
}
```

Conclusion

Chapter 7 was a doozy of a chapter! I realize most of the content was rather dry, but we've almost finished writing a working map editor and no map editor would be complete without these functions. I guess technically a map could do without some of these features, but where's the fun in having a map editor without items, start positions, entities, lights, sounds, etc.? If we didn't have any of those things in our map editor, you'd pretty much be a mouse running through a maze without the hope of any cheese!

From here on the map editor changes will be fairly simple and straight to the point. There isn't much left except for saving, opening, moving, deleting, and a few other minor topics. Then we begin the game itself, which compared to the map editor is a walk in the park!

This page intentionally left blank.

Chapter 8

Adding Useful Functionality

This chapter is where the magic of the map editor happens! We transform our basic map editor into a useful tool by adding many common functions like selecting, duplicating, deleting, and moving. These tools add the real functionality to any map editor and will help bring it to life. Without these functions we wouldn't be able to delete any mistakes nor would we be able to move objects around the screen or even duplicate objects for quick ease of use.

Selecting an Object

The ability to select objects is a major milestone in the development of our map editor. With this new function we can broaden the capabilities and functionality of our map editor by writing functions that take specific advantage of selected objects. Although the heading does specifically say selecting an object, the code we'll write to select the objects will also select items, entities, single-player/deathmatch start positions, and much more. This of course allows us to code the function once, which conserves time and extends the life of our keyboard!

There are three different ways in which we could write the object selection code. The first method requires you to assign numbers to each object that has the selection capability. Then we would check the status of the selection buffer to see which object, if any, is selected. This method of selecting objects is fairly complex for a beginner OpenGL programmer and really adds more source code than is necessary. The advantage to this method of selecting objects is that it isn't dependent on any specific resolution or bit depth, giving you the highest compatibility. We won't cover this method because it's too complicated for beginners, but I encourage you to research the selection buffer and update the final source code to take advantage of this style of selecting objects.

The next two styles of selecting objects are actually fairly similar. Essentially we would draw everything in the map using the unique selectable RGB values to easily distinguish between each thing in the map. After drawing everything in the map, we use the latest mouse coordinates to grab the current pixel data and then check to see if the pixel data received equals any of the values in the map. This would include looking through all the object, item, and entity arrays as well as the single-player/deathmatch start positions and checking each value. If any of the values equal that of the grabbed pixel data, we'll simply copy the new value into a global variable, which is used to describe the currently selected thing.

Now that we've discussed the basic theory behind the second and third styles of selecting the objects, we can discuss the differences between them. Essentially they both follow the same instructions; however, once everything is drawn we'll swap the buffers displaying the data on the screen and use the function GetPixel and the macros GetRValue, GetGValue, and GetBValue to retrieve the pixel data. The downside to this way of getting the pixel data is that it must draw the unique selectable colors to the screen before being able to get the data. Unfortunately, these colors will be shown to the user, which is a minor issue although it would be very annoying to see all the map data flash in color for a second whenever the user tries to select an object. Another drawback to this way of getting the pixel data is that it's slooooow! The GetDC function is rather slow by nature because there is normally no hardware acceleration as it's a standard Win32 API function call and not OpenGL related. This brings up another issue that may be something to consider for later on: Since the GetDC function is a Win32 API function call, it is not available on other platforms such as GNU/Linux. This may not be a major concern if you are only interested in Windows programming, but it's always nice to keep the possibilities open when writing your code! Because this style of grabbing the pixel data requires the redrawing of data to the screen, I don't recommend it. Although this may seem like a minor issue, it just doesn't look professional.

The other style of grabbing the pixel data follows the same basic ideas with the exception that we'll read from the back buffer instead of the front buffer by making a call to glReadBuffer. Once we've switched to the back buffer, we use the built-in OpenGL function glReadPixels to read the pixel data. The main difference between this style of grabbing the data versus the SwapBuffers style is that the data won't be drawn to the screen. The redraw isn't required because we're only rendering the data to the back buffer and then reading it. It's a rather sneaky method of grabbing the data but it works, and nicely I might add! This will be the method we'll use to grab the pixel data for the map editor because it allows the data to be drawn to the back buffer using the selectable RGB values without having to physically display it on the screen.

The problem with grabbing the pixel data over using the other methods of selecting objects is that it's dependent on the pixel data itself. If the user is in a desktop resolution that dithers the color data, as seen with any bits per pixel (BPP) resolution lower than 24-bit (16.7 million), the pixel data will be distorted. When the bit depth is 24- or 32-bit, each R, G, B (and A in the case of 32-bit) color component is represented as an 8-bit value. When your desktop bit depth is lower than 24-bit, each color component does not use the proper 8 bits per color. This leads to dithering, which basically chooses the closest color match to represent a specific color. This causes a problem when grabbing the data because if the returned pixel data is dithered, it won't be found in any of the selectable RGB values within the map! Most video cards (at the time of publication) support 24-bit if not 32-bit graphics resolutions, so we'll display a message at the beginning of the program launch that will inform the user that the map editor is designed for 24- and 32-bit bit depths. There should be very few compatibility issues as long as the user changes the resolution to any bit depth equal to or greater than 24-bit.

To begin coding we'll create a new global button called bSelectObject with the caption "Select Object." The button will be located at the top of the tool list above all the other function buttons. After creating the bSelect-Object button, we'll move to the OpenGL initialization routine and add a new line of code below it that will check the value of the current bit depth. If the bit depth is lower than 24-bit, we'll display a message on the screen informing the user that some functions of the map editor may not be possible without changing the current bit depth. To retrieve the current bit depth of the desktop we use the Win32 API function GetDeviceCaps and supply a device context (raster->hDC) for the first parameter and the predefined constant of which data you want returned. In the case of the bit depth, we'll use the constant BITSPIXEL. GetDeviceCaps can retrieve all sorts of information from the supplied device context that may be important to the development of your software, and I encourage you to research the function further. As mentioned previously, the function returns the data requested as an integer. Rather than declare another variable to store the data and then compare the value to 24, we'll simply embed the function call into an if statement, checking to see if the return value is below 24. If the value is below 24, then we'll display a message box that informs the user to switch resolutions. The following code will check the current device context bit depth and then display a message if the value returned is below 24.

```
if (GetDeviceCaps(raster.hDC, BITSPIXEL) < 24) MessageBox (Window, "This program
         is designed to run in 24/32-bit bit depths. To ensure compatibility,
         please change your bit depth to 24 or 32-bit color.", "Warning", MB_OK);
```

Now that we've finished updating the WinMain code to create the Select Object button, set the default selectable object values, and checked the desktop bit depth, we'll move to the WMCommand function where we'll add the user interaction to the newly created Select Object button. Much like all the other buttons we've created in the map editor so far, we'll add a new else-if statement to the enormous if statement, checking if Select Object (bSelect-Object variable) is equal to the lParam variable. If the values are equal, we'll set the current creation type (creation_coords.type) to NULL, which will indicate that the user is executing the default function — selecting an object. After setting the creation type to NULL, we'll call the function Show-SelectedButton to update the button highlights. The code for the new else-if statement is provided below for you to study.

```
else if (lParam == (LPARAM)bSelectObject)
{
   creation_coords.type = NULL;
   ShowSelectedButton();
}
```

There is no need to update the ShowSelectedButton function because it will now be the default function of the map editor. We can now assume that when there is no highlighted option or button we can select an object or item within the map without a problem! This style of user interface follows many popular 3D modeling programs where selecting is the default option. Realistically it makes sense to have this function treated as the default because it allows the user a quick and simple way to select and manipulate data. Now that we've written the button handling code for the Select Object button, we must update the WMLButtonUp function to physically start the selection process. To do this we'll add a new else-if statement to the already vast if statement, checking to see if the value of creation_coords.type is equal to NULL. If the values are equal, we'll call a new function called Select-AnObject. The function does not require any parameters. Once the user releases the left mouse button, the function SelectAnObject will be called, provided that the user isn't creating any type of object. If the user is creating some other type of object or placing some sort of item, he can click the Select Object button to set the map editor back to the default select object mode. Sounds simple enough, right? The code snippet added to the WMLButtonUp function is provided below.

```
else if (creation_coords.type == NULL) SelectAnObject();
```

When I originally wrote the chapter example I had named the function SelectObject but quickly realized this would be a mistake. Within the Win32 API there is a function called SelectObject, which is intended to copy objects such as brushes, pens, etc., into a specified device context. This could have posed a huge problem had we wanted to call the original Win32 function. This is just a friendly reminder to always be careful when you name your functions. It could end up biting you in the rear if you don't check it carefully! With this little reminder out of the way, we'll begin discussing the SelectAnObject function, which does the majority of the work involved in selecting an object.

The SelectAnObject function must be placed above the WMLButtonUp function to allow the function call from WMLButtonUp to take place. Otherwise, we must declare a prototype or forward declaration for the function to be called correctly. The function does not accept any parameters as input nor does it return any value once it's finished. Due to the nature of this function we must declare several variables to calculate the proper position of the mouse in relation to the drawing of the object on the screen. We must also have variables to store the returned pixel data as well as a variable to be used as a flag when the pixel data is found in the map. These variables are discussed in further detail below.

The first variable we'll declare in the SelectAnObject function is an array of three and is called rgb. The variable is of the GLubyte data type and will store the retrieved pixel data from the screen. This will be the array we'll use to compare the values of each selectable RGB value in the map arrays once the pixel data has been received. The next variable we'll declare is of the RECT data type. This variable, rect, will store the four location coordinates (left, right, top, and bottom) of a specified window. This information will be queried to help calculate the mouse coordinates.

After declaring the rect variable we'll declare two more variables, x and y. Both variables are of the long data type and will contain the specific X and Y coordinates of the mouse. Then we'll declare two more long variables, which will store the width and height of the RenderWindow window. The final variable we'll declare is of the bool data type and is called found. This variable has its default value set to false, which indicates the pixel data retrieved has not been found. If we search through the lists and find the values of the rgb array to be equal to one of the select_rgb arrays, then we'll set the found variable to true.

After declaring the variables inside the function we are ready to begin writing the code. The first thing we must do is set our globally defined select_rgb array to its default value by calling the memset function. After filling the array with the value 0xFF, we must get the location coordinates of the RenderWindow by making a call to the GetClientRect function and specifying the RenderWindow variable as the first parameter, followed by the address of the rect variable as the second. This will return the coordinates (top, bottom, left, and right) into the rect structure.

Now that we've filled the rect variable with the coordinates of the RenderWindow, we can begin the calculations to determine the mouse coordinates on the rendered screen. The first thing we'll do is calculate the width and height of the window by subtracting the right from the left and the bottom from the top and storing the results in the width and height variables. Next we'll calculate the X coordinate by using the mouse X coordinate stored in creation_coords.finish.mouse_x and subtracting the width of a button, which incidentally is the width of the RenderWindow indentation. After calculating the X coordinate, we'll calculate the Y coordinate by subtracting the value of creation_coords.finish.mouse_y from the height of the Render-Window. The end result will be stored in the y variable.

After calculating the Y coordinate we are ready to begin rendering everything in the map using the selectable RGB values. To begin the rendering process we'll clear both the depth and color buffers by making a call to glClear and specifying the appropriate buffers. Next we'll change the default line width to a thicker line by calling the OpenGL function glLineWidth and specifying the new width, which happens to be 4.0 in this case. We change the default line width because it's very difficult to select a 1-pixel line. A line width of 4.0 allows the user to select the line easily. With a thickness lower than 4.0, the line may be too thin to select when the pixel data is grabbed. A heavier line width may cause some lines to overlap and make them difficult or impossible to select. There is no technical explanation for using the value 4.0; I simply arrived at it through trial and error.

After setting the line width, we'll call the rendering functions for each type of item and object. The rendering will begin by calling the DrawSolid function, then we'll call the DrawEntities and DrawItems functions, provided the associated layering values permit them to be displayed. For those two function calls, the associated layering values must be true for them to be drawn. A simple if statement will suit this purpose. After drawing the items and entities in the level we'll call the DrawStartPosition and DrawDeath-MatchPositions functions to finish the rendering process. Since we've finished the rendering process we must remember to change the line width back to the original value by calling glLineWidth once again.

With the rendering finished we can finally write the code to grab the pixel data from the back buffer and attempt to find a match among all the object and item type arrays. To begin this process the first thing we'll do is switch to the back buffer by making a call to the OpenGL function glReadBuffer and specifying GL_BACK as the single parameter. The glReadBuffer function supports many different parameters to read from all sorts of buffers and places within the buffers. This is as far in depth as we'll get in this function; however, I encourage you to research it further if you would like to know the full potential of the function.

After switching to the back buffer it's finally time to grab the pixel data. We'll use the function glReadPixels to do this. Because of the flexibility of the function, there are many parameters that must be filled in for this function to work. The first two parameters of the function are the X and Y coordinates of the pixel data to read. We'll use the x and y variables declared earlier in the SelectAnObject function as these parameters since they contain the calculated XY coordinates for the mouse. The next two parameters are the width and height (in pixels) of the pixels you want to read. Not only can glReadPixels read a single pixel, it can read an entire user-defined area of data. This type of read can be very useful when you want to create a screen capturing function within a game. (Hint! Hint!) Since we only need a single pixel, we'll set both parameters to 1.

The fifth parameter is the format of the pixel data we are requesting. This is where we specify whether we want to grab just the red, green, blue, RGB values, BGR values, etc., of the pixel. If we were to select each color component separately (GL_RED for red, GL_GREEN for green, and finally GL_BLUE for blue), we would require three calls to the function to accomplish the same task as simply specifying GL_RGB. Of course specifying each color component on its own has its merits depending on what you are developing, but in our case the GL_RGB value is ideal because it returns all three values (red, green, blue) in an array.

After specifying the format of the pixel data we must specify the data type we want the pixel data returned as. Although it may seem logical to specify float or int, OpenGL has built-in constants that contain the valid data types you can request. In our specific case we will use the value GL_UNSIGNED_BYTE because our pixel data is stored entirely in GLubyte data types, which is the most compatible format. If for some reason we wanted to return the pixel data in float form, we could use the value GL_FLOAT instead of GL_UNSIGNED_BYTE.

The reason we didn't use the GLfloat data type to store the selectable pixel data instead of GLubyte is that some platforms will change data when it's stored in the variable. Although it sounds weird, I've noticed in certain cases that when you grab the pixel data it may be 0.01 off from its original screen color. This may not seem like a big issue, but it could cause major problems if you have several objects that have very close unique colors for selecting. In some cases the difference could be even greater than 0.01, depending on the manufacturer of the video card, so it's easier to use GLubyte, which from my experiences doesn't have the same problem.

The final parameter of the function is the address of the variable to which we want the pixel data stored, which happens to be the rgb variable. We've finally selected the pixel data, but what do we do now? We must loop through every array in the map that has selectable RGB values and check to see if the values in the rgb array are equal to those in each select_rgb variable. If the variables are equal, we'll set the found variable to true, indicating that we've got a match. Once all the looping has been finished, we check the value of the found variable to see if it's true. If the variable is true, we simply set the contents of the global select_rgb variable to the values of the local rgb variable, which has the retrieved pixel data. By copying the values from the rgb variable to the global select_rgb, we're storing the object or item that is selected, which will come in handy throughout the development of the map editor. This type of information is very useful when drawing items because it will help the user easily spot which object is selected. The source code for the SelectAnObject function is provided here for you to study.

```
void SelectAnObject()
{
   GLubyte  rgb[3];
   RECT     rect;
   long     x;
   long     y;
   long     width;
   long     height;
   bool     found=false;

   memset (&select_rgb, 0xFF, sizeof(select_rgb));

   GetClientRect (RenderWindow, &rect);
   width  = rect.right - rect.left;
   height = rect.bottom - rect.top;
   x      = creation_coords.finish.mouse_x - DEFAULT_BUTTON_WIDTH;
   y      = height - creation_coords.finish.mouse_y;

   glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
   glLineWidth(4.0);
      DrawSolid();
      if (layer.draw_entity) DrawEntities(true);
      if (layer.draw_item) DrawItems(true);
   glLineWidth(1.0);

   glReadBuffer (GL_BACK);
   glReadPixels (x, y, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &rgb);

   for (long i = 0; i < map->header.max_objects; i++)
   {
      if (map->object[i].select_rgb[0] == rgb[0] &&
         map->object[i].select_rgb[1] == rgb[1] &&
         map->object[i].select_rgb[2] == rgb[2]) found = true;
   }

   for (i = 0; i < map->header.max_entities; i++)
   {
      if (map->entity[i].select_rgb[0] == rgb[0] &&
         map->entity[i].select_rgb[1] == rgb[1] &&
         map->entity[i].select_rgb[2] == rgb[2]) found = true;
   }

   for (i = 0; i < map->header.max_items; i++)
   {
      if (map->item[i].select_rgb[0] == rgb[0] &&
         map->item[i].select_rgb[1] == rgb[1] &&
         map->item[i].select_rgb[2] == rgb[2]) found = true;
   }

   for (i = 0; i < map->header.max_lights; i++)
   {
      if (map->light[i].select_rgb[0] == rgb[0] &&
         map-> light [i].select_rgb[1] == rgb[1] &&
         map-> light [i].select_rgb[2] == rgb[2]) found = true;
   }

   for (i = 0; i < map->header.max_sounds; i++)
   {
      if (map->sound[i].select_rgb[0] == rgb[0] &&
         map->sound[i].select_rgb[1] == rgb[1] &&
         map->sound[i].select_rgb[2] == rgb[2]) found = true;
   }

   if ((map->details.single_player.select_rgb[0] == rgb[0] &&
      map->details.single_player.select_rgb[1] == rgb[1] &&
      map->details.single_player.select_rgb[2] == rgb[2]) ||

      (map->details.deathmatch[0].select_rgb[0] == rgb[0] &&
      map->details.deathmatch[0].select_rgb[1] == rgb[1] &&
      map->details.deathmatch[0].select_rgb[2] == rgb[2]) ||

      (map->details.deathmatch[1].select_rgb[0] == rgb[0] &&
      map->details.deathmatch[1].select_rgb[1] == rgb[1] &&
      map->details.deathmatch[1].select_rgb[2] == rgb[2])) found = true;

   if (found)
   {
      select_rgb[0] = rgb[0];
      select_rgb[1] = rgb[1];
      select_rgb[2] = rgb[2];
   }
}
```

The downside to writing the object selection code is that we must update all the drawing functions (e.g., DrawLights, DrawSounds, etc.) to handle a selected item that is stored in the select_rgb global variable. Each drawing function will be updated with one parameter of the bool data type. The variable parameter will be named want_srgb, which is an abbreviated description for "want selectable RGB." The parameter of the function will have a default value of false to keep the rest of the function compatible with all the function calls we've already made. An example of the original and updated DrawSounds function is given below so you can see the difference between the old and new functions.

Original function:

```
void DrawSounds()
```

Updated function:

```
void DrawSounds(bool want_srgb=false)
```

Besides updating each drawing function parameter, we must update the code within the function to actually draw a selected point. To update the code we'll begin with the main drawing loop of each draw function, where we'll check to see if the array variable select_rgb contents are equal to the values in the global select_rgb. If the contents are equal, we will draw them in red. If the values are not equal, we'll draw the objects or items in their normal colors and continue through the loop. When the user selects something in the level, it should immediately turn red to indicate to the user that it's selected, as seen in most 3D modeling/map editing tools. If the user attempts to select an object but misses, the selected object will return to its original color. Sounds simple enough, right?

I'm not going to display the source code for each function we've updated because the chapter would end up being enormous! Instead, we'll display the updated source code from the DrawSounds function. With every drawing function we would simply replace the references to the sound array and put the appropriate array in its place. The example source code is provided below.

```
if (map->sound[i].select_rgb[0] == select_rgb[0] && map->sound[i].select_rgb[1]
          == select_rgb[1] && map->sound[i].select_rgb[2] == select_rgb[2])
          glColor3f (1.0f, 0.0f, 0.0f);
else
{
   if (want_srgb) glColor3ubv (map->sound[i].select_rgb);
   else glColor3f (0.0f, 1.0f, 0.0f);
}
```

With the object selection code finished we can begin discussing more useful functions to implement!

Moving Objects

Now that we've got the object selection code out of the way, we can focus our efforts on putting useful functionality into the map editor. The first function we'll add to the map editor is the ability to move objects around the world. Unlike previous functions where we've added a new button to the screen, we're going to use an existing menu item that is available in the pop-up menu when the user presses the right mouse button. To begin the coding, the first thing we need to do is create a new enumeration to store the mode type information for the extra functions. The extra functions will include things like moving and deleting, which don't fit into our generic creation system but still require the same style interface. It's simply easier to create a subtype for the different creation modes than rewrite all our code to include one specific feature. This also gives us the ability to easily add more functions to our map editor without having to add many new lines of code.

The name of our first enumeration value will be MODE_TYPE_MOVE and have the starting value of 0. Even though the user clicks the Move button he or she is actually starting a creation mode, but rather than placing an item, entity, or wall, a mode type is assigned that will move a selected object. The source code for the newly written enumeration is below for you to examine.

```
enum { MODE_TYPE_MOVE = 0 };
```

The type variable in the CREATION_COORDS structure will be used to store the mode type enumerations for the extra functions we'll be creating in this chapter. When we want to use the Move function we set this variable to the newly created enumeration MODE_TYPE_MOVE and let the rest of the code do its magic! With the type variable now added to the CREATION_COORDS structure, we can begin writing the code necessary to move the objects.

Now that we've updated the appropriate structures with the new variable we can begin the actual coding for the Move function. This is where the real magic begins! The Move function is activated when the user right-clicks with the mouse, selects the Move menu item, and finally left-clicks the menu item. With this in mind, we'll move to the WMCommand function to update the source code. When the user clicks the Move menu item a WM_COMMAND message will be sent that will call the WMCommand function. We'll simply add a new else-if clause, which will check to see if the value of the wParam variable is equal to the resource ID of the Move menu item (ID_POPUP_MOVE).

If the values are equal, then the user has clicked the Move menu item and we must execute another if statement, checking the values of the select_rgb array. If all three indexes of the select_rgb array are not equal to 255, we'll execute the code written below it, starting with setting the creation_coords.start variable to the value of creation_coords.finish to set the start mouse coordinates to the current mouse coordinates. Then we set the creation_coords.mode to CREATE_MODE_START to indicate we want to track the mouse, and finally we set our newly created creation_coords.type variable to MODE_TYPE_MOVE to indicate we'll be moving an object through the world.

We'll also add an else clause to the if statement saying that if all three values of the select_rgb array are equal to 255, we'll display a message using a MessageBox stating that an object must be selected before an object can be moved. This eliminates the possibility of the user clicking the Move button and nothing happening. It's also a good idea to always check values like this just to be on the safe side. The source code for the WMCommand update is provided below for you to examine.

```
else if (wParam == ID_POPUP_MOVE)
{
   if (select_rgb[0] != 255 && select_rgb[1] != 255 && select_rgb[2] != 255)
   {
      creation_coords.start     = creation_coords.finish;
      creation_coords.mode      = CREATE_MODE_START;
      creation_coords.type      = MODE_TYPE_MOVE;
```

```
    }
    else MessageBox (hWnd, "An Object must be Selected", "Error", MB_OK);
}
```

With an object selected and the Move button clicked, we must write the code to physically move the object. To do this we must write a function called Move, which will move the selected object and then update the WMMouseMove function, which is called every time the mouse is moved. The Move function will be placed above the SelectAnObject function source code. There are no parameters required for the function and there is no return data type needed. Inside our newly declared function we'll declare two new variables of the float data type. Both variables will store the offset from the previous mouse position to the current mouse coordinates for the screen. This information is used to move the object around the screen in world coordinates. This method of moving objects is simple yet effective, which is what we want!

After declaring the variables, we subtract the value of creation_coords.start.world_x from the value of creation_coords.finish.world_x and store the result in the x variable. We'll do the same calculation for the z-axis, substituting z for the z-axis and storing the result in the z variable. These calculations give the X and Z offsets for moving the object in world coordinates. This information is crucial to moving the selected object because we must add both values to the current x- and z-axis coordinates. In the case of map objects, we must add these values to every single vertex in the object. In every other instance, we simply add the values to the main X and Z coordinate values to move the object.

Before we begin adjusting the current coordinates, we'll set the creation_coords.start variable to the value of creation_coords.finish, which allows us to reset the mouse coordinates for the next time the user moves the mouse. This will allow us to recalculate the proper offset in real time, so the user can see things moving visually. With the mouse coordinates reset we can begin moving the data. Unfortunately there's no short way of doing it! We must loop through every array in our MAP class, checking to see if the globally declared select_rgb array values are equal to the values in the individual map select_rgb arrays.

In the case of the single-player start position we can simply check it using an if statement without the for loop. If the select_rgb values are equal, we'll simply add the values of x and z to the .xyz[0] and .xyz[2] indexes of the array to move the object. As mentioned, if the select_rgb values are equal in the object's array we must create a second loop to loop through the entire vertex list to adjust each X and Z location. Once we've set the values we simply exit from the function using a no-parameter version of the return function. This concludes the code for the Move function, so we can display the source code as follows.

```
void Move()
{
   float     x;
   float     z;

   x = (float) creation_coords.finish.world_x - creation_coords.start.world_x;
   z = (float) creation_coords.finish.world_z - creation_coords.start.world_z;

   creation_coords.start = creation_coords.finish;


   if (map->details.single_player.select_rgb[0] == select_rgb[0] &&
      map->details.single_player.select_rgb[1] == select_rgb[1] &&
      map->details.single_player.select_rgb[2] == select_rgb[2])
   {
      map->details.single_player.xyz[0] += x;
      map->details.single_player.xyz[1] += z;
      return;
   }

   for (long i = 0; i < 2; i++)
   {
      if (map->details.deathmatch[i].select_rgb[0] == select_rgb[0] &&
         map->details.deathmatch[i].select_rgb[1] == select_rgb[1] &&
         map->details.deathmatch[i].select_rgb[2] == select_rgb[2])
      {
         map->details.deathmatch[i].xyz[0] += x;
         map->details.deathmatch[i].xyz[2] += z;
         return;
      }
   }

   for (i = 0; i < map->header.max_objects; i++)
   {
      if (map->object[i].select_rgb[0] == select_rgb[0] &&
         map->object[i].select_rgb[1] == select_rgb[1] &&
         map->object[i].select_rgb[2] == select_rgb[2])
      {
         for (long i2 = 0; i2 < map->object[i].max_vertices; i2++)
         {
            map->object[i].vertex[i2].xyz[0] += x;
            map->object[i].vertex[i2].xyz[2] += z;
         }
         return;
      }
   }

   for (i = 0; i < map->header.max_entities; i++)
   {
      if (map->entity[i].select_rgb[0] == select_rgb[0] &&
         map->entity[i].select_rgb[1] == select_rgb[1] &&
         map->entity[i].select_rgb[2] == select_rgb[2])
      {
         map->entity[i].xyz[0] += x;
         map->entity[i].xyz[2] += z;
         return;
      }
   }

   for (i = 0; i < map->header.max_items; i++)
   {
      if (map->item[i].select_rgb[0] == select_rgb[0] &&
         map->item[i].select_rgb[1] == select_rgb[1] &&
         map->item[i].select_rgb[2] == select_rgb[2])
      {
         map->item[i].xyz[0] += x;
         map->item[i].xyz[2] += z;
         return;
      }
   }

   for (i = 0; i < map->header.max_sounds; i++)
   {
      if (map->sound[i].select_rgb[0] == select_rgb[0] &&
         map->sound[i].select_rgb[1] == select_rgb[1] &&
         map->sound[i].select_rgb[2] == select_rgb[2])
      {
         map->sound[i].xyz[0] += x;
         map->sound[i].xyz[2] += z;
         return;
      }
   }

   for (i = 0; i < map->header.max_lights; i++)
   {
      if (map->light[i].select_rgb[0] == select_rgb[0] &&
         map->light[i].select_rgb[1] == select_rgb[1] &&
         map->light[i].select_rgb[2] == select_rgb[2])
      {
         map->light[i].xyz[0] += x;
         map->light[i].xyz[2] += z;
         return;
      }
   }
}
```

The modification we'll make to the source code in order to get the Move function working is to update the WMMouseMove function. As we already know, the WMMouseMove function is called every time the mouse is moved, hence the name! We'll use this to our advantage when moving objects in the world by adding a line in the function's if statement to check if the value of creation_coords.type is equal to the enumeration MODE_TYPE_MOVE. In the event the values are equal, we'll call the function Move to move the object. If the user doesn't have anything selected, the if statements to match select_rgb values will fail and nothing will be moved. This completes the Move object's functionality of the map editor — I told you it was simple!

Assigning Textures to Objects

Assigning textures to objects brings us another step closer to having the map editor finished and also makes our game objects look more realistic. In case you're wondering, a *texture* is a picture that we can assign to a wall, floor, ceiling, or anything else in the game to make it look like something. I admit it's a very vague description, but imagine this: You're playing the newest first-person shooter but all the walls in the game are solid colors. Rather than having textures of bricks, paint, rock, or even metal, you're stuck with pink and purple walls! Of course there's nothing wrong with having walls that are solid pink or purple, but you get my point!

By adding a texture to a wall, we can go from walking through a neon psychedelic acid trip dungeon to a dark medieval dungeon by simply adding a texture that looks like dark green stained brick to the walls. With this in mind, let's begin the coding process by moving to the map.h file where we'll create a new enumeration. This new enumeration will store several texture types that can be assigned to a texture. We use these enumeration values to customize the appearance of a texture. The first enumeration we'll write is called TEXTURETYPE_NONE, which will have a default value of 0. This value is used with texture layers 1 and above to specify whether or not there is a texture assigned.

Because our game engine supports multiple textures being drawn at once, commonly known as *multitexturing*, we must have an enumeration value that specifies if the second texture is assigned or not. If the second texture has the style TEXTURETYPE_NONE assigned to it, then the object doesn't have a second texture and we don't need to worry about rendering it. The next value we'll add to the enumeration list is the value TEXTURETYPE_REGULAR, which is the default value for texturing. The first texture style should always be set to this value. The user can then modify this to one of the other values afterward.

After adding the TEXTURETYPE_REGULAR value to the enumeration list, the next value we'll add to the list is TEXTURETYPE_MASKED, which draws the associated texture with transparent portions. The final value we'll add to the enumeration list is the value TEXTURETYPE_BUMPMAP, which specifies that the texture should be blended to look like it's slightly embossed. This value is only available with the second texture style because we must blend the second texture with the first to produce the appropriate embossing. In case you're wondering, embossing is the process that makes a picture look 3D, or as if it's got depth, by merely changing the outlining/shadowing. To do normal bumpmapping we would use two textures. For example, the first texture style would be set to TEXTURETYPE_REGULAR and be a picture of a ceramic tile, and the second texture style would be set to TEXTURETYPE_BUMPMAP and would show the surface of the tile, including small indentations. By shifting the blending options in OpenGL, we can accomplish a 3D look by mixing the two textures together. The process sounds much more complicated than it actually is. Now that we're finished creating the enumeration we can display its contents.

```
enum {
      TEXTURETYPE_NONE = 0,
      TEXTURETYPE_REGULAR,
      TEXTURETYPE_MASKED,
      TEXTURETYPE_BUMPMAP
   };
```

With the texture type enumeration finished we can add a new method to the MAP class to insert textures into a specified object. Although we only want to give support for two layers of textures, we will allow the user to dynamically insert as many triangles as needed. Of course we're designing our map editor to work with only two textures, but you can easily modify the code we're discussing to support 4, 8, or even 16 texture units. Unfortunately there are some limitations to how many texture units you can draw at a time. The standard (at the time of publication) is two to four, depending on your hardware vendor. I chose to support two because once you learn the basics you can modify the code to support many more units. When we get to the game engine itself, we'll discuss how to check the maximum texture units available in your video card, but that's beyond the scope of this chapter, so we'll go back to creating the new method.

The new method we'll be creating is called InsertTexture and will have three parameters. The first parameter is called obj and it's of the long data type. This variable is used to specify which object index to insert the textures into. The next variable is a pointer to a char data type called filename, which will store the input filename for the texture. The final parameter, style, is a long and has a default value of TEXTURETYPE_REGULAR. The newly created method has a bool return data type. The method itself acts like any other insertion method we've created so far. We declare a new storage variable (in this case new_texture) to store the data for the new record. We then copy all the parameters to the new storage variable. Here there are only two variables to copy (filename and style). The other variables in the new_texture variable will be calculated and set by the game engine once the maps load.

With the data structure filled in, we then check to see if the value of object[obj].max_textures is equal to 0. If the value is equal to 0, then we'll allocate memory for one index. If the value is not equal to 0, then we'll declare a new pointer variable called temp, which is of the MAP_TEXTURE type. This variable will have memory allocated to it, then we'll copy the data from the object[obj].texture array to the temp variable. Once the data is copied, we'll destroy the object[obj].texture array and reallocate memory, specifying two extra records. This gives us one buffer record plus one extra record for storing data. We then simply copy the data back from the temp variable to the object[obj].texture array and deallocate the memory. This will conclude the if statement and we simply set the value of object[obj].texture [object[obj].max_textures] equal to the value in the variable new_texture. This will set the last record in the object[obj].texture equal to the value of new_texture. Then we increment the object[obj].max_textures variable by one and exit the function by calling the return function with a parameter of true. The source code for the InsertTexture method is provided below.

```
bool MAP::InsertTexture(long obj, char *filename, long style)
{
   MAP_TEXTURE new_texture;

   strcpy (new_texture.filename, filename);
   new_texture.style = style;

   if (object[obj].max_textures == 0) object[obj].texture = new
        MAP_TEXTURE[object[obj].max_textures+1];
   else
   {
      MAP_TEXTURE *temp;

      temp = new MAP_TEXTURE[object[obj].max_textures+1];
      for (long i = 0; i < object[obj].max_textures; i++) temp[i] =
           object[obj].texture[i];

      delete [] object[obj].texture;
      object[obj].texture = new MAP_TEXTURE[object[obj].max_textures+2];

      for (i = 0; i < object[obj].max_textures; i++) object
           [obj].texture[i] = temp[i];
      delete [] temp;
   }

   object[obj].texture[object[obj].max_textures] = new_texture;
   object[obj].max_textures++;

   return (true);
}
```

With this source code the InsertTexture function is completed, and we can start the resource editor and insert a new dialog box into our resource code. The new dialog box will have a resource ID of IDD_ASSIGNTEXTURE and a caption of "Assign Texture." This dialog box will be the basis of assigning textures to the objects in the game. It is a good idea to change the properties of the dialog box to center it in the middle of the screen as well. This of course is not a necessity, but it makes life a lot easier when you are actually using the software. Now that we've created the base dialog box we can begin adding controls to it!

The first controls we'll add to the dialog box are two edit controls. The resource names of the two controls should be set to IDC_ASSIGNTEX-TURE_FILENAME1 and IDC_ASSIGNTEXTURE_FILENAME2. These two controls will store the filenames for the two texture units we'll allow the user to customize. Although it's personal preference where you put them in the dialog box, I prefer having the first edit control at the top, then placing the second control in the vertical middle of the dialog box.

Next we'll add two combo box controls to the dialog with the names IDC_ASSIGNTEXTURE_TYPE1 and IDC_ASSIGNTEXTURE_TYPE2. Both combo boxes should have their properties changed so they don't have the Sort option enabled. When the Sort option is enabled it causes problems when you try to quickly grab the currently selected items because they are in alphabetical order instead of the order in which they were inserted. When editing the properties, also make sure you change the type of combo box to Drop List to ensure it functions properly. Sometimes the function's default options will select a different type of combo box, and we want to use the Drop List because it looks just like a regular combo box. If you want to make your dialog box slightly easier to use, you may want to consider putting in static controls to describe the layout. I added two static controls to describe where texture 1 and texture 2 features begin. A screen shot of my dialog is provided below.

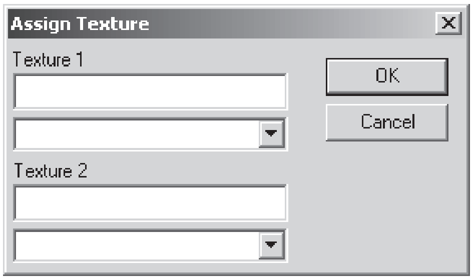


Figure 8.1: Assign Texture dialog box

Now that we've finished creating the dialog box, we can move to the main source file and begin updating the functions necessary to add assigning texture support. The first thing we'll code in the main source file is a new dialog procedure called AssignTextureDlgProc. This function will handle the messages sent from a dialog we'll be creating next. Within AssignTexture-DlgProc we'll add a case statement for the msg parameter to handle incoming messages for the WM_INITDIALOG and WM_COMMAND messages.

The WM_INITDIALOG case option will have a new variable declared called obj, which is of the long data type. The variable has a default value set to –1. After declaring the obj variable we'll create a for loop to loop through all the objects (header.max_objects) in the map. If values of the select_rgb array are equal to any of the map->object[i].select_rgb array values, then we'll set the value of obj to the index (i) in which it was found. This process will make sure that we've got a real object (wall, floor, ceiling) selected and return the number of that object into the obj variable. This information is critical since we'll be inserting data into different dialog controls in just a few moments. If we didn't have the object ID, then we wouldn't know which object index to use and couldn't display the current settings on the screen at all.

After retrieving the object index from the for loop, we'll create an if statement to ensure that the object actually found something! If the value of the obj variable is less than 0, we know that no object was selected and we should display a message box on the screen stating that a texture cannot be assigned to the object, then exit the dialog. At the bottom of the if statement we'll add an else clause to allow us to set up the dialog box to see the settings. Before we continue any further in setting up the dialog box, we should ensure that the object selected has textures already in it. To do this we'll write a small if statement to check the value of map->object[obj].max_textures. If the value is equal to 0, which is the default, then we'll call our newly created method map->InsertTexture to add some default textures for both the first and second texture units. As we know, the first parameter is the object number (obj), and the second parameter is the filename. Since we'll have two separate calls to the function, we'll simply call the first texture "MYBITMAP1.BMP" and the second "MYBITMAP2.BMP".

We must always ensure that the object texture arrays have two entries before we continue through the WM_INITDIALOG message. If we didn't check each time the dialog box loaded, we could cause the system to crash because we'd be accessing memory that hasn't been allocated. If the map editor crashed while the user was trying to texture a level, you couldn't imagine how annoyed he'd be. There's nothing worse than losing work due to a software bug or circumstances beyond the user's control.

Now that we've ensured that there's an object selected and there are textures in the object, we can begin setting the values in the dialog controls. The first two values we'll set are the filenames for the two texture units. Using the SetDlgItemText function, we'll set the IDC_ASSIGNTEXTURE_FILENAME1 and IDC_ASSIGNTEXTURE_FILENAME2 controls to the contents of map->object[obj].texture[0].filename and map->object[obj].texture[1].filename, respectively. Next, we'll set the contents of the combo box for the first texture unit. Before we add any data we'll send the combo box message CB_RESETCONTENT to delete all the contents inside the combo box. Then we'll send two messages using the CB_ADDSTRING message with the string "Regular" in the first message and "Masked" in the second. As mentioned before, the first texture unit only has two texturing options available. Finally, we'll set the current selection of the combo box using the message CB_SETCURSEL and specifying map->object[obj].texture[0].style–1 for the selection. We use –1 since the message starts at 1 and the selection message starts at 0. The messaging for the second texture unit works the same way except that we'll be adding the word "None" to the beginning of the list of texture styles and the word "BumpMap" to the end. This will give us a total of four available texture styles. Since we're using the None texture style we don't need the –1 at the end of the selection message. After updating the second CB_SETCURSEL message we're finished with the WM_INITDIALOG message and can write the code to handle the WM_COMMAND message.

The WM_COMMAND message is much easier to deal with than the beginning WM_INITDIALOG message. Within the message we check to see if the user pressed the Cancel button by checking if the value of wParam is equal to the value of IDCANCEL. If the values are equal, then the user clicked Cancel and we exit the dialog box without saving the changes. If the user clicks the OK button, then wParam would equal IDOK and we would have to execute several lines of code. First we would declare another local variable called obj, which is of the long data type and has a default value of –1. We'll search through the list of objects again to get the object ID number. We don't need to verify if the object number set in the obj variable is valid or not because we already know from WM_INITDIALOG that the user selected a proper object.

After retrieving the object number into the obj variable we simply get the data input from the two edit controls by calling GetDlgItemText, specifying IDC_ASSIGNTEXTURE_FILENAME1 and IDC_ASSIGNTEXTURE_FILENAME2 as the resource IDs, and returning the filename strings to the map->object[obj].texture[0].filename and map->object[obj].texture[1].filename variables. This sets the filenames for both variables so we know which textures are assigned to each texture unit/object. Then we send the CB_GETCURSEL message to the two combo boxes, IDC_ASSIGNTEXTURE_TYPE1 and IDC_ASSIGNTEXTURE_TYPE2, and store the returned values in the map->object[obj].texture[0].style, adding +1 to the first texture unit and map->object[obj].texture[1].style variables. We add +1 to the first texture unit because it doesn't use the TEXTURETYPE_NONE enumeration but rather TEXTURETYPE_REGULAR, and it keeps it compatible when we initially set the option when the dialog starts the next time the texture function is called. The last line inside the WM_COMMAND message simply ends the dialog box by calling the EndDialog function.

To finish the dialog we simply close the case statement brackets and add a return value of 0 to the bottom line of the dialog box code to allow messages to easily flow through the dialog box. This completes the source code for the AssignTextureDlgProc function, so we can display the source code below.

```
LRESULT CALLBACK AssignTextureDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:

         long obj = -1;

         for (long i = 0; i < map->header.max_objects; i++)
         {
            if (map->object[i].select_rgb[0] == select_rgb[0] &&
               map->object[i].select_rgb[1] == select_rgb[1] &&
               map->object[i].select_rgb[2] == select_rgb[2]) obj = i;
         }
         if (obj < 0)
         {
            MessageBox (hWnd, "Selected Object cannot be assigned a texture",
                        "Oops!", MB_OK);
            EndDialog (hWnd, 0);
         }
         else
         {
            if (map->object[obj].max_textures == 0)
            {
               map->InsertTexture (obj, "MYBITMAP1.BMP");
               map->InsertTexture (obj, "MYBITMAP2.BMP");
            }

            SetDlgItemText (hWnd, IDC_ASSIGNTEXTURE_FILENAME1, map->object
                           [obj].texture[0].filename);
            SetDlgItemText (hWnd, IDC_ASSIGNTEXTURE_FILENAME2, map->object
                           [obj].texture[1].filename);

            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE1, CB_RESETCONTENT,
                               0, 0);
            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE1, CB_ADDSTRING,
                               0, (LPARAM)"Regular");
            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE1, CB_ADDSTRING,
                               0, (LPARAM)"Masked");

            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE1, CB_SETCURSEL,
                               map->object[obj].texture[0].style-1, 0);

            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE2, CB_RESETCONTENT,
                               0, 0);
            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE2, CB_ADDSTRING,
                               0, (LPARAM)"None");
            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE2, CB_ADDSTRING,
                               0, (LPARAM)"Regular");
            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE2, CB_ADDSTRING,
                                0, (LPARAM)"Masked");

            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE2, CB_ADDSTRING,
                                0, (LPARAM)"BumpMap");
            SendDlgItemMessage (hWnd, IDC_ASSIGNTEXTURE_TYPE2, CB_SETCURSEL,
                                map->object[obj].texture[1].style, 0);
         }
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            long obj = -1;
            for (long i = 0; i < map->header.max_objects; i++)
            {
               if (map->object[i].select_rgb[0] == select_rgb[0] &&
                  map->object[i].select_rgb[1] == select_rgb[1] &&
                  map->object[i].select_rgb[2] == select_rgb[2]) obj = i;
            }

            GetDlgItemText (hWnd, IDC_ASSIGNTEXTURE_FILENAME1, map->
                            object[obj].texture[0].filename, 500);
            GetDlgItemText (hWnd, IDC_ASSIGNTEXTURE_FILENAME2, map->
                            object[obj].texture[1].filename, 500);

            map->object[obj].texture[0].style = SendDlgItemMessage (hWnd,
                 IDC_ASSIGNTEXTURE_TYPE1, CB_GETCURSEL, 0, 0) + 1;
            map->object[obj].texture[1].style = SendDlgItemMessage (hWnd,
                 IDC_ASSIGNTEXTURE_TYPE2, CB_GETCURSEL, 0, 0);


            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

To finish the texture assignment process, we need to update WMCommand and change its functionality from displaying a message box to running the dialog box we just created. It's simply a matter of deleting the already written line of code and replacing it with a line that calls the DialogBox function, specifying the resource ID IDD_ASSIGNTEXTURE as the second parameter inside the MAKEINTRESOURCE macro and using the AssignTextureDlgProc function as the fourth parameter. The updated WMCommand code is provided below.

```
else if (wParam == ID_POPUP_TEXTURE) DialogBox (GlobalInstance,
         MAKEINTRESOURCE(IDD_ASSIGNTEXTURE), NULL, (DLGPROC)AssignTextureDlgProc);
```

This concludes the section on assigning textures to objects, so we can move to the next topic.

Duplicating Objects

The Duplicate function is probably the easiest of all the main three functions (Move, Delete, and Duplicate) to write. It doesn't require adding code to the map.h file, nor do you have to worry about bringing up any new dialogs. It's simply a matter of creating a new function called Duplicate with a single parameter called hWnd, which is a window. Within the function we'll have a loop for every array in our map class, looping from 0 to the maximum value in the loop (e.g., 0 to the value of map->header.max_objects). Inside the loop we'll check to see if the value of the array select_rgb variable is equal to the globally declared select_rgb variable. If all three indexes of the array are equal to each other, then we know that this object is the one selected to be duplicated.

The duplication process is different for each array because we'll use the array's associated map Insert function and specify all the values from the currently selected entity index in the InsertEntity method. By reusing the Insert functions we've been creating over the last several chapters we can minimize the amount of work we actually have to do in the map editor. However, there are some exceptions to the way we duplicate things. Objects and lights work differently because they have embedded arrays in their data structures, which means when we insert the new light or object we must insert all the other data such as the light inclusion list or the textures, triangles, and vertices.

In order for you to understand how it works we'll discuss three different cases for duplicate. Because most arrays are very similar it's fairly easy to do the duplicate for others. The first piece of code we'll write will duplicate objects. As mentioned earlier, we'll create a for loop to loop from 0 to the value of map->header.max_objects. Within the for loop we'll check to see if any of the object select_rgb array values are equal to the global select_rgb array values. If we find a match, we begin the duplicate process. For convenience, we'll declare a new variable inside called last_entry before we duplicate the objects. The variable, which is of the long data type, will have a default value set to the value of header.max_objects. This variable will store the last entry in the object index before we inserted a new object. This information is useful because we would otherwise have to specify map->header.max_objects–1 as the object ID every time we wanted to insert triangles, vertices, and textures.

To begin the duplicate process, the first thing we'll do is insert a new object into the map. To do this we call map->InsertObject and specify the name (map->object[i].name) of the object, the type of object (map->object[i].type) as the second parameter, the special variable (map->object[i].special), the collidability of the object (map->object[i].is_collidable), and whether or not the object is visible (map->object[i].is_visible). This will insert a new object into our map, into which we must insert the data itself.

Inserting the data into the object is simply a matter of creating a new loop for the individual values of vertices, triangles, and textures. Each loop will call the appropriate Insert functions and use the data from the existing arrays to insert new data. In the case of vertices, we simply loop from 0 to map->object[i].max_vertices, calling map->InsertVertex with each iteration and specifying the object to be inserted to (last_entry), the XYZ coordinates (map->object[i].vertex[i2].xyz[0] for X, etc.), the RGBA values for the coloring (map->object[i].vertex[i2].rgba[0] for red, etc.), the normal values (map->object[i].vertex[i2].normal[0] for the X), and finally the fog depth for the vertex (map->object[i].vertex[i2].fogdepth).

The triangles array works in the same manner as vertices, except we'll obviously be calling the method InsertTriangle. The first parameter for the InsertTriangle function is the object number to be inserted into (last_entry). Then we include the three vertex points required to make the triangle (map->object[i].triangle[i2].point[0], map->object[i].triangle[i2].point[1], and map->object[i].triangle[i2].point[2]). The final six parameters are the first layer UV coordinates for each triangle point.

To finish the object duplication process we must create one last for loop to insert textures into the newly created object. The for loop will loop from 0 to the value of map->header.max_textures. With each iteration of the loop we'll call the map->InsertTexture method, supplying the last_entry variable as the first parameter, the map->object[i].texture[i2].filename as the second parameter, and map->object[i].texture[i2].style as the final parameter. Once the for loop finishes we'll call the return function without a parameter to exit the function. We exit the function because there's no need to continue since we've already inserted the object. This concludes the code for duplicating objects.

To duplicate regular arrays in the map such as entities, items, and sounds we simply create a new for loop, which loops from 0 to the value maximum array value (e.g., map->header.max_items if we are duplicating items). Within the for loop we'll create an if statement to check if the global select_rgb values are equal to the array select_rgb values. In the event the values are equal, then we've got a selection match and we can duplicate the appropriate item. Inside the if statement we'll call the appropriate map Insert method and supply all the parameters required to ensure there are no overloaded values being set. In the case of duplicating items we would call the map->InsertItem method and specify map->item[i].xyz[0] for the first parameter (X coordinate), map->item[i].xyz[1] for the second parameter (Y coordinate), map->item[i].xyz[2] (Z coordinate), map->item[i].type (type of item), map->item[i].respawn_wait as the fifth parameter, and finally map->item[i].respawn_time. Once we've filled in the insertion method we simply call the function return without a parameter to exit the function. This process works for all the arrays in the MAP class with the exception of the object and light arrays because they have arrays within themselves.

Light array duplication works in the same manner as for the other types of arrays in that we make a loop to go through all the indexes of the array, checking for a matched select_rgb array value. Like the insert object code, we'll declare a new variable called last_entry, which is of the long data type. The variable will have a default value set to the value of map->header.max_lights to allow us easy access to the newest entry in the light list. After inserting the new light, we'll check the value of map->light[i].max_inclusions to see if it's greater than 0. If the value is greater than 0, we'll set the newly created light's max_inclusions variable equal to the value of map->light[i].max_inclusions. After setting the variable we'll allocate memory for the map->light[last_entry].inclusions variable to hold the entire list for map->light[last_entry].max_inclusions. And as if you couldn't guess, we'll create another for loop to copy the data from the map->light[i].inclusions array to the newly created and allocated variable map->light[last_entry].inclusions array. Once the data is copied we simply call the return function without any parameters to exit from the Duplicate function.

At the bottom of the Duplicate function we'll display a message box that states that the object could not be duplicated. If any selected object was found and duplicated, it should have exited from the function already. If the message is displayed on the screen, then the object wasn't found in any of the select_rgb arrays. This completes the code necessary to duplicate objects and other items in our map and we can display the source code below.

```
void Duplicate(HWND hWnd)
{

   for (long i = 0; i < map->header.max_objects; i++)
   {
      if (map->object[i].select_rgb[0] == select_rgb[0] &&
         map->object[i].select_rgb[1] == select_rgb[1] &&
         map->object[i].select_rgb[2] == select_rgb[2])
      {
         long last_entry = map->header.max_objects;

         map->InsertObject (map->object[i].name, map->object[i].type, map->
              object[i].special, map->object[i].is_collidable,
              map->object[i].is_visible);
         for (long i2 = 0; i2 < map->object[i].max_vertices; i2++) map->
             InsertVertex (last_entry, map->object[i].vertex[i2].xyz[0],
             map->object[i].vertex[i2].xyz[1], map->object[i].vertex[i2].xyz[2],
             map->object[i].vertex[i2].rgba[0], map->object[i].vertex[i2].rgba[1],
             map->object[i].vertex[i2].rgba[2], map->object[i].vertex[i2].rgba[3],
              map->object[i].vertex[i2].normal[0], map->object[i].vertex
              [i2].normal[1], map->object[i].vertex[i2].normal[2], map->
              object[i].vertex[i2].fog_depth);
         for (i2 = 0; i2 < map->object[i].max_triangles; i2++) map->
              InsertTriangle (last_entry, map->object[i].triangle[i2].point[0],
              map->object[i].triangle[i2].point[1], map->object[i].triangle
              [i2].point[2], map->object[i].triangle[i2].uv[0].uv1[0],map->
              object[i].triangle[i2].uv[0].uv1[1], map->object[i].triangle
              [i2].uv[0].uv2[0],map->object[i].triangle[i2].uv[1].uv2[1],
              map->object[i].triangle[i2].uv[0].uv3[0],map->object[i].triangle
              [i2].uv[0].uv3[1]);

         for (i2 = 0; i2 < map->object[i].max_textures; i2++) map->
              InsertTexture (last_entry, map->object[i].texture[i2].filename,
              map->object[i].texture[i2].style);
         return;
      }
   }

   for (i = 0; i < map->header.max_entities; i++)
   {
      if (map->entity[i].select_rgb[0] == select_rgb[0] &&
         map->entity[i].select_rgb[1] == select_rgb[1] &&
         map->entity[i].select_rgb[2] == select_rgb[2])
      {
         map->InsertEntity (map->entity[i].type, map->entity[i].xyz[0], map->
              entity[i].xyz[1], map->entity[i].xyz[2], map->entity[i].angle[0],
              map->entity[i].angle[1], map->entity[i].angle[2], map->
              entity[i].health, map->entity[i].strength, map->entity[i].armour);
         return;
      }
   }

   for (i = 0; i < map->header.max_items; i++)
   {
      if (map->item[i].select_rgb[0] == select_rgb[0] &&
         map->item[i].select_rgb[1] == select_rgb[1] &&
         map->item[i].select_rgb[2] == select_rgb[2])
      {
         map->InsertItem (map->item[i].xyz[0], map->item[i].xyz[1], map->
              item[i].xyz[2], map->item[i].type, map->item[i].respawn_wait,
              map->item[i].respawn_time);
         return;
      }
   }

   for (i = 0; i < map->header.max_lights; i++)
   {
      if (map->light[i].select_rgb[0] == select_rgb[0] &&
         map->light[i].select_rgb[1] == select_rgb[1] &&
         map->light[i].select_rgb[2] == select_rgb[2])
      {
         long last_entry = map->header.max_lights;

         map->InsertLight (map->light[i].name, map->light[i].texture_filename,
              map->light[i].xyz[0], map->light[i].xyz[1], map->light[i].xyz[2],
              map->light[i].angle[0], map->light[i].angle[1], map->
              light[i].angle[2], map->light[i].rgba[0], map->light[i].rgba[1],
              map->light[i].rgba[2]);
         if (map->light[i].max_inclusions > 0)
         {
            map->light[last_entry].max_inclusions = map->light[i].max_inclusions;
            map->light[last_entry].inclusions     = new int[map->
                 light[i].max_inclusions+1];
            for (long i2 = 0; i2 < map->light[last_entry].max_inclusions; i2++)
                 map->light[last_entry].inclusions[i2] = map->light
                 [i].inclusions[i2];
         }
         return;
      }
   }

   for (i = 0; i < map->header.max_sounds; i++)
   {
      if (map->sound[i].select_rgb[0] == select_rgb[0] &&
         map->sound[i].select_rgb[1] == select_rgb[1] &&
         map->sound[i].select_rgb[2] == select_rgb[2])
      {
         map->InsertSound (map->sound[i].xyz[0], map->sound[i].xyz[1], map->
              sound[i].xyz[2], map->sound[i].filename, map->sound[i].angle[0],
              map->sound[i].angle[1], map->sound[i].angle[2]);
         return;
      }

   }

   MessageBox (hWnd, "Unable to Duplicate", "Error", MB_OK);
}
```

To finish the Duplicate function we'll update WMCommand and change the already handled ID_POPUP_DUPLICATE button click. We'll take out the call to MessageBox and replace it with the function Duplicate, passing the variable hWnd as the first parameter. Now when the user clicks the Duplicate button, it will duplicate the selected object and keep it in its current location!

Adding Skyboxes

A *skybox* is a set of polygons that appears on the farthest coordinates of the world. Normally a skybox is represented as a cube, with each side having a texture assigned to it and displaying 90 degrees of sky. When each side has a texture assigned to it, the world should nicely connect and give the illusion that the user is outside, or at the very least provide the user with the means to look in any direction upward and see the sky. Many games use a skybox as the background because it's a fast and simple method of filling the backgrounds of a level without having to create tons of background models. A simple skybox can change a single-color blue sky to a mountain sky or change a single-color evening night sky to a star-filled cityscape. Figure 8.2 displays a basic skybox in relation to a map.

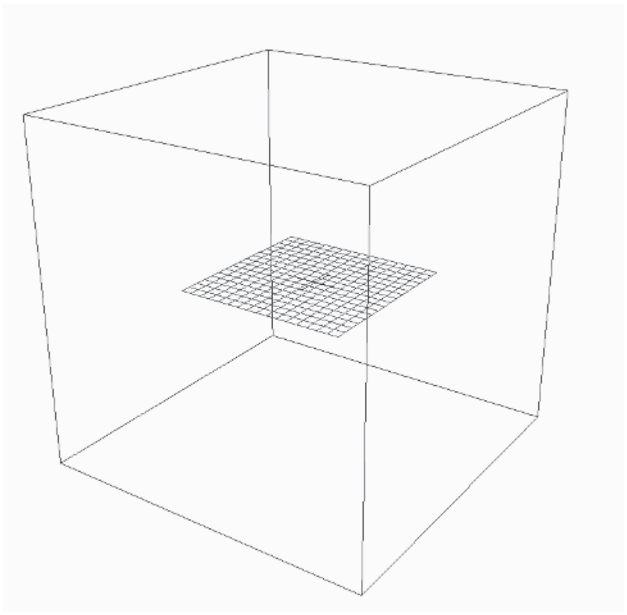


Figure 8.2: Skybox wireframe

In this section we'll add a dialog box to the map editor that will input the six bitmaps needed to make the skybox work in the game engine. Where do the skybox bitmaps come from, you may be wondering. There are several software packages that can create nice-looking skies. The first package is called Bryce, which is now a Corel product. This package makes very nice 3D terrains with great antialiasing; however, the price of the product is slightly higher than what the beginning game programmer/designer can afford. Another available software package is called Terragen, which is written by a small company called Planetside Software. I hate to be biased, but I prefer the interface and power of Terragen over other software packages on the market for making skyboxes. The company even has a shareware version that you can download from their web site to try out. I recommend you download the software package and read some tutorials about making skyboxes. In no time you'll be creating some awesome backgrounds!

As mentioned before, we'll be creating a new dialog box to handle the skybox filenames. The resource ID of the dialog box will be called IDD_SKYBOX, and will have a caption of "SkyBox Properties." As with every dialog box we create, I've checked the Center option to have the dialog box centered on the screen. After exiting the dialog properties we'll insert six edit controls into our newly created dialog box. These edit controls will store the filenames of the six directions (front, back, left, right, top, and bottom) we need bitmaps for. The resource names for the bitmaps are IDC_SKYBOX_FRONT, IDC_SKYBOX_BACK, IDC_SKYBOX_LEFT, IDC_SKYBOX_RIGHT, IDC_SKYBOX_TOP, and finally IDC_SKYBOX_BOTTOM.

We'll also add one check box to the dialog with the resource ID of IDC_SKYBOX_USE. This dialog box will be used to control whether or not the skybox is used in the game. Although it's not important, I've stacked the edit controls in a vertical column and left enough room to add six static controls for naming each GUI control. This of course won't affect the skybox data interaction at all; however, it's always nice to add that finishing touch! Figure 8.3 shows my skybox dialog.

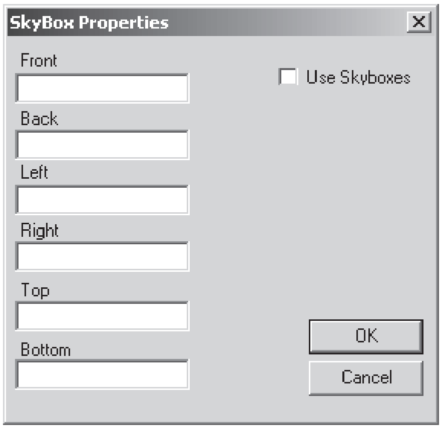


Figure 8.3: SkyBox Properties dialog

With the SkyBox Properties dialog design complete, we can begin the coding process. As with every dialog box, we'll create a dialog procedure to handle all messages sent to the dialog box. Our skybox dialog procedure will be called SkyBoxDlgProc and will be placed directly above the WMCommand function. The first Windows message we'll handle will be WM_INITDIALOG. Inside the WM_INITDIALOG message we'll create an if statement to see if the value of map->header.use_skybox is true or not. If the value is true, we'll set each of the edit controls in the dialog to the appropriate map->skybox filename using the SetDlgItemText function. In the case of the front side, we'd call SetDlgItemText, specifying IDC_SKYBOX_FRONT as the second parameter and map->skybox.front.filename as the NULL-terminated string for the filename. Each view has its own variable within the map->skybox structure.

After setting each of the edit controls to the appropriate filename we must turn on the checkmark in our IDC_SKYBOX_USE check box, signifying that we're using skyboxes in our level. To turn the checkmark on, we simply call the function SendDlgItemMessage and specify the resource ID IDC_SKYBOX_USE to send the message to, BM_SETCHECK as the message, and finally BST_CHECKED to tell the checkmark which state it should be in. This is a very handy message because it allows us to add some real functionality into our map editor without having to dig really deep into Win32 code. With the Use check box checked, we've finished the code necessary in the WM_INITDIALOG message and can move to the WM_COMMAND message to finish the dialog box functionality.

In the WM_COMMAND message we'll check to see if the value of wParam is equal to IDCANCEL. If the values are equal, we'll call the EndDialog function to exit the dialog since the user clicked Cancel. If the value of wParam is equal to IDOK, the user clicked the OK button and we must get the filenames from the controls. In the case of the front view, we would call GetDlgItemText, specifying IDC_SKYBOX_FRONT as the resource ID and map->skybox.front.filename for the string to place the data into. After grabbing the data we'll send the BM_GETCHECK message to our check box (resource ID IDC_SKYBOX_USE).

To send the message we'll once again use SendDlgItemMessage and specify the IDC_SKYBOX_USE resource ID as the second parameter and the BM_GETCHECK message as the third parameter. The returned value from the function will be used in an if statement and compared against the value BST_CHECKED. If the returned value from SendDlgItemMessage is equal to BST_CHECKED, the user checked the check box and we should set the variable map->header.use_skybox to true; otherwise, we'll set the value to false. After setting the map->header.use_skybox variable, we'll simply call the EndDialog macro to exit from the dialog box. The source code for the SkyBoxDlgProc is provided below.

```
LRESULT CALLBACK SkyBoxDlgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         if (map->header.use_skybox)
         {
            SetDlgItemText (hWnd, IDC_SKYBOX_FRONT, map->skybox.front.filename);
            SetDlgItemText (hWnd, IDC_SKYBOX_BACK, map->skybox.back.filename);
            SetDlgItemText (hWnd, IDC_SKYBOX_LEFT, map->skybox.left.filename);
            SetDlgItemText (hWnd, IDC_SKYBOX_RIGHT, map->skybox.right.filename);
            SetDlgItemText (hWnd, IDC_SKYBOX_TOP, map->skybox.top.filename);
            SetDlgItemText (hWnd, IDC_SKYBOX_BOTTOM, map->skybox.bottom.filename);

            SendDlgItemMessage (hWnd, IDC_SKYBOX_USE, BM_SETCHECK,
                     BST_CHECKED, 0);
         }
      } break;

      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            GetDlgItemText (hWnd, IDC_SKYBOX_FRONT, map->skybox.front.filename,
                     500);
            GetDlgItemText (hWnd, IDC_SKYBOX_BACK, map->skybox.back.filename,
                     500);
            GetDlgItemText (hWnd, IDC_SKYBOX_LEFT, map->skybox.left.filename,
                     500);
            GetDlgItemText (hWnd, IDC_SKYBOX_RIGHT, map->skybox.right.filename,
                     500);
            GetDlgItemText (hWnd, IDC_SKYBOX_TOP, map->skybox.top.filename, 500);
            GetDlgItemText (hWnd, IDC_SKYBOX_BOTTOM, map->skybox.bottom.filename,
                     500);

            if (SendDlgItemMessage (hWnd, IDC_SKYBOX_USE, BM_GETCHECK, 0, 0) ==
                     BST_CHECKED) map->header.use_skybox = true;
            else map->header.use_skybox = false;

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

To get the dialog box working we'll need to open the resource editor and add another menu item to the IDR_MENU menu. Under the Map pop-up menu we'll create a new menu item called "Skybox" with a resource ID of ID_MAP_SKYBOX. Moving back to the main source file, we'll add a new else-if clause to the WMCommand function, checking to see if the value of wParam is equal to ID_MAP_SKYBOX. If the values are equal, we'll call the DialogBox macro, specifying IDD_SKYBOX inside the MAKEINT-RESOURCE macro as the second parameter and the SkyBoxDlgProc dialog procedure as the fourth parameter. This concludes the code necessary to add skyboxes into our maps, which will come in handy later on.

Adding Fog

When using fog, every object in the scene is affected by its configuration. This means that if we specify red fog, all objects at one point or another will have red fog on them. Because of this simple configuration, we only need a basic configuration dialog box to set up the fog. To begin the fog development, we'll create a dialog box with a resource ID of IDD_FOG. Inside the box we'll add a check box control (resource ID IDC_FOG_USE) with a label called "Use Fog" to toggle fog on or off. We'll also add one combo box (IDC_FOG_MODE) to select the fog mode, which determines the falloff points and how it draws. Finally we'll add seven edit controls to specify the density of the fog (IDC_FOG_DENSITY), the starting position of the fog (IDC_FOG_START), the ending position of the fog (IDC_FOG_END), and of course the RGBA values to color the fog (IDC_FOG_R, IDC_FOG_G, IDC_FOG_B, and IDC_FOG_A). If you decide to get really creative you can add some static controls, as I've done, to easily distinguish between the different options. Remember to turn off the sorting operation for the combo box so it doesn't sort incoming data items.

With the Fog dialog box designed, we'll write the code for the dialog procedure (FogDlgProc) to handle its messages. Like any dialog procedure we've created in the map editor, we're only interested in the WM_INITDIALOG and WM_COMMAND messages. When the dialog box first appears (WM_INITDIALOG message), we're going to set the Use Fog (IDC_FOG_USE) check box based on the value of map->header.use_fog. Then we'll reset the combo box and insert the three default fog modes (Linear, Exp, and Exp2) to allow the user to customize the type of fog. Once the items have been inserted we'll set the current mode selection based on the value of map->fog.mode (GL_EXP = Selection 1, GL_EXP2 = Selection 2, otherwise use Selection 1). The final step is to fill in the edit controls with the content provided in the fog header variables. The proper values for each edit control will be explained in detail in Chapter 14, "Lights and Fog."

The code for the WM_COMMAND message works in a similar manner to the code in the WM_INITDIALOG message. First we set the fog value based on the checked/unchecked value returned from the Use Fog check box (IDC_FOG_USE). Next we set the OpenGL fog mode based on the current mode selection. Finally, we grab the text from each edit control, convert the values to floats, and store them in the appropriate fog structure variable. Before we can use this newly created function, we must update our menu resource with a new item in the Map menu called Fog (resource ID ID_MAP_FOG). In the WMCommand function, we'll run the Fog dialog box if the Fog menu item is pressed. The source code for the FogDlgProc function is provided here.

```
LRESULT CALLBACK FogDlgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
         {
            char temp[500];

            if (map->header.use_fog) SendDlgItemMessage (hWnd, IDC_FOG_USE,
                     BM_SETCHECK, BST_CHECKED, 0);
            else SendDlgItemMessage (hWnd, IDC_FOG_USE, BM_SETCHECK,
                      BST_UNCHECKED, 0);

            SendDlgItemMessage (hWnd, IDC_FOG_MODE, CB_RESETCONTENT, 0, 0);
            SendDlgItemMessage (hWnd, IDC_FOG_MODE, CB_ADDSTRING, 0,
                      (LPARAM)"Linear");
            SendDlgItemMessage (hWnd, IDC_FOG_MODE, CB_ADDSTRING, 0,
                      (LPARAM)"Exp");
            SendDlgItemMessage (hWnd, IDC_FOG_MODE, CB_ADDSTRING, 0,
                      (LPARAM)"Exp2");

            switch (map->fog.mode)
            {
               case GL_EXP: SendDlgItemMessage (hWnd, IDC_FOG_MODE, CB_SETCURSEL,
                         1, 0); break;
               case GL_EXP2: SendDlgItemMessage (hWnd, IDC_FOG_MODE, CB_SETCURSEL,
                         2, 0); break;
               default: SendDlgItemMessage (hWnd, IDC_FOG_MODE, CB_SETCURSEL,
                         0, 0);
            }

            sprintf (temp, "%1.2f", map->fog.density);
            SetDlgItemText (hWnd, IDC_FOG_DENSITY, temp);

            sprintf (temp, "%1.2f", map->fog.start);
            SetDlgItemText (hWnd, IDC_FOG_START, temp);

            sprintf (temp, "%1.2f", map->fog.end);
            SetDlgItemText (hWnd, IDC_FOG_END, temp);

            sprintf (temp, "%3.0f", (map->fog.rgba[0] * 255.0f));
            SetDlgItemText (hWnd, IDC_FOG_R, temp);

            sprintf (temp, "%3.0f", (map->fog.rgba[1] * 255.0f));
            SetDlgItemText (hWnd, IDC_FOG_G, temp);

            sprintf (temp, "%3.0f", (map->fog.rgba[2] * 255.0f));
            SetDlgItemText (hWnd, IDC_FOG_B, temp);

            sprintf (temp, "%3.0f", (map->fog.rgba[3] * 255.0f));
            SetDlgItemText (hWnd, IDC_FOG_A, temp);
         } break;
      case WM_COMMAND:
         {
            if (wParam == IDCANCEL) EndDialog (hWnd, 0);
            else if (wParam == IDOK)
            {
               char temp[500];

               if (SendDlgItemMessage (hWnd, IDC_FOG_USE, BM_GETCHECK, 0, 0) ==
                   BST_CHECKED) map->header.use_fog = true;
               else map->header.use_fog = false;

               switch (SendDlgItemMessage(hWnd, IDC_FOG_MODE, CB_GETCURSEL, 0, 0))
               {
                  case 0: map->fog.mode = GL_LINEAR; break;
                  case 1: map->fog.mode = GL_EXP; break;
                  case 2: map->fog.mode = GL_EXP2; break;
               }

               GetDlgItemText (hWnd, IDC_FOG_DENSITY, temp, 500);
               sscanf (temp, "%f", &map->fog.density);

               GetDlgItemText (hWnd, IDC_FOG_START, temp, 500);
               sscanf (temp, "%f", &map->fog.start);

               GetDlgItemText (hWnd, IDC_FOG_END, temp, 500);
               sscanf (temp, "%f", &map->fog.end);

               GetDlgItemText (hWnd, IDC_FOG_R, temp, 500);
               sscanf (temp, "%f", &map->fog.rgba[0]);
               map->fog.rgba[0] /= 255.0f;

               GetDlgItemText (hWnd, IDC_FOG_G, temp, 500);
               sscanf (temp, "%f", &map->fog.rgba[1]);
               map->fog.rgba[1] /= 255.0f;

               GetDlgItemText (hWnd, IDC_FOG_B, temp, 500);
               sscanf (temp, "%f", &map->fog.rgba[2]);
               map->fog.rgba[2] /= 255.0f;

               GetDlgItemText (hWnd, IDC_FOG_A, temp, 500);
               sscanf (temp, "%f", &map->fog.rgba[3]);
               map->fog.rgba[3] /= 255.0f;

               EndDialog (hWnd, 1);
            }
         } break;
   }
   return (0);
}
```

Editing Map Details

With most of the features finished in the map editor, we're going to get nostalgic and update the first dialog box we created in the map editor, the Map Details dialog box. We want to remove the code we wrote in the beginning chapters and replace it with usable features. Since we've already written the MapDetailsDlgProc function and have the menu item code finished, we simply need to modify our existing code, which makes life easy! The first piece of code we'll update is the WM_INITDIALOG message. Within the WM_INITDIALOG message we'll update the function call to SetDlgItem-Text, replacing the third parameter with the variable map->details.map_name. Obviously we don't want to have a hardcoded map name, especially something as dull as "Map Name"!

After updating the map name we'll delete the first string being inserted into the IDC_MAP_DETAILS_LEVEL_RULES list box control. There's obviously no point in having a message that inserts itself only to be reset in the next line. This code was great when we were discussing the basics of dialog box programming; however, that time has come and gone. Next we'll change the LB_SETCURSEL message so the fourth parameter (the selection) is the value of map->details.map_exit_rules, which will select the appropriate level rule in the list box. The combo box updates follow the same procedure as those for the list box. We'll delete the code to insert the "Erase Me" string, since it's redundant. The CB_SETCURSEL message will be changed so the fourth parameter is now the value of map->details.map_type, which will automatically select the proper map when the map loads. This completes the changes for the WM_INITDIALOG message, and we must now update our WM_COMMAND message to retrieve the information from the dialog controls.

There is slightly more work involved in updating WM_COMMAND. In the IDOK button click we'll add a new line to get the text from the resource ID IDC_MAP_DETAILS_NAME and store the string in the variable map->details.map_name. After retrieving the map name we'll replace the level_rule variable declaration with the variable map->details.map_type, and we'll also replace the level_type variable declaration with the variable map->details.map_exit_rules. The temp array declaration can be deleted and we can continue with the EndDialog macro. The IDCANCEL button click should have the MessageBox line removed, for obvious reasons, and only have the call to the EndDialog macro. This completes the update to the MapDetails dialog procedure. Now the dialog controls will be set to the internal map values when the dialog starts, and the internal map variables will retrieve the input values when the user presses the OK button. Simple enough! Since we've finished with the update we can display the source code here.

```
LRESULT CALLBACK MapDetailsDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, map->details.map_name);

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_RESETCONTENT,
                             0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                             0, (LPARAM)"Exit");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_ADDSTRING,
                             0, (LPARAM)"Get Fragged");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_RULES, LB_SETCURSEL,
                             map->details.map_exit_rules, 0);

         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_RESETCONTENT,
                             0, 0);
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                             0, (LPARAM)"Single Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_ADDSTRING,
                             0, (LPARAM)"Multi Player");
         SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_LEVEL_TYPE, CB_SETCURSEL,
                             map->details.map_type, 0);
      } break;

      case WM_COMMAND:
      {
         if (wParam == IDOK)
         {
            GetDlgItemText (hWnd, IDC_MAP_DETAILS_NAME, map->details.map_name,
                            500);
            map->details.map_type = SendDlgItemMessage (hWnd, IDC_MAP_DETAILS_
                            LEVEL_TYPE, CB_GETCURSEL, 0, 0);
            map->details.map_exit_rules = SendDlgItemMessage (hWnd, IDC_MAP_
                            DETAILS_LEVEL_RULES, LB_GETCURSEL, 0, 0);

            EndDialog (hWnd, 0);
         }
         else if (wParam == IDCANCEL) EndDialog (hWnd, 0);
      } break;
   }
   return (0);
}
```

Selecting and Assigning Objects to Lights

In Chapter 7 we wrote the code to insert lights into our map. Unfortunately we didn't write the code necessary to insert objects into the light inclusion list, thereby allowing the object to be lit. This section will discuss how we'll select which light we want to insert the object into and how to actually insert the light into the inclusion list. To begin the coding process we'll open the resource editor and create a new dialog box with the resource ID of IDD_SELECTLIGHT. The text caption of the dialog box will be "Select Light." Inside the dialog box we'll insert one list box control with the resource ID of IDC_SELECTLIGHT_LIGHTNAME. It is a good idea to change the properties of the light box so the Sort option is unchecked. With this option checked, the returned selected item order could vary quite differently from the insertion order, which is something we want to avoid at all costs!

The dialog box has the sole purpose of displaying the names of all the lights and allowing the user to choose one and return the selected value. Both the assign and remove light functions will use this function to get the selected light, then perform the appropriate algorithm to either insert into or delete from the inclusion list. This concludes the additions to the resource editor, so we should move to the main source file and begin writing the dialog procedure SelectLightDlgProc, which will handle the messages for the function. As with every dialog procedure, we can copy and paste the function definition from any other dialog box we've already created and simply rename the function itself. The SelectLightDlgProc function should be placed above WMCommand so we can easily access it later in the software.

As with the rest of our dialog boxes, we'll use WM_INITDIALOG to initialize the controls within the dialog. Thankfully there is only one dialog box control to worry about, which makes life much easier. Since our single control is a list box, we'll send the message LB_RESETCONTENT to erase the existing data in the control. (This is a simple reminder to always erase the existing content before using.) After erasing the data we simply create a for loop, looping from 0 to the value of map->header.max_lights. With each iteration of the loop we'll call the SendDlgItemMessage function, specifying LB_ADDSTRING as the message and map->light[i].name as the string to insert in the fifth parameter. This will fill the list box with every light in our map. Once it's finished we send one more message to set a default selection by sending the LB_SETCURSEL message to the list box control and specifying 0 for the fourth parameter. This completes the WM_INITDIALOG message handling code, so we can begin discussing the code necessary to handle the WM_COMMAND message.

Compared to other WM_COMMAND messages we've handled in the past, this one is really simple. If the user clicks the OK button (wParam equals IDOK), we'll declare a new variable of the long data type. The variable, cursel, will have a default value set by calling the function SendDlgItemMessage and specifying the resource ID of IDC_SELECT-LIGHT_LIGHTNAME and the LB_GETCURSEL message. This will return the currently selected light name, which will be stored in the cursel variable. Once the value is returned we simply call the EndDialog macro to exit from the dialog, supplying the cursel variable as the second parameter of the macro, which happens to be the return value for the dialog box.

If the user clicks the Cancel button (wParam equals IDCANCEL), we'll simply call the EndDialog macro and use a hardcoded value of –1 for the second parameter, indicating that no parameter was chosen. We cannot use the value of 0 because there technically can be a light 0. If the returned value is –1, then the user cancelled the selection process and we should exit from whatever function we're doing. If the returned dialog value is greater than or equal to 0, we should continue on our merry way. With the SelectLightDlg-Proc function finished, we can display the source code below.

```
LRESULT CALLBACK SelectLightDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         SendDlgItemMessage (hWnd, IDC_SELECTLIGHT_LIGHTNAME, LB_RESETCONTENT,
                   0, 0);
         for (long i = 0; i < map->header.max_lights; i++) SendDlgItemMessage
                   (hWnd, IDC_SELECTLIGHT_LIGHTNAME, LB_ADDSTRING, 0,
                   (LPARAM)map->light[i].name);
         SendDlgItemMessage (hWnd, IDC_SELECTLIGHT_LIGHTNAME, LB_SETCURSEL, 0, 0);
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDOK)
         {
            long cursel = SendDlgItemMessage(hWnd, IDC_SELECTLIGHT_LIGHTNAME,
                      LB_GETCURSEL, 0, 0);
            EndDialog (hWnd, cursel);
         }
         else if (wParam == IDCANCEL) EndDialog (hWnd, -1);
      } break;
   }
   return (0);
}
```

With the base code out of the way for selecting the light, we can begin writing the code to assign the object to a specific light. To start we'll create a new function called AssignToLight, which will be located between the WMCommand function and the SelectLightDlgProc dialog procedure. The new function will have one parameter, called hWnd, which is a handle to a window. Inside the function we'll declare two new variables, both of the long data type. The first variable is called light and will store the returned value from the call to the Select Light dialog box. The other variable is called obj and has a default value of –1. The obj variable will contain the object number for the currently selected object. This information is very useful because it allows us to easily set variables to the selected item without having to check through the loop of select_rgb values every time.

After declaring the new variables we'll create a for loop from 0 to the value map->header.max_objects. Within the loop we'll check to see if the values in the map->object[i].select_rgb array are equal to the globally declared select_rgb. In the event the values are equal, we'll set the obj variable to the value of the i variable, indicating that the currently selected object is i. At the end of the loop we'll check whether the value of obj is less than 0. If the value is indeed less than 0, then no object was found and we should display an error message and exit from the function using the return function without any parameters. If the obj variable is greater than or equal to 0, we'll continue through the function.

With the obj variable set to the currently selected object, we should prompt the user with a listing of all the lights to allow the selection of one to add the object to. This can be accomplished by simply calling the DialogBox macro and specifying the IDD_SELECTLIGHT dialog as the second parameter inside the MAKEINTRESOURCE macro and specifying the SelectLightDlgProc function as the fourth parameter. Keep in mind that the return value from the DialogBox macro should be stored in the light variable so we know which option the user chose. Once the dialog box closes we'll check the value of the light variable to see if it's less than 0. If light is less than 0, no light was selected and we should once again display an error message and exit from the function.

Before we do the final insertion code we'll perform another for loop from 0 to the value of map->light[light].max_inclusions. With each iteration of the loop we'll check to see if our selected object is in the map->light[light].inclusions array. In the event the values are equal we'll once again display an error message and exit from the loop. By now you're probably getting pretty bored of this phrase. I am happy to say that this will be the last time it's in this section!

The insertion code itself is no different from any other dynamically allocated array in our map editor code. We first check the value of map->light[light].max_inclusions for 0. If the value is 0, we allocate memory for one index. If the value of map->light[light].max_inclusions is greater than 0, we'll declare a pointer of an integer called temp. We allocate memory for the temp variable and copy the contents of the map->light[light].inclusions array into it. Then we'll delete the inclusions array and reallocate memory for it but with an extra record. After allocating memory, we'll simply copy the data back from the temp array and delete the temp array to finish the else-if clause.

With the memory allocation complete we'll simply add the new obj variable to the last record in the array, which happens to be stored in map->light[light].max_inclusions, and increment the value of map->light[light].max_inclusions so we include the new record for which we just allocated memory. With the max_inclusions variable incremented, we've completed the function and can display the source code below.

```
void AssignToLight(HWND hWnd)
{
   long light;
   long obj = -1;


   for (long i = 0; i < map->header.max_objects; i++)
   {
      if (map->object[i].select_rgb[0] == select_rgb[0] &&
         map->object[i].select_rgb[1] == select_rgb[1] &&
         map->object[i].select_rgb[2] == select_rgb[2]) obj = i;
   }
   if (obj < 0)
   {
      MessageBox (hWnd, "Could not find selection", "Oops!", MB_OK);
      return;
   }

   light = DialogBox (GlobalInstance, MAKEINTRESOURCE(IDD_SELECTLIGHT), NULL,
           (DLGPROC)SelectLightDlgProc);
   if (light < 0)
   {
      MessageBox (hWnd, "No light selected", "Error", MB_OK);
      return;
   }

   for (i = 0; i < map->light[light].max_inclusions; i++)
   {
      if (map->light[light].inclusions[i] == obj)
      {
         MessageBox (hWnd, "The object is already assigned to this light!",
                     "Oops!", MB_OK);
         return;
      }
   }

   if (map->light[light].max_inclusions == 0) map->light[light].inclusions
            = new int[1];
   else
   {
      int *temp;
```

```
        temp = new int[map->light[light].max_inclusions+1];
        for (i = 0; i < map->light[light].max_inclusions; i++) temp[i] = map->
             light[light].inclusions[i];

        delete [] map->light[light].inclusions;
        map->light[light].inclusions = new int[map->light[light].max_inclusions+2];

        for (i = 0; i < map->light[light].max_inclusions; i++) map->light
             [light].inclusions[i] = temp[i];
        delete temp;
    }

    map->light[light].inclusions[map->light[light].max_inclusions] = obj;
    map->light[light].max_inclusions++;
}
```

One last detail that we must concern ourselves with is the pop-up menu. Because this function will be accessed through the pop-up menu, we'll open the resource editor and add a new menu item called Assign to Light with a resource ID of ID_POPUP_ASSIGNTOLIGHT. This will be the method we'll use to assign the object to a light, once we update the WMCommand function. To update the WMCommand function, we'll add a new else-if clause, which will check the value of wParam against the value ID_POPUP_ ASSIGNTOLIGHT. If the values are equal, we'll call the AssignToLight function and specify the hWnd variable as the single parameter. This concludes the code necessary to add objects to the inclusion lists, allowing them to be lit.

Removing Objects from Lights

Removing objects from lights works along the same principles as assigning them; however, we must make several modifications to the code. With this in mind we'll simply copy and paste the code into a new function called RemoveFromLight. The code for the function remains the same until we hit the dynamic memory allocation at the bottom of the function. Instead of modifying this code we'll simply cut this section out of the function and write brand-new code to delete the object from the inclusion list. To begin, we must ensure the value of map->light[light].max_inclusions is greater than 0. If the value is 0, we'll crash the system by attempting to delete memory, which is not a good thing.

We'll declare another variable pointer called temp, which is of the int type. After the declaration we'll allocate memory to store the entire inclusion list and copy the contents of the inclusion list to the temp array. Then we'll delete the inclusion list memory and allocate memory for one less index. After allocating the memory we'll create a for loop to copy the data back to the inclusion list, with the exception that we'll check the value of i (our looping variable) with the obj variable. If the value of i is lower than the selected object (obj variable), we'll copy the data from one index to the other without any changes. If the value of the i variable is greater than the obj variable, then we'll copy the data into the inclusion array but subtract 1 from the i index variable for the inclusion list. This allows the deleted obj variable to be skipped without any harm to the rest of the data.

Once the loop is finished we simply delete the temp array and decrease the value of map->light[light].max_inclusions by one. The source code for the newly created function is shown here.

```
void RemoveFromLight(HWND hWnd)
{
   long light;
   long obj = -1;


   for (long i = 0; i < map->header.max_objects; i++)
   {
      if (map->object[i].select_rgb[0] == select_rgb[0] &&
         map->object[i].select_rgb[1] == select_rgb[1] &&
         map->object[i].select_rgb[2] == select_rgb[2]) obj = i;
   }
   if (obj < 0)
   {
      MessageBox (hWnd, "Could not find selection", "Oops!", MB_OK);
      return;
   }

   light = DialogBox (GlobalInstance, MAKEINTRESOURCE(IDD_SELECTLIGHT), NULL,
        (DLGPROC)SelectLightDlgProc);
   if (light < 0)
   {
      MessageBox (hWnd, "No light selected", "Error", MB_OK);
      return;
   }

   if (map->light[light].max_inclusions > 0)
   {
      int *temp;

      temp = new int[map->light[light].max_inclusions+1];
      for (i = 0; i < map->light[light].max_inclusions; i++) temp[i] = map->
           light[light].inclusions[i];

      delete [] map->light[light].inclusions;
      map->light[light].inclusions = new int[map->light[light].max_
           inclusions];

      for (i = 0; i < map->light[light].max_inclusions; i++)
      {
         if (i < obj) map->light[light].inclusions[i] = temp[i];
         else if (i > obj) map->light[light].inclusions[i-1] = temp[i];
      }
      delete temp;
      map->light[light].max_inclusions--;
   }
```

This doesn't mean we're finished with the section. We must update our pop-up menu to add a new menu item called Remove from Light with a resource ID of ID_POPUP_REMOVEFROMLIGHT. After creating the new menu item we'll move back to the main source file and update the WMCommand function with a new else-if clause to checking the value of wParam against ID_POPUP_REMOVEFROMLIGHT. If the values are equal, we'll call the RemoveFromLight function and supply the hWnd variable as the single parameter.

Editing Object Values

One of the final topics we'll discuss in this chapter is editing the values of objects. We'll focus specifically on editing the values of objects because they are the most complex data structure in our map format and probably the most important. We'll allow the user to choose whether the object is visible or collidable and to edit its name and the special value for the object. The special value is used to give the object special functionality such as losing 10% health. Whenever you design a new map format it's a good idea to add something similar to a special value for each object. This provides you with a simple method of adding functionality to a specific object without having to code all sorts of extra data.

To begin the coding process we'll create a new enumeration in the map.h file that will handle three different special values: SPECIAL_NONE, which has a default value of 0, SPECIAL_END_LEVEL, and SPECIAL_MINUS_10_HEALTH. If the object has no special value assigned, then the object special value will be equal to SPECIAL_NONE. If the user touches an object with a special value of SPECIAL_END_LEVEL, the level should end. If the special value is equal to SPECIAL_MINUS_10_HEALTH, we'll reduce the player's health by 10%. These three values are only a small sample of what's possible when you add special values to an object. This feature is more of a design objective when you're planning your games. Since we're learning the basics we won't bother ourselves with adding more values.

Next we'll create a new dialog box called Edit Objects with a resource ID of IDD_EDIT_OBJECT, which will handle the editing of the object. Inside the dialog box we'll insert six edit controls with resource IDs of IDC_EDITOBJECT_NUMBER (this control should be read only), IDC_EDITOBJECT_NAME, IDC_EDITOBJECT_TYPE (this control should be read only), IDC_EDITOBJECT_STARTY, IDC_EDITOBJECT_HEIGHT, and IDC_EDITOBJECT_FOG. These edit controls will contain some of the data we'll allow the user to edit. Besides the edit controls we'll add two check boxes called IDC_EDITOBJECT_VISIBLE and IDC_EDITOBJECT_COLLIDABLE, which will control the visibility and collidability. Finally, we'll create a single list box with the Sort option disabled called IDC_EDITOBJECT_SPECIAL, which provides a list of special options.

Moving to the source code side of things, we'll create a new dialog procedure called EditObjectDlgProc. Within the dialog procedure we'll handle two different messages sent to the dialog procedure, WM_INITDIALOG and WM_COMMAND. Within the WM_INITDIALOG we'll declare a new variable of the bool data type called found and give it a default value of false. Next we'll create a loop to find the currently selected item. If an object is found to be selected, we'll place the object number selected into a string and set it into the IDC_EDITOBJECT_NUMBER edit control. After setting the object number, we'll set the name of the object in the IDC_EDIT-OBJECT_NAME edit control.

We'll then set the object type in the IDC_EDITOBJECT_TYPE edit control. Next we'll set the IDC_EDITOBJECT_VISIBLE and IDC_EDIT-OBJECT_COLLIDABLE check box controls to checked if their respective map object variables are true. If the variables are false, then we won't bother with them. With the two check box controls set, we will take the Y coordinate of the last vertex in the object and place the value into a string, which will then be set in the variable IDC_EDITOBJECT_STARTY. This value will help us define the starting and ending Y coordinates for the selected object. If the current object selected is a wall (type == OBJECTTYPE_WALL), then we should subtract the last vertex Y coordinate from the second vertex Y coordinate to produce the height of the object and place the value in IDC_EDITOBJECT_HEIGHT. When dealing with floors and ceilings we know that the height of both will be static among all the vertices; however, walls are different because they have two vertices, which must be higher up the Y coordinate than the floors or ceilings. Because the user selected a wall, we fill in the height variable so the selected wall's height as well as its original starting Y coordinate can be easily changed.

Probably one of the coolest variables in the map format is the fog value. In our game engine we'll use two different types of fog. The first type of fog we'll be creating in our game engine is the generic fog calculation that OpenGL uses to simulate fog, which requires that many vertices be on the screen to produce a nice effect. Although having lots of vertices on the screen is nice, an insufficient vertex count can cause the fog to go haywire and not look good. The other type of fog we'll use comes as an extension to the OpenGL API, and allows us to specify the fog depth on a per-vertex level. This feature allows us to create awesome-looking fog on specific objects. Before we get into the specifics (which are discussed further in Chapter 14), we'll take the value of the first fog vertex and copy it to the IDC_EDITOBJECT_FOG edit control. We use the first vertex of the selected object because we'll set every vertex to the same value. If you wanted to get really technical in your game engine you would probably want to create a function to edit the values of specific vertices as well as editing objects, but this topic is beyond the scope of this book.

The final control we must set is IDC_EDITOBJECT_SPECIAL, which happens to be our list box. This control will contain the special functions for the object that the user has selected. Like all list box controls we'll send the LB_RESETCONTENT message to the control to delete all the data already in it. Next we'll add the three strings to display "None", "End Level", and "–10% Health". These three strings will be the special values available to be assigned to the object. By default the "None" option is assigned to the object, indicating that the object has no special value. With the strings added to the control we simply send the LB_SETCURSEL message to the control, specifying the object's special variable to select the appropriate special option.

After setting the current selection we set the found variable to true, indicating that we've found an object. Once the loop to find objects has finished we check to see if the value of the found variable is false. If it's false, we display a message box saying no object was found and we exit the dialog.

Within the WM_COMMAND message we must check for a Cancel button press. If Cancel was clicked, we must exit the dialog. If the OK button was pressed, we must declare a new string called temp, one long variable called obj to store the object number, and two floats called height and starty.

Once the variables have been declared we grab the object number from the IDC_EDITOBJECT_NUMBER edit control and store the integer value in the obj variable. This gives us a simple way to know which object has been selected. After grabbing the object number we'll get the new object name from the IDC_EDITOBJECT_NAME control and store the value directly in the map->object[obj].name variable. The collidable and visibility variables must be set by checking to see if the IDC_EDITOBJECT_VISIBLE and IDC_EDITOBJECT_COLLIDABLE variables are checked. If either value is checked we check the corresponding map object variable.

The special variable is set by getting the currently selected item in the IDC_EDITOBJECT_SPECIAL variable and storing it in the map->object[obj].special variable. After grabbing the special variable we'll get the input fog variable and store it in a string, then loop through all the object's vertices, setting the fog variable from the string data. Next we'll get the start Y coordinate information from the IDC_EDITCONTROL_STARTY control and store the floating-point value in the starty variable. After getting the starting Y coordinate we'll loop through all the vertices and set the Y coordinate to the value of the starty variable. If the object selected is in fact a wall, we'll grab the height value from the IDC_EDITCONTROL_HEIGHT variable and store it in the height variable. Then we'll set the Y coordinates of the first two indexes of the vertex array to equal the starty variable plus the height variable, which would make the object taller, as we want to do.

With all the object variables set we can exit the dialog box by calling EndDialog. This completes the source code for editing the object values.

```
LRESULT CALLBACK EditObjectDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         bool found = false;

         for (long i = 0; i < map->header.max_objects; i++)
         {
            if (map->object[i].select_rgb[0] == select_rgb[0] &&
               map->object[i].select_rgb[1] == select_rgb[1] &&
               map->object[i].select_rgb[2] == select_rgb[2])
            {
               char temp[500];

               sprintf (temp, "%i", i);
               SetDlgItemText (hWnd, IDC_EDITOBJECT_NUMBER, temp);

               SetDlgItemText (hWnd, IDC_EDITOBJECT_NAME, map->object[i].name);

               sprintf (temp, "%i", map->object[i].type);
               SetDlgItemText (hWnd, IDC_EDITOBJECT_TYPE, temp);

               if (map->object[i].is_visible) SendDlgItemMessage (hWnd,
                   IDC_EDITOBJECT_VISIBLE, BM_SETCHECK, BST_CHECKED, 0);
               if (map->object[i].is_collidable) SendDlgItemMessage (hWnd,
                   IDC_EDITOBJECT_COLLIDABLE, BM_SETCHECK, BST_CHECKED, 0);

               sprintf (temp, "%f", map->object[i].vertex[3].xyz[1]);
               SetDlgItemText (hWnd, IDC_EDITOBJECT_STARTY, temp);

               if (map->object[i].type == OBJECTTYPE_WALL)
               {
                  sprintf (temp, "%f", map->object[i].vertex[0].xyz[1]-map->
                           object[i].vertex[3].xyz[1]);
                  SetDlgItemText (hWnd, IDC_EDITOBJECT_HEIGHT, temp);
               }

               sprintf (temp, "%f", map->object[i].vertex[0].fog_depth);
               SetDlgItemText (hWnd, IDC_EDITOBJECT_FOG, temp);

               SendDlgItemMessage (hWnd, IDC_EDITOBJECT_SPECIAL, LB_RESETCONTENT,
                                   0, 0);
               SendDlgItemMessage (hWnd, IDC_EDITOBJECT_SPECIAL, LB_ADDSTRING,
                                   0, (LPARAM)"None.");
               SendDlgItemMessage (hWnd, IDC_EDITOBJECT_SPECIAL, LB_ADDSTRING,
                                   0, (LPARAM)"End Level");
               SendDlgItemMessage (hWnd, IDC_EDITOBJECT_SPECIAL, LB_ADDSTRING,
                                   0, (LPARAM)"-10% Health");
               SendDlgItemMessage (hWnd, IDC_EDITOBJECT_SPECIAL, LB_SETCURSEL,
                                   map->object[i].special, 0);

               found = true;
            }
         }

         if (!found)
         {
            MessageBox (hWnd, "Could not find an object to edit", "Oops!", MB_OK);
            EndDialog (hWnd, 0);
         }
      }break;
      case WM_COMMAND:
      {
         if (wParam == IDCANCEL) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            char temp[500];
            long obj;
            float height;
            float starty;

            GetDlgItemText (hWnd, IDC_EDITOBJECT_NUMBER, temp, 500);
            sscanf (temp, "%i", &obj);

            GetDlgItemText (hWnd, IDC_EDITOBJECT_NAME, map->object[obj].name,
                500);

            if (SendDlgItemMessage (hWnd, IDC_EDITOBJECT_VISIBLE, BM_GETCHECK,
                0, 0) == BST_CHECKED) map->object[obj].is_visible = true;
            if (SendDlgItemMessage (hWnd, IDC_EDITOBJECT_COLLIDABLE, BM_GETCHECK,
                0, 0) == BST_CHECKED) map->object[obj].is_collidable = true;

            map->object[obj].special = SendDlgItemMessage (hWnd, IDC_EDITOBJECT_
                SPECIAL, LB_GETCURSEL, 0, 0);

            GetDlgItemText (hWnd, IDC_EDITOBJECT_FOG, temp, 500);
            for (long i = 0; i < map->object[obj].max_vertices; i++) sscanf
                (temp, "%f", &map->object[obj].vertex[i].fog_depth);


            GetDlgItemText (hWnd, IDC_EDITOBJECT_STARTY, temp, 500);
            sscanf (temp, "%f", &starty);

            for (i = 0; i < map->object[obj].max_vertices; i++) map->
                 object[obj].vertex[i].xyz[1] = starty;
            if (map->object[obj].type == OBJECTTYPE_WALL)
            {
               GetDlgItemText (hWnd, IDC_EDITOBJECT_HEIGHT, temp, 500);
               sscanf (temp, "%f", &height);

               map->object[obj].vertex[0].xyz[1] = starty + height;
               map->object[obj].vertex[1].xyz[1] = starty + height;
            }

            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}
```

With the Edit Objects dialog box now having all the needed functionality, we must move to the resource editor, more specifically the IDR_POPUP_MENU menu, and add a new menu item called "Edit" with a resource ID of ID_POPUP_EDIT. Next we'll move back to the main source file and add a new line to the WMCommand function that says if the wParam variable is equal to the ID_POPUP_EDIT menu resource we'll call the IDD_EDIT_OBJECT dialog using the DialogBox macro. When the user clicks the Edit menu item when an object is selected, a dialog box will now appear with all the values available to be edited.

Deleting Objects

The final section of this chapter discusses the Delete option. Obviously, this function is a must in any game editor because it gives us the ability to delete unwanted objects. I don't know anyone who can sit down and create a level from start to finish without making even one mistake. If you can do this, you've got my applause, but the majority of game and level designers will need this function to keep their sanity! For the majority of items in our MAP class, deleting objects is simple. All the arrays of data, with the exceptions of lights and objects, are deleted in the same way. For this reason we'll discuss the generic approach to deleting all the objects, then focus on lights and objects specifically. Throughout the discussion we'll use the light array as the example.

Before we begin discussing the generic deletion process we'll declare a new variable called idx, which is of the long data type. This variable will be used to store the index used to copy the deleted/selected item data. After declaring the idx variable we'll create a loop that will loop from 0 to the maximum value of the appropriate array (e.g., map->header.max_sounds). As we loop through the array we check to see if the globally declared select_rgb array is equal to the values in the array's select_rgb (e.g., if (select_rgb[0] == map->sound[i].select_rgb[0])). If all three values are equal, then we've found the selected item and must delete it from the array and exit from the Delete function. Once we've established which item is selected, we'll declare a new pointer variable of the array type called temp (e.g., MAP_SOUND *temp). This variable will be used to back up all the data within the current array, allowing us to rebuild the map without losing data.

After declaring the temp variable we'll allocate memory to it using the maximum value of the array plus an extra index for a buffer (e.g., temp = new MAP_SOUND[map->header.max_objects+1]). With the memory allocated for the temp array we copy the contents of each index of the original array into the temp array (e.g., strcpy (temp[i].filename, map->sound[i].filename) to back up the data. This is an obvious must; otherwise, the user would lose all the data! Once the array has been copied to the temp array we destroy the original array (e.g., delete [] map->sound) and reallocate memory for the array with one less index (e.g., map->sound = new MAP_SOUND[map->header.max_sounds]). We allocate the array with one less index because we're deleting the selected index.

Now that the array has memory allocated to it we can set our newly declared idx variable to 0. I'll explain the reason behind this in just a moment. After setting the idx variable to 0, we'll create another for loop, which will loop from 0 to the maximum value of the array (e.g., map->header.max_sounds). This loop will be used to copy the data from the temp array back to the original map->sound array without copying the selected item. Within the loop we'll create an if statement to ensure the current value of the main loop (i) is different from the copying loop value (i2). If the values aren't equal, we'll set the array index map->sound[idx] equal to the current temp[i2] array index and increment the idx variable by one. This completes both the if statement and the for loop to copy the data from the temp array back to the original array. After copying the data we'll destroy the temp array and decrease the maximum array value (map->header.max_sounds) by one. Then we'll call the return function without any parameters to exit the function. There's no point in continuing since we've already deleted the item that was selected.

The logic behind the code is simple. First we back up the original array into the temp array. Then we destroy and reallocate memory to the original array. As we copy the data back to the original array we check to see if the selected item is different from the current item being copied. If the values are different, we copy the value back to the original array and increment the idx variable. We use the idx variable as the index for the newly copied data because we cannot rely on the value of the i2 variable since it loops through the entire list of items and we only want to copy the data for every array minus the selected one. The idx variable only increments when the selected item is not the same as the copied item, and therefore each item is placed in sequential order without copying the dreaded selected item. As mentioned before, this process works for all data arrays in the MAP class with the exception of lights and objects, which have sub-arrays that must be taken care of. To make the deletion code for every MAP array of data, we simply change the variable names mentioned to the appropriate array.

The deletion process for lights is slightly different from the regular data arrays in that it requires us backing up and restoring the inclusion list for each light. To begin the coding process we'll copy and paste the code from any of the previous deletion functions and modify it according to the new specifications. Obviously each array name must be changed to its appropriate light variable equivalent. After changing the names of the variables we'll update the backup for loop with a new if statement, which will check the value of map->light[i].max_inclusions. If the value is greater than 0, which means we have inclusions in the inclusions list, we'll allocate memory for the temp[i].inclusions array to the size of map->light[i].max_inclusions and copy the contents of the map->light[i].inclusions[i2] variable to the temp[i].inclusions[i2] variable. Once we've copied the data to the temp inclusion list, we'll destroy the original inclusion list because we're going to rebuild the entire array in a few moments. At the end of the if statement we'll add an else clause, which will set the temp[i].inclusions variable to NULL, indicating that the array is empty!

Just like the regular data arrays, we'll destroy the main array, then reallocate memory for it minus one record. With the array now allocated memory, we should set the idx variable to 0 to ensure the placements are proper, then copy the data back to the original light data array as we would with any other array. With each iteration of the loop we'll check to ensure the current copied light is not the selected one, which would defeat the purpose of trying to delete the selected light! After copying the main light record from temp[i] to its original map->light[i], we must create a new if statement once again to check the value of map->light[i].max_inclusions. If the value of map->light[i].max_inclusions is greater than 0, then the inclusion list has data and we must copy the entire contents from the temp[i].inclusions array to its original map->light[i].inclusions array. After copying the data back to the original inclusion list, we'll delete the temp variable inclusion list, exit the inclusion copying if statement, and increment the idx variable to change the placement of the next light in the light array. Once the deletion process is finished, we simply destroy the temp variable array, decrease the map->header.max_lights variable, and exit the function by calling the return function without a parameter.

The third and final deletion algorithm we'll write is specific to objects because it has three sub-arrays we must copy and because we must modify the light arrays to delete object references in the inclusion list that no longer exist. Before we begin writing the final deletion function we'll modify the RemoveFromLight function to add two new overloaded variables to the function declaration, which were originally locally declared variables. If we look at the RemoveFromLight function we'll see that we declared two variables of the long data type. The variables, obj and light, will now be declared in the function declaration with the default value of –1 assigned to each. The obj variable will be declared first, then the light variable. Besides altering where we've declared the two variables, we'll add a new if statement to the function that will check the value of the obj variable. If the value of obj is equal to –1, then the original function call has occurred and we'll get the currently selected object information and prompt the user to select a light. If the value of obj is greater than –1, which it will be when we call the function from the Delete function and supply the object number and light number, then we'll skip all the user prompt questions and move directly to the code involved in physically removing the object from the light inclusion list. This sums up the modifications to the RemoveFromLight function and we can move back to the Delete function and continue the code necessary to delete an object.

Deleting objects works in the same manner as deleting lights with the exception that once we've established which item is selected (through the loop of checking the select_rgb values), we'll do some modifications to the light arrays. At this point you may be asking yourself why we would bother to do that. If we don't modify the light arrays to remove all links to the object we want to delete, eventually we'll be storing object numbers in the inclusion lists that go far beyond the number of objects physically loaded. This in turn will cause crashes and give nightmares to the end user trying to make levels. To begin the process we'll create a for loop that will loop from 0 to the value of map->header.max_lights. Within the loop we'll create another for loop from 0 to the value of the current light's max_inclusions value, which is used to check if the selected object exists in the light inclusion list. If the currently selected light is in the inclusion list, then we'll call the newly modified RemoveFromLight function with the parameters hWnd for the window, obj as the object, and finally the light number itself. This will delete the reference to the object in the light inclusion list and complete the for loop.

After deleting the references to the object we're going to be deleting from the inclusion list, we'll create another for loop that will once again loop from 0 to the value of the current light's max_inclusions value. This loop will shift all values in the light inclusion list above our object number down by one. If we didn't have this loop and we deleted object number 6 from our map, the other light inclusion values from 7 to 10 would be off by one value. If we shift all light inclusion values that are greater than the selected object number down by one, then all the inclusion information will stay the same as we delete objects. This process will allow us to easily synchronize the arrays without having to manually redo all the lighting information every time the user deletes an object. If we didn't add this loop, then we'd have to delete the entire inclusion list and reassign all the objects to the specific lights, which would be a pain in the keister.

Within the loop we simply create an if statement that will check to see if the map->light[i].inclusions[obj] object number is greater than the currently selected object. If it is, we decrement the value of map->light[i].inclusions[obj] by one to shift it down. It's simple, straightforward, and gets the job done! This completes the for loop for the inclusion shifting and for the light management functionality of deleting objects entirely. The rest of the deletion code works in the same way as the previous examples with the exception that if map->object[obj].max_textures, map->object [obj].max_vertices, or map->object[obj].max_triangles has a value greater than 0, then we'll copy the data to the temporary array and delete the original arrays. When we're restoring the data we'll copy the data back to the original from the temp array, deleting the temp array as we copy the data. At the bottom of the object deletion code we'll decrement the value of map->max_objects by one and call the return function. At the very bottom of the Delete function we'll display a message box that will indicate that the object couldn't be found for deletion. This will let the user know that the deletion process failed since the Delete function couldn't find any suitable select_rgb matches in the arrays. We've now finished the code necessary to delete objects and display it here.

```
void Delete(HWND hWnd)
{
   long idx = 0;

   for (long i = 0; i < map->header.max_entities; i++)
   {
      if (map->entity[i].select_rgb[0] == select_rgb[0] &&
         map->entity[i].select_rgb[1] == select_rgb[1] &&
         map->entity[i].select_rgb[2] == select_rgb[2])
      {
         if (map->header.max_entities > 1)
         {
            MAP_ENTITY *temp;

            temp = new MAP_ENTITY[map->header.max_entities+1];
            for (long i2 = 0; i2 < map->header.max_entities; i2++) temp[i2] =
                 map->entity[i2];

            delete [] map->entity;
            map->entity = new MAP_ENTITY[map->header.max_entities];

            idx = 0;
            for (i2 = 0; i2 < map->header.max_entities; i2++)
            {
               if (i2 != i)
               {
                  map->entity[idx] = temp[i2];
```

```
                    idx++;
                 }
              }
              delete [] temp;
           }
           else
           {
              delete [] map->entity;
              map->entity = NULL;
           }
           map->header.max_entities--;

           return;
        }
     }


     for (i = 0; i < map->header.max_items; i++)
     {
        if (map->item[i].select_rgb[0] == select_rgb[0] &&
           map->item[i].select_rgb[1] == select_rgb[1] &&
           map->item[i].select_rgb[2] == select_rgb[2])
        {
           if (map->header.max_items > 1)
           {
              MAP_ITEM *temp;

              temp = new MAP_ITEM[map->header.max_items+1];
              for (long i2 = 0; i2 < map->header.max_items; i2++) temp[i] =
                   map->item[i];

              delete [] map->item;
              map->item = new MAP_ITEM[map->header.max_items];

              idx = 0;
              for (i2 = 0; i2 < map->header.max_items; i2++)
              {
                 if (i2 != i)
                 {
                    map->item[idx] = temp[i2];
                    idx++;
                 }
              }
              delete [] temp;
           }
           else
           {
              delete [] map->item;
              map->item = NULL;
           }
           map->header.max_items--;

           return;
```

```
      }
   }


   for (i = 0; i < map->header.max_sounds; i++)
   {
      if (map->sound[i].select_rgb[0] == select_rgb[0] &&
         map->sound[i].select_rgb[1] == select_rgb[1] &&
         map->sound[i].select_rgb[2] == select_rgb[2])
      {
         if (map->header.max_sounds > 1)
         {
            MAP_SOUND *temp;

            temp = new MAP_SOUND[map->header.max_sounds+1];
            for (long i2 = 0; i2 < map->header.max_sounds; i2++) temp[i2] =
                 map->sound[i2];

            delete [] map->sound;
            map->sound = new MAP_SOUND[map->header.max_sounds];

            idx = 0;
            for (i2 = 0; i2 < map->header.max_sounds; i2++)
            {
               if (i2 != i)
               {
                  map->sound[idx] = temp[i2];
                  idx++;
               }
            }
            delete [] temp;
         }
         else
         {
            delete [] map->sound;
            map->sound = NULL;
         }
         map->header.max_sounds--;

         return;
      }

   }


   for (i = 0; i < map->header.max_lights; i++)
   {
      if (map->light[i].select_rgb[0] == select_rgb[0] &&
         map->light[i].select_rgb[1] == select_rgb[1] &&
         map->light[i].select_rgb[2] == select_rgb[2])
      {
         if (map->header.max_lights > 1)
         {
```

```
      MAP_LIGHT *temp;

      temp = new MAP_LIGHT[map->header.max_lights+1];
      for (long i2 = 0; i2 < map->header.max_lights; i2++)
      {
         temp[i2] = map->light[i2];
         if (map->light[i2].max_inclusions > 0)
         {
            temp[i2].inclusions = new int[temp[i2].max_inclusions+1];
            for (long i3 = 0; i3 < map->light[i2].max_inclusions; i3++)
                 temp[i2].inclusions[i3] = map->light[i2].inclusions[i3];

            delete [] map->light[i2].inclusions;
         }
         else temp[i2].inclusions = NULL;
      }

      delete [] map->light;
      map->light = new MAP_LIGHT[map->header.max_lights];

      idx = 0;
      for (i2 = 0; i2 < map->header.max_lights; i2++)
      {
         if (i2 != i)
         {
            map->light[idx] = temp[i2];
            if (map->light[idx].max_inclusions > 0)
            {
               map->light[idx].inclusions = new int[map->
                    light[i2].max_inclusions+1];
               for (long i3 = 0; i3 < map->light[idx].max_inclusions;
                    i3++) map->light[idx].inclusions[i3] =
                    temp[i2].inclusions[i3];

               delete [] temp->inclusions;
            }
            idx++;
         }
      }
      delete [] temp;
   }
   else
   {
      if (map->light[0].max_inclusions > 0) delete [] map->
          light[0].inclusions;
      delete [] map->light;
      map->light = NULL;
      }
      map->header.max_lights--;

      return;
    }
  }


  for (i = 0; i < map->header.max_objects; i++)
  {
    if (map->object[i].select_rgb[0] == select_rgb[0] &&
      map->object[i].select_rgb[1] == select_rgb[1] &&
      map->object[i].select_rgb[2] == select_rgb[2])
    {
      if (map->header.max_objects > 0)
      {
        MAP_OBJECT *temp;

        for (long light = 0; light < map->header.max_lights; light++)
        {
          for (long obj = 0; obj < map->light[light].max_inclusions; obj++)
          {
            if (map->light[light].inclusions[obj] == i) RemoveFromLight
               (hWnd, obj, light);
          }

          for (obj = 0; obj < map->light[light].max_inclusions; obj++)
          {
            if (map->light[light].inclusions[obj] > i) map->
               light[light].inclusions[obj]--;
          }
        }


        temp = new MAP_OBJECT[map->header.max_objects+1];
        for (long i2 = 0; i2 < map->header.max_objects; i2++)
        {
          temp[i2].is_collidable =     map->object[i2].is_collidable;
          temp[i2].is_visible    =     map->object[i2].is_visible;
          temp[i2].max_textures  =     map->object[i2].max_textures;
          temp[i2].max_triangles =     map->object[i2].max_triangles;
          temp[i2].max_vertices  =     map->object[i2].max_vertices;
          strcpy (temp[i2].name, map->object[i2].name);
          temp[i2].select_rgb[0] =     map->object[i2].select_rgb[0];
          temp[i2].select_rgb[1] =     map->object[i2].select_rgb[1];
          temp[i2].select_rgb[2] =     map->object[i2].select_rgb[2];
          temp[i2].special       =     map->object[i2].special;
          temp[i2].type          =     map->object[i2].type;


          if (map->object[i2].max_textures > 0)
          {
```

```
         temp[i2].texture = new MAP_TEXTURE[map->
              object[i2].max_textures+1];
         for (long tex = 0; tex < map->object[i2].max_textures; tex++)
              temp[i2].texture[tex] = map->object[i2].texture[tex];
         delete [] map->object[i2].texture;
      }
      else temp[i2].texture = NULL;


      if (map->object[i2].max_triangles > 0)
      {
         temp[i2].triangle = new MAP_TRIANGLE[map->
              object[i2].max_triangles+1];
         for (long tri = 0; tri < map->object[i2].max_triangles; tri++)
              temp[i2].triangle[tri] = map->object[i2].triangle[tri];
         delete [] map->object[i2].triangle;
      }
      else temp[i2].triangle = NULL;


      if (map->object[i2].max_vertices > 0)
      {
         temp[i2].vertex = new MAP_VERTEX[map->object[i2].max_
              vertices+1];
         for (long ver = 0; ver < map->object[i2].max_vertices; ver++)
              temp[i2].vertex[ver] = map->object[i2].vertex[ver];
         delete [] map->object[i2].vertex;
      }
      else temp[i2].vertex = NULL;
   }


   delete [] map->object;
   map->object = new MAP_OBJECT[map->header.max_objects+1];


   idx = 0;
   for (i2 = 0; i2 < map->header.max_objects; i2++)
   {
      if (i != i2)
      {
         map->object[idx].is_collidable   =     temp[i2].is_collidable;
         map->object[idx].is_visible      =     temp[i2].is_visible;
         map->object[idx].max_textures    =     temp[i2].max_textures;
         map->object[idx].max_triangles   =     temp[i2].max_triangles;
         map->object[idx].max_vertices    =     temp[i2].max_vertices;
         strcpy (map->object[idx].name, temp[i2].name);
         map->object[idx].select_rgb[0]   =     temp[i2].select_rgb[0];
         map->object[idx].select_rgb[1]   =     temp[i2].select_rgb[1];
         map->object[idx].select_rgb[2]   =     temp[i2].select_rgb[2];
         map->object[idx].special         =     temp[i2].special;
         map->object[idx].type            =     temp[i2].type;
```

```
            if (map->object[idx].max_textures > 0)
            {
               map->object[idx].texture = new MAP_TEXTURE[map->
                    object[idx].max_textures+1];
               for (long tex = 0; tex < map->object[idx].max_textures;
                    tex++) map->object[idx].texture[tex] =
                    temp[i2].texture[tex];
               delete [] temp[i2].texture;
            }
            else map->object[idx].texture = NULL;


            if (map->object[idx].max_triangles > 0)
            {
               map->object[idx].triangle = new MAP_TRIANGLE[map->
                    object[idx].max_triangles+1];
               for (long tri = 0; tri < map->object[idx].max_triangles;
                    tri++) map->object[idx].triangle[tri] =
                    temp[i2].triangle[tri];
               delete [] temp[i2].triangle;
            }
            else map->object[i2].triangle = NULL;


            if (map->object[idx].max_vertices > 0)
            {
               map->object[idx].vertex = new MAP_VERTEX[map->
                    object[idx].max_vertices+1];
               for (long ver = 0; ver < map->object[idx].max_vertices;
                    ver++) map->object[idx].vertex[ver] =
                    temp[i2].vertex[ver];
               delete [] temp[i2].vertex;
            }
            else map->object[idx].vertex = NULL;
            idx++;
         }
      }
   }
   else
   {
      if (map->object[0].max_vertices > 0) delete [] map->object[0].vertex;
      if (map->object[0].max_triangles > 0) delete [] map->
          object[0].triangle;
      if (map->object[0].max_textures > 0) delete [] map->object[0].texture;
      delete [] map->object;
      map->object = NULL;
   }
   map->header.max_objects--;
```

```
            return;
         }
      }

      MessageBox (hWnd, "Unable to Delete", "Error", MB_OK);
   }
```

Chapter Example

Due to the size of the Chapter 8 example I won't show the source code here but rather direct you to the ex8_1 folder of the downloadable file.

Conclusion

In Chapter 8 we wrote the code necessary to duplicate, delete, move, select, and edit objects, and much more. Without these functions, we wouldn't have much of a map editor or be able to design great games with it. Of course there are still all sorts of things you could add to this map editor. We'll add saving/loading in the next chapter, then begin coding the engine itself, which is the fun part.

This page intentionally left blank.

Chapter 9

Saving and Opening Files

Saving and opening files are small but important features in any map editor. Without these features we'd have no method of executing the same map file over and over again. There would also be no point in creating our map editor if we couldn't read/write the data into a file. For this reason this chapter will discuss how we read and write data to the files. Unlike the previous chapters, this one is short, since saving and opening is straight to the point as long as you understand how to read and write files in the first place.

Saving Files

Saving the data from our maps to a file is a very important function. Without this functionality we would not be able to load our created map data into our game, which would make the map editor completely worthless. Before we begin writing the code for the Save function, we should consider the benefits of binary and text modes to store the data. If we save the map data in a text file, we are essentially opening up the doors to an easy and open format that allows the user to update the file without using the map editor. This is a great benefit to people who want to create their own maps without having to use the map editor or those who want to tweak certain settings without running the map editor.

If you don't want to share your map data with the end users, you can save the data in binary form. This does not protect you from crackers who may reverse engineer your map format, but it does protect you against the average user trying to edit the levels to his or her advantage. If you wanted to protect the data from crackers you would need to create a data encryption system for the structures/data in the map. Data encryption is beyond the scope of this book, so we'll continue discussing the pros and cons of saving the data in binary form. Another benefit to writing the data in binary form is that we can write the data structures directly to the file as opposed to writing formatted text, which is simply easier to work with in most instances. The downside to storing binary data is that it's harder to upgrade the file format later on. We've helped simplify that process by providing the versioning information inside the file format, helping us to choose which version of a specific structure we should use when we have multiple versions of our file format.

By now you've probably guessed that we're going to be saving our files in binary form instead of text. We'll be adding the saving code to the MAP class, allowing it to be reused later on. I can't think of a reason why you'd want to use the Save function outside the map editor, but by putting it in the MAP class, we're able to access it for whatever purpose we want. To begin the coding process, we'll add our Save function to the MAP class. The single parameter for the function will be a pointer of chars called filename, which as the name implies will store the filename of the map. Within the function we'll declare a new variable of the FILE type, which will be our file pointer to the location to which the information will be saved.

After declaring our file pointer variable, we simply open the file for binary writing and begin writing the data. We'll write the version variable (MAP_VERSION) structure to the file first. We write the version information first so we can compare this information to our version/revision constants and then load the appropriate structures for the map version. For example, if we write the version structure to the file first we can easily identify whether to load MAP_HEADER or MAP_HEADER_2 if we have an upgraded version of the header structure. With each write to our map file, we'll use the sizeof function on the structure rather than the variable to identify the size of the structure.

Now that we've saved the version information to the file, we can write the header and details to the file, keeping in mind that we use the appropriate structure for each sizeof function call. If the value of the use_skybox variable in the header structure is true, then we'll write the skybox structure variable to the file to complete the header section of the map format. Finally, the last structure to write is the fog variable, provided the value of header.use_fog is true. After writing the main headers to the file, we'll discuss writing the data itself. Like every other function that's had to use the object array, we'll spend a majority of our time working to save the objects. To save the objects we simply create a loop that will go through all the objects and write each object to the file. Following each object write, we'll loop through the vertices, triangles, and texture arrays, writing each index for each array. Although our dealings with the object array have been fairly complex in the past, saving the data is dead simple!

All the other arrays with the exception of the light array will simply loop through their maximum header value and write each index to the map file. The light array is different because in addition to looping through the maximum header value, we must also check to see if there is any data in the inclusion list. If there is data in the inclusion list, we must create a secondary loop after writing the light index to write each index in the inclusion list to the file as well. Once the arrays are all written to the file, we can close the file and exit the function. Wasn't that simple? Of course opening files is slightly tricky, but we'll get to that in a few minutes. For the moment we'll display the source code for the Save function below.

```
void MAP::Save(char *filename)
{
   FILE *fp;

   fp = fopen (filename, "wb");

      fwrite (&version, sizeof(MAP_VERSION), 1, fp);
      fwrite (&header, sizeof(MAP_HEADER), 1, fp);
      fwrite (&details, sizeof(MAP_DETAILS), 1, fp);
      if (header.use_skybox) fwrite (&skybox, sizeof(MAP_SKYBOX), 1, fp);
      if (header.use_fog) fwrite (&fog, sizeof(MAP_FOG), 1, fp);

      for (long i = 0; i < header.max_objects; i++)
      {
         fwrite (&object[i], sizeof(MAP_OBJECT), 1, fp);

         for (long i2 = 0; i2 < object[i].max_vertices; i2++) fwrite
             (&object[i].vertex[i2], sizeof(MAP_VERTEX), 1, fp);
         for (i2 = 0; i2 < object[i].max_triangles; i2++) fwrite
             (&object[i].triangle[i2], sizeof(MAP_TRIANGLE), 1, fp);
         for (i2 = 0; i2 < object[i].max_textures; i2++) fwrite
             (&object[i].texture[i2], sizeof(MAP_TEXTURE), 1, fp);
      }

      for (i = 0; i < header.max_entities; i++) fwrite (&entity[i],
           sizeof(MAP_ENTITY), 1, fp);
      for (i = 0; i < header.max_lights; i++)
      {
         fwrite (&light[i], sizeof(MAP_LIGHT), 1, fp);
         for (long i2 = 0; i2 < light[i].max_inclusions; i2++) fwrite
             (&light[i].inclusions[i2], sizeof(long), 1, fp);
      }

      for (i = 0; i < header.max_sounds; i++) fwrite (&sound[i],
           sizeof(MAP_SOUND), 1, fp);
      for (i = 0; i < header.max_items; i++) fwrite (&item[i], sizeof(MAP_ITEM),
           1, fp);

   fclose (fp);
}
```

With the source code to save the map files now written in the MAP class, we can write a small function to display the common Save dialog box, which is seen on almost every Windows program ever made. The first thing we'll do is create a new function called Save, which will have a single parameter of the HWND data type. Inside the function we'll declare two new local variables. The first variable is called ofn and is of the OPENFILENAME type, and the second variable is an array of 500 characters called filename, which, as the name suggests, will contain the filename. The OPENFILENAME structure contains all the settings required to display an Open/Save dialog box. All we must do is simply set the default values for the variables inside the ofn variable structure, then call either the GetOpenFileName function to open the file or GetSaveFileName to save the file. In case you're wondering, the only difference between the two dialog boxes is the default button that says either "Save" or "Open," depending on which option you chose.

Before we begin filling in the ofn variable, we must string copy a default filename to the filename variable. I used the name "myfile.map" as the default, but you can literally put anything you want in there. We fill this variable in because the dialog box wants some sort of default value in the filename. After setting a default filename we'll use memset to fill the ofn structure with 0s. We do this to avoid typing every member of the structure, which in this case is more than 10 options. It's much easier to simply fill it in with 0s, then fill in the details we need to be different. The first option we'll set is ofn.lStructSize, which must be set to the size of the OPENFILENAME structure. We set the structure size so the GetSaveFileName function knows how much memory the structure uses to easily determine which structure was passed. In some cases there can be several structure versions and some APIs use the sizing method to determine its size.

Next we'll set ofn.hwndOwner to our input parameter hWnd. This tells the dialog box which window is the owner of it. When this function isn't selected, odd results may sometimes occur. After setting the owner window, we set the filter types. In case you're wondering, a filter is the file information/extension details you see in the bottom part of Save dialog boxes. When you click the down arrow, normally you'll see several different types of formats supported. Since we're only going to concern ourselves with our proprietary map format, we'll set the ofn.lpstrFilter variable to "Map Files (*.map)\0*.map". The filter string is normally broken into two sections. The first section is the format description, which in our case is the "Map Files (*.map)" section. Then we add the "\0" to the string and the associated extension for the file format. We're not going to bother adding other formats to the list, but if you wanted to use the "All Files (*.*)" option you would simply add a semicolon after the extension definition and add your next file format.

After setting the file filter information, we set ofn.lpstrFile to our filename variable to set the default filename when the Save dialog box first starts. Following the filename, we'll set the ofn.nMaxFile variable to the size of the filename variable so we know how many characters can be accepted. We've now finished filling in the ofn structure and can call the function GetSaveFileName, specifying the address of the ofn structure. If the user clicks the Save button and the function succeeds, the return value is true and we can call our map->Save function. When the function succeeds, the save filename will be stored in the ofn.lpstrFile variable. When we call the map->Save function, we specify the returned filename ofn.lpstrFile as the single parameter. This completes the source code for saving the map data so we can display the source code for the Save function below.

```
void Save(HWND hWnd)
{
   OPENFILENAME ofn;
   char filename[500];

   strcpy (filename, "myfile.map");
   memset (&ofn, 0x0, sizeof(ofn));
   ofn.lStructSize  = sizeof(OPENFILENAME);
   ofn.hwndOwner    = hWnd;
   ofn.lpstrFilter  = "Map Files(*.map)\0*.map";
   ofn.lpstrFile    = filename;
   ofn.nMaxFile     = sizeof(filename);
   if (GetSaveFileName (&ofn)) map->Save (ofn.lpstrFile);
}
```

Clearing Map Data

Before we begin discussing the code for opening files we must discuss the addition of one new function to the map editor. The new function will be called Clear and will be placed in the MAP class. The Clear function will release all the arrays in our map class that have memory allocated to them, plus fill the header, details, and skybox variable structures with 0s. To begin we simply check to see if the maximum value of the array is greater than 0. If the value is greater than 0, we'll release the memory and set the pointer to NULL. In the cases of the object and light arrays, we'll check the values of their sub-arrays first and release their memory, then release the main arrays. After we release the arrays we simply call the memset function to fill in the header, details, and skybox variable structures with 0s to ensure that all the data has been set back to its original settings. We use this function to clear any loaded data before we attempt to either load another map or start a new map. Now that we've discussed the code for the Clear function we can display the source code below.

```
void MAP::Clear()
{
   if (header.max_objects > 0)
   {
      for (long i = 0; i < header.max_objects; i++)
      {
         if (object[i].max_vertices > 0) delete [] object[i].vertex;
         if (object[i].max_triangles > 0) delete [] object[i].triangle;
         if (object[i].max_textures > 0) delete [] object[i].texture;
      }
      delete [] object;
      object = NULL;
   }

   if (header.max_sounds > 0)
   {
      delete [] sound;
      sound = NULL;
   }

   if (header.max_entities > 0)
   {
      delete [] entity;
      entity = NULL;
   }

   if (header.max_lights > 0)
   {
      for (long i = 0; i < header.max_lights; i++)
      {
         if (light[i].max_inclusions > 0) delete [] light[i].inclusions;
      }
      delete [] light;
      light = NULL;
   }

   if (header.max_items > 0)
   {
      delete [] item;
      item = NULL;
   }

   memset (&header, 0, sizeof(MAP_HEADER));
   memset (&details, 0, sizeof(MAP_DETAILS));
   memset (&skybox, 0, sizeof(MAP_SKYBOX));
}
```

After writing the source code we'll move back to the resource editor and add a new menu item called "New" to the File menu. The New menu item will have a resource ID of ID_NEW. Moving to the coding side of things, the WMCommand function must be updated to make the New function available. As with button clicks, we'll check to see if the wParam variable is equal to the ID_NEW resource ID. If the values are equal, then we'll call the map->Clear method. By adding this last piece of functionality to the Clear function, we can now clear the current file at any time and start a new one. Since we'd already written the code to clear the file before we load a new one, it was easy to add a new menu resource to simply clear the file as the new function. See how things work out for the best? The update for the WMCommand function is given below.

```
else if (wParam == ID_NEW) map->Clear();
```

Opening Files

Obviously opening files is an important feature for the editor. It's important not only because it loads the files, but because it will complete our map editor, giving the user the ability to create maps for our game engine, which we'll be creating in Part II. The source code for opening files is similar to the saving code we wrote in the previous section, with the exception that we allocate memory for each array before we read it in. With this in mind, let's begin the source code by creating a new method in the MAP class called Open. The method will have one parameter called filename, which is a string pointer. The method also has a return type of bool, indicating whether the function has succeeded (true) or failed (false).

Inside the method we'll once again declare a new file pointer called fp. Before we open the file, we'll call the Clear method we created in the previous section to ensure there is no data already in the map. If we don't call the Clear method, we run the risk of allocating memory for arrays that have already been allocated for, and we risk having data corruption throughout our map data from a previously loaded map. After we call the Clear method, we can open the file. If the file doesn't exist, then we'll immediately exit the method with a false return value. We do this to avoid reading a file that doesn't exist and avoid a crash!

Each record will be read in the order it was written. The version structure will be read, then the header, then details, and then, if the values of header.use_skybox/header.use_fog are true, we'll read the skybox and/or fog data. In the event we changed the map format and had different structures for different versions of the editor, we would simply use the values we read from the file to determine which structure to use. This of course should be done before we read the other header structures. Once the headers have been read, we begin reading the map data itself. Before we attempt to read the data we must ensure the data structure we want to read contains data. If the maximum number of entries is greater than 1, then we'll allocate the appropriate memory for the data and read it from the file. Putting this to use, if the header.max_objects variable is greater than 0, then we allocate memory for the object array for the number of entries in the array. After allocating the memory we loop through each of the entries, reading each record into the array.

Unlike most of the other arrays in the map format, the object and light structures have arrays within the structure that must be checked for array entries. If the sub-arrays have entries, they should have memory allocated to them and read the data into each array. In the case of the object array, we would check to see if the variables max_vertices, max_triangles, and max_textures have a value greater than 0. If the values are greater than 0, we allocate the memory for the appropriate arrays, read the data for each array, and continue through the main iteration of the loop. The light inclusion list works in the same manner. As mentioned earlier, the sound, entity, and item structures would simply allocate memory if their max variable was greater than 0 and then read each record from the file. With every instance of an array, if the maximum entries value is 0, then we'll set the array to NULL as opposed to allocating memory for it. This happens for all arrays regardless of whether they are within one of our structures or a regular array. After reading all the arrays we simply exit the function, returning the value true to indicate that the function was successful. With the function complete, we display the source code below.

```
bool MAP::Open(char *filename)
{
   FILE *fp;

   Clear();

   fp = fopen (filename, "rb");
   if (fp == NULL) return (false);

      fread (&version, sizeof(MAP_VERSION), 1, fp);
      fread (&header, sizeof(MAP_HEADER), 1, fp);
      fread (&details, sizeof(MAP_DETAILS), 1, fp);
      if (header.use_skybox) fread (&skybox, sizeof(MAP_SKYBOX), 1, fp);

      if (header.use_fog) fread (&fog, sizeof(MAP_FOG), 1, fp);

      if (header.max_objects > 0)
      {
         object = new MAP_OBJECT[header.max_objects+1];
         for (long i = 0; i < header.max_objects; i++)
         {
            fread (&object[i], sizeof(MAP_OBJECT), 1, fp);

            object[i].vertex    = new MAP_VERTEX[object[i].max_vertices+1];
            object[i].triangle  = new MAP_TRIANGLE[object[i].max_triangles+1];
            object[i].texture   = new MAP_TEXTURE[object[i].max_textures+1];

            for (long i2 = 0; i2 < object[i].max_vertices; i2++) fread
                (&object[i].vertex[i2], sizeof(MAP_VERTEX), 1, fp);
            for (i2 = 0; i2 < object[i].max_triangles; i2++) fread
                (&object[i].triangle[i2], sizeof(MAP_TRIANGLE), 1, fp);
            for (i2 = 0; i2 < object[i].max_textures; i2++) fread
                (&object[i].texture[i2], sizeof(MAP_TEXTURE), 1, fp);
         }
      }
      else object = NULL;


      if (header.max_entities > 0)
      {
         entity = new MAP_ENTITY[header.max_entities+1];
```

```
            for (long i = 0; i < header.max_entities; i++)
            {
               fread (&entity[i], sizeof(MAP_ENTITY), 1, fp);
            }
         }
         else entity = NULL;


         if (header.max_lights > 0)
         {
            light = new MAP_LIGHT[header.max_lights+1];
            for (long i = 0; i < header.max_lights; i++)
            {
               fread (&light[i], sizeof(MAP_LIGHT), 1, fp);

               if (light[i].max_inclusions > 0)
               {
                  light[i].inclusions = new int[light[i].max_inclusions+1];
                  for (long i2 = 0; i2 < light[i].max_inclusions; i2++)
                  {
                     fread (&light[i].inclusions[i2], sizeof(long), 1, fp);
                  }
               }
               else light[i].inclusions = NULL;
            }
         }
         else light = NULL;


         if (header.max_sounds > 0)
         {
            sound = new MAP_SOUND[header.max_sounds+1];
            for (long i = 0; i < header.max_sounds; i++)
            {
               fread (&sound[i], sizeof(MAP_SOUND), 1, fp);
            }
         }
         else sound = NULL;


         if (header.max_items > 0)
         {
            item = new MAP_ITEM[header.max_items+1];
            for (long i = 0; i < header.max_items; i++)
            {
               fread (&item[i], sizeof(MAP_ITEM), 1, fp);
            }
         }
         else item = NULL;

      fclose (fp);
      return (true);
   }
```

With the source code for opening files complete, we can write the code to display the Open dialog box. As mentioned in the first section, the code for opening Save and Open dialog boxes is similar, with the exception of the final function call. In the case of the Save dialog box, we called the function GetSaveFileName and gave the address of the ofn structure. In the case of the Open dialog box, we can copy and paste the source code from the Save function, changing the function name to Open and the function call to GetOpenFileName, and of course we'd call our newly created map->Open filename. The source code for the Open function is shown below.

```
void Open(HWND hWnd)
{
   OPENFILENAME ofn;
   char filename[500];

   strcpy (filename, "myfile.map");
   memset (&ofn, 0x0, sizeof(ofn));
   ofn.lStructSize  = sizeof(OPENFILENAME);
   ofn.hwndOwner    = hWnd;
   ofn.lpstrFilter  = "Map Files(*.map)\0*.map";
   ofn.lpstrFile    = filename;
   ofn.nMaxFile     = sizeof(filename);
   if (GetOpenFileName (&ofn)) map->Open (ofn.lpstrFile);
}
```

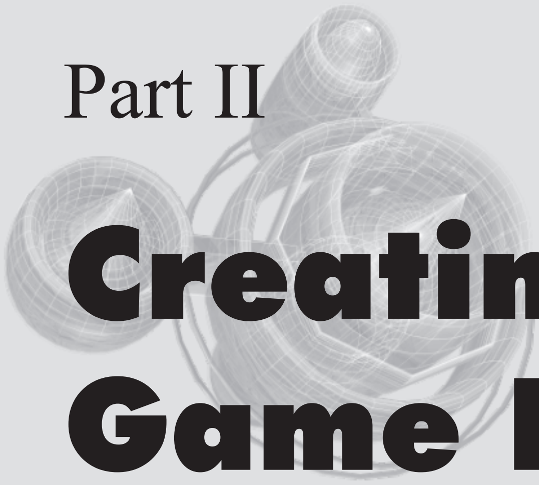
Chapter Example

Please see the ex9_1.cpp example from the downloadable file.

Conclusion

We're done! We've finished our basic map editor, which will allow us to make maps for our game engine. We won't be touching this source code after this point, and only make references to the editor as a program when we're talking about making the game and special effects/features. I thought adding the save/open functionality to the end of the map editor chapters would be a nice break because it is a simple topic to discuss and relatively short compared to some of the huge chapters earlier.

Part I of the book, "Creating the Map Editor," familiarized you with basic Windows programming concepts. The second part of the book, "Creating the Game Engine," will discuss intermediate graphical topics and how to apply them in your game engine. These two parts will teach you a great deal of the information you need to know about making your own 3D video games with OpenGL in the C language.



Part II

Creating the Game Engine

In this part of the book we'll create our game engine, reusing source code from the map editor to load the maps and learn how to load bitmaps, display textures, draw lighting, and add collision detection and all sorts of other nifty game-related material. Essentially this is where the bulk of the work gets done! Unlike the chapters in Part I, most sections in these chapters will be different and not as repetitive because we've done a majority of the boring work already.

In Chapter 10 we'll discuss the new base code we'll be using in the game, and the code required to check which graphics resolutions your video card can handle. Following chapters will discuss texturing, rendering, adding movement and lighting, and using 3D models, along with a variety of other topics that will help you create a game using OpenGL.

Chapter 10

Basic Game Setup

The New Base Code

All through the first part of the book we constantly reused the same source code while adding new functionality to it. Since we're starting a brand new program, this is obviously not the best idea since we'd spend more time chopping code out of the program than it's worth. We can, however, use the source code from our first example, ex1_1, as the base code for our game engine and modify it to our current needs. When we create the new project for the game engine we'll simply call the new file ex10_1 (following the current naming convention), and copy the source code from the file ex1_1.cpp to the ex10_1.cpp file.

Now we've got a basic Windows program, but you may be wondering how the heck we're going to turn this light-gray windowed program into a full-screen game engine. The answer to this question is simple — through modifying the class/window creation parameters! The first modification we'll make is in the code to fill in the window class data. We've set the window background color to be light gray (LTGRAY_BRUSH), which although it's not important will be changed to black (BLACK_BRUSH). We change the background to black since it's the easiest color to work with when putting colors on the background, and light gray doesn't look good (in my opinion) when it's full screen. After changing the background to black, we must change the class name of the window from "ME" to "GE". There are two instances of "ME" in the program. The first is when you register the class window, and the second is in the call to CreateWindow. Once the changes have been made we'll change the window title in the CreateWindow function from "Map Editor" to "Game Engine", for obvious reasons,

If we recompile the program and run it, you'll notice that the window title now says "Game Engine" instead of "Map Editor." This of course is more of a cosmetic adjustment than a necessity, but it will indicate that the program is completely different from the map editor. Since we want our game to be displayed in full-screen mode we'll change the window style variables from WS_OVERLAPPEDWINDOW, which produces a regular window, to WS_POPUP, which will make a borderless window on the screen. To make the window full screen we change the hardcoded width and height of the window from 640x480 by calling the function GetSystemMetrics and specifying the parameter SM_CXSCREEN for the width and SM_CYSCREEN for the height to retrieve the maximum width and height of the screen.

Rather than rewrite the source code for initializing OpenGL we'll simply reuse our OPENGL classes, which is one of the great features of modular programming. We'll simply include the raster.h file to complete our base code. The following will be used as our base code and we'll make additions to it from now on.

```
#include <windows.h>
#include <stdio.h>
#include "raster.h"

HWND Window;

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR
                    lpCmdString, int CmdShow)
{

   WNDCLASS   wc;
   MSG        msg;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "GE";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);
   }

   Window = CreateWindow("GE", "Game Engine", WS_POPUP | WS_VISIBLE, 0, 0,
            GetSystemMetrics(SM_CXSCREEN), GetSystemMetrics(SM_CYSCREEN), NULL,
            NULL, hInstance, NULL);
   if (Window == NULL)
```

```
    {
        MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
        return (0);
    }


    while (1)
    {
        if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
        {
            if (msg.message == WM_QUIT) break;
            TranslateMessage(&msg);
            DispatchMessage (&msg);
        }
    }

    return (1);
}
```

Using a Dialog Box for Game Configuration

If you've ever experimented with the Microsoft DirectX SDK you've probably noticed that most of the examples and even some commercial games and software that use the SDK have dialog boxes that display initial game settings such as the graphical resolution and other important details. The DirectX SDK uses enumeration to query the resolutions the video card supports. Unfortunately, OpenGL doesn't have enumeration support built into its API, which presents a problem to us game developers if we want to support multiple video resolutions. On one hand, we can hardcode several resolutions for the user to choose from, although we run into the issue that some users may want to play the game at resolutions higher or lower than the game supports.

The alternative to hardcoding the video resolution is to query the Windows graphics drivers through the EnumDisplaySettings function to list the resolutions the graphics adapter supports. Although the source code won't be cross-platform compatible with other operating systems like GNU/Linux, it does provide a good method for enumerating which graphic modes to use without hardcoding any specific resolutions or using that heinous SDK! To begin our new endeavor we'll create a small dialog box that has the window caption of "Game Config" with a resource ID of IDD_CONFIG. As I've stated frequently, I prefer having the dialog centered, but it's not necessary.

Inside the dialog box we'll change the Cancel button to have the caption "Exit" and change the resource ID to IDEXIT rather than IDCANCEL, which is the default. Next we'll insert a combo box called IDC_RESOLUTION. As with all our combo boxes in dialogs, it's a good idea to change the style to a drop-down list and turn off sorting so we don't end up with weird results when we view the listed resolutions. Figure 10.1 shows a mockup of how the dialog box could look.

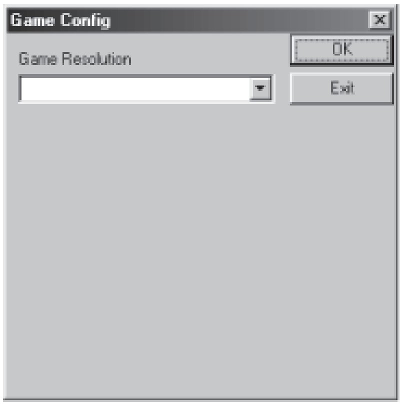


Figure 10.1: The Game Config dialog

In the dialog box procedure (ConfigDlgProc) we'll handle the messages WM_INITDIALOG and WM_COMMAND. Within WM_INITDIALOG we'll declare a long called mode, which will have a default value set to 0. We use this variable to increment through the different graphical modes the graphics adapter can support during the enumeration process.

The other variable we'll declare is called dm, which is of the DEVMODE data type. The DEVMODE type contains all the information we need to enumerate the values from the graphics adapter. We'll declare two new global variables called old_resolution and new_resolution, which will store the current resolution (default resolution) and the new one we'll switch to. We should save these values so we know which resolution to switch back to when we're done testing (playing) our game and so we can easily bring up details about the current resolution. A simple call to EnumDisplaySettings, specifying NULL for the display device (first parameter), mode variable for the second parameter, and the address of the old_resolution variable as the third parameter, will save the current display settings.

After saving the settings we create a while loop, which will loop while our function call to EnumDisplaySettings doesn't equal true. The EnumDisplaySettings function uses a reference to the display device, which we'll set to NULL for the default adapter, the mode number, and the address of the dm variable as the parameters to retrieve the current graphics settings. Inside the loop condition we'll place the dm.dmPelsWidth, dm.dmPelsHeight, and dm.dmBitsPerPel variables inside a string and add it to the combo box, provided the bits per pixel is greater than 15-bit color. This will give us a nicely built list of graphics modes that are supported by the hardware in a simple manner. In case you're wondering why we only want graphics resolutions above 15-bit, it's because 8-bit normally uses palettes to display colors, which means we have a list of colors we can use out of a possible 256 colors. In the day and age of hardware Transform & Lighting, shaders, and other graphical goodies, we can surely require a minimum of 16-bit color to make our game display properly without worrying if the average user can support it.

After inserting the data into the combo box we simply increase the mode number by one and repeat the process. When the function can't enumerate additional graphic modes from the adapter, the return value will return false, which will exit the loop. Once the loop has been exited we simply set the current combo box selection to the first item in the list to complete the WM_INITDIALOG Windows message code. When the user clicks the Exit button (IDEXIT) in the dialog box we'll simply exit the dialog box, returning the value 0 to indicate the user wants to exit the game. If the user clicks the OK button (IDOK), then we'll declare a variable to hold the currently selected item in the combo box. With the currently selected item we can now send CB_GETLBTEXT to get the resolution from the selected item and store it in a temporary string.

Once the data is put into the string we set the global new_resolution variable equal to the old_resolution variable to copy the default settings, then string scan the new width, height, and bits per pixel to our new_resolution variable. With the new_resolution variable now filled in, we can simply call the ChangeDisplaySettings function and specify the new_resolution variable as the display settings to change to and 0 as the second parameter, which indicates we want the function to change modes permanently. The ChangeDisplaySettings function has many different options that could be specified as the second parameter.

Many of the options will test or change the configuration of the computer, which is much more complicated than what we need. If we specify 0, the resolution will change without any special details. The downside to changing the resolution in this fashion is that if your game crashes your Windows resolution will be stuck with the settings from your game.

The final step is to exit the dialog, returning the value 1 to indicate the function was a success. In the base code we'll add the call to the newly created dialog box at the top, before we even register the class. Because OpenGL relies on the current desktop resolution as the main buffer, it's a good idea to change to the desired resolution after we register our window class and just before we create the window itself. If we change the resolution before we register the window class, the initialization of OpenGL will not work properly, which is why we place the dialog box between the two function calls. It is a good idea to check the return value from the dialog box call to see whether or not the value is equal to 0. In the event the value is equal to 0, we display some sort of message and then exit the game. If the return value does not equal 0, we can continue to load the game.

At the bottom of the main source file we'll restore our default resolution values by calling the ChangeDisplaySettings function, specifying the old_resolution variable as the display settings to change to and 0 once again so the system will change its default setting.

Chapter Example

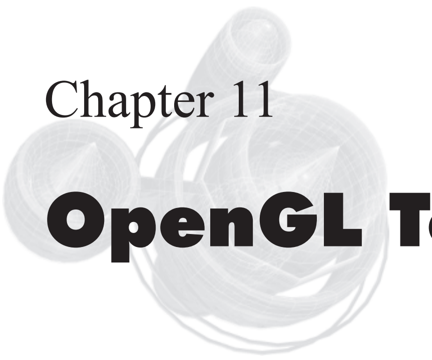
The following example code enumerates the graphics resolutions supported by the video hardware and waits for the user to select one. After the user clicks OK, the resolution will change to the desired mode with the game being full screen and with a black background. To exit the program we'll need to press Alt+F4, since we've not yet assigned a key to exit the game. The new base code with enumeration capabilities is provided below.

ex10_1.cpp

```
#include <windows.h>
#include <stdio.h>
#include "raster.h"

#include "resource.h"


HWND Window;
DEVMODE old_resolution;
DEVMODE new_resolution;


LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_DESTROY: PostQuitMessage(0); break;
   }
   return (DefWindowProc(hWnd, msg, wParam, lParam));
}


LRESULT CALLBACK ConfigDlgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case WM_INITDIALOG:
      {
         DEVMODE    dm;
         long       mode=0;

         EnumDisplaySettings (NULL, ENUM_CURRENT_SETTINGS, &old_resolution);
         while (EnumDisplaySettings (NULL, mode, &dm) != false)
         {
            char temp[500];
            sprintf (temp, "%ix%ix%i", dm.dmPelsWidth, dm.dmPelsHeight,
                     dm.dmBitsPerPel);
            if (dm.dmBitsPerPel > 15) SendDlgItemMessage (hWnd, IDC_RESOLUTION,
                     CB_ADDSTRING, 0, (LPARAM)temp);

            mode++;
         }
         SendDlgItemMessage (hWnd, IDC_RESOLUTION, CB_SETCURSEL, 0, 1);
      } break;
      case WM_COMMAND:
      {
         if (wParam == IDEXIT) EndDialog (hWnd, 0);
         else if (wParam == IDOK)
         {
            long    cursel = SendDlgItemMessage (hWnd, IDC_RESOLUTION,
                                                 CB_GETCURSEL, 0, 0);
            char    temp[500];


            SendDlgItemMessage (hWnd, IDC_RESOLUTION, CB_GETLBTEXT, cursel,
                                (LPARAM)(char *)temp);
            new_resolution = old_resolution;
            sscanf (temp, "%ix%ix%i", &new_resolution.dmPelsWidth,
                    &new_resolution.dmPelsHeight, &new_resolution.dmBitsPerPel);

            ChangeDisplaySettings (&new_resolution, 0);
            EndDialog (hWnd, 1);
         }
      } break;
   }
   return (0);
}


int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR lpCmdString,
                    int CmdShow)
{
   WNDCLASS   wc;
   MSG        msg;

   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
   wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance     = hInstance;
   wc.lpfnWndProc   = WndProc;
   wc.lpszClassName = "GE";
   wc.lpszMenuName  = NULL;
   wc.style         = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL, "Error: Cannot Register Class", "ERROR!", MB_OK);
      return (0);

   }


   if (!DialogBox (hInstance, MAKEINTRESOURCE(IDD_CONFIG), NULL,
                (DLGPROC)ConfigDlgProc))
   {
      MessageBox (NULL, "Thanks for running the game engine", NULL, MB_OK);
      return (0);
   }


   Window = CreateWindow("GE", "Game Engine", WS_POPUP | WS_VISIBLE, 0, 0,
         GetSystemMetrics(SM_CXSCREEN), GetSystemMetrics(SM_CYSCREEN), NULL,
         NULL, hInstance, NULL);
   if (Window == NULL)
   {
      MessageBox (NULL, "Error: Failed to Create Window", "ERROR!", MB_OK);
      return (0);
   }

   while (1)
   {
      if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
      {
         if (msg.message == WM_QUIT) break;
         TranslateMessage(&msg);
         DispatchMessage (&msg);
      }
   }

   ChangeDisplaySettings (&old_resolution, 0);

   return (1);
}
```

Conclusion

As you've probably noticed, the book is moving along more quickly now since we're not taking the time to discuss the basics and instead are talking at a more intermediate level about how to code our game. In this chapter we discussed the new base code for our game engine and wrote the code necessary to enumerate the graphics resolutions available with the default graphics adapter, just like many games/examples would do in that yucky DirectX SDK! In the next chapter we're going to get to the really fun stuff — texturing with OpenGL.



Chapter 11

OpenGL Texturing

The art of texture mapping significantly changed the gaming industry in the 1990s, first by transforming the bland, one-color walls to 8-bit masterpieces. As processors got faster and game developers found new and creative ways to optimize their code, consumers began experiencing games with irregular wall sizes and levels that were fully textured. In the mid-'90s several video card manufacturers, including 3DFX Interactive, released 3D accelerators, giving workstation-style rendering technologies to the masses at consumer's prices.

Along with the new video card technology, the manufacturers provided special 3D APIs such as GLide (3DFX's proprietary 3D API), OpenGL, and later Direct3D to interface with the new features. Unlike before, when programmers had to write code directly "to the metal," these higher-level 3D APIs provided many functions to create polygons and triangles, and light, shade, and of course texture them. In the past when you walked close to objects that were textured, you might have noticed that the texture blew up so big that the pixels became huge rectangles of color on the screen. Many 3D accelerators supported a texture filtering style called bilinear filtering, which would perform a calculation to average out the adjacent pixel colors, producing a texture that looked blurred instead of pixelated.

Although it was possible to do this before we had 3D accelerators, the time involved in calculating the bilinear filter was too substantial to implement it. The new line of video cards would calculate these values through the 3D API, allowing the developer to focus more time on other aspects of the game. Figure 11.1 shows two walls, one with bilinear filtering and one without.

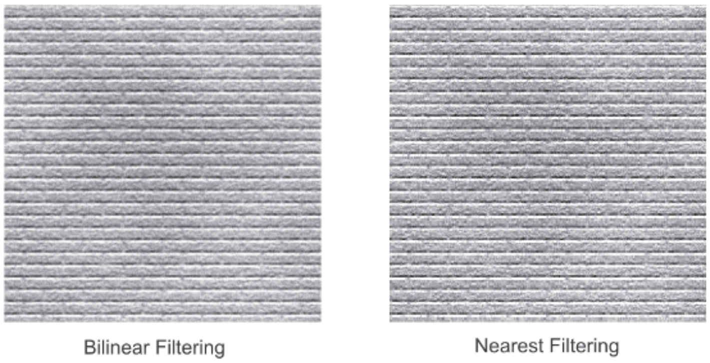


Figure 11.1: One wall with bilinear filtering and one without

As you can imagine, this new technology revolutionized the gaming business back in the day. One of the first games to implement 3D acceleration was GLQuake, which was a specially written version of id Software's Quake that supported OpenGL. The differences between the original version, which used 8-bit graphics, and the new 3D-accelerated 16-bit GL version of the game were substantial. No longer did we see major pixelation when we walked near the walls. Transparencies were now possible, and 16-bit color allowed programmers to use a wider palette to unleash their hell on the player! As time went by, 3D accelerators got more powerful, better features were added to the cards, and before long 32-bit color, texture compression, trilinear and anisotropic texture filterings, bumpmapping, and many other nifty features were all capable on 3D accelerators.

It's amazing to look back at games written in the early '90s and realize they were on the cutting edge, then fast-forward to today's technologies and see that the graphic gems we have today were all somehow inspired from those games made 14 years ago. In this chapter we'll be discussing how to initialize texturing in OpenGL, how to load a bitmap (BMP) file as a texture, and much much more.

How Texturing Works

Before we begin the coding process, we should discuss the steps involved in initializing, loading, and displaying textures in OpenGL. After initializing the OpenGL API we set the maximum number of textures for use in our game/application. Next we load the bitmap files we want to use as textures. We load the bitmaps after switching resolutions and initializing OpenGL so we don't make the user wait for the load only to find out his video card failed to initialize OpenGL. Could you imagine how annoyed you'd get if a game took two minutes to load, then finally displayed an error that it couldn't initialize OpenGL and must quit?

To put it bluntly, the average user would be pretty pissed off if he had to wait that long only to find out he can't play the game. This simple piece of info can make all the difference in that the user is only waiting a fraction of a second to see if the game can initialize to the desired resolution before loading the data. For simplicity we'll only load a couple of textures, but some games can literally load hundreds or even thousands, which could make a big difference in load time. Once the texture loading is complete, we set the default texturing settings, such as the filtering styles, and then we bind the bitmap RGB data to a texture number. If we don't specify the maximum number of textures before we attempt to bind the bitmap data to the texture numbers, the computer may not display the textures or could crash or behave in odd ways depending on how the video card manufacturer wrote the drivers. After binding the bitmap data to the texture number, we simply call built-in OpenGL functions to use the specified texture number when we want to texture a wall, which will, hopefully, transform a normally bland wall into a digital masterpiece.

Loading Bitmap Files

Before we can bind and use textures in OpenGL we must write the code to load a bitmap file. With all the different graphics formats available these days, it's hard to choose a specific format. In this book, we'll use the bitmap (BMP) format because it supports 24-bit color (which is the format we'll store our textures in), there is no proprietary compression involved, and the data can be easily read. Other popular formats for game texture use are Targa (TGA), JPEG (JPG/JPEG), and Graphics Interchange Format (GIF). Of course all graphics formats have advantages and disadvantages, but with the recent boom of software patents and companies protecting their intellectual property, you may want to consider using a format that has little to no IP claims. If you do use a format that has a copyrighted algorithm inside it, you may be required to pay a license fee to the owner of the IP. With this in mind, the following table displays the advantages and disadvantages to using each format.

Table 11.1: Graphics formats

	Advantages	Disadvantages
BMP	Supports 8-bit (w/palette) and 24-bit color	Very large file sizes (due to lack of compression)
GIF	Supports transparent color Decent compression rate	Only supports 8-bit (256) colors Contains a copyrighted compression algorithm (LZW compression scheme)

	Advantages	Disadvantages
JPG	Supports 24-bit color Variable compression settings (low-high compression)	File format is copyrighted
TGA	Supports 8-, 24-, and 32-bit color	Large file sizes (with no compression)

In addition to these standard graphics formats there are many others you could consider, such as Portable Network Graphics (PNG), Silicon Graphics Inc. (SGI), and Portable Pixel Map (PPM). There are literally hundreds of formats to choose from and each has its benefits and faults that can affect whether you choose to use it in a production game engine or in a simple texture demo. Obviously each file format structure is different, but the sheer simplicity of the bitmap format is one of the key advantages to using it over other formats. The bitmap format, depending on its size, has either two or three sections to it. The first section of a bitmap is always the header, which will store the important values of the format such as width, height, bits per pixel, compression data, etc. The next section depends specifically on the header's bits-per-pixel value. If the value is 24 (as in 24-bit color), then the pixel data loads next; if the value is 8 (as in 8-bit color), then a customized palette is loaded first, before the pixel data. Table 11.2 displays the main sections of 8-bit and 24-bit bitmaps.

Table 11.2: Bitmap sections

8-Bit Color	24-Bit Color
Header	Header
Palette	Data
Data	

The bitmap format does support other bits-per-pixel values in addition to 8 and 24; however, we're only concerned with 8-bit and 24-bit because they are the only formats that may be used in modern games. In this book, we'll deal specifically with 24-bit because we don't want to have to concern ourselves with palettes and converting indexes to RGB values.

Conveniently Microsoft provides the bitmap file format headers within the WINGDI.H header file. If you look at the header file, it's quite a big mess, but the structures BITMAPFILEHEADER and BITMAPINFOHEADER are the structures needed to load the bitmap file format. I could bore you for three or four paragraphs about defining our own bitmap header structure, but why write extra source code if it's not needed? If you are interested in the file format details and don't want to read through the header file, please visit the Sources section of this book for links about where to find file format info. The BITMAPFILEHEADER structure contains the file ID for the bitmap and the reserved variables, which are meant for future versions of the bitmap format. The BITMAPINFOHEADER structure contains important information such as the width, height, bits per pixel, and many other key pieces of data for loading and allocating memory for the bitmap data.

With this in mind, we'll create a new class called TEXTURE, which as the name implies will handle the texturing functionality. Inside the class we'll declare three public variables. The first variable is called file_header and is of the BITMAPFILEHEADER type. As stated earlier, this variable doesn't contain any essential data. The next variable, which is of the BITMAPINFOHEADER type, is called info_header and contains the important data for loading the bitmap RGB data into memory. The reason we're putting the bitmap headers in the public section of the class is that we want to be able to easily access the width, height, and bits-per-pixel variables without having to create any new methods in the class. It is possible to create a more structured class if you'd like, but keep in mind we'll limit access to the width, height, bits per pixel, and RGB data. I tried to keep the code as simple as possible, which is why everything is public.

After declaring the header variable, we'll declare another variable called data, which is a pointer to an unsigned character. When we load the physical RGB values from the bitmap file, we'll allocate the appropriate memory to the data variable and place the data into it. This of course means the variable should be placed in the public section of the class to ensure we can access the data from the main game code when we want to bind the texture to a number. Following the declaration of the two variables we'll create a default constructor for the class. Inside the default constructor we'll set the data variable to NULL and use the memset function to fill the bitmap header with 0s so there's no confusion over whether the data or header variables have valid data inside them.

Once we've set the variables to their default values, we'll create a new method called Load, which will have a pointer to a character as the single parameter. The parameter, named filename, will contain the bitmap filename to load. The method should have a bool return type to indicate whether the method failed or succeeded. Within the Load function we'll create a local pointer to a FILE type called fp, which is our file pointer. We'll also declare a variable called image_size, which is of the long type. This variable will be used to calculate the size of the image data once we load the bitmap header. Please keep in mind that when the designers of the bitmap format created it, for some really odd reason they stored the bitmap RGB data upside down, which means bitmap data is stored in BGR format. This of course isn't the format we want the data to be stored in, but we'll discuss a method of using BGR data for a texture.

With the local variables declared, we can begin the opening process. To begin this process we'll use the C function fopen, passing the filename variable as the first parameter and using the "rb" access mode as the second to allow us to read the data in binary. Obviously the return value from fopen will be our new filestream, which we'll store in the fp variable. If the returned filestream is NULL (file not found), then we'll exit the method by returning false immediately to avoid any crashes that may occur from trying to read data that isn't there.

As mentioned earlier in this section, we have two types of headers to read from the bitmap file. The first header we'll read is the file_header variable, then we'll read the info_header variable. If the info_header.biBitCount value doesn't equal 24 (for 24-bit), then we're not interested in using this lesser being of a file! Since the file isn't the type we wanted we should fill the header with 0s to ensure the data isn't mistakenly used and exit with an error. If the bits-per-pixel value is 24, our desired format, then we'll allocate memory to the data array variable. The amount of memory to allocate is the width times the height times 3 (because the bitmap has individual 8-bit red, green, and blue values) plus one extra record to be on the safe side. With the memory allocated for the data array, we simply read the data from the file using the C function fread and specifying the number of indexes as the size of the array (width * height * 3). Once the loop completes, we return a true value to indicate that the texture loading successfully completed.

When trying to load bitmap files, please be aware that there are many different versions and formats available to you. Although it may say "bitmap," it may be another format altogether. We're using the Windows version of the bitmap format, which is the common standard among Windows paint/graphics software; however, some programs use the OS/2 bitmap format, which is different from the Windows format. If you intend on using different graphics applications, it may be a good idea to find out which format is supported so you can adjust your application byte alignment settings and/or bitmap headers appropriately. Since we're using Microsoft's structures we don't need to adjust anything to have proper compatibility with the standard Windows bitmap format. To finish the task, click the OK button, then rebuild the entire workspace. Note that this is a mandatory practice after changing a compiler setting.

```
bool TEXTURE::Load(char *filename)
{
   FILE       *fp;
   long       image_size;

   Release();

   fp = fopen(filename, "rb");
   if (fp == NULL) return (false);
```

```
        fread (&file_header, sizeof(BITMAPFILEHEADER), 1, fp);
        fread (&info_header,sizeof(BITMAPINFOHEADER), 1, fp);
        if (info_header.biBitCount != 24)
        {
           memset (&file_header, 0, sizeof(BITMAPFILEHEADER));
           memset (&info_header, 0, sizeof(BITMAPINFOHEADER));
           fclose (fp);
           return (false);
        }

        image_size    = info_header.biWidth * info_header.biHeight * 3;
        data          = new unsigned char[image_size+1];
        fread (data, image_size, 1, fp);

     fclose (fp);


     return (true);
  }
```

Releasing Bitmap Data

Once we've loaded the bitmaps and bound the data to a texture number, there's no need to keep the bitmap data loaded in memory that could be used for other more important things like storing a level or audio files, which normally take up a lot of space. In our TEXTURE class we'll create a new method called Release, which as the name implies will release any allocated memory in the data file of the TEXTURE class. The new method does not have a return type nor does it have any parameters. Since our constructor sets the data pointer to NULL, the Release method will simply release the memory allocated to data if data isn't NULL and set the pointer to NULL. After setting the pointer to NULL, we'll fill both headers with 0s to ensure that everything was blanked.

It is a good idea to add a call to the Release method in the Load method before the code that opens the file to ensure the texture is blank. If we don't add the call to the Load function we could use the same texture variable to load multiple textures, then face a possible crash when the data pointer attempts to allocate memory to an already allocated pointer. It may seem farfetched, but it can happen, and by adding this simple step you can avoid a potential issue. The source code to release our bitmap data is complete and is displayed below.

```
void TEXTURE::Release()
{
   if (data != NULL)
   {
      delete [] data;
      data = NULL;
      memset (&file_header, 0, sizeof(file_header));
```

```
        memset (&info_header, 0, sizeof(info_header));
    }
}
```

Using Textures

With the code for loading and releasing bitmap data complete we can begin writing the OpenGL texturing code. When using texturing in OpenGL, the first thing you should always do is set the maximum number of OpenGL texture names. In case you're wondering, a texture name is a reference number we use to access the texture data in OpenGL. The texture names start at 1 (because 0 is a reserved name) and increment by one. In OpenGL we define an array of the GLint type to store the reference texture IDs for calling the texture data. After initializing OpenGL, we set the number of textures to reference in the array using the function glGenTextures. The first parameter in the function is the number of textures we can reference and the second parameter is the texture name array, which will store the reference names. This is an essential part of the texturing process because we cannot access and texture data or bind any texture data without specifying the number of textures. In some implementations of OpenGL, the computer even crashes if you attempt to access a texture name without first setting the number of texture names. For our game, we'll create a global variable of the GLint type called TextureName, which uses the constant MAX_TEXTURE_NAMES as its array size. The MAX_TEXTURE_NAMES constant will have a value of 5, giving us five texture names to work with. When using textures, please keep in mind that texture ID 0 is reserved and cannot have textures bound to it. If you attempt to bind a texture to 0, no texture will be displayed when you render an object.

Once the global variable has been declared we'll declare a new function called SetGLDefaults, which as the name implies will set our OpenGL defaults for rendering. It is a good idea to declare a function like this at some point to ensure that the default rendering information is set so we can display the graphics in the desired manner. Inside the function we'll set the default clear (background) color to black (0,0,0,1) using the function glClearColor, which will erase the screen and paint it black when we call the glClear function later on. After setting the clear color, we'll turn on depth buffering (Z-buffer) using the function glEnable and specifying GL_DEPTH_TEST. The depth buffer (Z-buffer) will automatically sort the objects being drawn from back to front, which is something of a luxury compared to the old DOS days when programmers would have to figure this stuff out manually. This is an essential function for us because in most cases the video card can calculate the objects from back to front much quicker than we can through regular algorithms. In some cases you may be able to sort the objects from back to front but this subject is beyond the scope of this book. If we didn't enable depth buffering, we could run into the issue that an object that's far away from the user is actually drawn in front of a wall closer to the user. If the objects aren't drawn from back to front, then they would be drawn in the order of their creation, which could be anywhere on the screen at any time. The depth buffer fixes this issue for us in a simple manner.

After enabling the depth buffer, we'll disable backface culling. Although the name sounds funny, backface culling will quickly determine which triangles are visible and which ones aren't based on your position relative to the triangle and the current cull style. There are essentially three styles available for backface culling: front, back and two-sided (which is technically disabling cull facing). When the three points that make up the triangle are in a clockwise order, they are using the cull style front. When the three points that make up a triangle are defined in a counterclockwise order, they are using the back cull style. When the application renders the triangles, it checks the cull style to interpret how the triangle should be drawn. If the user is directly in front of a triangle with the cull style of front, the triangle will be drawn. If the style is back, the object will be culled out of view. When backface culling is disabled, the triangles are drawn in the style selected, which essentially makes them double-sided walls. Many games use this technique to optimize level design because they can specify which ways walls point, therefore having them culled out of the scene when the user is behind them in another room. Since our map editor is very simple and only allows double-sided walls, we'll disable this function. If you are interested in seeing the true benefit of backface culling, you may want to add a wall facing feature to the map format, allowing you to easily specify which way the walls are pointing. Of course this is beyond the scope of this book. Figure 11.2 shows the two different styles of cull facing and how they work.

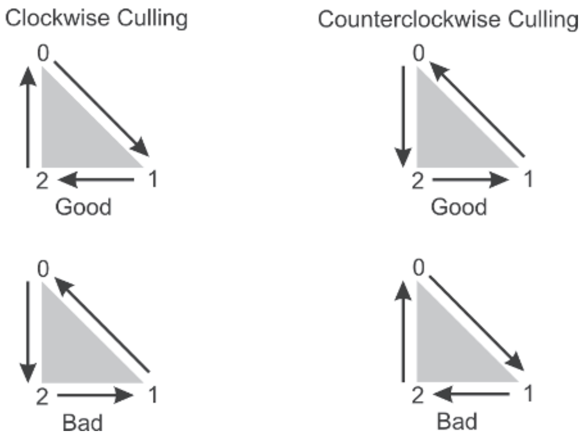


Figure 11.2: Two different styles of culling plus two instances of when they'd be drawn and culled out of the scene

The last rendering setting we must enable is 2D texturing itself, which is done by calling glEnable and specifying GL_TEXTURE_2D as the single parameter. This will enable 2D texturing in our applications. Similarly if you were interested in using 1D or 3D textures in your application you would specify GL_TEXTURE_1D or GL_TEXTURE_3D, respectively. The differences between the three texture types are fairly obvious by their naming. A 1D texture only specifies the width or height of a bitmap, whereas a 2D texture contains both the width and the height of the bitmap. To use 3D textures in hardware you'll require fairly sophisticated hardware, but they offer width, height, and depth for a texture, although there aren't many programs that support 3D textures as of yet and they do require a lot of texture memory (a 24-bit 64x64x64 3D texture requires 786 KB of memory).

With the default rendering settings now set, we can define the number of texture names available for the application. As mentioned previously, OpenGL requires you to set the number of texture names available in the program. To do this, we call the OpenGL function glGenTextures, specifying the number of textures as the first parameter (MAX_TEXTURE_NAMES constant) and the texture name array as the second parameter. The five texture references will be bound to the indexes in TextureName, allowing us to easily reference the texture data by using the different indexes of the TextureName array.

At this point we'll create a new function called LoadTexture, which will load a specified bitmap file as a texture and set the default texturing settings for it. Because of the functionality we want, LoadTexture will accept five parameters. The first parameter is a pointer to a character called filename. The next parameter is a long called texture_id, which as the name implies is the texture reference ID we want to bind the texture data to. The third and forth parameters are of the long data type and are called mag_filter and min_filter. These parameters set the magnification and minification texturing states. The final parameter is another long that contains the wrapping type for the texture.

Inside the function we'll declare a variable of the TEXTURE type and call the Load method, specifying the filename parameter as the filename for the method. If the return value from the method is false, we'll exit the function; otherwise we'll begin the texturing process. The first thing we should always do is set which texture is currently active by specifying the GL function glBindTexture. The first parameter is the dimension of the texture, which in our case is 2D so we specify GL_TEXTURE_2D. The second parameter is the texture name, which we reference by using the texture_id parameter as the index for the TextureName array. Once the texture is bound, we can define the texture's RGB data, width, height, and format by specifying the bitmap data we loaded. Using the GL function glTexImage2D, we define the texture's attributes. The first parameter must always be GL_TEXTURE_2D, as defined in the OpenGL SDK. The second parameter is the level of detail of the texture. This is used to define multiple levels of detail for textures, which we'll discuss later in this chapter. For the time being we'll specify 0, which states that the specified texture is the base (initial) level. The third parameter defines the internal pixel format for the data, which in our case is in RGB format, so we'll specify GL_RGB. In some cases you may want to specify individual red, green, blue, and even alpha channels here, which is possible by simply using the appropriate value. The next two parameters are the width and height of the texture, which can be found using the texture's header width/height values. After specifying the width and height, we'll set whether or not the texture has a border. In our case we'll set it to 0, stating there's no border. Next we specify the format of the texture data. Earlier in the chapter when we loaded the bitmap data from the file, I mentioned that the data is encoded upside down, which changes the format from RGB to BGR. Rather than mirror the texture data vertically, we can simply use the constant GL_BGR_EXT, which specifies the data as BGR instead of RGB. Originally the GL_BGR_EXT constant was an extension, but has since been adopted into the core API of OpenGL. After setting the texture format, we specify the data type of the texture data. In most cases the data type will be GL_UNSIGNED_BYTE, because most graphics file formats use unsigned characters to store their individual RGB data and/or indexed color data. Finally, the last parameter is the texture BGR data located in the texture class.

In the next sections we'll discuss how to apply the different texture settings available in OpenGL and how they work.

Texture Filtering

Texture filtering defines how the texture is drawn when it's scaled down (minified) and scaled up (magnified). OpenGL supports two styles of texture filtering: nearest and bilinear. The nearest filtering style draws textured objects without applying any type of antialiasing or blurring to the texture when it's scaled from its original size. This results in pixelation of the texture, which means the texture on the object interprets each pixel of the texture as a mini-rectangle on the screen when scaled. When bilinear filtering is enabled, the textured object will have a smooth blur applied to the texture, eliminated much of the pixelation seen in the nearest filtering style.

Many games shipping today use bilinear filtering (or even better filtering techniques) to make their textures look nice. The downside to bilinear filtering is that it takes more calculations to perform than nearest filtering; however, all 3D-accelerated video cards shipping for the past four or more years have supported bilinear filtering without major slowdowns. The min_filter and mag_filter parameters of the LoadTexture function will control how the texture filtering is applied when it's minified and magnified.

Figure 11.3 displays the two different styles of texture filtering compared to the original image.

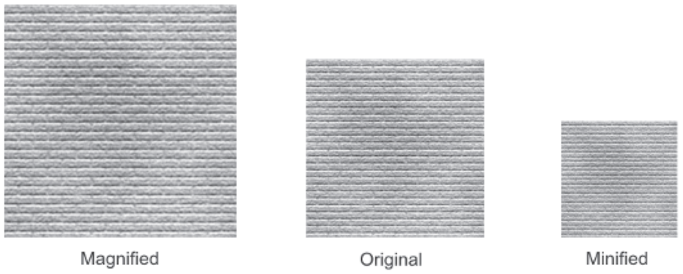


Figure 11.3: The original texture as well as two texture styles in scaled up/down form

Using the function glTexParameteri we can specify the texture filters for the current texture. The first parameter of the function is the texture dimension (1D, 2D, or 3D). Next we specify a texture parameter such as GL_TEXTURE_MAG_FILTER for magnification or GL_TEXTURE_MIN_FILTER for minification. Finally we specify the texture setting parameter, such as the parameter variable min_filter or mag_filter to specify the texture filtering style.

Texture Wrapping

After we set the texture filters we can set the texture wrapping properties. The texture wrapping properties affect the way the texture is tiled on the screen. Using the function glTexParameteri, we once again specify the texture dimension (1D, 2D, or 3D) as the first parameter. Rather than specify the filtering type, we'll specify the wrapping axis we're going to set. There are two options available in OpenGL v1.1: GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T. The S coordinate specifies the first texture axis wrapping style, which normally will be the horizontal direction with walls in front of you. The T coordinate specifies the second texture axis style, which normally will be the vertical direction with walls in front of you. Figure 11.4 displays the S and T coordinates for wrapping textures.

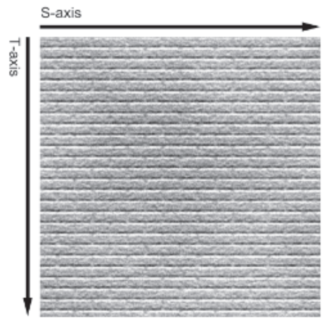


Figure 11.4: The S and T axes

At this point you may be wondering what type of wrapping options you can specify for textures. In OpenGL version 1.1, which we're using for now, we have two styles of wrapping available. The first texture wrapping type allows the texture to repeat itself (GL_REPEAT), which is commonly used when we want to tile a floor or wall in a game. The other wrapping type makes the texture clamp (GL_CLAMP), which means the image will only be drawn once, regardless of the number of times you specify it to tile. In the case of our game, we'll set both the S and T coordinate wrapping values to our wrap_type parameter. Figure 11.5 displays the two different wrapping styles available in OpenGL.

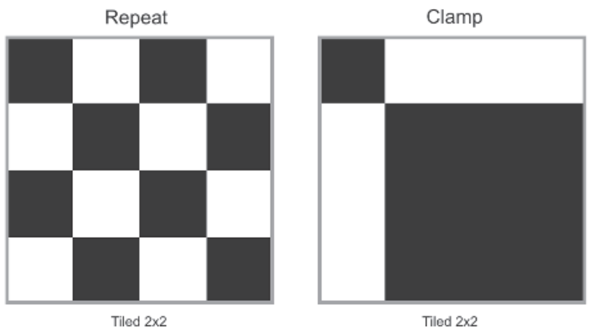


Figure 11.5: Texture wrapping options

Texture Environment Settings

The texture environment function (glTexEnv) allows the programmer to specify how the texture is to be drawn. When using this function, the first parameter must always be GL_TEXTURE_ENV to allow for future functionality. If we're interested in changing the texture environment settings, we must set the second parameter to GL_TEXTURE_ENV_MODE. The final parameter should be one of these four texture settings: GL_DECAL, GL_REPLACE, GL_MODULATE, or GL_BLEND. Each parameter will manipulate and blend the texture data differently to achieve the desired look. In the case of our game engine, we'll use the GL_DECAL and GL_MODULATE parameters because they provide the functionality needed for the features of our game. For regular generic textures we'll use the GL_MODULATE parameter because it draws the texture normally and without any special functionality. The GL_DECAL parameter allows us to specify a texture with transparent areas, which is a great feature to have available for special effects, projectiles, and other things.

Once we've set the texture environment settings, we should release the memory allocated in the texture variable to conserve RAM. In case you're wondering, once we've set the texture data by calling glTexImage, all texture data (regardless of its format) is stored in the video RAM and doesn't need to be stored in your regular RAM anymore, which is why we're releasing the data. We've now completed the code to load, bind, and set the default values for a texture. Remember that OpenGL requires the texture dimensions to be a power of two for it to correctly load and work, or otherwise your system won't display the texture or may even crash. The code for the LoadTexture function is provided below.

```
void LoadTexture(char *filename, long texture_id, long mag_filter,
                 long min_filter, long wrap_type)
{
   TEXTURE texture;

   if (!texture.Load (filename)) return;
   glBindTexture (GL_TEXTURE_2D, TextureName[texture_id]);
   glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, texture.info_header.biWidth,
            texture.info_header.biHeight, 0, GL_BGR_EXT, GL_UNSIGNED_BYTE,
            texture.data);
   gluBuild2DMipmaps (GL_TEXTURE_2D, GL_RGB, texture.info_header.biWidth,
            texture.info_header.biHeight, GL_BGR_EXT, GL_UNSIGNED_BYTE,
            texture.data);

   glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, mag_filter);
   glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, min_filter);
   glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap_type);
   glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap_type);

   glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   texture.Release();
}
```

Transparent Colors

Perhaps one of the best-kept secrets in OpenGL is user-defined transparent colors within a texture. In other SDKs like DirectX, the programmer can define which colors are transparent when loading the texture, therefore making certain colors invisible on the screen when the texture is drawn. This is most commonly seen in heads-up displays (HUDs), custom fonts, and the technique known as billboarding. By changing the blending options before we render the object in OpenGL, we can use a single transparent color, which is the color black (RGB 0,0,0). The problem with this method is that you are limited strictly to the color black. This can severely limit the number of uses for transparent colors within your game, because black is a commonly used color.

Alternatively, we can write a new function based on the previous texture loading function that will allow us to specify which color should be transparent. This will give us the freedom we want and/or need to use any color as a transparent color. The only downside to using this method of transparency is that it requires more memory to be allocated for the specific texture since an alpha channel must be specified for each texel of the texture. With many video cards today shipping with 128 MB and more of memory, the increase in memory allocation isn't that big of a deal. Using the LoadTexture function we previously wrote as the base code for the new function LoadTransparentTexture, we'll add three new parameters (called trans_r, trans_g, and trans_b) of the unsigned char type to the end of the function. Instead of storing the transparent texture in BGR format, we'll be storing it in BGRA format, which allows us to specify each pixel's alpha channel. When the BGR format is specified, the alpha channel is set to solid (1.0 or 255, depending on the data type) by default, where the alpha is completely user defined in BGRA. To accommodate this addition to the texture components we'll declare a new local variable called alpha_texture, which is a pointer of an unsigned char. This variable will store the original texture data with the alpha channel.

After loading the texture, we'll allocate memory to the alpha_texture pointer for the dimensions of the bitmap with four planes (width * height * 4) to accommodate the R, G, B, and A data being stored. Next we simply loop from 0 through the dimensions of the bitmap and copy each B, G, and R value to the new array. If the B, G, and R values are equal to the transparent B, G, and R values, then we'll set the fourth pixel index in the alpha_texture array to 0, indicating the texel should be transparent. Otherwise, the fourth index for the pixel should be set to 255 to indicate the texel should be solid. Since we've got an alpha channel as well as the standard BGR data, the texture data will use the internal format GL_RGBA, but we'll use the format GL_BGRA_EXT when we supply the actual texture data when using the function glTexImage2D. To complete the function, the final addition we need to make is to the bottom of the function, where we'll release the memory allocated for alpha_texture as well as memory for the original texture loaded. When adding bitmaps with transparent colors, it's a good idea to use the nearest filtering type so the other texture values won't be used when drawing the pixel information. In some cases when using the linear type with transparent textures, you may notice that some lines on the outside won't be transparent because of the calculations performed. The source code for the new transparent texture loading code is provided below.

```
void LoadTransparentTexture(char *filename, long texture_id, long mag_filter,
        long min_filter, long wrap_type, unsigned char trans_r, unsigned char
        trans_g, unsigned char trans_b)
{
   TEXTURE          texture;
   unsigned char    *alpha_texture=NULL;
   long             texel;

   if (!texture.Load (filename)) return;
```

```
    alpha_texture = new unsigned char[texture.info_header.biWidth*
              texture.info_header.biHeight*4+1];
    for (texel = 0; texel < texture.info_header.biWidth*texture.info_
              header.biHeight; texel++)
    {
       alpha_texture[texel*4] = texture.data[texel*3];
       alpha_texture[texel*4+1]= texture.data[texel*3+1];
       alpha_texture[texel*4+2]= texture.data[texel*3+2];

       if (texture.data[texel*3] == trans_b &&
          texture.data[texel*3+1]== trans_g &&
          texture.data[texel*3+2]== trans_r) alpha_texture[texel*4+3] = 0;
       else alpha_texture[texel*4+3] = 255;
    }

    glBindTexture (GL_TEXTURE_2D, TextureName[texture_id]);
    glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, texture.info_header.biWidth,
              texture.info_header.biHeight, 0, GL_BGRA_EXT, GL_UNSIGNED_BYTE,
              alpha_texture);

    gluBuild2DMipmaps (GL_TEXTURE_2D, GL_RGBA, texture.info_header.biWidth,
              texture.info_header.biHeight, GL_BGRA_EXT, GL_UNSIGNED_BYTE,
              alpha_texture);

    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, mag_filter);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, min_filter);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap_type);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap_type);
    glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    delete [] alpha_texture;
    alpha_texture = NULL;
 texture.Release();
```

Like the first version of transparent coloring I discussed, this version requires the blending function to be enabled. To enable the blending function we'll add a new line of code to the SetGLDefaults function that will enable blending (GL_BLEND) in OpenGL. Each time we want to render any transparent bitmaps (or to be on the safe side), we simply set the blending function to GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, which will make the transparent areas transparent. If we don't specify these values, the transparent area will be blended differently and won't be interpreted as transparent.

Using Mipmaps

In the previous section we discussed the two different filtering techniques that OpenGL supports natively in its API. In addition to nearest and linear, there are several other texture filtering options available when we use mipmapping in our game engine. The term *mipmapping* is used to describe a set of scaled textures that represent an original texture. For instance, when we load texture file ../media/hud.bmp, this is the base level, sometimes referred to as the first level (level 0), of the texture. When mipmapping is introduced, we would take that originally loaded texture (say 256x256) and generate an image half the size (now 128x128), then create another texture half its size (now 64x64), and continue the process until the texture is of 1x1 dimensions. If the texture dimension isn't square, the mipmaps should be generated until both sides are equal to 1x1. Figure 11.6 displays two different textures and their accompanying mipmaps.

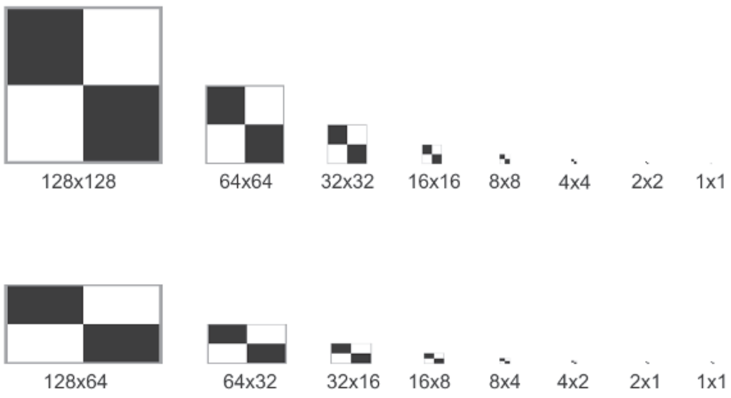


Figure 11.6: Two textures and their accompanying mipmaps

As a user walks through a level we've created, the video card is selecting different textures based on the texture size and distance. After selecting the specific mipmap to use, the video card will then perform more calculations to the texture to scale, rotate, and adjust where it needs to in order to draw it properly. If the video card has to scale an image 256x256 down to a resolution of 4x4, there's going to be an obvious performance difference when the card automatically chooses to use the 4x4 mipmap. In some platforms you can see a noticeable gain in performance when using mipmapping because the video card is optimized well for selecting the appropriate mipmap. When using mipmaps, we can specify four new texture filterings for the minification of the texture: GL_LINEAR_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, and GL_NEAREST_MIPMAP_NEAREST. Each filtering offers advantages and disadvantages. If you're interested in the best quality, you'll want to specify GL_LINEAR_MIPMAP_LINEAR, although the computation needed for its filtering is far higher than others and may result in slower rendering speeds. On the other hand, if you're looking for low-quality filtering but fast rendering speeds, you may want to consider using GL_NEAREST_MIPMAP_NEAREST. If you're interested in the different looks and speeds of each type of filtering, I encourage you to test each filtering option when you load the textures.

To build the different mipmap levels we can either calculate each level ourselves or we can use the GLU Library function gluBuild2DMipmaps, which will calculate each level automatically based on level 0's texture. When we call the function, we specify the texture target (GL_TEXTURE_2D), number of colors (GL_RGB/GL_RGBA), width (texture.info_header.biWidth), height (texture.info_header.biHeight), format (GL_BGR_EXT/GL_BGRA), data type (GL_UNSIGNED_BYTE), and finally the data (texture.data). You don't need to enable anything beyond the texture target (GL_TEXTURE_2D). Once gluBuild2DMipmaps has been specified, the video card will do the rest itself. If we were using 1D textures, we could alternatively use the function gluBuild1DMipmaps. To update our texture functions we'll add the call to the gluBuild2DMipmaps function right below the call to the glTexImage2D function to calculate the mipmaps after we initially bind the texture data.

Bitblitting in OpenGL

When the designers of OpenGL originally created the API, they added several functions for displaying 2D graphics on the screen. Unfortunately the native OpenGL function glDrawPixels is horrendously slow when drawing bitmap data to the screen. Some platforms and hardware vendors support the glDrawPixels function better than others, but in most cases there's still a considerable slowdown, which is undesirable in high-performance programming. Obviously we'd like to lose video processing power on something worthy of it rather than drawing a simple bitmap to a screen. Another way to draw 2D bitmaps to the screen besides using the built-in OpenGL functions is to change the projection of the screen to fake a 2D canvas, then draw the bitmap as an OpenGL quad and texture map the quad to appear correctly.

We'll use the second method to draw 2D bitmaps (or in our case textures) to the screen because it's fully 3D accelerated since we're using quads to draw it. This will minimize the number of frames lost when drawing things to the screen. The only limitations that we encounter is that the bitmap's dimensions must be a power of two, just like any regular texture would have. To begin the coding process we'll declare a new function called glBitBlt. The function will have five parameters that must be filled in for it to be used. All the parameters of the function are of the long type, because they are all integer data. The first parameter is called texture_id and is the reference number for the TextureName we want to display on the screen.

The second and third parameters are called start_x and start_y and they contain the starting coordinates for the bitmap. The final two parameters are the stretched width and height values for displaying the texture on the screen.

Inside the function, the first thing we'll do is bind the active texture we want by using the glBindTexture function, specifying the GL texture dimension (GL_TEXTURE_2D), and specifying the texture name reference using the texture_id parameter as the index. The appropriate texture is now ready to be used! Before we begin building textured quads, we must alter the reset projection matrices, set the new orthographic projection to emulate a 640x480 display screen, draw our quad, then restore everything back to the original settings. Although it sounds like a lot of work, it can be done in a fairly small amount of code.

As mentioned previously, the first thing we must do is reset the projection matrix. If we didn't reset the projection matrix, the displayed results may not be what was originally desired. The projection matrix can drastically alter the way data in OpenGL is drawn because it adds depth to the screen among other things. To reset the projection matrix the first thing we do is use the glMatrixMode function and specify the matrix (in our case GL_PROJECTION). We'll back up the current matrix by calling glPushMatrix, then reset the projection by calling glLoadIdentity, and switch back to the model view matrix. After switching back to the model view matrix, we'll back up the current matrices again, then set the orthographic projection using the settings 0, 639, 479, 0, 0, 1. As mentioned in Chapter 2, the glOrtho command sets the width, height, and depth of the screen like a huge cube. When we use the parameters specified above, we're setting the width to be from 0 to 639, the height to be from 0 to 479, and the depth to be from 0 to 1.

In case you're wondering, the 479 is placed as the first Y coordinate parameter because if we set it to 0, 479 the data would be drawn upside down at the bottom of the screen because of the orientation of the y-axis. The x- and z-axes don't have this issue though. Once the ortho has been set, we now have a 640x480 screen in OpenGL on which to draw textures. The width and height of the ortho can be changed to whatever resolution you'd like, but it's a good idea to keep it as a hardcoded number to make it easier to draw items on the screen. Even if the resolution is 1024x768, the screen ortho will automatically stretch the target texture across the full width of the screen when you bitblit an image to the edge of the ortho. Figure 11.7 shows how the textures are drawn on the screen.

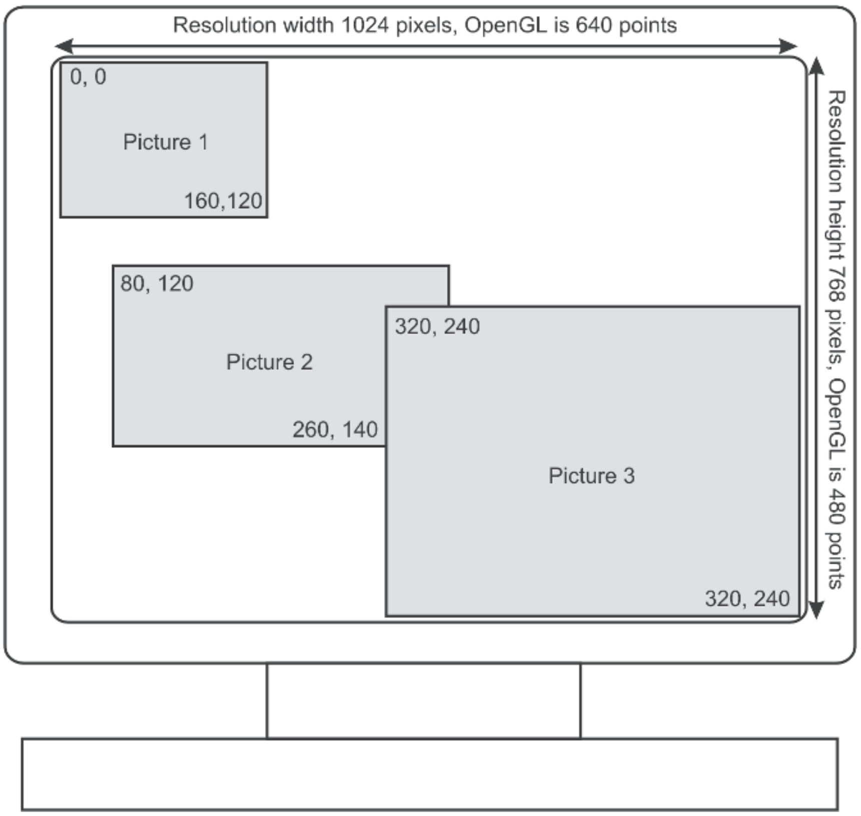

Resolution width 1024 pixels, OpenGL is 640 points
Resolution height 768 pixels, OpenGL is 480 points
0, 0
Picture 1
160,120
80, 120
Picture 2
260, 140
320, 240
Picture 3
320, 240

Figure 11.7: The width, height, and relation to the ortho

With the ortho now specified, we simply change the color to white to ensure the texture won't be affected by any previous colors being set, and begin drawing our quads. Before each point in the quad is drawn, we must specify the texture coordinate of the point. A *texture coordinate* is the location in the texture the point is supposed to represent. For instance, if the texture coordinate is 0,1, the point is located at the top-left corner of the texture. If the texture coordinate is 1,1, the point is 100% of the width of the texture but the height of the point is located at the top of the texture. If the texture coordinate is 2,3, the texture would be tiled two times on the width and three times on the height. Of course this assumes the texture wrapping is set to repeat. If the texture wrapping were set to clamp, then the object would only have a texture coordinate of 1,1 and the last pixel in each direction would be stretched the rest of the distance.

The texture coordinates are normally (but not always) stored as floating-point variables so you can specify 0.5,0.75, which would locate the texture coordinate in the middle of the texture horizontally and three-quarters of the way down vertically. To texture an object properly, each point in the object needs a texture coordinate so the map can be applied to it. Each coordinate must be specified before we specify the vertex. Figure 11.8 illustrates how to texture map a triangle and a quad and their respective texture coordinates for each point.

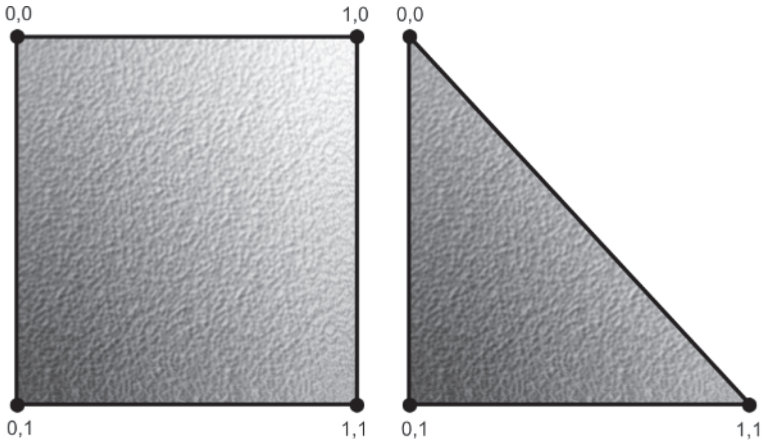


Figure 11.8: Texture coordinates for a quad and a triangle

In the case of our bitblitting routine, we'll be drawing a quad that's square and displays the entire texture so we can loop from the top-left corner (0,1) to top-right (1,1) to bottom-right (1,0), then finally to the bottom-left (0,0). After each texture coordinate we specify the vertices, which follow the same path of top-left (start_x, start_y), top-right (start_x + width, start_y), bottom-right (start_x+width, start_y+height), then finally bottom-left (start_x, start_y+height). Once the texture coordinates and vertex data have been specified we end the drawing process, restore the model view matrices (using glPopMatrix), switch to the projection matrix and restore it (once again using glPopMatrix), then switch back to the model view matrix, and exit the function. When the function exits, the matrices are back to the way they were when we entered the function in the first place. The source code for the function is written below for you.

```
void glBitBlt(long texture_id, long start_x, long start_y, long width,
       long height)
{
   glBindTexture (GL_TEXTURE_2D, TextureName[texture_id]);

   glMatrixMode (GL_PROJECTION);
      glPushMatrix();
      glLoadIdentity();
   glMatrixMode (GL_MODELVIEW);
   glPushMatrix();
   glOrtho (0,639, 479,0, 0,1);

   glColor3f (1.0f, 1.0f, 1.0f);
```

```
    glBegin (GL_QUADS);
       glTexCoord2f (0.0f, 1.0f);
       glVertex2i (start_x, start_y);

       glTexCoord2f (1.0f, 1.0f);
       glVertex2i (start_x+width, start_y);

       glTexCoord2f (1.0f, 0.0f);
       glVertex2i (start_x+width, start_y+height);

       glTexCoord2f (0.0f, 0.0f);
       glVertex2i (start_x, start_y+height);
    glEnd();

    glPopMatrix();
    glMatrixMode (GL_PROJECTION);
       glPopMatrix();
    glMatrixMode (GL_MODELVIEW);
}
```

Using Texture Lists

Now that we've discussed how to load bitmaps, bind them as textures, and draw them to the screen, we can apply this knowledge to creating the chapter example. The chapter example will load several textures from a texture list and use our glBitBlt function to display the HUD texture onto the screen with the transparent color enabled. To begin the coding process we'll declare a global constant called MAX_TEXTURE_FILES with the value 1. As the name implies, this constant will be used to control the number of texture files we'll load. After declaring the new constant, we'll declare a new structure called TEXTURE_FILE, which will store the filename as a string, and a Boolean value (called is_transparent) to determine whether the texture should be loaded regularly or using the transparent texture loading code. We'll also store the texture filtering information (GL_NEAREST or GL_LINEAR) in the variable texture_filter, which is of the long data type. The final variable in the structure is a long called wrap_type, which stores either GL_CLAMP or GL_REPEAT. Using the TEXTURE_FILE structure, we'll create a new global pointer array (called texture_file) that will be initialized with a single record of "hud.bmp" and true.

As we add more textures to future examples we simply add more entries to the texture_file variable and alter MAX_TEXTURE_FILES as needed. This provides us with a simple manner for loading textures, in addition to a way to easily bind textures to add custom textures later. With the texture_file variable globally declared we can move to the SetGLDefaults function to add the code to physically load the texture data. Within the function we'll declare a string (called texture_path) to store the path to the texture files and another string (called filename) to store the filename (including the path) of the texture. It is better to include the files without paths in the texture list so we don't have to worry about trimming the strings later to get just the filename when trying to find the proper texture for objects.

After declaring the variables, we'll create a loop at the bottom of the function that loops from 0 to the value of MAX_TEXTURE_FILES. Within the loop we store the path/filename of the current texture file into the string. Based on the texture_files is_transparent value (true or false), we'll load the appropriate texture loading function, keeping in mind that when we specify the texture number to load, we must specify the current loop value plus one (to avoid texture 0) and the texture filter value for each texture.

To finish the example we'll create a function called Render, and add the code necessary to render the HUD texture to the screen. To begin the code we'll clear the depth and color buffers using glClear. Next we'll set the blending function to GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA to eliminate our transparent color from the screen, then bitblit the HUD texture onto the bottom half of the screen. At the bottom of our Render function we'll swap the buffers to display the final rendered image to the screen. We've now finished the source code to load and display a texture to screen and easily add more textures without having to add many new lines of code. I told you texturing was easy! The source code for the updated SetGLDefaults function is shown here:

```
void SetGLDefaults()
{
   long tex_id;
   char texture_path[] = "../media/";
   char filename[50];

   glClearColor (0.0f, 0.0f, 0.0f, 1.0f);
   glEnable (GL_DEPTH_TEST);
   glDisable (GL_CULL_FACE);
   glEnable (GL_TEXTURE_2D);
   glEnable (GL_BLEND);

   glGenTextures (MAX_TEXTURE_NAMES, TextureName);
   for (tex_id = 0; tex_id < MAX_TEXTURE_FILES; tex_id++)
   {
      sprintf (filename,"%s%s", texture_path, texture_file[tex_id].filename);

      if (texture_file[tex_id].is_transparent) LoadTransparentTexture
          (filename, tex_id+1, texture_file[tex_id].filter_type, texture_file
          [tex_id].filter_type, texture_file[tex_id].wrap_type, 0,0,255);
      else LoadTexture (filename, tex_id+1, texture_file[tex_id].filter_type,
          texture_file[tex_id].filter_type, texture_file[tex_id].wrap_type);
   }
}
```

Chapter Example

Please see the example from the downloadable companion files (ex11_1).

Conclusion

In this chapter, we learned how to load bitmap files, bind them as textures, and draw them on the screen. We also learned how to use transparent color to draw a cutout texture to the screen. If you're interested, you can change the texture file to the HUD texture by changing the file listed in the texture_file array. Ideally, if you were to write a game using a texture list you'd probably want to add separate R, G, B values for the transparent color, which would give you the ability to customize the transparent color on a per-texture basis, instead of hardcoding the value like we did. In the next chapter we'll write the core of the rendering engine and add other rendering features and functions.

Chapter 12

Rendering Scenes

When developing video games, one of the main turning points in the development cycle happens when the programmers get the rendering engine working, since it is what displays the fully textured maps.

Setting the Perspective View

With the knowledge we gained in the previous chapter, we can begin writing the code to load maps, bind textures to objects, and draw the map to the screen. To begin the coding process, we must update our SetGLDefaults function to have a perspective view calculated in the projection matrix. When we draw objects to the screen, this projection will allow depth to be displayed. To update the function we'll add three new lines of source code to the top of the function to switch to the projection matrix, use the gluPerspective function with the proper parameters, then switch back to the model view matrix. The first parameter is the field of view, which allows you to view *n* degrees of a scene. Normally a game's field of view is anywhere from 60 to 90 degrees. In our game we'll use 90 degrees. Figure 12.1 displays the field of view using 45, 60, and 90 degrees of view.

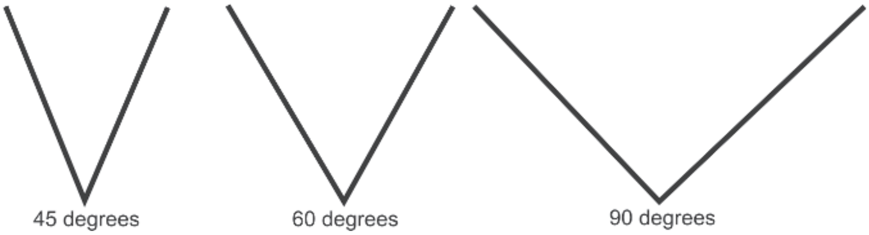


Figure 12.1: Three field of view settings

The second parameter is the aspect ratio for the perspective view, which is calculated by dividing the width of the screen (new_resolution.dmPelsWidth) by the height (new_resolution.dmPelsHeight). The final two parameters for the function are the near and far planes, which determine when things are in view and out of view. Since our coordinate system is really small, we'll set the near plane to something ridiculously low

(0.00001f) and the far plane to 5.0f, which is far beyond the constraints of our level, so we can always display our scenes properly. The source code for the updated SetGLDefaults function is provided below.

```
void SetGLDefaults()
{
   long tex_id;
   char texture_path[] = "../media/";
   char filename[50];

   glMatrixMode (GL_PROJECTION);
      gluPerspective (90.0f, (float)new_resolution.dmPelsWidth /
                (float)new_resolution.dmPelsHeight, 0.00001f, 5.0f);
   glMatrixMode (GL_MODELVIEW);

   glClearColor (0.0f, 0.0f, 0.0f, 1.0f);
   glEnable (GL_DEPTH_TEST);
   glDisable (GL_CULL_FACE);
   glEnable (GL_TEXTURE_2D);
   glEnable (GL_BLEND);

   glGenTextures (MAX_TEXTURE_NAMES, TextureName);
   for (tex_id = 0; tex_id < MAX_TEXTURE_FILES; tex_id++)
   {
      sprintf (filename,"%s%s", texture_path, texture_file[tex_id].filename);

      if (texture_file[tex_id].is_transparent) LoadTransparentTexture (filename,
                tex_id+1, texture_file[tex_id].filter_type, texture_file
                [tex_id].filter_type, texture_file[tex_id].wrap_type, 0,0,255);
      else LoadTexture (filename, tex_id+1, texture_file[tex_id].filter_type,
                texture_file[tex_id].filter_type, texture_file[tex_id]
                .wrap_type);
   }
}
```

Loading Maps

After updating the SetGLDefaults function, we now can display rendered data with a perspective view. Unfortunately there's nothing being rendered that takes advantage of the view at this time, so we'll fix that right now! With the perspective view in place, we can begin discussing the steps needed to load our level. Obviously we'll need to include the map.h header and declare a new global variable called map, which is of the MAP type. We'll also create a new function with a Boolean return value called LoadMap. The function will accept a single string parameter for the filename. Within the function, we'll run the map.open method. If the return value is false we'll exit the function, returning false to indicate failure.

If the map loads successfully, then we'll loop through each object and its texture layers to properly assign each texture ID based on the texture layer's filename.IDs. We're using multiple texture layers because eventually (in Chapter 16), we'll write the code to use the multitexturing OpenGL extension. In the meantime, we'll create a loop from 0 to the number of objects in the map (header.max_objects). Inside the loop we'll create another loop from 0 to the value of MAX_TEXTURE_LAYERS (hardcoded for 2). Within the function we'll run the function GetTextureID and specify the texture filename (map.object[obj].texture[tex_layer].filename) to get its texture ID. The returned texture ID value is saved (to map.object[obj].texture[tex_layer].id) and the loop proceeds to the next texture/object.

The code for the GetTextureID function is rather straightforward. First we loop from 0 to the number of textures in our list (MAX_TEXTURE_FILES). Within the loop we simply check to see if any of the texture files are inside the input texture filename. If the file is found, we return the current loop value plus one (because it's the texture number and 0 is reserved). If the value isn't found, the default return value is 0, so the texture won't be displayed. This method of texture assignment allows for a great deal of flexibility. For instance, rather than hardcode our HUD texture as 1, we could simply use the GetTextureID function to get the texture value in real time, which allows us to place the "hud.bmp" file anywhere in the texture list. Furthermore, it allows the game programmer to create a PAK/WAD system (similar to Quake/Doom games) to encapsulate many files into one massive file. Since the game data is loaded directly from the file, you could specify that "hud.bmp" is the bitmap for the HUD and simply use the function to get the texture ID. The source code for the GetTextureID and LoadMap functions is given below.

```
long GetTextureID(char *filename)
{
   for (long tex_id = 0; tex_id < MAX_TEXTURE_FILES; tex_id++)
   {
      if (strstr(filename, texture_file[tex_id].filename) != NULL) return
               (tex_id+1);
   }
   return (0);
}

bool LoadMap(char *filename)
{
   if (!map.Open(filename)) return (false);

   for (long obj = 0; obj < map.header.max_objects; obj++)
   {
      for (long tex_layer = 0; tex_layer < MAX_TEXTURE_LAYERS; tex_layer++)
      {
         map.object[obj].texture[tex_layer].id = GetTextureID(map.object
                  [obj].texture[tex_layer].filename);
      }
```

```
    }


     if (map.header.use_skybox)
         {
         map.skybox.front.texid = GetTextureID(map.skybox.front.filename);
         map.skybox.back.texid = GetTextureID(map.skybox.back.filename);
         map.skybox.left.texid = GetTextureID(map.skybox.left.filename);
         map.skybox.right.texid = GetTextureID(map.skybox.right.filename);
         map.skybox.top.texid = GetTextureID(map.skybox.top.filename);
         map.skybox.bottom.texid = GetTextureID(map.skybox.bottom.filename);
         }

     return (true);
}
```

Since we're now able to assign textures to objects, we can add our base textures to the texture list. We've got several base textures (wall.bmp, floor.bmp, and ceiling.bmp) that we can use to texture an object. Although it's not a huge library like some games have, it's enough to get the point across when designing levels.

Rendering Scenes

Now that we've got the code written to load and texture maps, we can begin writing the rendering engine. First, we'll create a function called RenderMap, which as the name implies will render our map data. The function doesn't require any parameters or return values. The first thing we'll do within the function is check the number of objects in the map. If the maximum number of objects is 0, then we'll exit the function to avoid wasting any further processor cycles on nothing. After the object check, we'll set the color to white to keep the data from being rendered in the current color. Next we write the rendering loop, which is the main body of code that renders the map. The main rendering code would loop through each object of the map. With each iteration of the loop we would set any default values. In our case we'll bind the first texture layer for the object, then call the function RenderObject and pass the object number (obj variable) to render each object individually. It's a good idea to create a function to render each object individually because we will need to reuse this code later when we write the lighting system. Moreover, the code used to render each object can also be optimized easily because there is less overhead since we're not incorporating the rendering code in one huge chunk.

Within the RenderObject function we'll loop through the object's triangles, inputting their UV texture coordinates and vertices into OpenGL to build the fully textured object. Once the for loop finishes we end the drawing process and begin another iteration of the object loop until we've gone through all the objects in the map. This is an oversimplified explanation of the rendering source code, but it's still relevant to most pieces of code. Doesn't the process sound simple enough?

To make life easier we've declared three variables to store the vertex references within the triangle loop. Although it won't make any different speed-wise, it's much easier to read the code when using smaller reference variable names (e.g., vertex_1) rather than conventional variable names (e.g., map.object[obj].triangle[tri].point[0]) within the object vertex array. Since each object is made up of many triangles (or two in the case of our objects) we use vertex references, which are taken from the triangle point values, to easily determine what vertices are to be drawn for each triangle. Thankfully the UV coordinates don't use the vertex references since they are stored in the triangle data structure. Once we've finished rendering the specified object, we exit the function and return to the main rendering loop to begin another iteration. The source code for rendering the map is below.

```
void RenderObject(long cur_obj)
{
   glBegin (GL_TRIANGLES);
   for (long tri = 0; tri < map.object[cur_obj].max_triangles; tri++)
   {
      long vertex_1 = map.object[cur_obj].triangle[tri].point[0];
      long vertex_2 = map.object[cur_obj].triangle[tri].point[1];
      long vertex_3 = map.object[cur_obj].triangle[tri].point[2];

      glTexCoord2fv (map.object[cur_obj].triangle[tri].uv[0].uv1);
      glVertex3dv (map.object[cur_obj].vertex[vertex_1].xyz);

      glTexCoord2fv (map.object[cur_obj].triangle[tri].uv[0].uv2);
      glVertex3dv (map.object[cur_obj].vertex[vertex_2].xyz);

      glTexCoord2fv (map.object[cur_obj].triangle[tri].uv[0].uv3);
      glVertex3dv (map.object[cur_obj].vertex[vertex_3].xyz);

   }
   glEnd();
}

void RenderMap()
{
   if (map.header.max_objects == 0) return;

   for (long obj = 0; obj < map.header.max_objects; obj++)
   {
      glColor3f (1.0f, 1.0f, 1.0f);

      glBindTexture (GL_TEXTURE_2D, TextureName[map.object[obj].texture
                [0].id]);
      RenderObject ( obj );
   }
}
```

Skyboxes

When writing games that view the outside world, it's a good idea for you to show something more elaborate than a solid background color. To tackle this minor issue we can use a skybox, which draws a cube around the boundaries of the world using a set of six bitmaps that connect seamlessly. When the skybox is drawn with the proper textures, a three-dimensional sky is born! Regardless of what direction you turn, a new part of the sky will be visible to you and the six bitmaps will connect perfectly together to produce an effect similar to what you'd see if you looked at the sky outside. Figure 12.2 displays a skybox and a player in the middle of the world.

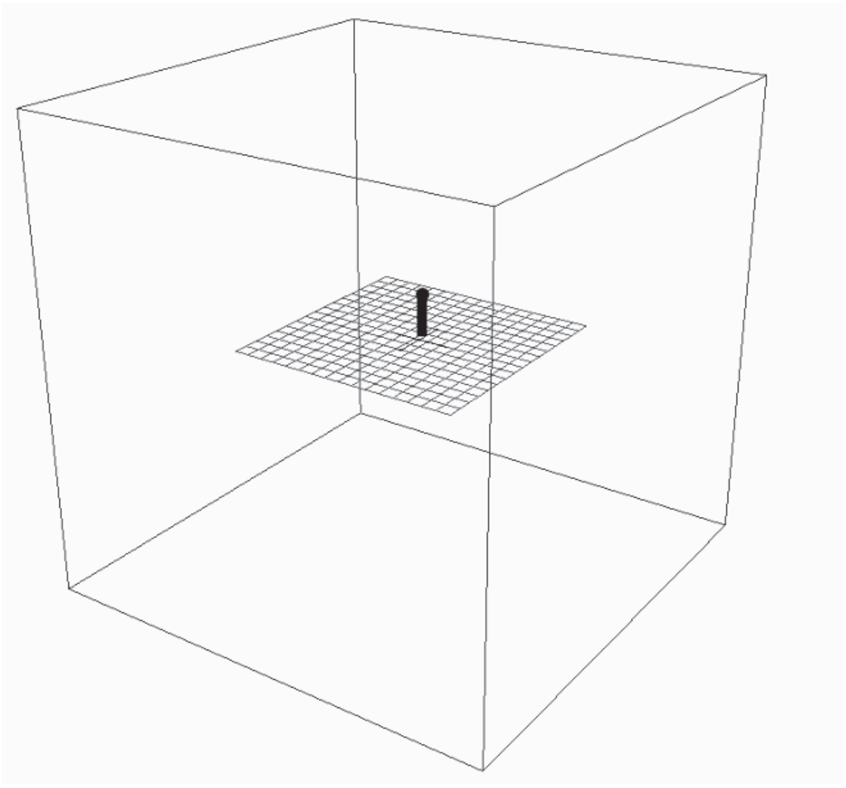


Figure 12.2: Skybox wireframe with a player in the middle of the screen

The effect is quite easy to render but fairly difficult to design when making the graphics.

Fortunately, there are several software packages available that produce high-quality landscape and sky pictures that can be used for skyboxes. When creating skyboxes, we need six bitmaps for the different views of the world (up, down, left, right, front, back). When the bitmaps are put together, they'll draw the set seamlessly so the sky will look like one high-resolution sky. If you're interested in learning about software packages that create landscapes and skies, please refer to the Further Reading section of the book. Creation of the skyboxes is beyond the scope of this book, but I encourage you to search on the Internet for tutorials on the subject. I've created a six-bitmap skybox, located in the media directory of the companion files. The bitmap files sky_rt, sky_lf, sky_up, sky_dn, sky_fr, and sky_bk are standard 24-bit 256x256 bitmap files. To use the files, we'll simply add them to the texture list and modify the MAX_TEXTURE_FILES constant to use the new files. We'll also need to update the MAX_TEXTURE_NAMES constant to give us more texture names. It is a good idea to set the constant to 50, since we'll begin to burn through the textures fairly quickly through the rest of the book.

After inserting the skybox textures into the texture file list, we must update the source code in the LoadMap function to get texture IDs for each side of the skybox. To update the function, we'll create an if statement at the bottom of the function to check whether the map has enabled the skybox function. If the value is skybox enabled (map.header.use_skybox is true), then each side of the skybox will call the GetTextureID function to get the texture ID to render its bitmap (e.g., map.skybox.front.texid = GetTextureID(map.skybox.front.filename)). Once all six bitmaps have their texture IDs assigned, the source code for texturing skyboxes is finished. The updated LoadMap function is shown below.

```
bool LoadMap(char *filename)
{
   if (!map.Open(filename)) return (false);

   for (long obj = 0; obj < map.header.max_objects; obj++)
   {
      for (long tex_layer = 0; tex_layer < MAX_TEXTURE_LAYERS; tex_layer++)
      {
         map.object[obj].texture[tex_layer].id = GetTextureID(map.object
                   [obj].texture[tex_layer].filename);
      }
   }


   if (map.header.use_skybox)
   {
      map.skybox.front.texid = GetTextureID(map.skybox.front.filename);
      map.skybox.back.texid = GetTextureID(map.skybox.back.filename);
      map.skybox.left.texid = GetTextureID(map.skybox.left.filename);
      map.skybox.right.texid = GetTextureID(map.skybox.right.filename);
      map.skybox.top.texid = GetTextureID(map.skybox.top.filename);
      map.skybox.bottom.texid = GetTextureID(map.skybox.bottom.filename);
   }

   return (true);
}
```

As mentioned earlier in the chapter, to render the skybox we simply draw a cube (six quads) around the outer boundaries of our map. The outer boundaries for each axis of our map are simplified to –1 and 1. This makes life much easier because drawing quads along these lines is very simple! When texturing the data, we'll simply use a tile of 1.0 since we don't want to tile the textures. We'll place the skybox code in a function called RenderSkyBox. Within the function, we'll bind the current texture side (such as map.skybox.front.texid), draw the quad with the proper texture/vertex coordinates, and end the creation of the quad. I used the front side as the first quad, but it doesn't matter what side you start with. Once we've rendered one side of our skybox we can copy and paste the code to another side and alter the vertex coordinates and texture ID where needed to make it fit into another skybox side. We repeat this process until each side of the skybox has been rendered, and we've now got a fully implemented skybox! The source code to render the skybox is shown here:

```
void RenderSkyBox()
{
   glBindTexture (GL_TEXTURE_2D, TextureName[map.skybox.front.texid]);
   glBegin (GL_QUADS);
      glTexCoord2f (0.0f, 1.0f);
      glVertex3f (-1.0f, -1.0f, -1.0f);

      glTexCoord2f (1.0f, 1.0f);
      glVertex3f (1.0f, -1.0f, -1.0f);

      glTexCoord2f (1.0f, 0.0f);
      glVertex3f (1.0f, 1.0f, -1.0f);

      glTexCoord2f (0.0f, 0.0f);
      glVertex3f (-1.0f, 1.0f, -1.0f);
   glEnd();

   glBindTexture (GL_TEXTURE_2D, TextureName[map.skybox.back.texid]);
   glBegin (GL_QUADS);
      glTexCoord2f (1.0f, 1.0f);
      glVertex3f (-1.0f, -1.0f, 1.0f);

      glTexCoord2f (0.0f, 1.0f);
      glVertex3f (1.0f, -1.0f, 1.0f);
      glTexCoord2f (0.0f, 0.0f);
      glVertex3f (1.0f, 1.0f, 1.0f);

      glTexCoord2f (1.0f, 0.0f);
      glVertex3f (-1.0f, 1.0f, 1.0f);
   glEnd();
```

```
glBindTexture (GL_TEXTURE_2D, TextureName[map.skybox.right.texid]);
glBegin (GL_QUADS);
   glTexCoord2f (0.0f, 1.0f);
   glVertex3f (1.0f, -1.0f, -1.0f);

   glTexCoord2f (1.0f, 1.0f);
   glVertex3f (1.0f, -1.0f, 1.0f);

   glTexCoord2f (1.0f, 0.0f);
   glVertex3f (1.0f, 1.0f, 1.0f);

   glTexCoord2f (0.0f, 0.0f);
   glVertex3f (1.0f, 1.0f, -1.0f);
glEnd();

glBindTexture (GL_TEXTURE_2D, TextureName[map.skybox.left.texid]);
glBegin (GL_QUADS);
   glTexCoord2f (1.0f, 1.0f);
   glVertex3f (-1.0f, -1.0f, -1.0f);

   glTexCoord2f (0.0f, 1.0f);
   glVertex3f (-1.0f, -1.0f, 1.0f);

   glTexCoord2f (0.0f, 0.0f);
   glVertex3f (-1.0f, 1.0f, 1.0f);

   glTexCoord2f (1.0f, 0.0f);
   glVertex3f (-1.0f, 1.0f, -1.0f);
glEnd();


glBindTexture (GL_TEXTURE_2D, TextureName[map.skybox.top.texid]);
glBegin (GL_QUADS);
   glTexCoord2f (0.0f, 0.0f);
   glVertex3f (-1.0f, 1.0f, -1.0f);

   glTexCoord2f (1.0f, 0.0f);
   glVertex3f (-1.0f, 1.0f, 1.0f);

   glTexCoord2f (1.0f, 1.0f);
   glVertex3f (1.0f, 1.0f, 1.0f);

   glTexCoord2f (0.0f, 1.0f);
   glVertex3f (1.0f, 1.0f, -1.0f);
glEnd();


glBindTexture (GL_TEXTURE_2D, TextureName[map.skybox.bottom.texid]);
glBegin (GL_QUADS);
   glTexCoord2f (0.0f, 0.0f);
   glVertex3f (-1.0f, -1.0f, -1.0f);
```

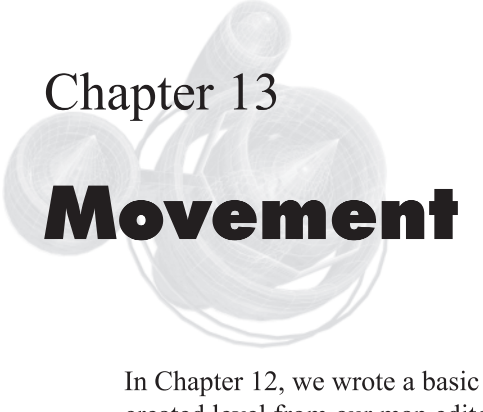
```
        glTexCoord2f (1.0f, 0.0f);
        glVertex3f (-1.0f, -1.0f, 1.0f);

        glTexCoord2f (1.0f, 1.0f);
        glVertex3f (1.0f, -1.0f, 1.0f);

        glTexCoord2f (0.0f, 1.0f);
        glVertex3f (1.0f, -1.0f, -1.0f);
    glEnd();
}
```

Chapter Example

To complete the example for this chapter (see ex12_1 from the companion downloadable files), we'll add an if statement to the RenderMap function to check whether or not the map should display the skybox. If the value (map.header.use_skybox) is true, then we'll run RenderSkyBox; otherwise we'll start rendering the map object itself. We're done with the example for the chapter and we can now load and texture maps, render them to the screen, and display a skybox in the background as well!

Conclusion

In this chapter, we learned how to load and texture files, how map rendering works, and how to assign texture IDs to objects based on their filename. These are the core features that will allow us to build new functionality into our game engine in the coming chapters. In the next chapter we'll learn how to walk around our map using keyboard and mouse input. This is essential so we can easily move around in the world to view new special effects since they'll be placed at certain locations.



Chapter 13
Movement

In Chapter 12, we wrote a basic rendering engine that displayed a user-created level from our map editor that was fully textured. Before we begin adding any special effects like lighting, fog, models, or other goodies, we'll discuss the code needed to move the user around the level. Obviously this is an essential feature for any 3D first-person shooter. If we didn't write any code to move the player, then we would have made one heck of a boring first-person shooter game.

The Math Behind Movement

Remember way back in 10th-grade math class when you learned about trigonometry and you thought to yourself "Why in the world would I ever need this?" Well prepare yourself, because this chapter will be using it! When a player walks through a 3D world, we're constantly using trigonometry to calculate new X/Z positions. Whether the player is walking in a straight line or strafing (moving sideways), we use trigonometry to compute the new values. The basic calculation used when the player walks forward is:

```
radians = pi / 180.0 * x_angle
new_x -= (sin(radians) * movement_speed)
new_z += (cos(radians) * movement_speed)
```

Now isn't that calculation simple? The calculation is derived straight from the mathematical calculation for drawing a circle. The next point is based solely on the current x-axis angle and the current location. To walk backward we simply swap the X/Z math symbols so we travel in the opposite direction (backward) as seen in the calculation below:

```
radians = pi / 180.0 * x_angle
new_x += (sin(radians) * movement_speed)
new_z -= (cos(radians) * movement_speed)
```

Strafing is a definite requirement for any first-person shooter. In case you're not familiar with the term, *strafing* allows the user to move from side to side. It seems like every first-person shooter since Wolfenstein has come with the strafing function, which is a definite advantage when you're trying to dodge bullets being shot by Nazis. The movement calculation for strafing right is as simple as the forward movement calculation. Before we calculate the number of radians, we add 90 degrees to the current x-axis angle, then calculate the new X/Z coordinates. By adding 90 degrees to the x-axis angle, we're turning the player 90 degrees (or to the right), then walking forward using the original calculation. The strafe right calculation is provided below:

```
radians = pi / 180.0 * (x_angle + 90)
new_x -= (sin(radians) * movement_speed)
new_z += (cos(radians) * movement_speed)
```

To strafe left, we simply subtract 90 degrees from the current x-axis angle, then perform the same walking forward calculation, which will fake a left turn and then move the player forward. Alternatively you could add 270 degrees to the current x-axis angle. The calculation to strafe left is provided below:

```
radians = pi / 180.0 * (x_angle - 90)
new_x -= (sin(radians) * movement_speed)
new_z += (cos(radians) * movement_speed)
```

Although the calculations are fairly close to C programming code, there are several additions to each calculation that we'll have to make when we write the movement code. To begin writing code, we'll declare two global variables (player and enemy) of the MAP_ENTITY type. One variable will be used for the local player data (player) and the other will contain the remote user's data when we're playing a deathmatch game (enemy), which we'll code in Chapter 17.

In the meantime we'll update the Render function source code so we can use the player's angle and XYZ position. To modify the source code, after clearing the depth/color buffers we'll back up the matrix (glPushMatrix) and rotate the view around the x-axis using the player's x-axis angle. Then we'll translate the matrix to the current player position using the values in the player.xyz array, render the map, and restore the original matrix. It's that simple. Of course we must next write the calculations to physically walk around the level.

```
void Render()
{
   glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
   glPushMatrix();
      glRotatef (player.angle[0], 0.0, 1.0, 0.0);
      glTranslated (player.xyz[0], player.xyz[1], player.xyz[2]);
      RenderMap();
   glPopMatrix();

      glEnable (GL_BLEND);
      glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
      glBitBlt(1, 0,240,640,240);
   SwapBuffers(raster.hDC);
}
```

Moving Forward and Backward

To put the calculations into code, we must declare three new global constants. The first constant is PI, the second is MOVEMENT_SPEED, which is 0.125, and the third is TURN_SPEED, which is 12.0. MOVEMENT_SPEED is the number at which the player is moved when a button is pressed to move forward or backward or to strafe left or right. Since the map has a limitation of 2.0, I chose 0.125 as an individual step, requiring 16 steps to walk from one side to the other. The TURN_SPEED constant refers to the number of degrees rotated left or right when the user turns. Although it doesn't provide 360 degrees of freedom like many games, 12-degree increments still gives you 30 different angles when walking. This value can be changed if you'd like to tweak the settings to get the best turning speed available.

After creating the constants we'll create our first movement function called MoveForward with a parameter reference of the MAP_ENTITY type. We'll simply execute the calculation we discussed earlier, saving the new location back to the entity X/Z position. This code will therefore work when the user moves his player or if we add projectiles (like rockets) that will move until they hit an object. The source code for walking forward is provided below.

```
void MoveForward(MAP_ENTITY &entity)
{
   double radians = PI / 180.0 * entity.angle[0];
   entity.xyz[0] -= sin(radians) * MOVEMENT_SPEED;
   entity.xyz[2] += cos(radians) * MOVEMENT_SPEED;
}
```

As mentioned in the first section of the chapter, the move backward functionality works in the same way as the move forward function, with the exception that the add and subtract symbols are reversed so the movement will move backward. With this in mind, we can copy and paste the code from the MoveForward function call and simply change the function name and reverse the symbols to finish the code for walking backward. The source code to walk backward is provided below.

```
void MoveBackward(MAP_ENTITY &entity)
{
   double radians = PI / 180.0 * entity.angle[0];
   entity.xyz[0] += sin(radians) * MOVEMENT_SPEED;
   entity.xyz[2] -= cos(radians) * MOVEMENT_SPEED;
}
```

Strafing

As mentioned earlier, the strafing function allows the user to slide left or right without having to turn. This function is a must when writing a first-person shooter because it allows the user to easily dodge oncoming projectiles like rockets. As with every movement function, we'll use the source code from the MoveForward function as the base code for this function and then modify it. The strafing function will place the current x-axis angle plus 90 degrees (so we're walking to the right) into a new angle variable, and then we run the calculation to move forward. When we calculate the radians, we simply use the variable containing the new angle. The rest of the calculation can remain the same because after calculating the radians, we're just moving forward. This is a cheap trick for strafing but it serves its purpose! The source code for the StrafeRight function is shown below.

```
void StrafeRight(MAP_ENTITY &entity)
{
   double new_angle = entity.angle[0] + 90;
   double radians = PI / 180.0 * new_angle;
   entity.xyz[0] -= sin(radians) * MOVEMENT_SPEED;
   entity.xyz[2] += cos(radians) * MOVEMENT_SPEED;
}
```

To strafe left, we simply alter the new angle calculation to subtract 90 from the current x-axis angle, which will rotate the user to the left and then perform the movement calculation. Obviously we'll copy the code from the StrafeRight function and simply modify the function name and math symbol in the first calculation. The source code for the StrafeLeft function is given below.

```
void StrafeLeft(MAP_ENTITY &entity)
{
   double new_angle = entity.angle[0] - 90;
   double radians = PI / 180.0 * new_angle;
   entity.xyz[0] -= sin(radians) * MOVEMENT_SPEED;
   entity.xyz[2] += cos(radians) * MOVEMENT_SPEED;
}
```

Turning

The source code to turn the user left and right might be the easiest source code of all the movement functions we've written so far. To turn to the right, we simply increment the x-axis angle by the value in TURN_SPEED and set the value to 0 if the angle is greater than 360 degrees. To turn left, we simply decrease the x-axis angle by the value in TURN_SPEED and set the value to 360 if the angle is less than 0. No trig calculations need to be performed since we're simply rotating around the x-axis. The source code to turn left and right is shown here.

```
void TurnLeft(MAP_ENTITY &entity)
{
   entity.angle[0] -= TURN_SPEED;
   if (entity.angle[0] < 0.0) entity.angle[0] = 360.0;
}


void TurnRight(MAP_ENTITY &entity)
{
   entity.angle[0] += TURN_SPEED;
   if (entity.angle[0] > 360.0) entity.angle[0] = 0.0;
}
```

Checking for Keypresses

In the previous sections we wrote the back-end code that turned, strafed, and moved an entity in our map. Since the back-end functionality has been written, we can begin writing the code to handle keyboard input. In the beginning chapters of the book, we learned that Windows is an event-driven operating system. When the user presses or releases a key on the keyboard, a message is sent. Using Windows messages is great when writing applications like map editors where the keyboard performance isn't an issue, but it can be rather slow compared to other methods available in the Win32 SDK. We'll use the function GetAsyncKeyState to query the state of a specified key, which can be done at any time, and it simply replies back with a 1 (pressed) or 0 (not pressed) depending on its state.

When checking the key state we can use the key's ASCII character or the Windows-based virtual key code. Typically buttons such as up, down, left, right, and escape are specified by their virtual key codes using the naming convention "VK_" then the name. Putting this knowledge to good use, we'll create a function (called CheckInput) to handle all aspects of input, whether it be from the mouse or from the keyboard. Within the function we'll add several function calls to GetAsyncKeyState to see the key state for escape (VK_ESCAPE), up (VK_UP), down (VK_DOWN), left (VK_LEFT), and right (VK_RIGHT). In the event the key is pressed, then we'll run a specific movement function (i.e., up = MoveForward, down = MoveBackward, left = StrafeLeft, right = StrafeRight, and escape = post a quit message). Within the main loop we'll run the CheckInput function to give us continuous input information. The source code for the CheckInput function is below.

```
void CheckInput()
{
   if (GetAsyncKeyState (VK_ESCAPE)) PostQuitMessage(0);
   else if (GetAsyncKeyState (VK_UP)) WalkForward(player);
   else if (GetAsyncKeyState (VK_DOWN)) WalkBackward(player);
   else if (GetAsyncKeyState (VK_LEFT)) StrafeLeft(player);
   else if (GetAsyncKeyState (VK_RIGHT)) StrafeRight(player);
}
```

Tracking Mouse Movements

There is a simple trick to tracking mouse movements. Before we begin the main loop we'll hide the mouse cursor (using ShowCursor(false)), then position the mouse (using the function SetCursorPos) in the middle of the screen. It's not essential for us to hide the mouse cursor, but it makes the game look better because the cursor won't look stuck in the middle. Each time the CheckInput function is run, we'll check the current position of the mouse (using the function GetCursorPos). If the mouse X is greater than the middle, we must turn to the right. If the mouse X is less than the middle, we must turn to the left. After turning the appropriate way, we reset the mouse to the middle of the screen and start the process again. The updated CheckInput source code is shown here:

```
void CheckInput()
{
   POINT point;

   if (GetAsyncKeyState (VK_ESCAPE)) PostQuitMessage(0);
   else if (GetAsyncKeyState (VK_UP)) MoveForward(player);
   else if (GetAsyncKeyState (VK_DOWN)) MoveBackward(player);
   else if (GetAsyncKeyState (VK_LEFT)) StrafeLeft(player);
   else if (GetAsyncKeyState (VK_RIGHT)) StrafeRight(player);

   GetCursorPos (&point);
   if (point.x > (int)(new_resolution.dmPelsWidth/2)) TurnRight(player);
   else if (point.x < (int)(new_resolution.dmPelsWidth/2)) TurnLeft(player);
   SetCursorPos (new_resolution.dmPelsWidth/2, new_resolution.dmPelsHeight/2);
}
```

Checking Mouse Button Input

Checking the status of each mouse button is essential when your game uses mouse input for movement or an action. Most first-person shooters, if not all, use the first (usually left) mouse button as the shoot option. The second and third buttons normally control a variety of other things like jump, switch weapons, etc. In the case of our game, we're going to make the middle mouse button move forward and the right mouse button move backward. Rather than handle the Windows messages for button press down and button press up, we'll once again use the function GetAsyncKeyState, but specify either VK_MBUTTON for the middle mouse button or VK_RBUTTON for the right, allowing us to easily get the status of whether the button is pressed. To update our CheckInput function, we'll update the if statements to check the values of these two buttons and perform the appropriate action. Under most circumstances, using GetAsyncKeyState is a great method of getting the status of the mouse buttons; however, if the mouse buttons are swapped (as a Windows option), then you'll have to adjust the buttons appropriately to handle the proper input. The updated source code for the CheckInput function is given below.

```
void CheckInput()
{
   POINT point;

   if (GetAsyncKeyState (VK_ESCAPE)) PostQuitMessage(0);
   else if (GetAsyncKeyState (VK_UP) || GetAsyncKeyState(VK_MBUTTON))
         MoveForward(player);
   else if (GetAsyncKeyState (VK_DOWN) || GetAsyncKeyState(VK_RBUTTON))
         MoveBackward(player);
   else if (GetAsyncKeyState (VK_LEFT)) StrafeLeft(player);
   else if (GetAsyncKeyState (VK_RIGHT)) StrafeRight(player);

   GetCursorPos (&point);
   if (point.x > (int)(new_resolution.dmPelsWidth/2)) TurnRight(player);
   else if (point.x < (int)(new_resolution.dmPelsWidth/2)) TurnLeft(player);
   SetCursorPos (new_resolution.dmPelsWidth/2, new_resolution.dmPelsHeight/2);
}
```

Chapter Example

Please see the example from the companion files (ex13_1).

Conclusion

In this chapter, we learned how to move, strafe, and turn the players in our map. This is essential for the game since we cannot display certain functions without having the ability to move around the level to display them, or at the very least turn to view things behind us. In the following chapters we'll discuss how to draw lights, extend beyond OpenGL 1.1, code multiplayer functions, and of course find out about tips and tricks for game development.

Chapter 14

Lights and Fog

In the past couple of chapters we've learned how to load and display textures, render our maps, and check the keyboard and mouse input in order to walk around the level. In this chapter, we'll discuss the different types of lighting systems we can use to light our levels. We'll also write the code to light objects using projected texturing.

Dynamic Lighting

For many games, lighting adds depth and/or mood that would otherwise be lacking. OpenGL has a simple form of lighting built into the API that is commonly referred to as dynamic lighting. This style of lighting is calculated on a per-vertex basis, which means we must have many vertices in a scene to accurately draw the light. In many cases you'll be limited to eight lights due to the limitations of most video hardware. This limitation isn't as big a concern as the per-vertex calculations, which can drastically drain the performance of the system with high numbers of vertices. One advantage to dynamic lighting is that it requires a minimal amount of code to get running, which is great when you're prototyping code. Some features and special effects also use dynamic lighting for the backbone of their operation. Figure 14.1 displays a cube rendered using dynamic lighting.

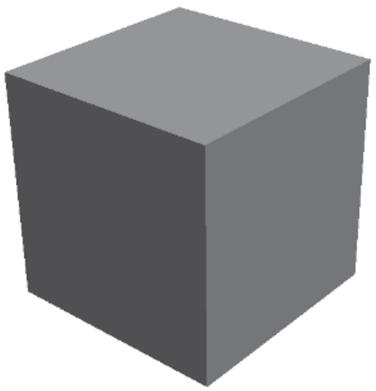


Figure 14.1: Dynamically lit cube

In our game engine, we'll use a different method for lighting instead of using dynamic lighting. Other methods of lighting include vertex coloring, radiosity, per-pixel lighting, and projected texturing. Some of these lighting techniques (e.g., vertex coloring, radiosity, and projected texturing) have been used in 3D games for several years, while others are new and emerging technologies.

Vertex-Colored Lighting

Vertex coloring is calculated in a similar fashion to dynamic lighting. When each triangle of an object is drawn, the three vertices that make up the triangle are shaded different colors to emulate lighting. This is a great method of lighting if you want to shade an entire area a single color for ambience, but it stinks if you want precise lighting because it only shades along the three sides of the triangle. Obviously this isn't an ideal lighting type for us, since we want our light to cast itself on as many objects as possible with as little code/time involved in working with it as possible. The problem with vertex-colored lighting is that if we decided to change the color of the light, then we'd have to change every vertex that's supposed to be colored. In a real-world situation where you have 10,000+ objects in a level, you'd develop carpal tunnel syndrome before you'd be able to fix every vertex in the map by clicking each vertex! Since vertex-colored shading isn't processor intensive with today's graphics processors, this type of lighting is ideal for adding ambience to smaller areas on a map, but it's not practical for large areas or entire levels. In the case of our game engine, we'll use vertex coloring on objects but only for the purposes of darkening objects for our real lighting system. Figure 14.2 displays a vertex-colored scene to fake lighting.

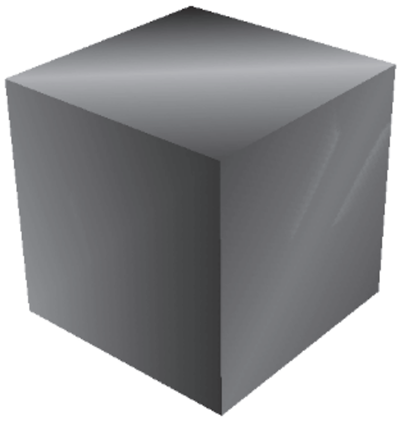


Figure 14.2: Vertex-colored lights

Radiosity Lighting

Unlike vertex-colored lighting, radiosity lighting requires a tremendous amount of calculation because of its complexity. When using radiosity lighting, each beam of light shines down a direct path. As the light continues down the path, each intersecting object and the light's location on the object are recorded. Depending on the algorithm used to calculate the lighting, some objects will bounce the light fragment in another direction until the fragment is nonexistent. Since each light contains many smaller rays, which also check for intersections, the calculations are painfully intense for the processor to compute. The more rays used in the light, the slower the calculations become. Although the lighting looks really awesome, using this technique for many lights cannot be done in real time because of the enormous number of calculations being performed. Depending on how precise the calculations are, it is possible to create a real-time radiosity engine, as can be seen in many radiosity tutorials, but normally the demos draw just a couple of lights and only display a few 3D objects. Figure 14.3 displays a working radiosity engine.

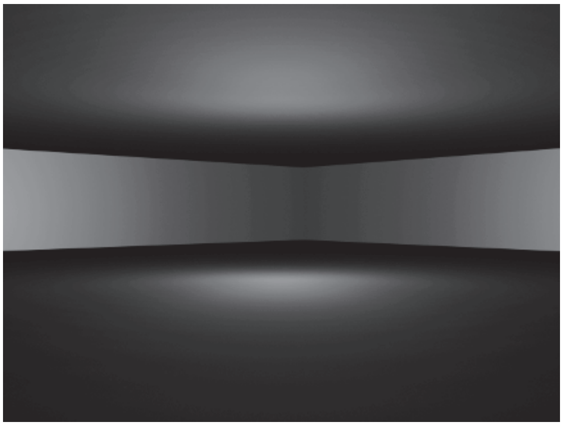


Figure 14.3: Radiosity light example

Projected Texturing

With all the different types of lighting systems we could use in our game engine, we're going to use a system called projected texturing, which projects a texture onto any number of surfaces. This technique can be used to project spotlights (black to white gradient textures) onto walls to emulate lighting, or it can be used to create elaborate shadowing effects that project onto adjacent walls. Figure 14.4 shows an example of a scene using projected texturing for lighting.

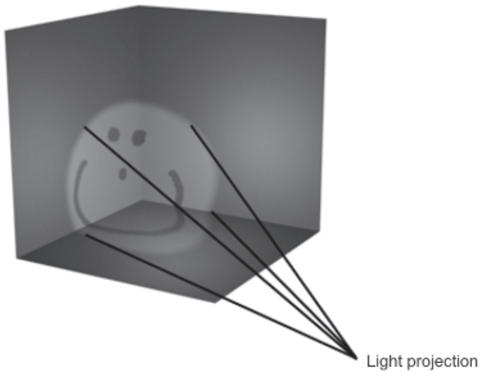


Figure 14.4: Projected texturing example

Since we're using a texture to project light onto a surface/object, we have the ability to choose exactly how we want the texture to look, which can drastically alter how it's projected onto an object/surface. In most cases, a simple circular gradient from black to white will suffice for a light, but it's up to the designer of the level or scene to decide what to use for the spotlight. For our game we'll use the file light.bmp, which is located in the media directory of the downloadable file (www.wordware.com/files/openglgd), for our projected texture. Using the map editor, we'll insert a light and add several objects to the light's include list. We include objects into its list so the light knows which objects to redraw with the lighting information. We can redraw an object 10, 20, 50, 100, or even 1,000 times using different light settings, but it must be done for each light. Because we're manipulating the texture matrix (by rotating/translating it), we must re-render each object that will be affected by this change. As an example, say we inserted one light and included two of our four walls in the include list. First we would render the level normally, then we would render each light. When the light is drawn, we rotate and translate the texture matrix to the specified location. Then each object in the include list is drawn so it can be potentially drawn with the projected texture. If the object is not in the projected texture's viewing area, it will not be drawn with the projected texture, as seen in Figure 14.5.

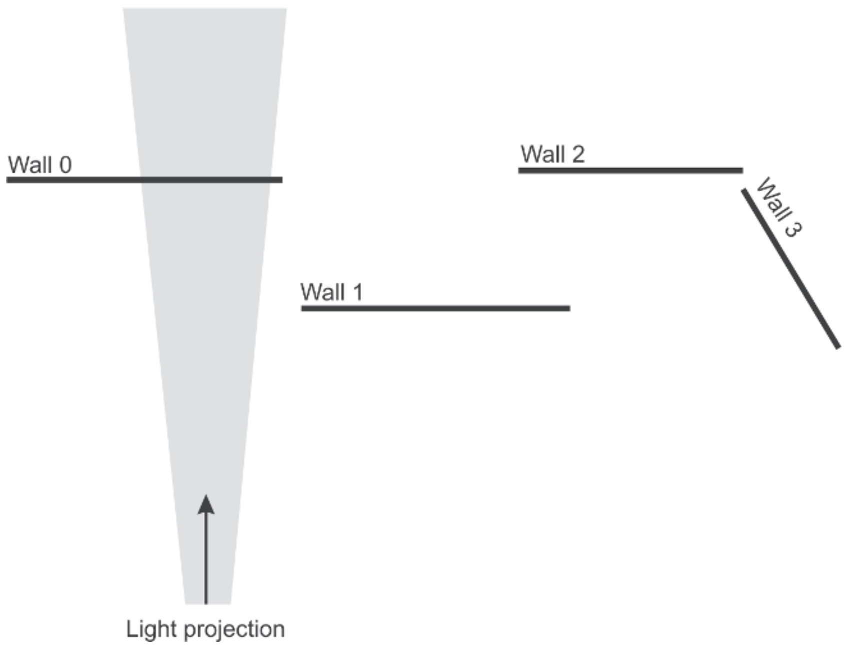


Figure 14.5: Four walls, one inline of projection, one not inline, and two not in the include list

You may want to include other objects within the include list that are not inline with the light so you can rotate the light later. This type of lighting provides great flexibility for many types of special effects such as emergency vehicle type lights, because you can easily rotate any axis of the light. Putting this new knowledge into coding terms, projected texturing is fairly simple to implement. First we'll create a new function called RenderLight, which will render a user-specified light. This allows us to easily add special effects to the light such as flickering by simply rendering the light every second or third frame.

To easily project textures onto any object, we use automatic texture coordinate generation to calculate the texture coordinates for the projected texture. Rather than trying to calculate the new coordinates for the projected texture on each object we can let the video card do the processing for us. With the power of today's graphics processors, the calculations for the texture coordinate generation take a minimal amount of time under normal circumstances. If you were to use auto texture coordinate generation to texture hundreds or thousands of objects with hundreds or thousands of triangles each, you might run into some performance issues. Thankfully the only objects in our game are two triangles, so we won't have to worry about any major performance hit when using auto tex-gen.

Auto texture generation uses four planes to calculate the texture coordinates. In order for the calculation to work, we must set the default parameters for each plane (S/T/R/Q). To define the plane parameters, we need each plane to have its own array of four (GLfloat type), which specifies the plane vector. In the case of the S coordinate, the plane array would contain the values 1.0, 0.0, 0.0, 0.0. The T plane (the next one) would have the values 0.0, 1.0, 0.0, 0.0 in its array. As you can see, setting up each array is rather simple. The R/Q plane arrays work in the same manner, moving the 1.0 value over one slot. Using the function glTexGen, we must specify the texture coordinate (GL_S, T, R, or Q) to generate coordinates for and the type of coordinate generation, which in our case is calculated from the eye-plane (GL_EYE_PLANE). This value can be changed to calculate from the object plane (GL_OBJECT_PLANE) to produce a different effect. The final parameter is the variable plane arrays.

After configuring the texture coordinate generation defaults, we simply enable texture coordinate generation using glEnable with the parameter GL_TEXTURE_GEN_S/T/R/Q. Everything we draw after enabling texture coordinate generation will automatically have texture coordinates calculated by the video card. This enables us to draw all our objects in the include lists without having to worry about calculating the texture coordinates. Texture coordinate generation will override any texture coordinates that you specify manually. This means that even though you decide to use glTexCoord to texture something while tex-gen is enabled, the coordinates will still be automatically calculated.

With texture coordinate generation ready for objects, we can begin manipulating the texture matrix. This is the key to projected texturing because most of the magic within projected texturing comes from texture matrix manipulation. Under normal circumstances the texture matrix will display a texture directly on the object specified. Since the texture matrix is a regular matrix like the model view, we can use the GL commands glRotate and glTranslate to position the texture in its new location. We rotate the texture matrix by the values in the light's angle array so we can position the light into the proper direction as seen in Figure 14.6.

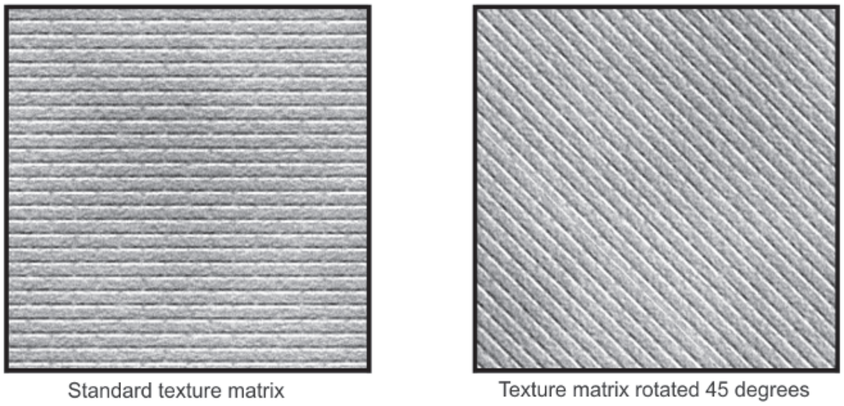


Figure 14.6: A bitmap with a regular texture matrix and one with a texture that's been rotated

Putting this information to use, first we'll switch to the texture matrix and correctly rotate/translate/scale the matrix into its desired position. Remember to always rotate the matrices and then translate to avoid having the position change. When you translate first and then rotate the matrices, their position will change because of the rotation. To avoid this, simply rotate first and then translate. Although it's not necessary, it's a good idea to scale the matrix down so the texture doesn't project itself as a huge texture. Depending on the size of the light, you can scale the texture matrix appropriately. Once we're finished modifying the texture matrix, we switch back to the model view matrix and loop through the objects in the include list, drawing each object so it will be affected by the lighting. Once we finish drawing the lights we switch back to the texture matrix and reset it to the default values by loading the identity matrix and then disable the texture generation so nothing looks messed up when we redraw the scene. Now didn't I say it would be simple! The source code for the RenderLight function is given below.

```
void RenderLight(long cur_light)
{
   GLfloat s_plane[] = {1.0f, 0.0f, 0.0f, 0.0f};
   GLfloat t_plane[] = {0.0f, 1.0f, 0.0f, 0.0f};
   GLfloat r_plane[] = {0.0f, 0.0f, 1.0f, 0.0f};
   GLfloat q_plane[] = {0.0f, 0.0f, 0.0f, 1.0f};

   glTexGenfv (GL_S, GL_EYE_PLANE, s_plane);
   glTexGenfv (GL_T, GL_EYE_PLANE, t_plane);
   glTexGenfv (GL_R, GL_EYE_PLANE, r_plane);
   glTexGenfv (GL_Q, GL_EYE_PLANE, q_plane);
```

```
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    glEnable(GL_TEXTURE_GEN_R);
    glEnable(GL_TEXTURE_GEN_Q);

    glMatrixMode (GL_TEXTURE);
       glPushMatrix();
          glRotated (map.light[cur_light].angle[0], 0.0f, 1.0f, 0.0f);
          glRotated (map.light[cur_light].angle[1], 0.0f, 0.0f, 1.0f);
          glRotated (map.light[cur_light].angle[2], 1.0f, 0.0f, 0.0f);
          glTranslated (map.light[cur_light].xyz[0], map.light
                    [cur_light].xyz[1], map.light[cur_light].xyz[2]);
          glScalef (0.25f, 0.25f, 0.25f);
    glMatrixMode (GL_MODELVIEW);


    glColor4fv (map.light[cur_light].rgba);
    for (long obj = 0; obj < map.light[cur_light].max_inclusions; obj++)
    {
       RenderObject (map.light[cur_light].inclusions[obj]);
    }

    glMatrixMode (GL_TEXTURE);
       glPopMatrix();
       glLoadIdentity();
    glMatrixMode (GL_MODELVIEW);

    glDisable (GL_TEXTURE_GEN_S);
    glDisable (GL_TEXTURE_GEN_T);
    glDisable (GL_TEXTURE_GEN_R);
    glDisable (GL_TEXTURE_GEN_Q);
}
```

In order for the light to render we must update the RenderMap function to loop through each light, binding the light texture (in our case the file light.bmp) and of course setting any blending settings (in our case to eliminate black) before we render each light. After binding the texture and setting any blending values we can render the light and begin the next iteration of the loop. To finish up the lighting code, we must update the SetDefaults function and change the depth function so it's less than or equal to (GL_LEQUAL) when comparing objects. This will allow our projected textures to be drawn over the original objects in the level. If we didn't change the depth function, then the projected textures wouldn't be drawn because the depth function calculation would eliminate the include list objects from the scene automatically. One big issue with projected texturing is that it doesn't dim other parts of the scene. To get around this, one simple trick is to darken the vertices of everything in the room by lowering the RGBA values for each vertex. This is a small hack of the vertex-colored lighting, but it does a wonderful job of drawing dark areas and it's super fast! Figure 14.7

displays two scenes, one with the vertices at full intensity and one with the vertex colors at half their original value.

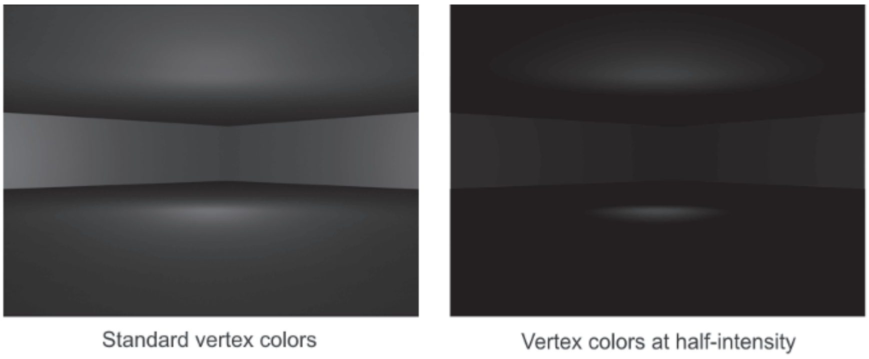


Figure 14.7: Examples of full intensity and half-intensity

Fog

Perhaps one of the easiest special effects to add to any game in OpenGL is fog. Using fog in OpenGL is as simple as six function calls to set up and start the fog rendering. This allows for quick and easy support for fogging in any game or application. Similar to dynamic lighting, fog is better calculated when there are more vertices in the scene; however, it does not need the large number that dynamic lighting requires. Fog can be used to great effect in games to darken the atmosphere, for example, or to add haze to the background of a racing game to cover up objects that haven't entered the viewing frustum yet.

Since our map files contain the fog configuration information, we can easily configure our fog through the map editor. Each level has different fog properties, which would cause issues if we set the defaults in the SetDefaults function. Instead we'll set the fog properties after loading the map data in the LoadMap function. This will change any necessary fog details or disable fog based on the value of map.header.use_fog. When specifying any fog parameters, we use the function glFog(i/f). The first parameter of the function is always the fog parameter to specify (i.e., GL_FOG_MODE, GL_FOG_DENSITY, etc.) and the second parameter is the value of the parameter.

The first fog parameter we'll set is the fog mode, which is the equation used to calculate the fog. This parameter will directly affect how the fog looks. You can choose one of three constants to use: GL_LINEAR, GL_EXP, or GL_EXP2. Figure 14.8 shows the three different modes in a real-world rendering situation.

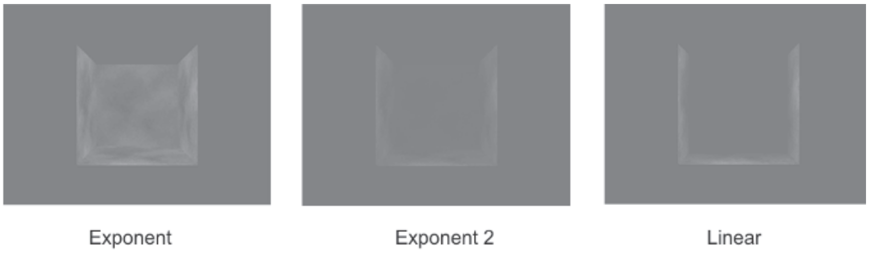


Figure 14.8: Three different modes of fog

Each fog mode produces its own unique look, so it's a good idea to test all three types before you make a decision on which fog to use. Since each fog value is set in the map editor, we'll use the fog variable map.fog.mode as the second parameter. The next value we'll set is the density, which as the name suggests controls how dense or thick the fog is on the screen. Normally the fog density is 1.0, but you can specify any number you'd like to customize the look of the fog. When specifying the fog density we use the parameter GL_FOG_DENSITY and the variable map.fog.density as the value.

When using the linear fog mode, we must specify the start (near) and end (far) distances for the fog calculation. Typically the start value is set to 0 and the end value is 1. Depending on the game you're creating, you may want to adjust the far value to reflect a very far end distance for a background haze or you may want to have a very close far value to display thicker fog. When specifying the start and end values we use the parameters GL_FOG_START and GL_FOG_END and the appropriate map fog value (map.fog.start and map.fog.end). Although these values are only for the linear fog mode, we can still specify them regardless of the fog mode because the other modes will not accept their values.

The fog color is specified using the parameter GL_FOG_COLOR, with an RGBA color array as the value. The color of the fog is completely up to the designer of the level, obviously! Just remember not to have the alpha at 0.0, because it won't be visible! If we were using indexed color (as opposed to RGB), we would use the parameter GL_FOG_INDEX and specify the indexed color number as the second parameter. Since we're using RGB values we don't need to concern ourselves with indexing.

Like any state modifier in OpenGL, we must enable or disable fog by calling glEnable or glDisable and specifying GL_FOG as the parameter. If the value of map.header.use_fog is true, we'll configure the fog's properties and enable it; otherwise we'll disable fog. In Chapter 16, we'll discuss an OpenGL extension that will allow you to specify fog values for each vertex. In the meantime the updated source code for the LoadMap function is shown below.

```
bool LoadMap(char *filename)
{
   if (!map.Open(filename)) return (false);

   for (long obj = 0; obj < map.header.max_objects; obj++)
   {
      for (long tex_layer = 0; tex_layer < MAX_TEXTURE_LAYERS; tex_layer++)
      {
         map.object[obj].texture[tex_layer].id = GetTextureID(map.object
                   [obj].texture[tex_layer].filename);
      }
   }


   if (map.header.use_skybox)
   {
      map.skybox.front.texid    = GetTextureID(map.skybox.front.filename);
      map.skybox.back.texid     = GetTextureID(map.skybox.back.filename);
      map.skybox.left.texid     = GetTextureID(map.skybox.left.filename);
      map.skybox.right.texid    = GetTextureID(map.skybox.right.filename);
      map.skybox.top.texid      = GetTextureID(map.skybox.top.filename);
      map.skybox.bottom.texid   = GetTextureID(map.skybox.bottom.filename);}


   if (map.header.use_fog)
   {
      glFogi (GL_FOG_MODE, map.fog.mode);
      glFogf (GL_FOG_DENSITY, map.fog.density);
      glFogf (GL_FOG_START, map.fog.start);
      glFogf (GL_FOG_END, map.fog.end);
      glFogfv (GL_FOG_COLOR, map.fog.rgba);
      glEnable (GL_FOG);
   }
   else glDisable (GL_FOG);


   return (true);
}
```

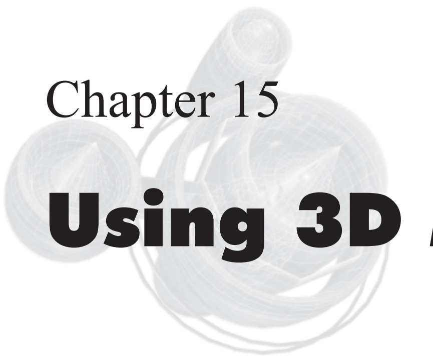
Chapter Example

Please see the example from the companion files (ex14_1).

Conclusion

In this chapter, we learned how to project textures onto objects, allowing us to create really amazing light without any major hassles. We also learned how to implement fog that's been configured through our map editor. These two special effects can help define the entire "feel" of a game. In Chapter 16 we'll discuss how to use OpenGL extensions that can add more features to both texturing and fog.

This page intentionally left blank.



Chapter 15

Using 3D Models

When creating any type of 3D video game, the use of 3D models is essential. Some popular modeling and animation tools for game development include 3ds max (by Discreet), Maya (by Alias Systems Corp.), LightWave (by NewTek), and MilkShape 3D (by chUmbaLum sOft). Each modeling tool provides a good tool set for modeling and animating characters. The first three packages are mostly used in a commercial environment because of their price (usually starting at $1,000 U.S.). If you don't have that type of money, there is an inexpensive alternative called MilkShape, which is used by both hobbyists and professionals. I'm not going to start a 3D modeling tool war by saying which one is better, because they're all good tools.

Exporting from 3ds max

For our game engine, we'll export data directly from 3ds max using the ASCII export file format. Although 3ds max supports many different formats including 3DS, MAX, and many more, the ASE format is perhaps the easiest format to work with because it's straight text and uses keywords to specify information. The following list gives some of the keywords used to specify the data in the format:

```
*GEOMOBJECT
*MESH_VERTEX
*MESH_FACE
*MESH_TVERT
*MESH_TFACE
```

As you can see by the keywords listed, the format is quite simple. If you use 3ds max 3.x or higher, the ASE export filter comes natively with the software. To use the filter with an already created model, simply choose the Export option from the File menu.

When the Export dialog box displays on the screen, change the filter to the ASCII Scene Export option as seen in Figure 15.1.

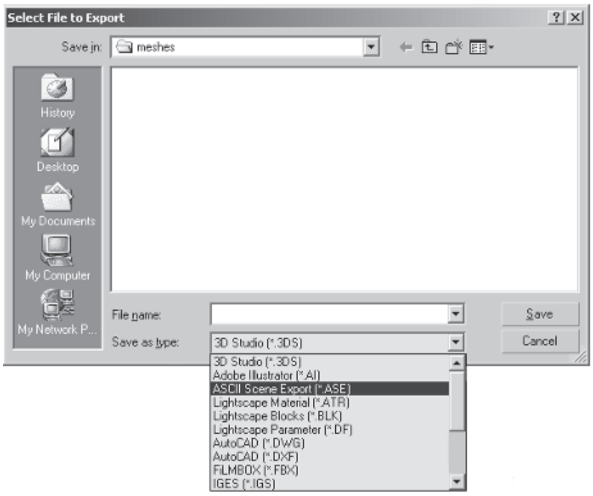
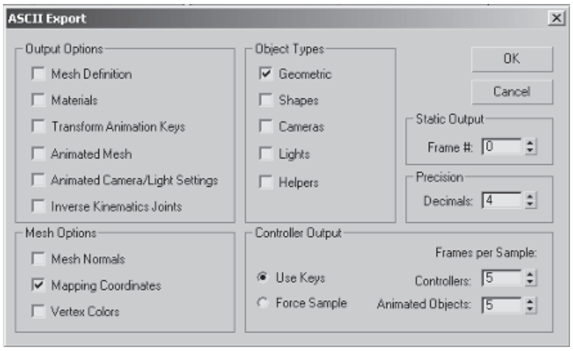


Figure 15.1: 3ds max Export dialog with the ASE filter selected

After selecting the filename and clicking Save, the ASE Export dialog will appear, requesting information about which features to export. Since we're only interested in modeling information, we'll uncheck all the options except Geometric and Mapping Coordinates as seen in Figure 15.2.


ASCII Export

Output Options
- [ ] Mesh Definition
- [ ] Materials
- [ ] Transform Animation Keys
- [ ] Animated Mesh
- [ ] Animated Camera/Light Settings
- [ ] Inverse Kinematics Joints

Mesh Options
- [ ] Mesh Normals
- [x] Mapping Coordinates
- [ ] Vertex Colors

Object Types
- [x] Geometric
- [ ] Shapes
- [ ] Cameras
- [ ] Lights
- [ ] Helpers

Controller Output
- ● Use Keys
- ○ Force Sample

Frames per Sample:
Controllers: 5
Animated Objects: 5

Static Output
Frame #: 0

Precision
Decimals: 4

OK
Cancel

Figure 15.2: The ASE Export dialog box with the export options selected

Once you click the OK button, the data will be written to the file and we're ready to write the code to import this file format.

Loading ASE Models

Since the ASE file is straight text, we can easily parse the information from the file to make a model. After loading the text from the file into a buffer, we must count the number of objects by counting the instances of the keyword *GEOMOBJECT, which is the identifier for each object. Since we don't want to hardcode the number of objects in our models, we'll count the number of objects to allocate memory for in an array of objects.

After allocating the memory for the object array, we begin parsing the information from the text buffer using string tokens. The first keyword we'll encounter while parsing the text buffer is *MATERIAL_COUNT, which specifies the number of materials (textures) the file uses. The next token is the integer number of materials. When the keyword is specified we allocate memory for all the materials in the model. I've always found it weird that there is a material count keyword but nothing for the geometry. Depending on how you want to implement your model structure, you may want to take advantage of graphics layers for multitexturing. Each material can have textures specified for the ambient, diffuse, etc., values, allowing multiple textures to be assigned to one material. Since we're making a basic parser, we'll only use the first *BITMAP keyword for the given material. Then the bitmap filename token is specified and we'll parse it to get just the filename so we can easily get the texture ID later on. After the material properties have been parsed, we'll enter the geometry stage of the file. The keyword *GEOMOBJECT is the new object identifier. When we come across this string, we simply switch to the next object and NULL the new object's arrays.

When we encounter the keyword *MESH_NUMVERTEX, the object's number of vertices is being defined, which is specified as the next token. Obviously we'll use the next token to set the number of vertices for the current object and allocate memory for the number of vertices required. The keyword *MESH_NUMFACE specifies the number of triangles in the current object. When the number of triangles is specified, we allocate memory for the triangle array so we can store triangle data, which is about to be read from the text buffer. A vertex is specified with the keyword *MESH_VERTEX. The first token is the vertex index for the array and the next three values are the X, Z, and Y (yes, X, Z, Y) coordinates in float form. A sample vertex line is shown here:

*MESH_VERTEX	0	–111.7155	–0.0000	–120.0837
Keyword	Index	X	Z	Y

The triangle data is specified using the keyword *MESH_FACE. Unlike the vertex line information, the triangle data is specified with colons for the different points in the triangle. The first token is the triangle index in the array to store the data. The second token specifies point A (point 1) of the triangle, with the vertex reference as the next token after that. Point 2 and point 3 of the triangle are specified in the same manner. A sample triangle line is shown here:

*MESH_FACE	0:	A:	0	B:	2	C:	3
Keyword	Index		P1		P2		P3

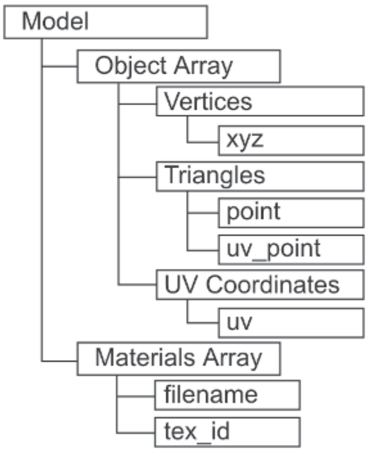
With the vertices and triangles specified we could draw the model without any texture coordinates, but what would the fun be in that? If we're going to take the time to load the basic information, we can certainly take a few extra moments to discuss important features of the ASE format such as UV coordinates. The texture coordinates in the ASE file format are stored in a similar manner to the standard geometry. First the keyword *MESH_NUM_TVERTEX is specified to set the number of UV coordinates in the array, allowing us to allocate the appropriate memory. Next we read the UV data itself, which is specified in the *MESH_TVERT keyword. The token of the keyword is the vertex index for the array, while the next three tokens are the U, V, and W coordinates for texturing. Although we're not interested in the W coordinate, it's good to know if you ever need it! A sample UV coordinate line is shown here:

*MESH_TVERT	0	0.0000	0.0000	0.0000
Keyword	Index	U	V	W (N/A)

Since the UV coordinates are stored in an array (like the vertices), the keyword *MESH_TFACE is used to store the UV references for each triangle point. The UV array references will be stored inside the triangle itself, rather than creating another structure to do the same thing. When the *MESH_TFACE keyword is specified, the next token is the current triangle reference, and the next three tokens are the UV references for the different points of the triangle. A sample textured triangle line is shown here:

*MESH_TFACE	3	11	10	8
Keyword	Index	P1	P2	P3

The final keyword is *MATERIAL_REF, which is the material reference number for the object. Obviously this is important since it allows each object to have different textures assigned to it. It's pretty amazing that these simple keywords can help bring an entire game to life, especially considering the source code involved in this process is very small (~250 lines). The following chart shows a breakdown of the different types and how they are stored to produce the model.



To begin coding, the first thing we'll do is create a new class called MODEL. Inside the class, we'll create 11 constants, which will contain the keywords we'll use to load the file format. To make life easy when coding this ASE loader, all keywords will use the format KEYWORD_function. In the case of a new object, the keyword for *MESH_GEOMOBJECT would be KEYWORD_OBJECT. This will help reduce the chance of mistakes using the different keywords. After creating the keyword constants, we can begin creating the structures to store the data. Luckily there are only five small structures that need to be created. The first structure (called MODEL_MATERIAL) contains the material information. Within the structure there is a string (called filename) and a long (called tex_id). The next string is the vertex structure (MODEL_VERTEX), which has an array (of three) called xyz that is of the GLfloat type.

The UV coordinates are stored in a structure called MODEL_UV_COORD and have one array, called uv, that is of the GLfloat type. The triangle structure (MODEL_TRIANGLE) has two arrays (point and uv_point), which are both arrays of the long type. The object structure (MODEL_OBJECT) contains four long variables (max_triangles, max_vertices, max_uv_coord, and material) and three pointer arrays of the other three structure types called vertex (MODEL_VERTEX type), triangle (MODEL_TRIANGLE type), and uv_coord (MODEL_UV_COORD). By allowing the model to contain multiple objects, we could in theory allow certain objects to be culled from the scene because they are behind other objects, change textures on the fly for animated effects, and more. You could also allow certain parts of the body to be shot/blown off by simply having the arm fall to the ground and not move anymore. With the structures created we'll declare a pointer array of the MODEL_OBJECT type called mdl_object, another array of the MODEL_MATERIAL type called mdl_material, and two long local variables called max_objects and max_materials to set the number of the respective arrays.

In the constructor of the MODEL class, we'll set the object variable to NULL and set the number of objects to 0. Next, we'll create a method called Load, which returns a bool type (true = success, false = failure). Before we begin parsing the ASE information we first open the input file, then calculate the filesize to allocate memory for our text buffer. Then we read the file text into the text buffer. Next we count the occurrences of the KEYWORD_OBJECT constant in the text buffer, allocate memory for the objects, and begin parsing the data. I created a function called CountStrings, which returns the number of occurrences of a specified string. Once the model information has been parsed, we release the memory allocated from the buffer variable. Remember when we begin counting the object numbers to set the local variable to –1 and not 0, because once we hit the KEYWORD_OBJECT constant we'll increase the value of the current object, which would make the value 0 and not 1. The source code to load the model is shown below.

```
bool MODEL::Load(char *filename)
{
   FILE *fp;
   long filesize;
   char *buffer;
   char *tmp, token[]      = " \\\":\t\r\n";
   long cur_material       = -1;
   long cur_object         = -1;

   Release();

   fp = fopen (filename, "rb");
   if (fp == NULL) return (false);
      fseek (fp, 0, SEEK_END);
      filesize = ftell(fp);
      buffer = new char[filesize+1];

      fseek (fp, 0, SEEK_SET);
      fread (buffer, 1, filesize, fp);
   fclose (fp);

   max_objects = CountStrings (buffer, KEYWORD_OBJECT);
   mdl_object = new MODEL_OBJECT[max_objects+1];

   tmp = strtok(strdup(buffer), token);
   while (tmp != NULL)
   {
      if (strcmp(tmp, KEYWORD_MATERIAL_COUNT) == 0)
      {
         max_materials     = strtol(strtok(NULL,token), NULL, 10);
```

```
        mdl_material     = new MODEL_MATERIAL[max_materials+1];
     }
     else if (strcmp(tmp, KEYWORD_BITMAP) == 0)
     {
        cur_material++;
        strcpy (mdl_material[cur_material].filename, strtok (NULL, token));
     }
     else if (strcmp(tmp, KEYWORD_OBJECT) == 0)
     {
        cur_object++;

        mdl_object[cur_object].max_vertices      = 0;
        mdl_object[cur_object].max_triangles     = 0;
        mdl_object[cur_object].max_uv_coord      = 0;
        mdl_object[cur_object].vertex            = NULL;
        mdl_object[cur_object].triangle          = NULL;
        mdl_object[cur_object].uv_coord          = NULL;
     }
     else if (strcmp(tmp, KEYWORD_NUM_VERTICES) == 0)
     {
        mdl_object[cur_object].max_vertices = strtol(strtok(NULL,token),
                   NULL, 10);
        mdl_object[cur_object].vertex      = new MODEL_VERTEX
                   [mdl_object[cur_object].max_vertices+1];
     }
     else if (strcmp(tmp, KEYWORD_NUM_TRIANGLES) == 0)
     {
        mdl_object[cur_object].max_triangles = strtol(strtok(NULL,token),
                   NULL, 10);
        mdl_object[cur_object].triangle = new MODEL_TRIANGLE
                  [mdl_object[cur_object].max_triangles+1];
     }
     else if (strcmp(tmp, KEYWORD_VERTEX) == 0)
     {
        long cur_vertex = strtol(strtok(NULL,token), NULL, 10);

        sscanf (strtok(NULL,token), "%f", &mdl_object[cur_object].vertex
               [cur_vertex].xyz[0]);
        sscanf (strtok(NULL,token), "%f", &mdl_object[cur_object].vertex
               [cur_vertex].xyz[2]);
        sscanf (strtok(NULL,token), "%f", &mdl_object[cur_object].vertex
               [cur_vertex].xyz[1]);
     }
     else if (strcmp(tmp, KEYWORD_TRIANGLE) == 0)
     {
        long cur_tri = strtol(strtok(NULL,token), NULL, 10);

        strtok(NULL, token);
        mdl_object[cur_object].triangle[cur_tri].point[0] = strtol(strtok
                  (NULL,token), NULL, 10);

        strtok(NULL, token);
        mdl_object[cur_object].triangle[cur_tri].point[1] = strtol(strtok
                  (NULL,token), NULL, 10);

        strtok(NULL, token);
        mdl_object[cur_object].triangle[cur_tri].point[2] = strtol(strtok
                  (NULL,token), NULL, 10);
     }
     else if (strcmp(tmp, KEYWORD_NUM_TVERTEX) == 0)
     {
        mdl_object[cur_object].max_uv_coord = strtol(strtok(NULL,token),
                  NULL, 10);
        mdl_object[cur_object].uv_coord = new MODEL_UV_COORD[mdl_object
                  [cur_object].max_uv_coord+1];
     }
     else if (strcmp(tmp, KEYWORD_TVERTEX) == 0)
     {
        long cur_vertex = strtol(strtok(NULL,token), NULL, 10);

        sscanf (strtok(NULL,token), "%f", &mdl_object[cur_object].uv_coord
               [cur_vertex].uv[0]);
        sscanf (strtok(NULL,token), "%f", &mdl_object[cur_object].uv_coord
               [cur_vertex].uv[1]);
     }
     else if (strcmp(tmp, KEYWORD_TFACE) == 0)
     {

        long cur_tri = strtol(strtok(NULL,token), NULL, 10);

        mdl_object[cur_object].triangle[cur_tri].uv_point[0] = strtol(strtok
                  (NULL,token), NULL, 10);
        mdl_object[cur_object].triangle[cur_tri].uv_point[1] = strtol(strtok
                  (NULL,token), NULL, 10);
        mdl_object[cur_object].triangle[cur_tri].uv_point[2] = strtol(strtok
                  (NULL,token), NULL, 10);
     }
     else if (strcmp(tmp, KEYWORD_MATERIAL_REF) == 0) mdl_object
              [cur_object].material = strtol(strtok(NULL,token), NULL, 10);

     tmp = strtok(NULL, token);
  }

  delete [] buffer;
  buffer = NULL;

  return (true);
}
```

Releasing Model Data

Since we are dealing with allocated memory, we'll create a method called Release to allow us to easily release all the memory allocated for the model. The method will loop through the entire model structure, releasing the memory for every pointer array. The source code for our Release method is given below.

```
void MODEL::Release()
{
   if (max_objects > 0)
   {
      for (long obj = 0; obj < max_objects; obj++)
      {
         if (mdl_object[obj].max_vertices > 0) delete [] mdl_object
                [obj].vertex;
         if (mdl_object[obj].max_triangles > 0) delete [] mdl_object
                [obj].triangle;
         if (mdl_object[obj].max_uv_coord > 0) delete [] mdl_object
                [obj].uv_coord;
      }
      delete [] mdl_object;
      mdl_object = NULL;
      max_objects = 0;
   }

   if (max_materials > 0)
   {
      delete [] mdl_material;
      mdl_material = NULL;
      max_materials = 0;
   }
}
```

With the back-end source code written to load the ASE format, we'll move back to the main source file and create a new function called LoadModel. This function will load the specified model, then loop through the materials, getting the texture IDs for each material. If we didn't assign the texture IDs for each material specified, the model wouldn't have any texture assigned to it, which isn't terribly useful. The source code for the LoadModel function is provided below.

```
bool LoadModel (char *filename, MODEL & model)
{
   if (!model.Load(filename)) return (false);

   for (long mat = 0; mat < model.max_materials; mat++)
   {
      model.mdl_material[mat].tex_id = GetTextureID(model.mdl_material
               [mat].filename);
   }
   return (true);
}
```

In our example we'll load the file item_gun.ase, which is located in the media directory of the downloadable companion files. This file contains a rather basic gun, which we'll use as an item in our game. I suggest placing the LoadModel function call right after loading the map so there is a standard pattern for loading data.

Drawing Models

The drawing of each model will be done through the function RenderModel. Similar to the map rendering loop, to render the model data we simply loop through the list of objects where we bind the object's texture, then begin to loop through triangles. With each iteration of the triangle loop we specify the three UV coordinates as well as vertex points for the triangle. It's that simple! The source code for our RenderModel function is shown here:

```
void RenderModel(MODEL model)
{
   for (long obj = 0; obj < model.max_objects; obj++)
   {
      glBindTexture (GL_TEXTURE_2D, TextureName[model.mdl_object
                     [obj].material]);
      glBegin (GL_TRIANGLES);
      for (long tri = 0; tri < model.mdl_object[obj].max_triangles; tri++)
      {
         long p1    = model.mdl_object[obj].triangle[tri].point[0];
         long p2    = model.mdl_object[obj].triangle[tri].point[1];
         long p3    = model.mdl_object[obj].triangle[tri].point[2];
         long uv1   = model.mdl_object[obj].triangle[tri].uv_point[0];
         long uv2   = model.mdl_object[obj].triangle[tri].uv_point[1];
         long uv3   = model.mdl_object[obj].triangle[tri].uv_point[2];


         glTexCoord2fv (model.mdl_object[obj].uv_coord[uv1].uv);
         glVertex3fv (model.mdl_object[obj].vertex[p1].xyz);

         glTexCoord2fv (model.mdl_object[obj].uv_coord[uv2].uv);
         glVertex3fv (model.mdl_object[obj].vertex[p2].xyz);

         glTexCoord2fv (model.mdl_object[obj].uv_coord[uv3].uv);
         glVertex3fv (model.mdl_object[obj].vertex[p3].xyz);
      }
      glEnd();
   }
}
```

Putting this knowledge to good use, we'll create a function called RenderItems, which simply loops through the items in the map, drawing them at a specific location. To draw an object at a specific location, we simply save the current matrix, call glTranslate to move to the desired location, then draw the item using RenderModel. When finished rendering the model in question, simply restore the saved matrix and restart the loop. The source code for the RenderItems function is shown below.

```
void RenderItems()
{
   for (long itm = 0; itm < map.header.max_items; itm++)
   {
      glPushMatrix();
         glTranslated (map.item[itm].xyz[0], map.item[itm].xyz[1],
                       map.item[itm].xyz[2]);
         RenderModel( item_gun );
      glPopMatrix();
   }
}
```

To make the items display on the screen we simply add a call to the RenderItems function in the RenderMap function.

Chapter Example

Please see the example from the companion files (ex15_1).

Conclusion

In this chapter we learned how to load and draw a model exported from 3ds max in the ASCII Export file format into our game. As an alternative to using our map editor, you may want to consider writing your own ASE loading code to make levels. Many professionals use 3ds max and other content creation tools as opposed to creating their own software to save time and money, and for general ease of use.

This page intentionally left blank.

Chapter 16

Beyond OpenGL 1.1

When OpenGL was originally created, its engineers added the functionality to use vendor-created extensions to the original API. This allowed different hardware vendors the opportunity to add their own special features for their hardware without having proprietary features in the API. A group of hardware vendors, including 3Dlabs, ATI, nVIDIA, Matrox, and many more, shapes how future versions of OpenGL look by voting on which GL extensions should be part of the core API. Normally when an extension is first created the name of the extension begins with GL_EXT or the hardware manufacturer's name, for example NV_EXT for nVIDIA. Every six months the members of the ARB get together and discuss the future of OpenGL. Often within the meetings they'll vote on which extensions to add and how they should be adjusted to meet ARB standards. If a particular extension wins the vote to be included into the API, the name uses the ARB prefix GL_ARB to indicate it's an ARB-sanctioned extension.

In the case of the multitexturing extension GL_EXT_MULTITEXTURE, the name changed to GL_ARB_MULTITEXTURE when it was voted in by the ARB. Although OpenGL only has a few versions of the API (currently at 1.5), there are well over 150 extensions in the OpenGL Extension Registry located on SGI's web site. This registry allows developers to easily find information about each extension, including who created it and the added functions/constants to the API needed to make it work. When designing new software packages it's a great place to visit to get ideas of which extensions to support.

One of the biggest problems with OpenGL, albeit an annoyance rather than a major issue, is that there are no updated libraries available. When a developer begins writing OpenGL code, he or she is using the same library introduced in the mid-90s using the 1.1 standard. This appears to make OpenGL outdated because the libraries haven't been updated in years, but it's quite the contrary. Although the libraries shipped with Microsoft Visual C++ 6.0/.NET don't come with the updated OpenGL headers (which they should) we can use any function in the newer 1.2, 1.3, 1.4, and even 1.5 versions by checking the availability of the function, then dynamically loading it from the video card drivers.

To use the new features of OpenGL or any extensions, we must add several new OpenGL headers to our project. The files GL_EXT.H and WGL_EXT.H contain updated constants, API functions, and more for the newest versions of OpenGL (currently 1.5). Although OpenGL is available on numerous platforms, the WGL_EXT.H header contains proprietary extensions/versions of extensions for the Windows platform. Because of the design decisions used when creating OpenGL, some of these new extensions can make all the difference to performance and visual quality. Some extensions are actually standard cross-platform extensions that use the Win32 API for proper configuration. A good example of this is the GL/WGL_ARB_MULTISAMPLE extension, which is used for antialiasing primitives drawn on the screen. Although the extension is supported on multiple platforms, the Windows version requires that you initialize OpenGL, query the video card for support, and then redefine the pixel descriptors to support the updating multisampling values. Since each platform works differently in its initial setup of OpenGL, there are different steps required for the extension to work, and therefore this version of the extension should be classified as a Windows-specific (hence the WGL) extension.

To query information from the video card/OpenGL driver about the type of video card (GL_RENDERER), hardware manufacturer (GL_VENDOR), OpenGL version supported (GL_VERSION), and the supported extensions (GL_EXTENSIONS), we use the glGetString command and specify the appropriate GL value or values. The returned value is a NULL-terminated string that indicates the values requested. Each extension is separated by a space. If you query the information from a computer that doesn't have a video card with OpenGL drivers, you'll most likely receive the string "Microsoft Generic" as the vendor. This value is returned when Microsoft's software implementation is used to draw OpenGL. If you were to design a complex 3D game engine, you may want to exit the game with an error if an OpenGL software implementation is found, because of potential performance or reliability issues that may occur.

As a side note, Microsoft has stopped shipping the software drivers with the newer versions of its operating systems (i.e., Windows XP). If you are seriously interested in a software implementation of OpenGL for Windows, you may want to research the Windows version of Mesa that has a software implementation. Of course Mesa and software implementations are beyond the scope of this book, but the Mesa web site is included in the Sources section of this book.

Once the extension information has been saved to a string, we simply see if the desired extension is in the list. In the event the extension name is in the list, we load function stubs from the driver so we can easily run the new extension when needed. When using OpenGL extensions it's a good idea to declare the extension function stubs as global variables so you can use them anywhere in your application/game.

To begin coding we'll create a new function called LoadGLExtensions, which as the name implies will load the GL extensions for the new functionality we want to support. If any essential extensions cannot be loaded, we'll exit with a failure; otherwise, a default return value of true will be returned. Before we begin loading any extensions we'll get the list of extensions from the video card and store them in a local string so we can easily check which extensions are supported and which ones aren't. If the function returns with a failure, we'll display an error message and exit the software (remembering to release OpenGL and restore the graphics settings first). If the extension is an essential part of the game engine (e.g., multitexturing) and the video card doesn't support it, there's no reason to continue because the software could potentially crash.

Texturing

Many of today's video cards support great new texturing features such as anisotropic filtering (GL_EXT_texture_filter_anisotropic), irregular texture sizes (GL_ARB_texture_non_power_of_two), multitexturing (GL_ARB_multitexture), and texture compression (GL_ARB_texture_compression). These extensions are only a few of the extension features available with video hardware.

Anisotropic Filtering (GL_EXT_texture_filter_anisotropic)

In Chapter 11 we learned about the different types of texture filtering styles (i.e., linear and nearest) that OpenGL supports. Although linear filtering does a decent job of drawing textures, a newer filtering calculation was added through the extension GL_EXT_texture_filter_anisotropic. This new extension allows us to use anisotropic filtering when texturing images. The calculation for anisotropic filtering is much more intense than the original linear calculations; however, the visual results are far superior. When rendering a scene using linear filtering, images (especially mipmaps) will become blurred. The anisotropic calculation produces far less blur, giving the textures a nice, clear look. This extension is great to use if you want the best visual quality on the screen with almost no coding overhead.

Since this feature isn't vital and can potentially hurt performance, we're not going to make it a standard feature. Instead we'll allow the user to enable or disable the feature through a check box (resource ID IDC_ANISOTROPIC) in the dialog box that appears at the beginning of the game. When the dialog closes we set the value of our newly declared global use_anisotropic to true or false, based on the check box state. When we begin loading GL extensions, if the anisotropic extension is supported, then we keep use_anisotropic as true; otherwise we set it to false. The extension is used when loading textures, so it would have to be done in both LoadTexture and LoadTransparentTexture.

If anisotropic filtering is enabled, it is a good idea to get the highest level of anisotropy (the highest calculation value the video card can use) using glGetFloatv with the parameter GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT. Each video card has different maximum values for anisotropy and we don't want to assume any general numbers. The second parameter of glGetFloatv returns the value requested. Using glTexParameterf, we specify the type of texture (GL_TEXTURE_2D), the anisotropic filtering (GL_TEXTURE_MAX_ANISOTROPY_EXT), and finally the max value. This will add the anisotropic filtering calculation to every texture, which will reduce the blurred texture effect. The source code for the updated LoadTexture function is written below. Remember that both functions have been updated to support anisotropic filtering.

```
void LoadTexture(char *filename, long texture_id, long mag_filter, long
         min_filter, long wrap_type)
{
   TEXTURE texture;
   long    internal_format;

   if (!texture.Load (filename)) return;
      glBindTexture (GL_TEXTURE_2D, TextureName[texture_id]);

      if (use_texture_compression) internal_format = GL_COMPRESSED_RGBA_ARB;
      else internal_format = GL_RGBA;
      glTexImage2D (GL_TEXTURE_2D, 0, internal_format, texture.info_
               header.biWidth, texture.info_header.biHeight, 0, GL_RGB,
               GL_UNSIGNED_BYTE, texture.data);
      gluBuild2DMipmaps (GL_TEXTURE_2D, internal_format, texture.info_
               header.biWidth, texture.info_header.biHeight, GL_RGB,
               GL_UNSIGNED_BYTE, texture.data);

      glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, mag_filter);
      glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, min_filter);
      glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap_type);
      glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap_type);
      if (use_anisotropic)
      {
         float anisotropy;
         glGetFloatv (GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &anisotropy);
         glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT,
                  anisotropy);
      }
      glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   texture.Release();
}
```

Irregular Texture Sizes (GL_ARB_texture_non_power_of_two)

In Chapter 11, we learned how to load and use textures in OpenGL. Using power-of-two textures was the standard for many years until the extension GL_ARB_texture_non_power_of_two was approved in 2003 by the ARB. This new extension allowed OpenGL to accept non-power-of-two textures, mipmaps, cubemaps, and more. As you can tell, this adds a great deal of flexibility to the development of a game because you can let your artists create a texture any size they'd like without having to worry about scaling or stretching it to conform to the power-of-two standard. To make life even easier, there are no added constants or functions needed to use this extension! You simply query the extension list for the extension name GL_ARB_texture_non_power_of_two. If the value is present, then we can use irregular textures; otherwise we cannot. It's that simple!

To use this extension in our game engine we'll simply add a new non-power-of-two bitmap called npot_hud.bmp to the texture list. Next we'll update our Render function to check for the GL_ARB_texture_non_power_of_two extension. If the extension is found, then we'll bitblit the npot_hud.bmp file to the screen. If the extension isn't found in the extension list, then we'll bitblit texture 1 (the original HUD) to the screen as we've done in the previous chapters. If you intend on using this extension in a game with more than one or two textures, you may want to consider writing an algorithm to check for the extension and, if it's not present, automatically scale the images to a power-of-two value rather than storing two different versions. If you keep two different versions of the textures, you may find that you'll take up much more space than you originally anticipated, which could be avoided with a few extra seconds (per texture) of calculation.

The updated Render function source code with GL_ARB_texture_non_power_of_two extension support is provided below.

```
void Render()
{
   glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

   glPushMatrix();
      glRotatef (player.angle[0], 0.0, 1.0, 0.0);
      glTranslated (player.xyz[0], player.xyz[1], player.xyz[2]);
      RenderMap();
   glPopMatrix();

   glEnable (GL_BLEND);
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

```
    if (strstr(gl_extension, "GL_ARB_texture_non_power_of_two") != NULL)
            glBitBlt(GetTextureID("npot_hud.bmp"), 0,240,640,240);
    else glBitBlt(1, 0,240,640,240);

    SwapBuffers(raster.hDC);
}
```

Multitexturing (GL_ARB_multitexture)

Before the adoption of multitexturing, game developers would have to re-render scenes several times (normally called passes) using a new texture/blending feature for each pass to accurately draw the scene. The big problem with re-rendering scenes or objects is that we lower the performance because we have to render more objects in the scene. To solve this issue, an OpenGL extension (originally called GL_EXT_multitexture) was created to give the developer the ability to use multiple textures or UV coordinates at once.

Rather than render an object two, four, or even eight times, we can query the number of texture layers supported, then set each layer's texture and UV coordinates at once.

Depending on the hardware, some video cards will render the specified layers in one pass and some will automatically break it up into several quick passes done by the hardware.

When the original extension was released, most graphics accelerators supported only two texture layers. The high-end graphics hardware being released at the time of publication supports from 16 to 32 layers. This adds a tremendous amount of customization because you can use as many or as few texture layers as you want. Some objects may use 24 and some may only use two.

In 1998, the ARB added the GL_SGIS_multitexture extension directly into the OpenGL 1.2.1 API specification (now calling it GL_ARB_multitexture) as core functionality. Any OpenGL driver that supports version 1.2.1 must have the multitexturing functions in it to conform to the 1.2.1 specification. The new functions added to the specification are: glActiveTextureARB, glClientActiveTextureARB, and glMultiTexCoord[1/2/3/4][i/f/s/v]. Before we bind the necessary texture for use, we must activate the texture layer for use (using glActiveTextureARB) so the OpenGL driver knows which texture layer to place the bound texture into. The glext.h header file contains the texture layer constants (GL_TEXTURE0_ARB through GL_TEXTURE31_ARB) that can be activated. Although there are 32 constants to potentially use, you should always ensure the hardware supports the number of layers you need. If you try to reference a layer that doesn't exist, the driver may become unstable or crash.

Once each texture layer has been initialized we begin the rendering loop to render each object. In the past we used the function glTexCoord to specify the UV coordinates for the objects. When using the multitexturing extension we use the glMultiTexCoord function, specifying the texture layer as the first parameter and the UV coordinates as the next parameters. As you can see, it's very similar to the original method with the exception of specifying the texture layer, which is a must for use of multitexturing! To use the multitexturing extension we'll declare the global function stubs (glActive-TextureARB and glMultiTexCoord2fARB) so we can load each extension. The function types are PFNGLACTIVETEXTUREARBPROC and PFNGLMULTITEXCOORD2FARBPROC.

We're not going to load every version of the glMultiTexCoord function, because we're not going to use them to texture the objects. If for some reason you wanted to use one of the other versions of the function, you would simply declare the appropriate function as a global and load it from the driver. Like the point parameters extension, we must load the function addresses from the driver using wglGetProcAddress. If the extension isn't found in the list, then we'll exit the function returning false, because multitexturing is an essential function for our game engine. The updated source code for the LoadGLExtensions function is given below.

```
bool LoadGLExtensions()
{
   gl_extension = (char *)glGetString(GL_EXTENSIONS);

   if (strstr(gl_extension,"GL_ARB_point_parameters") != NULL)
   {
        glPointParameterfARB = (PFNGLPOINTPARAMETERFARBPROC)wglGetProcAddress
                  ("glPointParameterfARB");
        glPointParameterfvARB = (PFNGLPOINTPARAMETERFVARBPROC)wglGetProcAddress
                  ("glPointParameterfvARB");
   }

   if (strstr(gl_extension,"GL_ARB_multitexture") != NULL)
   {
      glActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC)wglGetProcAddress
                  ("glActiveTextureARB");
      glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)wglGetProcAddress
                  ("glMultiTexCoord2fARB");
   }
   else return (false);

   return (true);
}
```

With the extension stubs now loaded from the video card we can begin updating our rendering source code. Unlike previous examples where we've been able to add several new lines of functionality to the engine, here we'll be adding and replacing source code in many different areas. To update the rendering process we'll first edit the main rendering loop for the objects in the RenderMap function. Since we want to use multitexturing in our game, we'll create a loop from 0 to the value of MAX_TEXTURE_LAYERS, where we'll call glActiveTexture and specify the layer number (GL_TEXTURE0_ARB + loop number), then we'll use our regular glBindTexture function to bind the texture to the current texture layer. Using this method instead of hardcoding the values will allow us to add more texture layers (by increasing MAX_TEXTURE_LAYERS) without having to update our source code. With every iteration of the loop, the texture layer is binding the associated texture to itself, which will then be used when we draw the object. The code snippet below displays the updated rendering loop for the game.

```
for (long obj = 0; obj < map.header.max_objects; obj++)
{
   glColor3f (1.0f, 1.0f, 1.0f);

   for (long layer = 0; layer < MAX_TEXTURE_LAYERS; layer++)
   {
      glActiveTextureARB (GL_TEXTURE0_ARB + layer);
      glBindTexture (GL_TEXTURE_2D, TextureName[map.object[obj].texture
               [layer].id]);
   }

   RenderObject ( obj );
}
```

When we wrote the code to assign texture numbers to each object we included each layer, so both layers of each object will automatically be assigned its texture number. This assumes the texture specified is in the texture list; otherwise the texture ID 0 (full-intensity white) will be returned. With the code updated to bind multiple textures to a single object, we update our RenderObject function. Besides binding each texture to its own layer, the UV coordinates must be set for each layer. This means that we'll have to update our RenderObject function to specify the UV coordinates for each layer. Before specifying each vertex, we'll loop to the number of texture layers (MAX_TEXTURE_LAYERS) and specify the UV coordinates for each layer using the function glMultiTexCoord.

To keep things consistent with the code we've already written, we'll use the glMultiTexCoord2f version of the command. This version of the command has three parameters. The first is the texture layer number (we'll use GL_TEXTURE0_ARB + loop number), and the other two are the UV coordinates for the layer. Each of the three glTexCoord calls should be updated to coincide with this new way of assigning UV coordinates. Once the three coordinates are specified, we've finished the code necessary to use multitexturing in our game engine. The code for the RenderObject function is provided here.

```
void RenderObject(long cur_obj)
{
   glBegin (GL_TRIANGLES);
   for (long tri = 0; tri < map.object[cur_obj].max_triangles; tri++)
   {
      long vertex_1 = map.object[cur_obj].triangle[tri].point[0];
      long vertex_2 = map.object[cur_obj].triangle[tri].point[1];
      long vertex_3 = map.object[cur_obj].triangle[tri].point[2];

      for (long layer = 0; layer < MAX_TEXTURE_LAYERS; layer++)
              glMultiTexCoord2fvARB (GL_TEXTURE0_ARB+layer, map.object
              [cur_obj].triangle[tri].uv[layer].uv1);
      glVertex3dv (map.object[cur_obj].vertex[vertex_1].xyz);

      for (layer = 0; layer < MAX_TEXTURE_LAYERS; layer++) glMultiTexCoord2fvARB
              (GL_TEXTURE0_ARB+layer, map.object[cur_obj].triangle[tri].uv
              [layer].uv2);
      glVertex3dv (map.object[cur_obj].vertex[vertex_2].xyz);

      for (layer = 0; layer < MAX_TEXTURE_LAYERS; layer++) glMultiTexCoord2fvARB
              (GL_TEXTURE0_ARB+layer, map.object[cur_obj].triangle[tri].uv
              [layer].uv3);
      glVertex3dv (map.object[cur_obj].vertex[vertex_3].xyz);

   }
   glEnd();
}
```

After rendering the objects in the map, it's a good idea to set the active texture to layer 0 to keep things compatible with our texturing code throughout the engine. If we set the active texture to layer 0, then functions such as our glBitBlt function won't need to be changed to accommodate the use of multitexturing.

Texture Compression (GL_ARB_texture_compression)

When writing 3D video games, one method of lowering the texture memory usage and optimizing rendering is to use texture compression. The extension GL_ARB_texture_compression is a framework for texture compression in OpenGL. If texture compression is specified when loading each texture, the video card will compress the texture data automatically using the generic texture compression algorithm. When compressed, the textures will take up less texture memory (sort of like zipping a texture in memory) and therefore takes less time/bandwidth for use.

Some graphics vendors supply proprietary compression algorithms (such as S3TC, FXT1, etc.) in their cards, which can be queried and used to compress the texture data. Although these proprietary algorithms are great for compression, using the generic compression algorithm allows you to write cross-platform code without having any issues. The new texture compression extension adds several new functions including glCompressedTexImage [1/2/3]DARB, glCompressedTexSubImage[1/2/3]DARB, and glGetCompressedTexImageARB. The glCompressedTexImage function is used to define an already-compressed texture, the glCompressedTexSubImage function defines an already-compressed texture subimage, and the glGetCompressedTexImageARB function gets compressed image data from the video card. To have the texture data stored in the video RAM, we change the internal format from the standard GL_RGB/A to GL_COMPRESSED_RGB/A_ARB. This will compress the texture data using the generic texture compression algorithm. It can be used in straight glTexImage calls or with auto-mipmap generation functions as well.

In the event we wanted to query the number of textures or get compressed texture data from the video card we would load the functions from the driver. However, we're only interested in storing the data in compressed form, which can be done using the generic constant listed in the extension header file. To update our engine to support texture compression we'll add a check box to our setup dialog box with the label "Use Texture Compression" (ID is IDC_TEXTURE_COMPRESSION) to allow the user to select whether they want to use it. It's a good idea to give users the option because there is a small loss in visual quality, and some gamers prefer quality and others speed. We'll add one new global called use_texture_compression, which is of the bool type. If the box is checked when the dialog closes we'll set the value to true; otherwise it's false. In the LoadGLExtensions function, if GL_ARB_texture_multitexture is supported and use_texture_compression is true, then everything is great! If either value is different, then use_texture_compression should be set to false so we don't attempt to use texture compression.

With the use_texture_compression variable finally set, we can update the LoadTexture and LoadTransparentTexture functions. Inside each function we'll declare a local variable called internal_format, which will store the internal format of the texture RGB/A data. If use_texture_compression is true, then we set internal_format to GL_COMPRESSED_RGB/A_ARB to make the video card compress the data in memory. If the value of use_texture_compression is false, we'll set internal_format to its original value of GL_RGB/A. The internal_format variable will replace the third parameter of glTexImage2D and the fourth parameter of gluBuild2DMipmaps. With these small enhancements, our game engine can now compress textures when they're loaded or keep them in their original state.

Fog

In Chapter 14 we learned how to initialize fog and display a per-vertex calculated fog on the screen. In this section, we examine the new volumetric fog extension.

Volumetric Fog (GL_EXT_fog_coord)

The GL_EXT_fog_coord extension takes fog to the next level, allowing us to specify the depth of the fog on a per-vertex basis, rather than having the video hardware calculate this information for us. This great extension allows us to add fog only to certain objects or vertices, which gives us really amazing effects. Imagine walking down a stairwell into a dungeon and having mist (fog) just above the floor. This type of fog is called volumetric fog and has been seen in games like Quake 3 Arena by id Software. The fog coordinate extension was promoted to the core API as of OpenGL 1.5. Any video card that supports OpenGL 1.5 will support this feature; otherwise it's available through its original extension.

The fog coordinate extension adds two new functions to the API. The first function, glFogCoord[f/d/v]EXT, specifies the depth for each vertex. This is the function we'll use to add fog depth for each object. The second function, glFogCoordPointerEXT, is used to specify the fog coordinate values when using vertex arrays. Like every extension, before we can begin using it we must first ensure that the extension is supported. In our LoadGLExtensions function we'll check for the extension (GL_EXT_fog_coord) in the extension list. If it's found, then we'll load the glFogCoord function into a global variable (of the PFNGLFOGCOORDFPROC type).

After loading the function stub we must update our RenderObject function to set the fog depths for each vertex, which is where the magic happens. In the rendering loop of RenderObject, before specifying the vertices we'll add a call to glFogCoordfEXT, specifying the vertex's fog_depth value. This will specify the fog depth for each vertex, which will affect how the object is drawn using fog. If we don't want fog on the object, we simply leave the fog depth as 0.0; otherwise, we can change the value appropriately to customize the fog. In our map editor we've coded it to set a generic fog depth for an entire object. Ideally, if you wanted really awesome fog you'd want to modify the map editor to set the fog depth for each vertex, but setting the fog depth on a per-object basis is good enough for our purposes. The updated RenderObject function is written below.

```
void RenderObject(long cur_obj)
{
   glBegin (GL_TRIANGLES);
   for (long tri = 0; tri < map.object[cur_obj].max_triangles; tri++)
   {
      long vertex_1 = map.object[cur_obj].triangle[tri].point[0];
      long vertex_2 = map.object[cur_obj].triangle[tri].point[1];
      long vertex_3 = map.object[cur_obj].triangle[tri].point[2];

      for (long layer = 0; layer < MAX_TEXTURE_LAYERS; layer++)
             glMultiTexCoord2fvARB (GL_TEXTURE0_ARB+layer, map.object
             [cur_obj].triangle[tri].uv[layer].uv1);
      glFogCoordfExt (map.object[cur_obj].vertex[vertex_1].fog_depth);
      glVertex3dv (map.object[cur_obj].vertex[vertex_1].xyz);

      for (layer = 0; layer < MAX_TEXTURE_LAYERS; layer++) glMultiTexCoord2fvARB
             (GL_TEXTURE0_ARB+layer, map.object[cur_obj].triangle[tri].uv
             [layer].uv2);
      glFogCoordfExt (map.object[cur_obj].vertex[vertex_2].fog_depth);
      glVertex3dv (map.object[cur_obj].vertex[vertex_2].xyz);

      for (layer = 0; layer < MAX_TEXTURE_LAYERS; layer++) glMultiTexCoord2fvARB
             (GL_TEXTURE0_ARB+layer, map.object[cur_obj].triangle[tri].uv
             [layer].uv3);
      glFogCoordfExt (map.object[cur_obj].vertex[vertex_3].fog_depth);
      glVertex3dv (map.object[cur_obj].vertex[vertex_3].xyz);
    }
    glEnd();
}
```

Once the RenderObject function has been updated, we simply modify our LoadMap function, more specifically the fog configuration, to tell GL that we are going to specify the fog coordinate sources. To do this we use the constant GL_FOG_COORDINATE_SOURCE_EXT as the first parameter in the call to glFog. When we use GL_FOG_COORDINATE_SOURCE_EXT, we can use one of two constants to specify the fog depth calculation: GL_FOG_COORDINATE_EXT and GL_FRAGMENT_DEPTH_EXT. Each constant slightly alters the calculation to give different fog effects. In our game we'll use the GL_FOG_COORDINATE_EXT constant as seen in the LoadMap source code below.

```
bool LoadMap(char *filename)
{
    if (!map.Open(filename)) return (false);

    for (long obj = 0; obj < map.header.max_objects; obj++)
    {
      for (long tex_layer = 0; tex_layer < MAX_TEXTURE_LAYERS; tex_layer++)
      {
        map.object[obj].texture[tex_layer].id = GetTextureID(map.object
                   [obj].texture[tex_layer].filename);
      }
    }


    if (map.header.use_skybox)
    {
      map.skybox.front.texid    = GetTextureID(map.skybox.front.filename);
```

```
        map.skybox.back.texid     = GetTextureID(map.skybox.back.filename);
        map.skybox.left.texid     = GetTextureID(map.skybox.left.filename);
        map.skybox.right.texid    = GetTextureID(map.skybox.right.filename);
        map.skybox.top.texid      = GetTextureID(map.skybox.top.filename);
        map.skybox.bottom.texid   = GetTextureID(map.skybox.bottom.filename);
    }


    if (map.header.use_fog)
    {
        glFogi (GL_FOG_MODE, map.fog.mode);
        glFogf (GL_FOG_DENSITY, map.fog.density);
        glFogf (GL_FOG_START, map.fog.start);
        glFogf (GL_FOG_END, map.fog.end);
        glFogfv (GL_FOG_COLOR, map.fog.rgba);
        glFogi (GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT);
        glEnable (GL_FOG);
    }
    else glDisable (GL_FOG);


    return (true);
}
```

Windows Extensions

Each operating system has its own unique extensions designed to bring new functionality to that specific operating system, including Microsoft Windows. The extensions for this operating system use the "wgl" naming convention as opposed to the "arb" or "ext" naming conventions for generic extensions.

Swap Control (WGL_EXT_SWAP_CONTROL)

When OpenGL renders each frame, the rendering is synchronized with the vertical synchronization of the screen. Unless it's disabled, this will cap the frame rate at the V-sync rate. For instance, if the V-sync rate is 65 Hz, then we'll only be able to achieve a maximum of 65 frames per second, regardless of the graphics hardware we're using. To fix this, either we can have the user adjust the graphics card settings (if the drivers permit this) to turn off OpenGL V-sync redrawing or we can use the new Windows GL extension Swap Control (WGL_EXT_SWAP_CONTROL). This extension adds two new functions to the API: wglSwapIntervalExt and wglGetSwapIntervalExt. The first function, wglSwapIntervalExt, allows you to set the swap interval for the synchronization. If the value specified is 0, then there is no synchronization, which is what we want. The function wglGetSwapIntervalExt will return the current swap interval value.

When using extensions that have new GL commands, we must declare the function stubs as global variables as mentioned earlier in the chapter. In the swap control extension, we'll declare the two new functions using their stub names (PFNWGLSWAPINTERVALEXTPROC and PFNWGLGET-SWAPINTERVALEXTPROC). Both functions will be set to NULL when declared. With the new function stubs declared, we can check the gl_extension string to ensure the extension (WGL_EXT_SWAP_CONTROL) is supported and load the functions. As mentioned earlier in the chapter, to load the extension from the GL driver we use the function wglGetProc-Address and specify the address of the function to load. The return value is the address of the function, which should be cast to the function stub type as seen in the sample below.

```
wglSwapIntervalExt = (PFNWGLSWAPINTERVALEXTPROC)wglGetProcAddress
         ("WGLSWAPINTERVALEXT");
wglGetSwapIntervalExt = (PFNWGLGETSWAPINTERVALEXTPROC)wglGetProcAddress
         ("WGLGETSWAPINTERVALEXT");
```

After loading the functions, we simply run wglSwapIntervalExt with a parameter of 0 to turn off V-sync. Our game will now run as fast as possible without having to worry about syncing with the vertical refresh rate.

Chapter Example

Please see the example from the companion files (ex16_1).

Conclusion

With all these great new extensions, we've added some wonderful new functionality to our game. There are well over 200 extensions to OpenGL, available on numerous hardware and software platforms. When designing a new application, remember to research which hardware supports the extensions being considered for use to avoid any compatibility issues. Although it can limit your capabilities, it's a good idea to stick with ARB/EXT extensions and avoid using proprietary extensions from video card vendors unless you can replicate the functionality without the feature or by using another proprietary feature. In the next chapter we'll discuss how to write multiplayer source code.

Chapter 17

Multiplayer Gaming

When designing multiplayer games there are many things to take into consideration such as the protocol, port, packet size, and number of players to support. These are common issues to consider because they can drastically alter how the game plays over a LAN or the Internet. We're going to use the Uniform Data Packet (UDP) protocol because it's connectionless and slightly quicker than using TCP since it doesn't send back an acknowledgment. Because UDP doesn't send an acknowledgment we may have some packets that get lost in transmission, which is a cause for concern since packet loss can drastically alter the way games are played.

Like any multiplayer video game, our code will need a client and a server. When the game engine is started the user will choose to be either the client or the server. If the user chooses the client, he'll be prompted for the IP address for the server so the client can connect to the server. If the user chooses the server option, the server code initializes and waits for an incoming connection (as a Windows message) while we walk around the level. When the client connects to the server, the client and server will begin talking back and forth to each other.

There are many ways to write the networking code for games. You can send packets on a timed basis, on a per-movement basis, a combination of the two, and many other ways. Each method has benefits and drawbacks. One method I strongly discourage is sending packets each cycle through the main loop or even on a per-frame basis because of possible lag issues. Imagine your friend bought the latest and greatest video card from the local computer store and you decide to go head to head in a deathmatch game. If your friend's video card is double the speed of yours, he'll be sending you twice the number of packets of his movements. On his end, you'll end up moving in choppy blocks because your video card is only processing half as many frames per second.

One method that is quite simple to use and fairly packet efficient is sending the data after a certain elapsed time period. For instance, after 250 milliseconds we send a packet, then we count the time until 250 ms has elapsed to send another. When the other player moves, his current coordinates/angle information will be sent four times a second, just like yours, to ensure a basic movement operation. Adding the multiplayer capability to the game engine is a simple matter of adding around 200 lines of code to handle all the functionality. Although it doesn't sound small, it's fairly simple to understand.

To begin we'll create a new class called MULTIPLAYER, which will store all our functions for the multiplayer capability. Because UDP uses the Winsock 2 drivers for connectivity, we dynamically load the library (ws2_32.lib) into our multiplayer header file to avoid including it in the workspace library list. After including the library into the header, we'll declare the constants that will control which port, protocol, packet size, and message we'll use for multiplayer support. Earlier in the chapter I mentioned that we're going to use the UDP protocol to transfer data. The global constant for UDP in the Winsock 2 header is SOCK_DGRAM, which will be stored in our constant MULTIPLAYER_PROTOCOL. To use TCP you'd simply specify SOCK_STREAM instead of SOCK_DGRAM. Keep in mind that you'll need to make minor changes to the source code for TCP to work!

The port we'll use for the game is 6001, and is stored in the constant MULTIPLAYER_PORT. Most games use a port that is above the 1000 mark so they don't conflict with other common server ports that may be running on the same computer. The size of each packet is defined in the constant PACKET_SIZE, which is 75. We'll use this value to declare local arrays for incoming and outgoing packets later on. The packets can be any size you'd like, but keep in mind that it's a good idea to keep them relatively small so they don't kill computers with slow uploading bandwidth. Rather than wait for an event to happen on the socket, we can specify our own Windows message that will be signaled when the event happens. For instance, in our game we set up our own Windows message called MM_PACKET_RECEIVED, which will be signaled when a packet is received from the other player. The constant MM_PACKET_RECEIVED has a value of 5000, but it is completely customizable. I wouldn't recommend using message values lower than 1000; otherwise you may inadvertently use an existing Windows message, which may screw things up!

After specifying the constants, we've got two sets of enumerations to create. The first defines the two types of connections (MULTIPLAYER_CLIENT for client and MULTIPLAYER_SERVER for server) and the other defines the type of packet being sent. Since we're only creating a basic function multiplayer game, we'll have two types of packets: PACKETTYPE_MOVEMENT for moving players and PACKETTYPE_SHOOT for shooting your gun. Since the initialization for the server is different from the source code to initialize the client, I've broken the process down into two functions: InitServer and InitClient.

Initializing the Server

There are three simple steps for initializing the server. First we attempt to start the Winsock by calling the function WSAStartup and specifying the version (0x202 for 2.0) to get the variable to store the returned Winsock information, which is of the WSADATA type. The Winsock information contains things like vendor strings to help identify what type of card, Winsock version, and other data is supported by the card. If the function fails, the returned value is SOCKET_ERROR and we'll display an error message and return false to indicate a failure. Although it's not required at this point, it's a good idea to call the function WSACleanup to reset the Winsock sockets created.

In the second step we create the socket, which the game uses to send and receive the data. To create the socket we fill out the address family in sin_family (AF_INET for Internet), the incoming IP address sin_addr.s_addr (INADDR_ANY for any address), and finally the sin_port, which must be encoded using the proper network order. To convert the port to the proper network order, pass the MULTIPLAYER_PORT constant to the htons function, which will then return the proper value. After filling the structure we call the socket function, specifying the address family (AF_INET) and the protocol of the socket (specified in the MULTIPLAYER_PROTOCOL constant). The final parameter specifies the protocol to use for the specified address, but since we didn't specify any specific ones we can set it to 0. The return value from the socket function is the socket itself, which we'll store in the variable game_socket. If the returned value is INVALID_SOCKET, then something failed when attempting to create the socket and we should exit the function immediately, display an error, clean up Winsock, and of course exit with a failure.

The final step in the server initialization process is to bind the local address information with the socket data so we can receive data. Using the bind function the first parameter is the socket to bind (game_socket variable), the second parameter is the local socket address (sock_addr variable), and the final parameter is the size of the sock_addr variable. If the value returned from bind is SOCKET_ERROR, then there's been an error and we must exit immediately, closing the socket and returning false. If the function succeeds, we'll set our local variable active to true, stating the initialization is active, and set our connection_type variable to MULTIPLAYER_SERVER to indicate this object is the server. This is important since the sending and receiving of packets is different for the client and server. Of course the last line of the source code will return a true value, indicating the function succeeded. The code for the InitServer function is provided on the following page.

```
bool MULTIPLAYER::InitServer()
{
   WSADATA wsaData;

   if (WSAStartup(0x202, &wsaData) == SOCKET_ERROR)
   {
      MessageBox (NULL, "Error: Unable to Start Winsock", NULL, MB_OK);
      WSACleanup();
      return (false);
   }

   sock_addr.sin_family         = AF_INET;
   sock_addr.sin_addr.s_addr    = INADDR_ANY;
   sock_addr.sin_port           = htons(MULTIPLAYER_PORT);
   game_socket                  = socket(AF_INET, MULTIPLAYER_PROTOCOL, 0);
   if (game_socket == INVALID_SOCKET)
   {
      MessageBox (NULL, "Error: Unable to Create Listening Socket", NULL, MB_OK);
      WSACleanup();
      return (false);
   }

   if (bind(game_socket, (struct sockaddr*)&sock_addr, sizeof(sock_addr))
         == SOCKET_ERROR)
   {
      MessageBox (NULL, "Error: Failed to Bind Listening Socket", NULL, MB_OK);
      WSACleanup();
      return (false);
   }

   active          = true;
   connection_type = MULTIPLAYER_SERVER;

   return (true);
}
```

Initializing the Client

Initializing the client version of multiplayer support is similar in design to the server initialization code. We'll create a function called InitClient, which has a single parameter (called hostname) that is an array of characters. The parameter contains the IP address of the server to connect to. This is obviously important because we won't be able to connect if we don't specify an IP. Like the server, the first step for initialization calls the WSAStartup function to start up Winsock and grab the Winsock information. Next we use the function gethostbyname and specify our hostname string to retrieve the host information to connect to the server. The return value from the function is of the hostent type, which we'll store in a local variable called hp. The gethostbyname function fails when the return value is NULL. If we encounter a

NULL return value (hp is NULL), then we'll display an error message and reset the Winsock information.

After getting the host information we'll set up the socket address (sock_addr variable). First we'll set the values of the structure to 0 as a default value. Next we'll copy the host address from the hp->h_addr variable to our sock_addr.sin_addr so we know where to connect to. Once we've copied the address, we set the address family to the value specified in the host information address type (hp->h_addrtype). The last variable to fill out in the structure is the port we're using for the connection, which is the return value from using htons and specifying the MULTIPLAYER_PORT constant. With the structure filled out we call the socket function to create the game socket, specifying the address family specification (AF_INET for Internet). The second parameter is the socket (game_socket variable), and the final parameter is the specific protocol used for the address family, which can be set to 0 since we're not using it.

The final step when initializing the client code is to call the connect function with the socket (game_socket variable) as the first parameter, the socket address (sock_addr variable) as the second parameter, and the size of the socket address as the third parameter. If the function fails, the return value will be SOCKET_ERROR. If the function succeeds, we'll attempt to connect to the server. The active variable should be set to true and we'll set the connection type to MULTIPLAYER_CLIENT to indicate we're the client. Since the function succeeded we'll return a value of true to finish the InitClient function. Although it's been slightly wordy, the code for initializing the client code is fairly small and easy to understand. The source code for the function is provided below.

```
bool MULTIPLAYER::InitClient(char *hostname)
{
   WSADATA wsaData;
   struct hostent *hp;

   if (WSAStartup(0x202, &wsaData) == SOCKET_ERROR)
   {
      MessageBox (NULL, "Error: Unable to Start Winsock", NULL, MB_OK);
      WSACleanup();
      return (false);
   }

   hp = gethostbyname(hostname);
   if (hp == NULL )
   {
      MessageBox (NULL, "Error: Cannot resolve address", NULL, MB_OK);
      WSACleanup();
      return (false);
   }

   memset(&sock_addr, 0, sizeof(sock_addr));
   memcpy(&(sock_addr.sin_addr), hp->h_addr, hp->h_length);
   sock_addr.sin_family   = hp->h_addrtype;
   sock_addr.sin_port     = htons(MULTIPLAYER_PORT);

   game_socket = socket(AF_INET, MULTIPLAYER_PROTOCOL, 0);
   if (game_socket < 0 )
   {
      MessageBox (NULL, "Error: Cannot open socket", NULL, MB_OK);
      WSACleanup();
      return (false);
   }

   if (connect(game_socket, (struct sockaddr*)&sock_addr, sizeof(sock_addr))
            == SOCKET_ERROR)
   {
      MessageBox (NULL, "Error: Failed to connect to server", NULL, MB_OK);
      WSACleanup();
      return (false);
   }

   active          = true;
   connection_type  = MULTIPLAYER_CLIENT;

   return (true);
}
```

Sending Packets

Sending outgoing data packets, whether they're from the client or server, is relatively simple. In both cases we'll use the MULTIPLAYER class variable outgoing_packet, which is an array of chars, based on the size specified in the PACKET_SIZE constant. In our MULTIPLAYER class we'll create a function called Send, which will send the outgoing data regardless of the connection type. When we want to send data to the other person we simply put the data inside the outgoing_packet string, then call our Send method to transfer it to our opponent. In case you're wondering, you can put both binary data and NULL-terminated strings into the outgoing_packet variable because we're using the size of the string to send it as opposed to string length (through strlen). For our game, we'll be sending straight text because it's the easiest to work with, although it's not very secure or efficient when packing the data. The first thing we'll do in the Send function before worrying about sending the packet is check to see if the connection is even active (active variable is true); if not, we'll exit. This prevents us from sending data to a socket that's not even initialized and helps to avoid unnecessary crashing.

If the connection_type variable value is MULTIPLAYER_SERVER, then we'll use the function sendto. The parameters of the function are the socket (game_socket variable), outgoing packet data (outgoing_packet variable), size of the data to send (sizeof (outgoing_packet)), the flags for sending (0 since there are none), the socket address (sock_addr variable), and finally the size of the socket address (sizeof(sock_addr)) variable. The server functions are always slightly more complicated than those on the client side of things because they need all the information in order to send data back through the socket since we're using a connectionless protocol. The return value from the sendto call will be stored in a local variable called retval, which we'll discuss in further detail in a few moments. If the connection_type variable value is MULTIPLAYER_CLIENT, then we'll call the function send rather than sendto. The call to the send function only has four parameters as opposed to the six parameters in sendto. The difference between the two calls is the use of the socket address and its size in the sendto call. Except for the last two, the parameters are identical for both functions. Once again, the returned value from send is returned to the variable retval. If the value of the variable is SOCKET_ERROR, we'll display an error on the screen but *not* reset the socket. This is only so we can track if there is a problem with our sending code. At the bottom of the function we'll return a true value to indicate the function succeeded. The source code for the function is written below.

```
bool MULTIPLAYER::Send()
{
   int retval;

   if (!active) return(false);

   if (connection_type == MULTIPLAYER_SERVER) retval = sendto(game_socket,
            outgoing_packet, sizeof (outgoing_packet), 0, (struct
            sockaddr*)&sock_addr, sizeof(sock_addr));
   else retval = send(game_socket, outgoing_packet, sizeof(outgoing_packet), 0);

   if (retval == SOCKET_ERROR) MessageBox (NULL, "Error: Failed to send packet",
            NULL, MB_OK);

   return (true);
}
```

Receiving Packets

Receiving data packets works in a similar fashion to sending them, with the exception that we'll store the incoming packets in a variable called incoming_packet. We'll also use the functions recvfrom and recv. To begin, we'll create a new method called Receive, which will have a bool return type. Within the function we'll once again ensure the connection is active (if active is false, then exit with failure). If the connection type is server, then we'll use the recvfrom function to receive the data. The first parameter of the function is the socket (game_socket variable), the second parameter is the incoming packet variable (incoming_packet), and the third parameter is the size of the packet (sizeof(incoming_packet)). The next parameter contains the flags for receiving the data, but since we're not interested in using any of the flags, we'll set this parameter to 0. After the flags parameter, we specify the socket address (sock_addr variable) so the server knows what address to get the information from, and finally we specify the size of the socket address, which we'll store in a variable called sock_addr_len).

If the connection type is a standard client, then we'll use the function recv instead of recvfrom. As with the structure of the send/sendto functions, the recv/recvfrom functions are the same with the exception that recv doesn't need the socket address information. So we can simply copy and paste the source code from the recvfrom code and erase the last two parameters. In the calls to receive the information we'll store the returned value in a variable called retval. If the returned value (value of retval) is SOCKET_ERROR, then there's something wrong with receiving and we'll simply display an error message and return with a failure. If the return value is anything other than SOCKET_ERROR, we'll return a value of true to indicate the receiving succeeded. The source code for the Receive function is written below.

```
bool MULTIPLAYER::Receive()
{
   int retval;
   int sock_addr_len = sizeof(sock_addr);

   if (!active) return (false);

   if (connection_type == MULTIPLAYER_SERVER) retval = recvfrom(game_socket,
            incoming_packet, sizeof (incoming_packet), 0, (struct
            sockaddr*)&sock_addr, &sock_addr_len);
   else retval = recv(game_socket, incoming_packet, sizeof (incoming_packet), 0);

   if (retval == SOCKET_ERROR)
   {
      MessageBox (NULL, "Error: Failed to Receive UDP Packet", NULL, MB_OK);
      return (false);
   }

   return (true);
}
```

Releasing Open Sockets

Releasing the open sockets is a necessary evil to ensure the operating system functions properly if other applications need to use this specific port. To do so we'll create a new method called Release. If the connection isn't active (active variable isn't true), then we'll exit the function immediately. This will eliminate any problems that may occur from an attempt to close the socket when it's not open. After the check we'll close the game socket by calling the function closesocket and using the game_socket variable as the single parameter. Once the game socket is closed, we'll call the Winsock function WSACleanup to clean up any extra Winsock tidbits that may still be in use. Finally, we'll set the active value to false so the different methods inside the MULTIPLAYER class won't be run and to avoid potential crashes with a closed socket. The source code for the Release method is written here.

```
void MULTIPLAYER::Release()
{
   if (!active) return;

   closesocket(game_socket);
   WSACleanup();

   active = false;
}
```

Multiplayer Setup

With the back-end source code for multiplayer support finished, we can begin working on the code necessary to enable multiplayer support in our game. The first thing we'll do is add two check boxes (IDC_CONFIG_IS_CLIENT and IDC_CONFIG_IS_SERVER) to our configuration dialog box. We'll also add one edit control (called IDC_CONFIG_HOSTNAME), which will hold the IP address of the host we want to connect to if we've selected the client connection type. When the game initially starts and the dialog box displays, the user can choose between being a server or a client. If the user selects the client check box, then we'd get the string entered in the hostname edit control as the connection address for the server computer.

If you want to run the server/client locally you can use the IP address 127.0.0.1 to test the multiplayer support. When the dialog box closes, we simply get the status of each check box and set the connection type appropriately (MULTIPLAYER_SERVER or MULTIPLAYER_CLIENT). If the user doesn't click either button, we don't set the connection type and therefore don't have to worry about setting up the multiplayer connection. When the dialog box finishes, we call the appropriate Init method based on the connection type. In the event the user didn't select any multiplayer connection type, then we won't bother to initialize either connection type. After initializing the connection, we check to see if the connection is active (through the multiplayer.active variable), and set up a custom Windows message to handle incoming packets being received from the other user.

To create the custom Windows message we use the function WSAAsyncSelect with the socket (multiplayer.game_socket) as the first parameter. The second parameter for the function is the window (Window) to be received from the packet, the custom message constant (MM_PACKET_RECEIVED), and finally the Winsock event that will cause this message.

We could specify all sorts of events such as receiving data, sending data, the initial connection, and closing the connection, but we are only interested in the receiving data (FD_READ) event. Now that the custom message is set up, when we receive data from the socket, our custom MM_PACKET_RECEIVED Windows message will be signaled in the main Windows procedure.

When the message is signaled, we immediately invoke multiplayer.Receive to receive the incoming packet so we don't lose it with another possible transmission. After receiving the packet, we decode the individual packet parameters into the local variables packet_type, x, y, z, xa, ya, and za. If packet_type is a standard movement, we set the enemy XYZ/angle information to the values extracted from the string. After we've finished decoding our packet, we simply call the WSAAsyncSelect function to restart our custom receiving message. Technically this step doesn't need to be done, but it ensures the custom message is reset and works properly for the next packet that's received. The source code for the updated WndProc function is given below.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
      case MM_PACKET_RECEIVED:
      {
         long packet_type;
         float x;
         float y;
         float z;
         float xa;
         float ya;
         float za;

         multiplayer.Receive();

         sscanf (multiplayer.incoming_packet, "%i %f %f %f %f %f %f",
                 &packet_type, &x, &y, &z, &xa, &ya, &za);


         if (packet_type == PACKETTYPE_MOVEMENT)
         {
            enemy.xyz[0]   = x;
            enemy.xyz[1]   = y;
            enemy.xyz[2]   = z;
            enemy.angle[0]= xa;
            enemy.angle[1]= ya;
            enemy.angle[2]= za;
         }
```

```
            WSAAsyncSelect (multiplayer.game_socket, Window, MM_PACKET_RECEIVED,
                            FD_READ);

        }break;
        case WM_DESTROY: PostQuitMessage(0); break;
    }
    return (DefWindowProc(hWnd, msg, wParam, lParam));
}
```

When the game starts we'll load the model player.ase (available from the download) into the global variable player_model. We'll use this model to represent the other user or enemy in the game. Unfortunately, rendering this model requires slightly more work than simply rendering the items because we want to rotate the model to face the direction the other user is facing. To do this, we'll back up our current matrix, then rotate the x-axis around the current player's x-axis value just as we would for the regular view. Next we'll position the enemy by adding the current player's XYZ coordinates to the enemy's XYZ. We do this to compensate for the movements that will occur when we later move the view to render the map. Next we rotate the x-axis again, using the enemy's x-axis coordinate this time. This will rotate the model to the appropriate location that the enemy is facing, which is the key. Finally we call our RenderModel function and then restore the matrix so we can begin altering the matrices to render the map data. This code will only be run if we're in a multiplayer game (multiplayer.active is true). The source code for the updated Render function is below.

```
void Render()
{
    glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);


    if (multiplayer.active)
    {
        glPushMatrix();
            glRotatef(player.angle[0], 0.0f, 1.0f, 0.0f);
            glTranslated (player.xyz[0]+enemy.xyz[0], player.xyz[1]+enemy.xyz[1],
                          player.xyz[2]+enemy.xyz[2]);
            glRotatef (enemy.angle[0], 0.0f, 1.0f, 0.0f);

            RenderModel (player_model);
        glPopMatrix();
    }


    glPushMatrix();
        glRotatef (player.angle[0], 0.0, 1.0, 0.0);
        glTranslated (player.xyz[0], player.xyz[1], player.xyz[2]);
        RenderMap();
    glPopMatrix();
```

```
   glEnable (GL_BLEND);
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   if (strstr(gl_extension, "GL_ARB_texture_non_power_of_two") != NULL)
            glBitBlt(GetTextureID("npot_hud.bmp"), 0,240,640,240);
   else glBitBlt(1, 0,240,640,240);

   SwapBuffers(raster.hDC);
}
```

In our main loop of the game, we'll add a call to a function called SendMultiplayerPacket. This function will compare the current time with the time that the last packet was sent, which is stored in the global variable time_of_last_packet_sent. When the variable is first declared we'll set the default value to 0 so we can send an initial packet when we start the game engine. If the current time is greater than 250 ms between sends, we'll construct a new packet in the multiplayer.outgoing_packet variable and send it. Once we've sent the packet we'll save the current time to start the process again. To get the current time we use the function timeGetTime, which requires us to include the header mmsystem.h into our list of headers and include the library winmm.lib. You can include the library in the source file using #pragma or include it in the workspace library list. If we wanted to send more packets for smoother movement, we'd lower the time between each packet being sent. One thing to keep in mind is that if you send too many packets at once you may run into difficulties processing the information on the other computer. By using a time-based packet sending system, we're eliminating some of the issues surrounding sent packets being synchronized with the frame updates. The source code to send a multiplayer packet is shown below.

```
void SendMultiplayerPacket()
{
   if (!multiplayer.active) return;

   if (timeGetTime() > (unsigned long)time_of_last_packet_sent+250)
   {
      sprintf (multiplayer.outgoing_packet,"%i %f %f %f %f %f %f",
               PACKETTYPE_MOVEMENT, (float)player.xyz[0], (float)player.xyz[1],
               (float)player.xyz[2], (float)player.angle[0],
               (float)player.angle[1], (float)player.angle[2]);
      multiplayer.Send ();

      time_of_last_packet_sent = timeGetTime();
   }
}
```

Chapter Example

Please see the example from the companion files (ex17_1).

Conclusion

In this chapter we learned the basics of how to use UDP Winsock code with OpenGL to move and rotate characters around our 3D environment. Using this chapter as a simple guide, you could create very elaborate multiplayer experiences simply by adding more packet types and/or features to each packet. It would even be simple to modify the packets being sent to include the gun being held by the user. With this information you could draw the player model with a specific gun in his hands, which is a great feature to add to any game. This is just one simple idea that came to mind for customizing the packets, but the possibilities with networking code are almost endless. As mentioned earlier, if you'd like to get the highest performance out of the transmission code, it's recommended you store the data as binary rather than text to save space.

Chapter 18

Using OpenAL

When designing the audio engine for your game engine, there are many APIs to consider including the Windows sound system, DirectSound 3D, SDL, OpenAL, and many others. Every sound API has benefits and drawbacks. For instance, the Windows sound system doesn't have hardware acceleration or 3D audio playback. DirectSound 3D has a wide market because it's included in the DirectX SDK; however, it doesn't work on non-Windows operating systems (excluding the Wine/WineX projects). The Simple DirectMedia Layer (SDL) library works on many platforms (e.g., Windows, GNU/Linux, MacOS/X, BSD, Playstation 2, Linux, etc.), but it lacks 3D audio support, which is something that 3D video games in this day and age must have.

The OpenAL API provides a simple syntax (similar to OpenGL's) to play audio in both 2D and 3D environments. OpenAL is freely available under the LGPL license for the Windows, GNU/Linux, and Mac OS X platforms. If you're interested in using OpenAL when developing console games, you can create OpenAL implementations for the Sony PlayStation 2, Microsoft Xbox, and Nintendo GameCube console systems using Sensaura's gameCODA technology. This puts a tremendous amount of power at your fingertips because you can design your sound engine without worrying about using proprietary sound engine code for each platform. In some cases you will still need to write proprietary sound code to accommodate any special features that may be available, but this will drastically cut down on extra code writing. The time we save not having to write extra code could be used for more important issues like writing new graphics functions or testing our game quality.

OpenAL Installation

You can download OpenAL from http://www.openal.org. Since we fall into the developer category we'll download and install the OpenAL Developer SDK 1.0 (the latest version at the time of publication), which provides us with the run-time libraries, development SDKs, and programmer's reference manuals. Depending on how daring you are, you can download newer beta copies of the SDK; however, the added functionality will not be discussed in this book. OpenAL, unlike many SDKs, provides excellent documentation that is simple to understand and relevant code examples.

In the past we've simply included the required headers in the source code without any trouble; however, if we include the OpenAL header files al.h and alc.h (the main headers of OpenAL), the compiler won't find them. To resolve this issue we must add the path to the OpenAL headers (normally C:\Program Files\OpenAL 1.0 Software Development Kit\Include) into the workspace. To do this in Visual C++, click Project, then Settings (or Alt+F7), choose the C/C++ tab, and select the Preprocessor category. At the bottom of the dialog the Additional Include Directories text area will appear, allowing you to add the location of the OpenAL header files. Although this is annoying, it's a one-time deal for the workspace. The OpenAL libraries must also be included into the workspace. Since the libraries (openal32.lib and alut.lib) are located in the OpenAL installation directory, we'll simply add the two files to the Object/Library modules section of the Link tab in the Project Settings dialog box. We can add the OpenAL installation directory path to the workspace by changing the combo box category to Input and adding the installation path to the Additional library path section. This will avoid having to specify the directory for the libraries when adding them to the library list.

Initializing OpenAL

Like the other components in our game engine, the sound system has its own class (called SOUND) to interface with OpenAL and the audio hardware. To initialize OpenAL, we'll create a new method called Init, which has a bool return type. This function will contain all the code necessary to start and use OpenAL. When initializing OpenAL, the first thing we do is open the selected audio device using the function alcOpenDevice, specifying the device as the single parameter. Since each sound card has a different name, it would be very difficult to know what sound card we want to use. In this type of situation we can use a NULL value, which will use the preferred audio device of the computer. If you are interested in enumerating the audio devices for OpenAL, I would recommend researching information on the ALC_ENUMERATION_EXT extension, which provides the names of the different devices for the alcOpenDevice function call.

The returned value from the alcOpenDevice function is the open device of the ALCdevice type. If the returned value is NULL, then opening the device failed and we should display an error and exit. If the function succeeded, we create the OpenAL context from the device by calling the function alcCreateContext. The first parameter of the function is the device we're creating the context from, which in our case will be the variable alDevice. The second parameter is the special attribute list used when creating the context. The attributes allow you to specify frequency, refresh, and sync of the context. Because we're just trying to create a simple sound system, this information isn't important to us. The return value is a pointer to the new context, which is of the ALCcontext type. If the returned value is NULL, then an error occurred and we'll display the error and exit, like usual.

Because we can create multiple contexts, we must specify which context to make the current one. To do this we use the function alcMakeContextCurrent and specify our current context (alContext variable). Unlike in the past where we checked the return value for failure, we'll use a unique function called alcGetError, which gets the context error codes. If the returned value is ALC_NO_ERROR, no context errors were found. Any other return value is the AL context error code, for which we display an error message and exit the function. Once the context has been made current, we reset the error codes through the use of alGetError. After resetting the error codes, we'll use the function alGenBuffers to specify the number of audio buffers we want in our game. At this point you may be wondering what the heck an audio buffer is. An *audio buffer* stores the pulse-code modulated (PCM) data from audio files such as those in the wave (.wav) format. When we define how many buffers we want, we're defining how many different sound effects we are going to use. Like textures in OpenGL, we'll create a constant called MAX_AUDIO_BUFFERS, which will store the default number of buffers (2). The filenames of the sounds we're going to load will be stored in a pointer array called sound_files. We'll also declare another variable called SoundBuffers, which is an array of the ALuint type. The size of the array is the value of our MAX_AUDIO_BUFFERS constant.

When we call the alGenBuffers function, we specify the number of buffers (MAX_AUDIO_BUFFERS) as the first parameter and the buffer array as the second array, similar to the glGenTextures function in OpenGL. Using the function alGetError, we'll check for errors that occurred from using alGenBuffers. If the returned value from alGetError is AL_NO_ERROR, no error occurred. Otherwise an error occurred and we'll display the error and exit. At this point we'll loop through the buffers and load the .wav files defined in the sound_files array, using the method LoadWav, which we'll create in a few moments. After loading the wave data we'll simply exit the Init method, returning a true value to indicate initialization was a success.

As you can see, the initialization of OpenAL is fairly simple. If we were interested in using special features such as EAX or extensions, the initialization would be slightly more complicated, but that's another story. The source code for the function is written below.

```
bool AUDIO::Init()
{
   alDevice = alcOpenDevice(NULL);
   if (alDevice == NULL)
   {
        MessageBox (NULL,"Error: Failed to Open Device", NULL, MB_OK);
        return (false);
    }

    alContext = alcCreateContext(alDevice, NULL);
    if (alContext == NULL)
    {
        MessageBox (NULL, "Error: Failed to initialize OpenAL", NULL, MB_OK);
        return (false);
    }

    alcMakeContextCurrent(alContext);
    if (alcGetError(alDevice) != ALC_NO_ERROR)
    {
        MessageBox (NULL, "Error: Failed to Make Context Current", NULL, MB_OK);
        return (false);
    }

    alGenBuffers(MAX_AUDIO_BUFFERS, SoundBuffer);
    if (alGetError() != AL_NO_ERROR)
    {
        MessageBox (NULL, "Error: Failed to Generate Buffers", NULL, MB_OK);
        return (false);
    }

    for (long buffer = 0; buffer < MAX_AUDIO_BUFFERS; buffer++)
    {
        LoadWav (sound_files[buffer], buffer);
    }

    return (true);
}
```

Using Audio Files

At the end of the previous section we looped through the different audio buffers and loaded audio data from .wav files into the desired buffer using the method LoadWav. In this section we'll write the LoadWav method code to load the specified .wav files. I chose to use .wav files for sound effects because OpenAL has a utility library called alut that contains functions to easily load them into a sound buffer. If we were interested in playing sound clips that were fairly long (10 seconds or longer), it would be a good idea to use another format because .wav files are stored as uncompressed data and can sometimes take an enormous amount of space. With the proper libraries you can use any audio file format in OpenAL sound buffers including the popular MPEG Layer 3 (.mp3) and Ogg Vorbis (.ogg) formats. The newer beta versions of the OpenAL SDK have included support to decode the open-source compression audio format ogg/vorbis. More information about the newer versions of OpenAL can be retrieved on their web site.

To begin the loading process we'll create a new method called LoadWav, which has a bool return type. The first parameter of the new method is a pointer of a char (called filename) and the second is a long (called buffer) that specifies the buffer number to place the audio data in. The AL utility library, or alut, supports two methods of loading wave data. The first is through the function alutLoadWAVMemory, which loads wave audio data from a memory location. This is useful if you're storing your wave data in a resource script, embedding it directly into a program, or loading data from a packed file like the PAK/WAD files seen in many popular first-person shooters. The other function, alutLoadWAVFile, loads wave data directly from a specified file. This is the method we'll be using in our game engine. The first parameter for the function is the filename to load. The next five parameters return the format, file size, bit depth, frequency, and looping information about the sound. These variables are essential for configuring the sound buffer so we can copy the wave to the sound buffer without causing an error. The size of the sound buffer depends directly on those variables.

After loading the .wav file data we run the function alGetError and check for any errors. If the returned value isn't AL_NO_ERROR, then we've experienced an error and we'll display it on the screen and exit the function with a failure. If the function has succeeded, we'll run the function alBufferData, which will copy the wave data returned from alutLoadWAVFile to the sound buffer specified. The first parameter of the function is the sound buffer (SoundBuffer[buffer]) and the second is the format of the .wav file. The third parameter is the actual data from the .wav file and the fourth parameter is the size. The final parameter for the function is the frequency of the wave. If the function fails (alGetError doesn't return AL_NO_ERROR), then we'll display an error and exit. If the function succeeds (alGetError return value is AL_NO_ERROR), then the wave data returned from alutLoadWAVFile is no longer needed because we're storing it in memory as a sound buffer.

To release the sound data we use the function alutUnloadWAV and specify the format, data, size, and frequency of the loaded data. Much like OpenGL textures, there's no sense in keeping the data loaded in the SDK buffers and in conventional memory as well. If the function fails (alGetError doesn't return AL_NO_ERROR), we'll display an error and exit; otherwise we'll exit the function, returning true to indicate success. As you can see by the following source code, loading .wav files is quite simple when using the alut functions.

```
bool AUDIO::LoadWav (char *filename, long buffer)
{
   char        filepath[500];
   char        errmsg[500];
   ALsizei     size;
   ALsizei     freq;
   ALenum      format;
   ALvoid      *data;
   ALboolean  loop;


   sprintf (filepath, "../media/%s", filename);
   alutLoadWAVFile(filepath, &format, &data, &size, &freq, &loop);
   if (alGetError() != AL_NO_ERROR)
   {
      sprintf (errmsg, "Error: Failed to Load WAV (%s)", filename);
      MessageBox (NULL, errmsg, NULL, MB_OK);

      return (false);
   }

   alBufferData(SoundBuffer[buffer], format, data, size, freq);
   if (alGetError() != AL_NO_ERROR)
   {
      sprintf (errmsg, "Error: Failed to Copy Buffer Data %i (%s)", buffer,
               filename);
      MessageBox (NULL, errmsg, NULL, MB_OK);

      return (false);
   }

   alutUnloadWAV(format, data, size, freq);
   if (alGetError() != AL_NO_ERROR)
   {
      sprintf (errmsg, "Error: Failed to Unload WAV Data (%s)", filename);
      MessageBox (NULL, errmsg, NULL, MB_OK);

      return (false);
   }
   return (true);
}
```

Playing Sources

To play the different sound sources we'll create a function called PlaySource, which accepts a single parameter (called source) that specifies the source index in the source array to play. Using the function alSourcePlay, we specify the sound source (SoundSource[source]) to begin playing the source. The source settings we specified in the SetSource method will be applied to the output of the source to modify its sound. To check for errors, we'll use the function alGetError. If the returned value isn't AL_NO_ERROR, then an error occurred and we'll display an error message. Otherwise life is peachy. The source code for the PlaySource method is provided here.

```
void AUDIO::PlaySource(long source)
{
   if (source > max_sources) return;

   alSourcePlay(SoundSource[source]);
   if (alGetError() != AL_NO_ERROR)
   {
      char errmsg[50];
      sprintf (errmsg, "Error: Failed to Play Source %i", source);
      MessageBox (NULL, errmsg, NULL, MB_OK);
   }
}
```

Stopping Sources

Because we can specify whether a source automatically loops when playing, we must have a mechanism to turn individual sources off. To do this we'll create a new method called StopSource, which has a single long parameter storing the source number to shut off. To stop the source from playing we use the function alSourceStop with the source reference number. Like the other functions in the class, we'll use the alGetError function to check for any errors. If an error occurs (return value isn't AL_NO_ERROR), then we'll display an error message. There's no need to add the extra line to exit the function because there's no more code to write anyway! The source code for the StopSource method is given below.

```
void AUDIO::StopSource(long source)
{
   if (source > max_sources) return;

   alSourceStop(SoundSource[source]);
   if (alGetError() != AL_NO_ERROR)
   {
      char errmsg[50];
      sprintf (errmsg, "Error: Failed to Stop Source %i", source);
      MessageBox (NULL, errmsg, NULL, MB_OK);
   }
}
```

Getting the Buffer ID

When we wrote the code to load bitmap files as textures we created a function called GetTextureID, which retrieved the texture number of a specified file. Although this isn't a necessary function to have when implementing OpenAL code, it's a good idea to create a new method to search through the list of audio filenames and return the buffer number. Essentially we'll loop through the number of audio buffers (which is the same as the number of filenames in the sound_files array) and check for the specified filename. If we find a match, we'll return the buffer number. If a match isn't found, we'll return a value of –1 to indicate failure, since 0 is reserved as the first sound buffer. The source code for the GetBufferID function is written below.

```
long AUDIO::GetBufferID(char *filename)
{
   for (long buffer = 0; buffer < MAX_AUDIO_BUFFERS; buffer++)
   {
      if (strcmp(sound_files[buffer], filename) == 0) return (buffer);
   }
   return (-1);
}
```

Setting Source Information

In OpenAL, sources control where a sound is played, which sound buffer is to be played, the velocity of the sound, the pitch, gain, and even the looping information. They are the life of the audio system. When creating the sound environment for a map or level, it's a good idea to position sounds around the world to help build the atmosphere. For instance, if you were walking through a power generation station you would want looped sounds being played as you walk by the generators. As you move away from the sound source the audio would fade away, just like in real life.

Since we've added the support in our map format to add sound effects throughout the map, we're going to take advantage of this by creating a method to set the position, velocity, pitch, gain, and loop information for the sound. To begin we'll create a new method called SetSource, which has a bool return type. The new method will accept eight parameters. The first is the source (long type), then the filename (array of chars), then the X, Y, and Z position information (which are floats). After setting the position of the source we set the pitch (float), the gain (float,) and the looping value (bool). Inside the method, the first thing we do is get the buffer number for the specified filename by calling the method GetBufferID. If the return value is less than 0, the file wasn't found and we'll exit to avoid any problems.

After getting the sound buffer number, we'll begin configuring our source by attaching the specified buffer to the source. To do this we use the AL function alSource and specify the source (SoundSource[source]) to configure, then use the constant AL_BUFFER to specify that we want to attach the buffer, and finally provide the sound buffer number (buffer variable) for the filename specified. After attaching the buffer, we'll set the pitch of the source by using alSourcef with the constant AL_PITCH and the floating-point pitch (pitch variable) value. Although we're not going to edit the pitch value, I encourage you to play around with the pitch to see how it affects the audio files you're using in the game engine. Under normal circumstances when you modify the pitch value for the source, it will be played at either a higher or lower tone based on the value you specify (the default value for the parameter in the declaration is 1.0).

We'll use alSourcef in the same manner as before to specify gain, with the exception that we'll use the constant AL_GAIN, then specify the gain value. After specifying the gain, we set the position information using alSourcefv (for a floating-point array) with the constant AL_POSITION and the position information (position variable) as the final variable. The looping information is next to be specified, using the constant AL_LOOPING with the looping value (true or false).

When we play the sound source, OpenAL will play the sound buffer specified for the sound source, applying the configuration we specified for it. At any time we can change the position or any information about the source by simply calling the SetSource method again for the specific sound source. This allows us to easily move sources around our map without having to code all sorts of workarounds, which is something you always want to avoid doing. The source code for the SetSource function follows.

```
bool AUDIO::SetSource(long source, char *filename, float x, float y, float z,
          float pitch, float gain, bool loop)
{
   ALfloat position[] = {x, y, z};
   char errmsg[50];
   long buffer;

   buffer = GetBufferID(filename);
   if (buffer < 0)
   {
      sprintf (errmsg, "Error: Failed to Find (%s)", filename);
      MessageBox (NULL, errmsg, NULL, MB_OK);
      return (false);
   }

   alSourcei(SoundSource[source], AL_BUFFER, SoundBuffer[buffer]);
   alSourcef(SoundSource[source], AL_PITCH, pitch);
   alSourcef(SoundSource[source], AL_GAIN, gain);
   alSourcefv(SoundSource[source], AL_POSITION, position);
   alSourcei(SoundSource[source], AL_LOOPING, loop);

   return (true);
}
```

Creating Sources

Because the number of sound sources is dependent on the value in the map, we'll create a method called CreateSources that allocates memory for the number of sources specified as the parameter. This allows us to specify the number of sources we want without having to hardcode any values. After allocating the memory we call the AL function alGenSources to generate the source names. Like alGenBuffers, we must call this to activate the number of sources required. The source code for the function is written below.

```
void AUDIO::CreateSources(long sources)
{
   if (max_sources > 0) ReleaseSources();

   max_sources = sources;
   SoundSource = new ALuint[max_sources+1];

   alGenSources (max_sources, SoundSource);
}
```

Releasing Sources

Since sources is a dynamically allocated array, we'll create a method called ReleaseSources to easily release its memory. We're keeping it separate from the standard release code because we may want to release the sources if we're loading a new level and have another number or other uses for it. To release the sources we first delete them by calling the function alDelete-Sources, specifying the number of sources to release and the sound sources array (SoundSource). After deleting the sources we release the memory in the SoundSource variable and set the number of sources to 0. The source code for the ReleaseSources method is shown below.

```
void AUDIO::ReleaseSources()
{
   if (max_sources == 0) return;

   alDeleteSources(max_sources, SoundSource);

   delete [] SoundSource;
   SoundSource = NULL;

   max_sources = 0;
}
```

Releasing OpenAL

Releasing OpenAL is quite simple. Before we release the sources, context, and device data, we'll first check to ensure that OpenAL was initialized. If the device or context are NULL, we'll exit to avoid crashing the software because AL didn't initialize. After the check we release the sound source, which was allocated when we loaded the level. With the sound sources released, we can delete the sound buffers containing the .wav files by calling the AL function alDeleteBuffers, with the number of buffers (MAX_AUDIO_BUFFERS) as the first parameter and the sound buffer (SoundBuffer) array as the second. Next we destroy the context by calling alcDestroyContext and specifying the AL context. After destroying the context we close the AL device using alcCloseDevice with the device as the parameter. Although it's not required, it's a good idea to set the context and device variables to NULL for tidy coding. The source code for the Release method is shown here:

```
void AUDIO::Release()
{
   if (alDevice == NULL || alContext == NULL) return;

   if (max_sources > 0) ReleaseSources();

   alDeleteBuffers(MAX_AUDIO_BUFFERS, SoundBuffer);
   alcDestroyContext(alContext);
   alcCloseDevice(alDevice);

   alContext  = NULL;
   alDevice   = NULL;
}
```

Setting Up the Listener

The listener is similar to the camera in OpenGL in that it is the main recipient of the audio code. Like the camera in OpenGL, you must set the position and orientation (angles) as well as the velocity of the listener to accurately place the sound sources. As our player walks through the map we would continuously update our listener position with the newest information so the sounds could be played properly. To do all this, the final method we'll create in the AUDIO class is called SetListener, which will set up listener information. The method contains two sets of parameters, the first being the XYZ position (three floats) and the second being the current angle of the player (called player_angle) in float form.

When specifying listener data, we use the function alListener and specify the data constant as the first parameter and the value as the second parameter. The position of the listener is a simple array of three floats containing the XYZ coordinates of the listener (player). The orientation of the listener is stored in an array of six floats because we're storing two vectors in it. The first vector set defines the forward orientation, and the second set defines the up orientation. Since our player angle information is stored in degrees, we must convert the X/Z coordinates to their proper sine/cosine representations by calculating the radians for the angle, then set the X value to the sine of the radians and the Z value as the -cosine radians. By plotting the orientation this way, we'll guarantee that our orientation will automatically rotate as we turn left and right. This will then pan the sound regardless of the number of speakers the user has.

Moving this knowledge to code, we'll first set the listener position by calling alListenerfv and specifying the position with the constant AL_POSITION and the value as the second parameter. After setting the value we'll check the return value from alGetError for any errors. If we encounter an error, we'll display an error message and exit the function. If no errors occur, we'll set the orientation of the listener by specifying the constant AL_ORIENTATION and the value as the second parameter. As these three functions are being specified, the audio playback should adjust itself automatically, provided there are no errors. Ideally this method would be called every frame update to give us the fastest update with our sounds. The source code for the SetListener method is written below.

```
void AUDIO::SetListener(float x, float y, float z, float player_angle)
{
   ALfloat rad           = (3.141592654f / 180.0f) * player_angle;
   ALfloat position[]    = {x, y, z};
   ALfloat orientation[] = {(float)sin(rad), 0, (float)-cos(rad), 0,0,0};

   alGetError();
   alcGetError(alDevice);

   alListenerfv(AL_POSITION, position);
   if (alGetError() != AL_NO_ERROR)
   {
      MessageBox (NULL, "Failed to Set the Listener Position", NULL, MB_OK);
      return;
   }

   alListenerfv(AL_ORIENTATION, orientation);
   if (alGetError() != AL_NO_ERROR)
   {
      MessageBox (NULL, "Failed to Set the Listener Orientation", NULL, MB_OK);
      return;
   }
}
```

Adding Sound to Our Game

With the back-end source code written in our AUDIO class we can finally add sound to our game engine. To do this we'll include the audio header and declare a global variable called audio, which is of the AUDIO type. We'll initialize the audio after we create the main window in our game. If the audio fails, we'll exit the software. We don't need to worry about displaying an error since our Init method will do this for us. If any of the other functions fail (such as initializing OpenGL), then we'll run the audio.Release method to release the allocated devices of OpenAL. At the bottom of the WinMain function we'll use audio.Release to release the AL device.

Since we want to project the audio based on the current location of the player, we'll update the CheckInput function to add a single call to the audio.SetListener method. When our player walks around the map, the listener will automatically be updated with the newest XYZ/orientation information. If we compile this demo right now, our audio system would initialize and set the listener position as we move but no sound sources will play. To fix this we'll update our LoadMap function to add support for sound sources. When the map is loaded we'll check the value of map.header.max_sounds. If the value is greater than 0, we'll create the sources (audio.CreateSources) using the value specified in map.header.max_sounds. If we wanted to add other sources into our game we'd simply add the number we want to map.header.max_sounds. When using other sources we would start the value at map.header.max_sounds so the original sources of the map would be easily loaded and set.

After creating the sources we'll set the sources by using the audio.SetSource method, then run the PlaySource method to start each source. Upon exit of the LoadMap function we'll have our sound system completely working. We'll be able to walk around the map and hear different sounds based on the movement of our player. The sounds will also be projected across the different speakers based on the orientation of the player. The source code for the updated LoadMap function is written below for you to study.

```
bool LoadMap(char *filename)
{
   if (!map.Open(filename)) return (false);

   for (long obj = 0; obj < map.header.max_objects; obj++)
   {
      for (long tex_layer = 0; tex_layer < MAX_TEXTURE_LAYERS; tex_layer++)
      {
         map.object[obj].texture[tex_layer].id = GetTextureID(map.object
                  [obj].texture[tex_layer].filename);
      }
   }
```

```
    if (map.header.use_skybox)
    {
       map.skybox.front.texid    = GetTextureID(map.skybox.front.filename);
       map.skybox.back.texid     = GetTextureID(map.skybox.back.filename);
       map.skybox.left.texid     = GetTextureID(map.skybox.left.filename);
       map.skybox.right.texid    = GetTextureID(map.skybox.right.filename);
       map.skybox.top.texid      = GetTextureID(map.skybox.top.filename);
       map.skybox.bottom.texid   = GetTextureID(map.skybox.bottom.filename);
    }


    if (map.header.use_fog)
    {
       glFogi (GL_FOG_MODE, map.fog.mode);
       glFogf (GL_FOG_DENSITY, map.fog.density);
       glFogf (GL_FOG_START, map.fog.start);
       glFogf (GL_FOG_END, map.fog.end);
       glFogfv (GL_FOG_COLOR, map.fog.rgba);
       glFogi(GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT);
       glEnable (GL_FOG);
    }
    else glDisable (GL_FOG);


    if (map.header.max_sounds > 0)
    {
       audio.CreateSources (map.header.max_sounds);
       for (long snd = 0; snd < map.header.max_sounds; snd++)
       {
          audio.SetSource (snd, map.sound[snd].filename, (float)map.sound
                           [snd].xyz[0], (float)map.sound[snd].xyz[1],
                           (float)map.sound[snd].xyz[2]);
          audio.PlaySource (snd);
       }
    }

    return (true);
}
```

Chapter Example

Please see the example from the companion files (ex18_1).

Conclusion

In this chapter, we discussed the OpenAL SDK and how to initialize and release hardware from it. We also learned how to load wave audio files, position audio sources, and play them. This chapter contains everything you'd need to create your own basic sound system and much more.

Chapter 19

Tips and Tricks

Displaying Text

At times during the development of your game you'll need to display text on the screen. Unfortunately OpenGL doesn't provide any means to do this. In this chapter, we'll discuss several methods for displaying text on the screen. Perhaps the easiest but least likely method to be used is the Win32 SDK function TextOut, using the current device context from OpenGL to display the text. Under normal circumstances we'd want to avoid using this method because it's very slow, doesn't provide a great deal of customization, and is not portable to other platforms. By customization, I'm referring to the ease of sizing, coloring, and creating the font itself. Although it's possible to create your own Windows font for use with this method, that's a pretty time-consuming task in itself. Especially considering you may potentially have a speed hit when drawing it to the screen.

An alternative method to using the TextOut function is to use bitmap files as font characters and simply bitblit the characters of a string to the screen. Using your favorite graphics package (such as Photoshop, PaintShop Pro, GIMP, or Microsoft Paint) we'll create the characters for our font. There are two methods of storing the bitmapped character data. The first method is to create a massive file that contains each character available in the font. To support a font we have to either write a special texturing function to parse each character from the bitmap or write a new bitblit function that can easily map the different characters in our bitmap file. The advantage of this first method is that it takes less space on the hard drive and has fewer file dependencies. The other method is to create a set of bitmaps containing a single character in each file. Although there are more file dependencies than the other method, we can easily add this method into our game without much difficulty. The character must fit the dimensions of the bitmap, but we can also use different sized bitmaps for certain characters like numbers so we can scale them up without having them draw pixelated. This is the method we'll be using in our game engine.

To save us some time I've already created the characters for our font. The font files use the filename notation font1_[character].bmp. These premade files are located in the media directory of the downloadable file and contain their appropriate character. Since these files are no different from any other texture files, we'll add them to our texture list with the transparent color Boolean set to true. Since the loading and binding of the bitmap characters is done automatically by our texture loading system, we can focus on writing the code to display text. Keeping with our GL-style Win32 calls we'll create a function called glTextOut, which will display text in a similar fashion to the Win32 function. The first two parameters of the function are the starting X/Y positions of the text. The final parameter of the function is the NULL-terminated character array containing the text to display.

To display the characters of a string, we simply loop through the characters of the string, bitblitting each to the screen. To accommodate upper- and lowercase letters we can either have individual font characters for each or use the function tolower to set the characters to lowercase. Of course this only applies if you want to load the files with their proper filenames. One character we'll try to avoid is the space character because we don't have a corresponding character for it. When we attempt to get the texture ID for the character, the return value will be 0, causing a blank, white colored texture. As we progress through the loop we'll shift the X coordinate to the right by the width of a character by multiplying the current character index by the width and adding the starting coordinate to it. Remember that if you're planning on using transparent colors in your font textures (as we are) you'll need to enable blending as we've done with our HUD texture to eliminate the transparent color. In our Render function we'll call our glTextOut function to display some generic text and the map name below it. Since we're using the glBitBlt code, we're simply bitblitting to the screen, which has a screen canvas of 640x480 as we defined when setting up the orthographic projection for blitting. This is a simple method of use for the function, but there are many great uses for it. The source code for the function is written below.

```
void glTextOut(long start_x, long start_y, char *string)
{
   char file[50];
   long h    = 12;
   long w    = 8;


   glEnable (GL_BLEND);
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   for (long idx = 0; idx < (signed)strlen(string); idx++)
   {
      if (string[idx] != ' ')
      {
         sprintf (file, "font1_%c.bmp", tolower(string[idx]));
         glBindTexture (GL_TEXTURE_2D, TextureName[GetTextureID(file)]);
         glBitBlt (GetTextureID(file), start_x+(idx*w), start_y, w, h);
      }
   }

   glDisable (GL_BLEND);
}
```

Adding a Frame-per-Second Counter

Although it's not an essential tool for developing games, adding a frame-per-second (FPS) counter can be very useful when trying to optimize games or rendering engines. Furthermore, it provides a means for benchmarking your software on different pieces of hardware. To create an FPS counter we need to declare three global variables (cur_fps, old_fps, and old_time) of the long type. Before we enter the main loop of our game, we'll set the old_time variable to the returned value from the function timeGetTime (which stores time since the computer was powered in milliseconds). With each pass of the loop, we'll run the CalculateFPS function, which will increase the cur_fps variable by one to increase the current frame per second. After increasing the current FPS, we'll calculate the elapsed time (by subtracting the old time from the current time). If the elapsed time is greater than or equal to 1000 ms (one second), then we've clocked one second's worth of frame updates, which allows us to accurately display the FPS by saving the current FPS to the old_fps variable, resetting cur_fps to 0, and finally setting the old_time variable to the current time to begin the frame counting process again. Since we wrote a function in the previous section to display a font on the screen, we'll display the last calculated FPS at the bottom of the function. If we displayed the current FPS, each frame update would continuously increment and not provide a real-world FPS value. It wouldn't be very useful to see "FPS 1," then "FPS 2," then "FPS 3," etc., as the frames are counted on the screen. By using the old FPS value, we eliminate this minor issue and display a solid value on the screen. Every second the FPS value will change, which is good enough for most people! Since this feature is specific to rendering, we'll call it in the Render function so it will be updated as we render data to the screen. The source code for the CalculateFPS function is written below.

```
void CalculateFPS()
{
   char temp[50];

   cur_fps++;
   if (timeGetTime()-old_time >= 1000)
   {
      old_fps      = cur_fps;
      cur_fps      = 0;
      old_time     = timeGetTime();
   }

   sprintf (temp, "FPS %i", old_fps);
   glTextOut (0,30, temp);
}
```

Holding an Item

Holding an item such as a gun is always a nifty feature in any first-person shooter video game. There are several different ways we can display the gun on the screen. First, we could create a 2D bitmap of the gun in the desired view, but it wouldn't be a 3D game if we relied on 2D technology! Rather than create a 2D bitmap for the gun, we'll use our existing 3D model and manipulate it to display in front of the character. To implement this feature we'll create a new function called DrawPlayerGun. Before we begin manipulating the model/camera information we'll save the current matrix. Next we'll scale the gun so it'll be very small on our screen. In our case we'll scale it to be 0.0003f of its normal size. This number doesn't hold any special value; I just thought it looked nice when I was tweaking the example. You'll probably want to do the same because each model will be a different size and therefore need special tweaking for each model player. After scaling the current matrix, we'll position the gun on the right-hand side using the GL command glTranslate. Using tweaked numbers again, we're able to position the gun on what appears to be the right hand of the player. If you're interested in having left-hand support, you'd want to negate the X coordinate. After positioning the gun, we render the model and restore the original matrix. In our Render function, we'll call the DrawPlayerGun function so we can display our gun. The source code for DrawPlayerGun is written below.

```
void DrawPlayerGun()
{
   glPushMatrix();
      glScalef (0.0003f, 0.0003f, -0.0003f);
      glTranslatef(0.03f, -0.03f ,0.0f);
      RenderModel (item_gun);
   glPopMatrix();
}
```

Time-Based Movement

When we wrote our original movement code we designed it to continually poll the keyboard and mouse positions after every frame update. Although this is great for testing software, it can cause serious issues on different hardware. Imagine that one person has the latest video card with 100 FPS and his deathmatch buddy is playing on older hardware with 25 FPS. The player with the good hardware would walk circles around the other player, since he is getting four times as many frames per second. This is a huge issue in our game because we don't want someone with better hardware to have a big advantage over others. For this reason, we'll implement a time-based movement system in our game so the movement speed is adjusted based on the current frames per second. This feature is rather simple to implement. We simply update our movement functions and divide the movement speed by the old frames-per-second value. Based on the old FPS value, the movement speed will be adjusted, then multiplied by the sine/cosine values. So if our old FPS was 1, then our movement speed would be the full size. If the FPS was 2, our movement speed would be half its original value. As I mentioned earlier, this is quite simple and every movement function must be updated to support this function. The source code for the updated MoveForward function is given below.

```
void MoveForward(MAP_ENTITY &entity)
{
   double radians = PI / 180.0 * entity.angle[0];
   long elapsed_time = timeGetTime() - old_time;
   entity.xyz[0] -= sin(radians) * (MOVEMENT_SPEED / old_fps);
   entity.xyz[2] += cos(radians) * (MOVEMENT_SPEED / old_fps);
}
```

Setting Start Positions

In the past when starting our game we've used the coordinates 0,0,0 for the starting coordinates of the player. Since we want to add custom start points based on the type of game we're playing (single player or deathmatch), we'll update our LoadMap function to set the starting coordinates for the player. If we're playing a multiplayer game (multiplayer.active is true), then we'll set the player coordinates to the map.details.deathmatch[1] value if the user is the client of the game. If the user is the server, we'll set the starting position to the map.details.deathmatch[0] value. If we're not in a multiplayer game, we can simply set the starting position value to the value stored in map.details.single_player. This provides a great deal of flexibility and allows us to put players at different parts of the level so they don't immediately see each other. The source code for this is written below.

```
if (multiplayer.active)
{
   if (connection_type == MULTIPLAYER_CLIENT)
   {
      player.xyz[0] = map.details.deathmatch[1].xyz[0];
      player.xyz[2] = map.details.deathmatch[1].xyz[2];
   }
   else
   {
      player.xyz[0] = map.details.deathmatch[0].xyz[0];
      player.xyz[2] = map.details.deathmatch[0].xyz[2];
   }
}
else
{
   player.xyz[0] = map.details.single_player.xyz[0];
   player.xyz[2] = map.details.single_player.xyz[2];
}
```

Optimization Techniques

There are many methods of optimization that can be performed in OpenGL, including using display lists, sorting texture IDs, using vertex arrays, and testing for occlusion. This section discusses these techniques.

Using Display Lists

Although our rendering engine isn't complex compared to many retail games, there are still many optimizations that can be completed very easily. In Chapter 12, we created the function RenderObject to render an object's triangles. Without adding too much new code, we can optimize this rendering process by placing each object's triangle data into a display list instead. A display list allows you to cache a compiled list of OpenGL commands into an OpenGL display list number. When rendering a scene in OpenGL, you can call the display list number as opposed to continuously specifying each GL command. From the main rendering loop you can then call the display list number to render the data. When using display lists (depending on the hardware), you'll notice some increases in speed. In some cases the speed increase is minor, and in some cases it's quite drastic. The performance gains all depend on your graphics hardware.

To begin the coding process we'll create a new function called BuildDisplayLists, which as the name implies will build the display lists for our map data. In the function we'll loop through the different objects in the map, compiling each display list using the object number. To compile a display list we use the OpenGL command glNewList with the object number plus one to avoid a list name of 0. The second parameter of the GL command is the compilation mode, which describes how the list will be used. In our case we'll use the constant GL_COMPILE_AND_EXECUTE to compile and execute the display list. After we've called glNewList, we must run the OpenGL commands that we want the display list to process, which in our case are stored in the RenderObject function. Following the call to RenderObject, which will write the GL map rendering calls to the display list, we run the GL command glEndList to tell OpenGL we're done compiling our display list. Once the loop finishes compiling the different lists, we're finished with the function. The source code for the BuildDisplayLists function is shown below.

```
void BuildDisplayLists()
{
   for (long obj = 0; obj < map.header.max_objects; obj++)
   {
      glNewList (obj, GL_COMPILE_AND_EXECUTE);
         RenderObject (obj+1);
      glEndList();
   }
}
```

With the functionality now created to compile the GL display lists, we'll call the BuildDisplayLists function inside the LoadMap function to compile the display lists when we load our map data. The final step to implementing display lists is to update our rendering code to call the display lists instead of calling the RenderObject function. The functions that would be affected by this change in our game engine are the RenderLight function and the main render loop where we call RenderObject to render each object. To execute the commands in the display list, we use the GL command glCallList and specify the list number. For instance, if we were rendering the objects in the map, we would comment the RenderObject function call and simply add the call glCallList. The single parameter of the command is the list number, which is the object number plus one. Pretty simple, huh? The updated source code for the main object rendering loop is written below to show how to call our display lists.

```
for (long obj = 0; obj < map.header.max_objects; obj++)
{
   glColor3f (1.0f, 1.0f, 1.0f);

   for (long layer = 0; layer < MAX_TEXTURE_LAYERS; layer++)
   {
      if (map.object[obj].texture[layer].id > 0)
      {
         glActiveTextureARB (GL_TEXTURE0_ARB + layer);
         glBindTexture (GL_TEXTURE_2D, TextureName[map.object[obj].texture
                        [layer].id]);
      }
   }

   // RenderObject ( obj );
   glCallList ( obj );
}
```

Sorting Texture Binds

Another optimization that can be performed in our game engine is the sorting of the texture IDs. If we sort our objects by their texture IDs, we can then limit the number of texture binds that are being performed with each frame being rendered. This can have a huge effect on performance since we are only transferring bitmaps a few times instead of potentially hundreds. Even with compressed textures, this method can help. The sort simply involves classing the bitmaps by their numbers and checking the previous texture against the current one. If the texture IDs are different, then we'll bind a new texture; otherwise, we'll continue using the texture. This is slightly more difficult when using multitexturing because we must sort each texture layer, which can become annoying at times, but it's part of programming games.

Vertex Arrays

Another method of optimizing a rendering engine is by using vertex arrays. A vertex array allows you to push many vertices for an object at once. Rather than specifying 100 calls to glVertex for a specific object, you can specify the data in an array and the hardware will automatically run through them. Some video cards process this data faster than display lists and some have the opposite effect. This technique is ideal for use with static geometry because you don't have to worry about updating the arrays.

Occlusion Querying

Occlusion querying allows you to test if a triangle is behind another. This is a fantastic feature and is especially great for detailed landscapes because we can easily cut certain objects or triangles out of a scene using new extensions. Through occlusion testing we can simply see if the "bad guys" are on one side of a mountain when we're on the other side. If we cannot see them, we don't have to worry about binding the textures and drawing them. Although the hardware does attempt to do this on its own, in some cases we can create faster algorithms by generalizing information. For instance, if the characters on one side of the mountain haven't moved since the last frame rendered, we can ignore drawing them. As our user walks we could test for occlusion every tenth of a second, which would provide us a simple means for culling our models.

Chapter Example

Please see the example from the companion file (ex19_1).

Conclusion

In this chapter we discussed simple techniques that add some spice to our game. In game development there are literally hundreds and even thousands of techniques that could have been included in this chapter. I focused on the techniques that would be the easiest for beginners yet still useful to intermediate and advanced programmers. With the functionality discussed in the book, you should be able to write your own basic OpenGL video game with all the trimmings!

In writing this book I tried to discuss how to create games in OpenGL in a simple, plain-English manner. I hope you've gotten everything you wanted from this book and more! Of course that would be learning how to make games in OpenGL. It's been a long journey to this point, and I'd like to thank you for reading this book!

Sources

Michael Abrash's Graphics Programming Black Book, Special Edition, Michael Abrash, Coriolis Group Books, 1997.

OpenGL Programming Guide: The Official Guide to Learning OpenGL Version 1.1 (second edition), Mason Woo, Jackie Neider, Tom Davis, Addison-Wesley Publishing, 1997.

Windows 95 API How-To, Matthew Telles, Andrew Cooke, Waite Group Press, 1996.

"The OpenGL Graphics System: A Specification (Version 1.5)," Mark Segal, Kurt Akeley, Silicon Graphics, Inc., 2003 (http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf).

"OpenGL BOF," Jon Leech, SGI/OpenGL ARB Secretary (http://www.opengl.org/developers/code/features/siggraph2002_bof/sg2002bof_sgi.ppt).

"Creative OpenAL Programmer's Reference (Version 1.0)," Creative Technology Limited, 2001 (http://chopenal.sourceforge.net/OpenAL.pdf).

"OpenAL Specification and Reference (Version 1.0 Draft Edition)," Loki Software, 2000 (http://www.openal.org/oalspecs-annote/).

GL_ARB_texture_compression Extension Specification (2000), GL_EXT_abgr Extension Specification (1995), GL_EXT_fog_coord Extension Specification (1999), WGL_EXT_swap_control Extension Specification (1999) (http://oss.sgi.com/projects/ogl-sample/registry).

"BMP Format, Windows Bitmap File Format Specifications, V1.1," Wim Wouters (atlc.sourceforge.net/bmp.html).

"The Design of the OpenGL Graphics Interface," Mark Segal, Kurt Akeley, Silicon Graphics Systems (http://www.opengl.org/developers/documentation/white_papers/opengl/).

Microsoft Developer Network — http://msdn.microsoft.com

Silicon Graphics Inc. (SGI) — http://www.sgi.com

OpenAL — http://www.openal.org

OpenGL — http://www.opengl.org

Mesa — http://www.mesa3d.org

This page intentionally left blank.

Further Reading

Official Web Sites

Official OpenGL web site — http://www.opengl.org

Official Mesa web site — http://www.mesa3d.org

OpenGL Extension Registry — http://oss.sgi.com/projects/ogl-sample/registry/

Official OpenAL web site — http://www.openal.org

Hobbyist OpenGL Web Sites

NeHe — http://nehe.gamedev.net

Game tutorials — http://www.gametutorials.com

Code Sampler — http://www.codesampler.com

DevMaster — http://www.devmaster.net

News Groups and Mailing Lists

OpenGL news groups — comp.graphics.api.opengl and www.opengl.org/community/newsgroups.html

OpenGL Vendor Developer Sites

3Dlabs developer site — http://developer.3dlabs.com

This site has a lot of information about 3Dlabs' video cards and capabilities, along with example code and ideas on the future of OpenGL.

ATI developer site — http://www.ati.com/developer/

A great resource for developers with an interest in ATI-specific OpenGL features. Some of the examples here are wild!

Intel developer site — http://www.intel.com/developer/

Matrox developer site — http://developer.matrox.com

nVIDIA developer site — http://developer.nvidia.com

A great resource for any developer wanting to learn about OpenGL coding for nVIDIA video cards and about the Cg Shader language.

SiS developer site — http://www.sis.com

Software Sites

3D Modeling Software

Alias|Wavefront — http://www.aliaswavefront.com
Makers of Maya

Avid — http://www.avid.com
Makers of SoftImage

Blender Foundation — http://www.blender3d.com
Makers of Blender

chUmbaLum sOft — www.swissquake.ch/chumbalum-soft
Makers of MilkShape, an inexpensive modeling tool

Discreet — http://www.discreet.com
Makers of 3ds max and gmax

NewTek — http://www.newtek.com
Makers of LightWave

Side Effects Software — http://www.sidefx.com
Makers of Houdini

Paint Software

Adobe Systems — http://www.adobe.com
Makers of Photoshop, Illustrator, and many other packages

GIMP Foundation — http://www.gimp.org
Makers of the free GNU Image Manipulation Program (GIMP)

Jasc Software — http://www.jasc.com
Makers of Paint Shop Pro

Terrain Generation Software

Corel — http://www.corel.com
Makers of Corel Bryce

Terragen — http://www.planetside.co.uk/terragen
Makers of Terragen

Index