游戏编程精粹 1

GAME PROGRAMMING Gems

[美] Mark DeLoura 编
王淑礼 张 磊 译
姚 勇 审校

内附光盘

人民邮电出版社
POSTS & TELECOM PRESS

游戏编程精粹 1

GAME PROGRAMMING Gems

作为游戏程序员，您是否常常苦于在开发中遇到难题却无处获取真正有用的帮助？Game Developer 杂志主编 Mark DeLoura 首次将全球游戏开发业 40 多位顶尖高手聚集在一起，站在技术开发者的立场上，为着促进整个行业发展的理想，把大家多年摸爬滚打积累的经验和成果无私贡献出来，创作了这本赢得全世界游戏程序员无数叫好声的《游戏编程精粹 1》。

本书覆盖了游戏编程的各个关键领域，涉及动画、人工智能、Z 缓冲、光照计算、气象效果、Internet 多玩家游戏、音乐与音效等。所提供的不仅是解决难点的思路和理论方法，而是能立即放入您代码中的实用工具。

这些前辈高手们智慧的结晶，无论对行业高手还是入门新手，都将是笔宝贵的资源财富。

ISBN 7-115-12587-2

9 787115 125873 >

ISBN7-115-12587-2/TP·4164
定价：80.00 元（附光盘）

人民邮电出版社网址 www.ptpress.com.cn

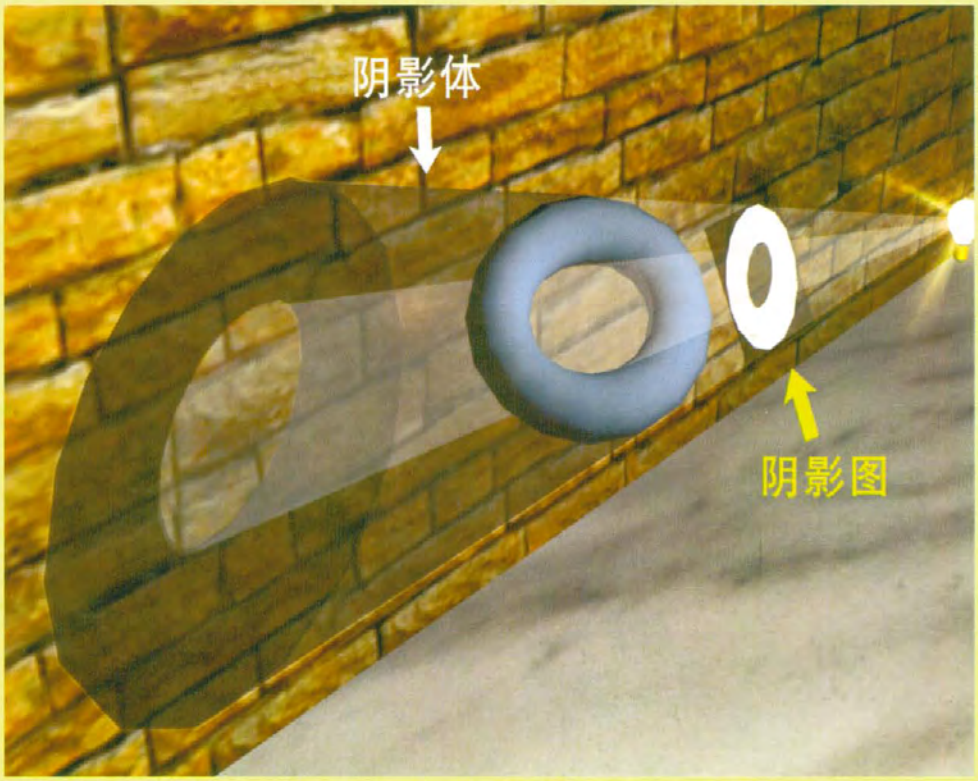
彩图 1
使用断层构造生成的分形风景 3D 渲染

彩图 2
使用中点置换生成的分形风景 3D 渲染

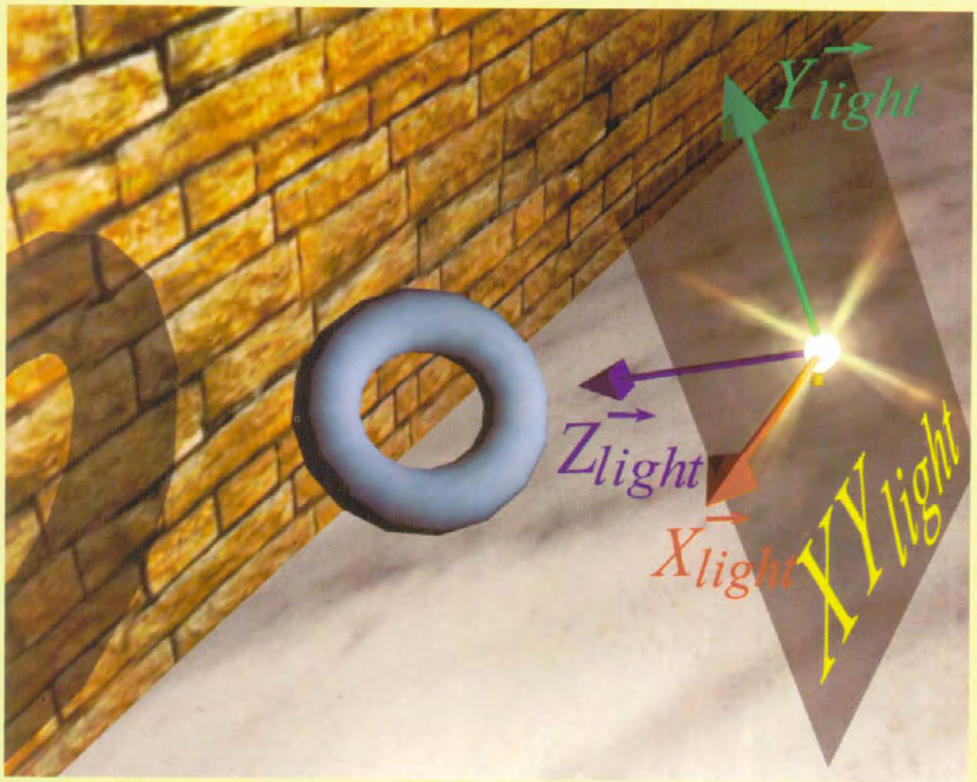
彩图3
使用粒子沉积生成的分形风景3D渲染

彩图4
将镜头光晕特效作为2D渲染，重叠在3D场景上

彩图 5
室外环境的环境图渲染在圆环上

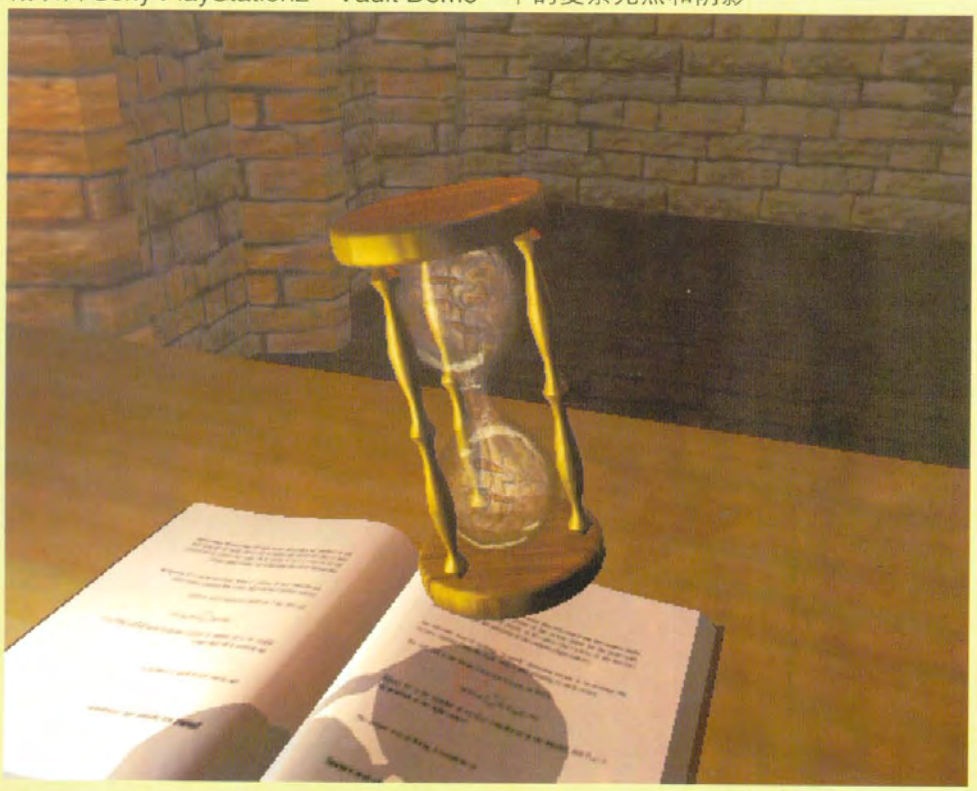


彩图 6
圆环的阴影图投影在周围环境中



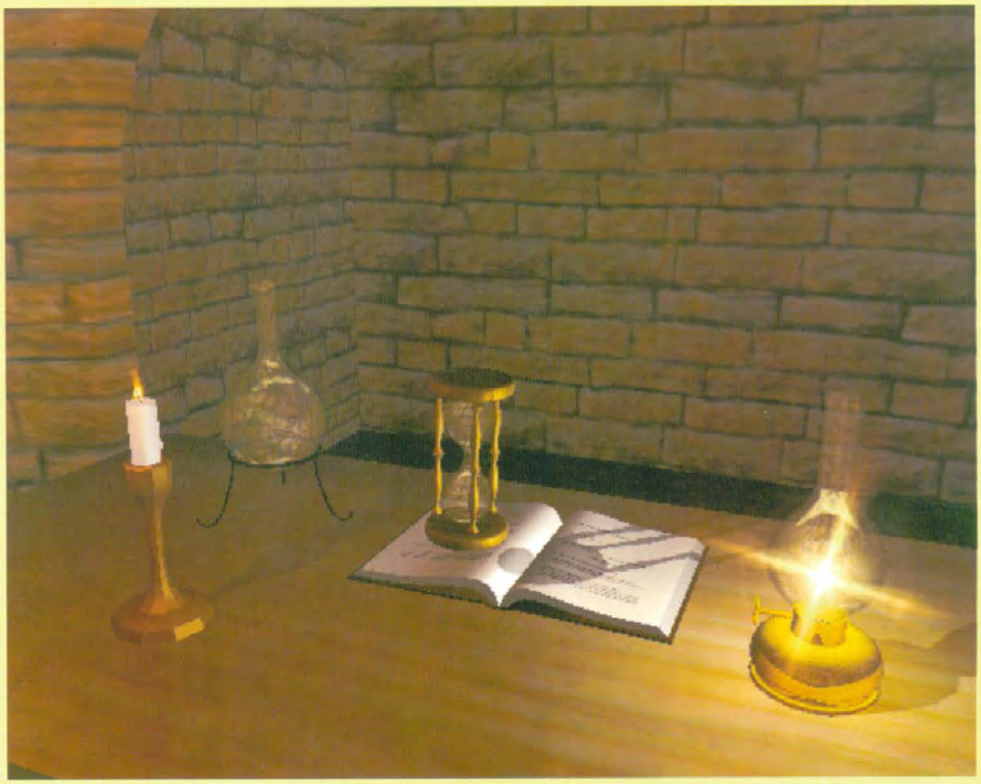
彩图 7
渲染某光源在环境中的投影所使用的光坐标系统

彩图 8 ~11
截取自 Sony PlayStation2“Vault Demo”中的复杂光照和阴影

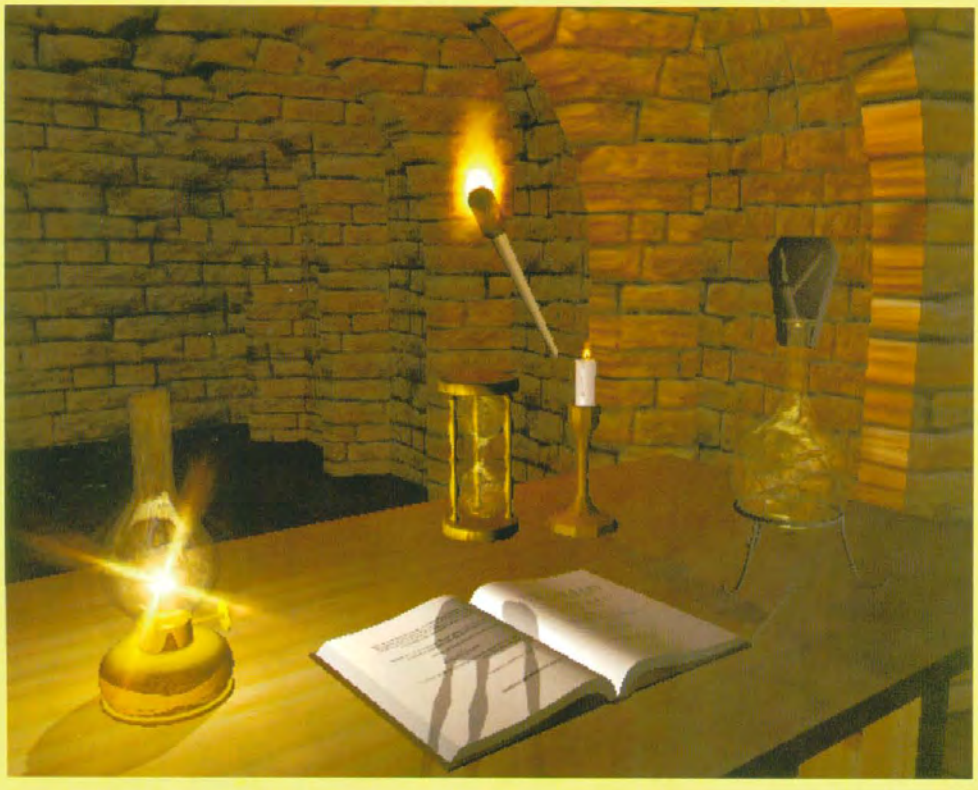


彩图 8

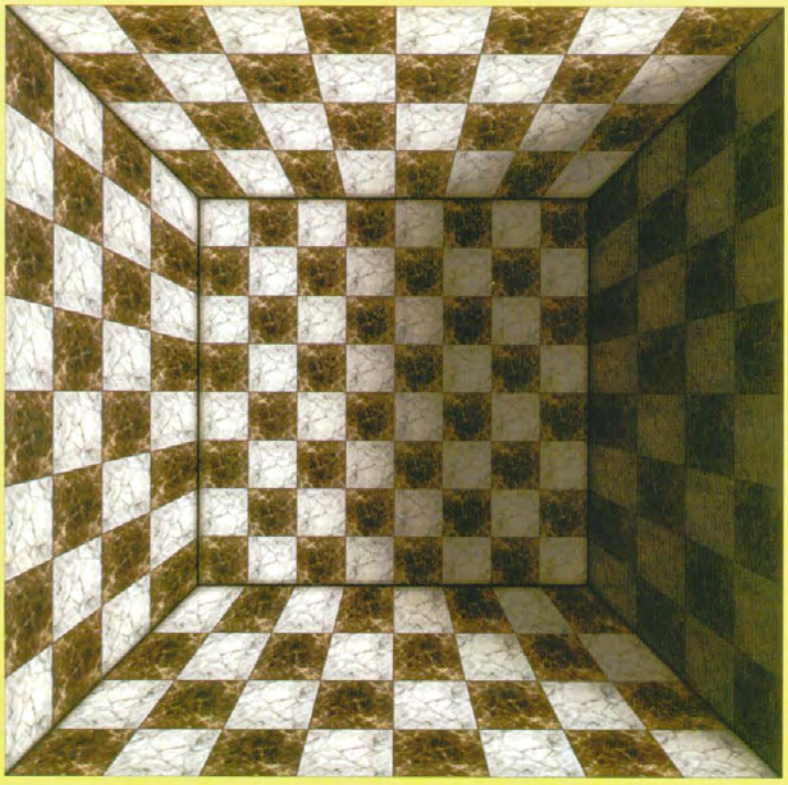
彩图 9



彩图 10



彩图 11



彩图 12
用于实时折射映射演示的纹理图

彩图 13~14

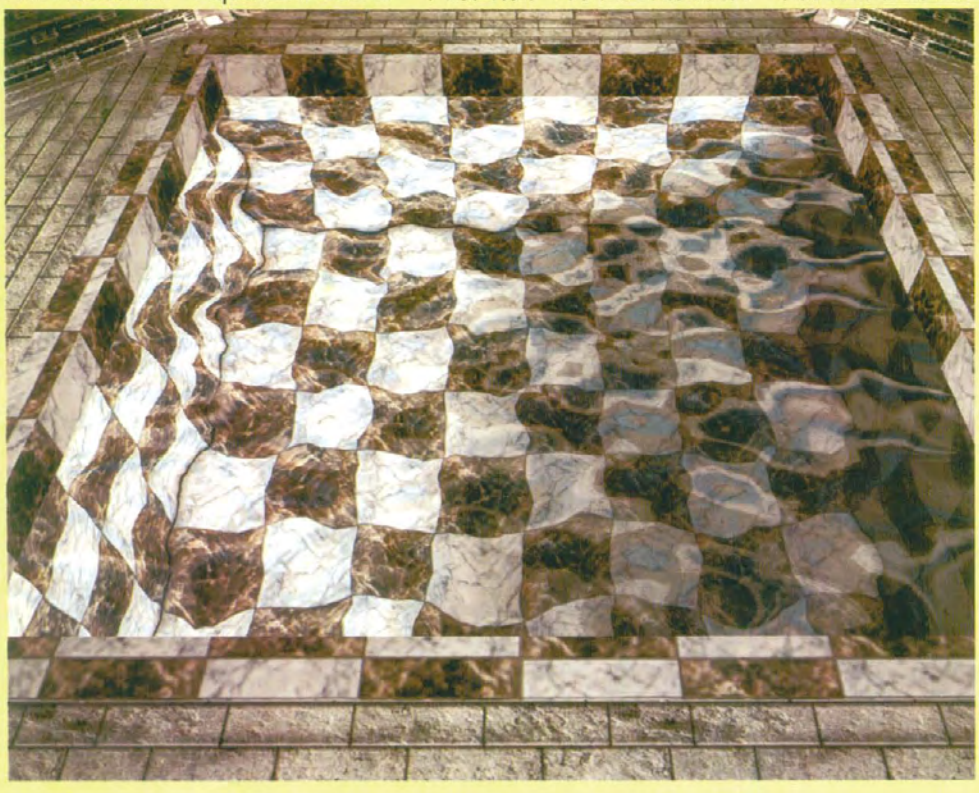
radiAtIon 图形引擎中的图像，对实时折射映射、环境映射和刻蚀进行了展示

彩图 13

彩图 14

彩图 15~16

ATI Research“Aquarium Demo”中的图像，对实时折射映射、环境映射和刻蚀进行了展示



彩图 15

彩图 16

内容提要

本书是由 40 多位国外游戏开发行业最为优秀的程序员撰稿的技术文集。每篇文章都针对游戏编程中的某个特定问题，不仅提供了解决思路，还给出了能立即应用到代码中的算法和源码。全书分为通用编程技术、数学技巧、人工智能、多边形技术和像素特效五章；附录部分提供了两个非常有用的工具库，矩阵工具库和文本工具库；随书附带光盘中包含有全书所有的源程序、演示程序、附录中的两个工具库以及 glSetup 和 GLUT 等开发工具。

本书适合游戏开发专业人员阅读。专家级开发人员可以立刻应用书中介绍的技巧，而初中级程序员通过阅读本书将增强其技能和知识。本书是游戏程序员必备的参考资料。

关于翻译审校

姚勇，1998 年从清华大学毕业并留校工作，从那时起开始研究 Realtime Rendering 和游戏引擎技术。2000 年他组建 H3D Studio，开始正式研发 3D 游戏引擎。经历了 4 年多的起伏跌宕，H3D 由当初 2、3 人的小组发展成今天的大型游戏软件开发公司，组建起了一支技术强劲的引擎开发及游戏产品制作队伍，并立志成为中国游戏制作的 HARDCORE。如今姚勇带领下的 H3D 正在用历经 4 年研发的 H3D 大型 3D 网络游戏引擎平台，制作一款重型 3D 网络游戏。

H3D 公司的网址：http://www.hardcore3d.net。

姚勇的 Email 地址：puzzy@hardcore3d.net。

前　言

Mark DeLoura

欢迎来到《游戏编程精粹 1》！我很自豪地将本书贡献给读者。在本书中，你将读到结合了 40 多位天才游戏开发者的智慧结晶。这些开发者聚集在一起创作了这本游戏编程技巧图书，可以给你来一次“POWER UP”！如果实现了这些他们通过长期经验积累而开发出的技术，你游戏中的敌人会更加聪明，你的英雄刺穿精灵的动作会更平滑，你的 3D 环境能吓得玩家在夜里跑去关灯。

作为任天堂公司的一名主力工程师，我与很多游戏开发者有过沟通。不论他们的问题是关于我们最新游戏机硬件的，还是关于复杂的游戏算法的，有一点很明确——我们都有很多问题。作为游戏开发者，我们常常不得不完成很多不知道如何下手的任务。当然，那是作为游戏程序员的部分乐趣所在。可是如果这些情况发生，你会向何处寻求帮助呢？在书架上找，还是上网？也许你会去搜遍大堆的杂志。对于游戏开发来说，根本就不存在最可靠的资源。不过，如果有一个地方，你知道可以先去那里找找看，岂不是很棒？这就是我们编写此书的出发点。

游戏开发是一个广泛的领域，它将相同的计算机图形学技术与很多来自其他领域的复杂算法（如人工智能和交互式音乐）结合。但当我问游戏程序员“你们真正希望在你们书架上有什么样的书？”时，众多的回答是——一本集专业编程技术之大成，又专供游戏编程所用的。本书饱含经由长期编程而得来的技术，无论对专家还是新手，都会有帮助。

本书中的文章阐述了游戏程序员可能面对的大部分技术性问题，其中有很多是关于具体技术的，也有不少章节的内容比较通用。这样做的意图是：不论读者的编程水平如何，通过阅读本书，专业水平都能上一个新的台阶。例如在技术较为通用的文章中，我们先给出了关于技术的概述，随之对此技术的使用进行更详细的探讨（如关于四元数的文章以及关于 A* 路径策略算法的系列文章）。

关于标准

在创作本书时有一个棘手的问题，在技术发展如此迅猛的环境下，我们如何展示这些技术资料。显然，答案是立足于现有的标准。所以本书中的所有文章均采用 C 或 C++来编程，用 OpenGL 作为图形语言，以 Windows

和 Linux 作为运行环境。在我们行业里，工作平台将不断地发展，这一点是肯定的。我们希望通过尽可能使用可移植性最好的语言、以最流行的操作系统为平台，本书的知识在今后的 10 年中仍然具实用价值。（实际上本书中有几篇优秀的文章是严格绑定于 Windows 和 Visual C++环境的，但是它们太优秀了，我们也就顾不得平台独立性的要求了。）

标准在我们行业里非常重要。随着大公司之间以专利注册的形式将新技术瓜分殆尽，我们之间将知识拿出来共享就变得格外重要了。为什么？在我最近玩过的最好的游戏中，有些是出自一些小的游戏开发工作室的，如果他们不得不担心专利方面的限制以及许可费用的话，很有可能一些真正的“精粹”永远不会产生！我们可以避开软件专利，与同仁共享信息资源，使用第三方的标准，互相支持鼓励，制作出更好的游戏。

我们可以用不属于任何一家公司的技术（如 C++程序语言）作为我们的工具，而不要去碰某些公司赖以扩张势力的那些玩意儿。随着游戏开发周期变得越来越长，是不是得把游戏导出到竞争对手的操作系统或库上，这是我们最用不着担心的问题。谁需要那样的压力啊？我们已经受够了。

你也许听说过，直到通用胶片的尺寸和技术标准出台后，电影业才真正起步。这当然是那些提倡创建游戏业标准的人传出来的。不过，没问题，让我们来面对它吧。我只会采用那些对我有益的标准。如果有些标准库，我不能看到它的代码，或是它编写得很糟且运行得很慢，我为什么要使用它呢？这就是为什么公开源码的思想在我们业内如此关键的原因。学习一项技术的最好可能方法就是阅读源代码。标准的确很重要，但是除非我们能看到源代码并放心它不会阻碍我们游戏的性能，否则它起不了什么作用。

GLUT（OpenGL 实用工具包）的例子很能说明问题。它告诉我们，软件是怎样为我们服好务，而又因其优异的性能成为标准的。Mark Kilgard 的工具包之所以广泛流行，就得益于他将源码完全公开，这个软件已经设计得如此精确完美，如果你在做简单跨平台应用的地方不使用它，那你真是太蠢了。今天，GLUT 是一个独立于平台的工具包，它非常健壮而且实用。本书中的源代码使用 GLUT 来在多平台上运行。如果没有 GLUT 的话，说实话，让我们自己来做这项工作将是件非常费力的事情。

好了，已经说了很多题外话了，我希望本书可以激发读者把自己的优秀技术写出来，给《游戏开发者》（*Game Developer*）杂志（www.gdmag.com）、Gamasutras（www.gamasutra.com）投稿，或提交给游戏开发者会议（Game Developer's Conference）（www.gdconf.com）。你把专业知识传播给同仁了，每个人都是赢家：你赢得了声望，与他人建立了联系；学习到你知识的人则在你的智慧中受益。

故事的力量

好的故事可以深深地影响人们。人们记得他们读过的最好的书、看过的最好的电影，他们把这些故事与自己联系起来，好像就是他们现实生活中的一部分。在某种意义上来说，也确实如此。好的故事中包含有信息，或含蓄或明显，人们领悟并将其融入自己的生活中。

我们都喜欢让自己置身于好故事中，游戏则是最好的故事。在游戏中，你不再是旁观者——你是主角！在这种情况下，我们传达的信息更具影响力。我们应该对自己创作的游戏内容负责，这一点很重要。在又一起校园枪杀暴力事件引起的一片责骂声中，我们游戏行业

很可能首当其冲成为人们攻击的目标。其他的故事传播媒体（如图书、电视、电影），都有严格的规定和分类，其目的就是为了保护公众不受到负面讯息的影响。对我们来说极其重要的一点是，要清楚自己游戏创作的意图，明智、合理地表述故事，以避免其他媒体所遭遇的那类文化审查。随着游戏开发业规模不断壮大，地位日益显著，这肯定会成为我们将要面临的问题。

现在正是加入我们行业的绝佳时机，我们正处在打进主流技术的入口处。你把最新的 Miyamoto 下载到你的桌面上，而不用给所看的节目付费，这个日子已为时不远。5 年或 10 年后的世界将会怎样？我已经迫不及待想要看到了。

致谢

创作这样的一本书当然不是一个人所能完成的任务。事实上，如果让我独自来做，我很有可能要去跳楼了。幸运的是，在本书的制作过程中我得到了很多能人的帮助。当然，我不可能在此处对他们一一致谢（听上去好像奥斯卡颁奖演说了……），但我还是要说：首先，非常感谢书中文章的作者们，没有他们多年的经验以及为写这些文章付出的辛劳，也就不会有这本书；同样，非常感谢出版者 Charles River Media，特别是 Jenifer Niles 和 Dave Pallai，当我首次告诉他们这个想法的时候，他们非常支持，并自始至终给予我热情的帮助，给我出点子提建议。有些游戏开发者尽管没有参加本书的写作，但也积极参与并对本书提供了大量的帮助，他们是：Jeff Lander、Chris Hecker、Chris Taylor、Mark Haigh-Hutchinson、Richard Nelson、Stephen White、Mark Kilgard 以及 Elaine Hutchison。Dante Treglia 和 Steve Rabin 各写了很多篇文章，并且当我需要为本书挑选文章的时候，他们给了我很多宝贵意见。还有许多人尽管对本书没有直接的影响，但对我个人却帮助很大：Anderw Glassner 的“图形学精粹（Graphics Gems）”系列图书给了我很大帮助；Adrienne McEntee 和 Blyth Benshoof 给了我精神上的支持；Sonja 给了我友情支持；任天堂公司的 Jim Merrick 帮助我处理了法律方面的麻烦，使这本书得以创作；我的父母和朋友们则让我时刻保持清醒的头脑。感谢 Jennifer Pahlka、Alan Yu 和 CMP 公司游戏媒体组（Game Media Group）的 Brad Kane 为促进我们行业的交流所做的杰出工作。

希望你能喜欢这本书！

—Mark A. DeLoura

关于封面图案

Andrew Kimse

该场景截取自《地牢危机》（*Dungeon Siege*），Gas Powered Games 公司 2001 年发布的一款游戏。《地牢危机》是一款动作类角色扮演游戏，结合了角色扮演（RPG）和实时策略（RTS）游戏的特点。该游戏使用用户控制的浮动摄像机，提供 3D 沉浸式场景。此场景位于游戏世界的北部区域。作为区域的通用图块是使用 3D Studio Max 软件创建并贴图的，然后通用室内编辑器导出并构造。人物动画是通过在骨架系统上使用单皮肤网格实现的。动画系统检测人物的武器类型，然后相应地调整人物姿势和攻击方法。游戏使用了高级粒子系统来实现雪、雨、火、魔法的特殊效果。

目　录

第 1 章　通用编程技术

第 2 章 数学技巧

第 3 章 人工智能

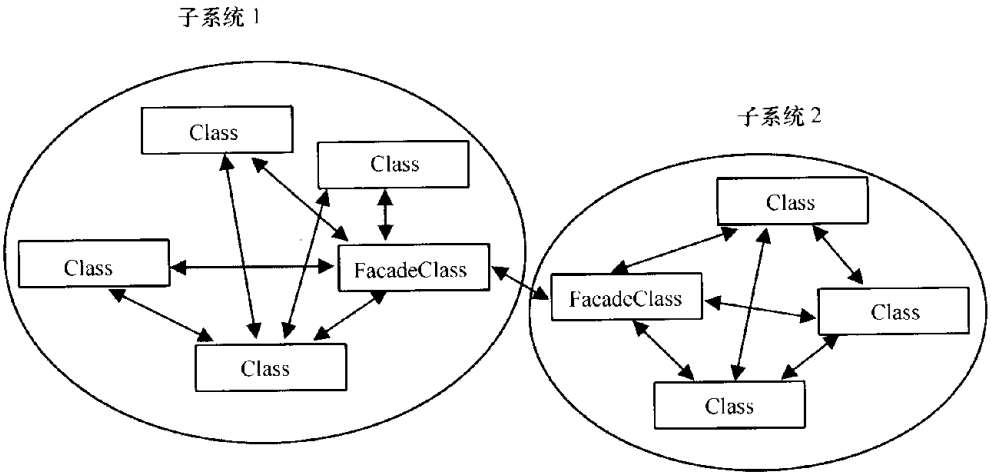
第 4 章 多边形技术

第5章 像素特效

第6章 附 录

第 1 章

通用编程技术


子系统 1
Class
Class
Class
FacadeClass
Class
子系统 2
Class
FacadeClass
Class
Class

1.0 神奇的数据驱动设计

Steve Rabin

游戏由两部分组成，逻辑和数据。这是一种有效的划分。分开来，它们一点用处也没有，不过合在一起，它们就能让你的游戏充满生机。逻辑部分定义游戏引擎的核心原则和算法，数据部分则提供其内容和行为的具体细节。当逻辑和数据相互分离开来，并能独立发展，则会产生神奇的效果。

显然，游戏数据应该从文件载入，而不应该内嵌在代码中。如何适度把握其中的平衡则是问题的关键。本篇文章给出了 7 个点子，它们将革新你做游戏的方式，或者至少能解决你的一些困惑。

1.0.1 点子 1——基础

创建一个能按需（而不是仅仅从系统启动的时候起）解析文本文件的系统。这对于数据驱动设计的有效应用来说至关重要。每个游戏都需要有一种清晰的方式读取通常意义上的数据，游戏最终是以二进制文件来读取的，但在开发期间能以文本文件来读至关重要。文本文件如果只用作编辑修改是没有什么实际意义的，它的真正意义在于——不用修改一行代码，你的整个小组，包括测试人员和游戏设计人员就能进行各种新尝试，并改变不同的东西来进行测试。这样，实现起来很简单的文本文件按需解析系统就成了一个必不可少的工具。

1.0.2 点子 2——最低标准

不要硬编码常量；把常量放进文本文件中，这样进行改变时就不用重新编译代码了。例如，像摄像机（camera）行为这样的基本功能就应该完全暴露。如果处理得当，游戏设计者、制作者，甚至街上的小孩都能最多使用一下记事本就能改变摄像机行为。游戏设计者和制作者通常会受控于程序员。通过暴露算法常量，那些非程序员们就可以用各种值来进行调试以达到他们期望的确切行为，而不用打扰任何一个程序员。

1.0.3 点子 3——杜绝硬编码

要假定任何东西都可能（并且很有可能）改变。如果游戏调用屏幕拆

分，不要用硬编码来实现。把你的游戏写成能支持任意数量的视区，每个视区都有其自身的摄像机逻辑。只要采用正确的设计，这并不会增加什么工作量。通过文本文件的神奇作用，你可以任意定义游戏采用单屏、双屏或四分屏。文件中还应该定义所有摄像机的初始值，如位置、方向、视野以及倾角。这样，你的游戏设计师在文本文件中能对所有的元素进行改变。

如果核心设计决策很灵活，游戏就可以逐渐演化，最终发挥其最大的潜力。事实上，对游戏进行抽象，将其核心部分抽象出来，这个过程本身就可以对设计产生极大的帮助。你可以把每个组件设计成其通用的功能，而不用为了一个单一的目标对其进行设计。其效果是，设计的灵活性迫使你去认识什么才是真正应该建立的，而不只是设计文档中所列举的具有局限性的行为。

举例来说，如果游戏仅采用4种类型的武器，你可以编写一个包括这些类型武器的完美的系统。不过，如果你将每种武器的功能抽象化，使用数据来定义其行为，就可以允许无数种有其具体特性的武器生成，所需的仅是在文本文件中进行一点修改，以对新的想法和游戏运行时的改变进行测试。这种智能性允许游戏逐渐发展并最终成为一个更好的游戏。

我用“杜绝”二字，你是否表示怀疑?

事实上，游戏需要进行调试，优秀的游戏会逐渐发展，并与最先的预想相差甚远。你的游戏应该有能力对规则、角色、种族、武器、关卡、控制方案以及对象的改变进行处理。没有这种灵活性，改变的代价很大，并且每一个改变都需要有程序员参与——这完全是资源浪费。如果难以进行改变，游戏很难从原始的版本有大的改变，它就不能逐步发展以挖掘出其潜力。

1.0.4 点子4——将控制流写成脚本

脚本只是一种在代码外定义行为的方法。脚本对于定义游戏中的步骤顺序或需要触发的游戏事件非常有用。例如，游戏中的cut-scene应该写成脚本。简单的因果逻辑也应该写成脚本，如一个搜索的完成状态或环境触发器。这些都是应用数据驱动思想的极好例子。

设计脚本语言时，需要考虑一下分支指令。分支有两种办法：一种是把变量保留在脚本语言中，用数学运算符（如等于“=”或小于“<”）进行比较；另一种是直接调用独立存在于代码中用于比较变量的评价函数，如IsLifeBelowPercentage(50)。你应该使用二者的结合，但要保持脚本的简洁。游戏设计师对处理评价函数要比声明变量、更新它们并进行比较轻松得多。这对于调试也更容易。

不幸的是编写脚本需要一种脚本语言，这表明你需要创建一种全新的语法来定义你的行为。脚本语言还需要脚本分析器，还可能需要一个编译器来将脚本转化成二进制文件以便能快速执行。另一种选择是使用一种现在已有的语言，如Java，不过这也需要大量的外围支持。为了不在这上面花费太多的时间，我们应该设计一个简单的系统。现在的总体趋势偏向于让脚本语言过于强大。在下一个点子中将讲解一些复杂脚本语言的陷阱。

1.0.5 点子 5——什么时候不适合使用脚本?

使用脚本编写数据驱动行为是使用数据驱动方法的自然后果。不过你需要记住数据驱动的核心思想：将逻辑和数据分开，复杂的逻辑在代码中运行，数据则保留在外面。

如果过于想以数据驱动游戏，就会有问题产生。有些时候，你可能会试图将一些复杂的逻辑放在脚本中。当脚本中出现状态信息并需要进行分支时，脚本就成了一个有限状态机（finite state machine）。当状态的数量增多时，那些无辜的脚本编写者（可怜的游戏设计师）就得进行编程的工作。如果脚本的编写变得足够复杂，这项工作就得转交给程序员，而他们必须用这种有很大局限性的语言来编程。脚本是为了让大家的工作更容易，而不是更困难。

为什么把复杂的逻辑保持在代码中有这么重要？其原因就在于功能性和调试方面的问题。由于脚本不直接在代码中，它们需要重复很多编程语言中已有的概念。这种倾向的结果就是，让其拥有越来越多的功能，直到能与真实语言媲美。脚本变得更复杂，我们就需要更多的调试信息，以用来找出脚本中的错误。这种附加信息最终将迫使越来越多的精力投入到了对脚本运行时各方面的监控上。

你可能已经想到了，脚本中复杂的逻辑会耗去很多精力，可能数月的工作就浪费于编写脚本分析器、编译器和调试器中。就好像程序员没有认识到他们本来就拥有一个很好的编译器。

模糊的边界

代码和脚本之间的边界很模糊，这是勿庸置疑的。通常来说，将人工智能（AI）行为写进脚本中并不好，然而拥有一个脚本触发系统使得世界能够进行互动却是个好主意。其原则应该是，如果逻辑太复杂，它应该放在代码中。脚本语言需要保持简洁，所以它们不会消耗你的游戏（以及你所有的程序资源）。

但是，有些游戏设计让玩家来写他们自己的 AI。一般来说，在第一人称视角射击游戏中允许创建机器人。当这种设计成为目标时，脚本语言不可避免会很像真正的编程语言。这种情况的一个例子是 Quake C。因为机器人创建是设计提出的需求，所以必须投入资源和精力开发脚本语言，使其和 C 语言一样有用。开发这种重量级的脚本语言是一项巨大的负担，你千万不要忽视其可能消耗的精力。

总之要记住，你不希望自己的游戏设计师或脚本编写人员对游戏进行编程。有些时候，当程序员创建脚本语言时，他们会试图逃避责任。“引诱”游戏设计师对游戏进行编程实在是太容易了。理想的情况是，程序员应该处理问题并把基本控制暴露出来以便对逻辑进行操纵。

1.0.6 点子 6——避免重复数据

绝对不要复制代码，这是一条编程实践标准。如果在两个不同的地方你需要用相同的行为（例如一个公共函数），这个行为只能存在于一个地方。这种思想也可应用于数据，可以采用引用来指向全局数据块。甚至，通过使用引用指向数据块并修改其中的一些值，可以实现类似继承的概念。

继承是一种伟大的思想，它可以应用到数据上。假想你的游戏中有生活在地下城的精灵，

在任何特定地下城中，有数据定义每个精灵站在哪里、其特性如何。封装这些数据的正确方法是使用一个全局的精灵定义。每个地下城的数据仅仅有一个引用，指向这个用于所有精灵实例的全局定义。为了让每个精灵具有独特性，指针可以伴随一个属性列表以进行重载。这种技术在避免复制数据的同时还允许每个精灵都不相同。

让每个数据块都有一个引用，这种思想就可以用于多重关卡了。采用这种技术，你可以使用精灵的全局定义，并伴随一个快速精灵的全局定义，这种快速精灵由基本的精灵继承而来。然后在每个地下城定义中，常规精灵和快速精灵的实例创建就很容易了。图 1.0.1 展示了这种采用引用和值覆盖的继承概念。

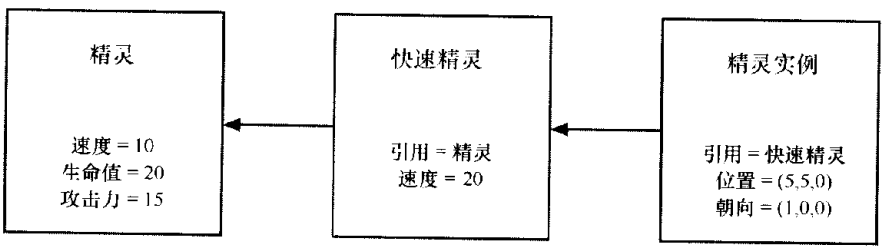


图 1.0.1 数据继承

1.0.7 点子 7——开发工具来生成数据

在大型游戏中，文本文件最终会变得不好控制和处理。实际的解决方法是采用一个工具来写这个文本文件。这种工具可以叫作游戏编辑器、关卡编辑器或脚本编辑器。采用工具来写并不会改变数据驱动的思想，它只会使得数据驱动更健壮、更高效。比较你所节省的时间，这额外工具开发所耗的时间是值得的。

1.0.8 结论

采用数据驱动方法很容易，但要得到显著的效果却不容易。当所有的东西都采用数据驱动的时候，你将拥有无限种发展的可能。

采用这条原则的一个例子是游戏 *Total Annihilation*（横扫千军），该游戏的设计师 Chris Taylor 将数据驱动设计推到了极限。*Total Annihilation* 是个 RTS（实时策略）游戏，它定义了两个不同的族，Arm 和 Core。虽然游戏以这两个种族为核心，但它们都没有被硬编码到游戏中。在理论上，甚至当游戏封装好了以后，还可以再加进新的数据使得游戏支持 3 个族。虽然这种可能性没有被挖掘出来，但 *Total Annihilation* 也充分利用了它的灵活性。由于它所有的单元都完全由数据定义，新的单元每周在该游戏网站上发布。事实上，很多人创建了他们自己的单元，其功能甚至震撼了游戏开发者。

数据驱动使得 *Total Annihiliation* 拥有了一群争先恐后的追随者。在 *Total Annihiliation* 之后，其他的游戏，如 *The Sims*，使用了同样的思想，在他们的网站上提供新的数据内容。没有开发者对数据驱动思想的投入与实践，这种空前的可扩展性是不可能的。

1.1 面向对象的编程与设计技术

James Boer

专业的游戏编程普遍采用C++是不难理解的。在没有减少太多C语言高效性和优秀的可移植性的同时，C++还提供了面向对象语言的设计优势。继承也就成了正确设计与实现C++代码的要求。虽然面向对象程序设计（OOP）是用来提高程序设计、可移植性和可维护性的，但设计得不好的C++程序要比设计得不好的C程序更糟，这是个残酷的事实。

很多文章和书籍都对通用的面向对象设计实践给出了很好的建议，但我印象中很少有针对游戏编程实践的。游戏程序员与典型的应用程序员有些不同。由于游戏程序员总是工作在最前沿的，要把人类和硬件的限制推到极限，因而他们往往更愿意改变甚至打破传统的程序设计原则。不幸的是，这种倾向常常会由于对基本OOP原则理解或实现的不当而产生负面效应，产生不可维护的代码。

随着游戏变得越来越复杂，游戏公司希望通过复用更多的代码来减少持续增长的开发费用。随着公司对内容及游戏可玩性的关注，引擎授权变得越来越受重视了，而且在不久的将来肯定会成长为一个主要且独立的产业。这类开发工作相比以前的游戏开发来说，则要求更高的稳定性与长期的规划性。每做一个新游戏就完全废弃你以前的代码，这在不久的将来可是不可行的。

本篇文章显然不可能覆盖游戏程序员需要知道的所有知识，但它指出了一些关键领域，使游戏程序员和游戏公司能提高产品代码的质量和一致性，从而产生更健壮且复用性更好的库和游戏引擎。我们还给你提供了一些资源，它们对该主题有更完整的讨论。

1.1.1 代码风格

编程风格常常会演变成“信仰”上的争论。我并不是想卷入此类争论，但是对于一个公司来说，采用一种风格，并让公司的每个人都使用这种风格是很重要的。

公司（不是指个人）应该全力争取类、函数以及变量命名约定上的一致性。许多公司采用一种简化的匈牙利标注法（Hungarian notation）方案。匈牙利标注法是由Charles Simonyi博士，微软公司的首席软件架构师，在数年前为了标准化变量命名约定而发明的。有争论说这样的命名约定对于像C++这样的“类型安全”语言来说并不必要，而且会在转换数据类型时增加

工作量（因为需要改变变量前缀）；但也有人非常欣赏这种数据类型一目了然的方便与快速。

匈牙利标注法的基本内容是，在变量名的前面加上描述该变量数据类型的标识。例如，一个叫作 SomeVariable 的整型变量应该命名为 iSomevariable。除变量类型外，指针也应该被表示出来。一个指向 Foo 类的指针可以为 pFooObj。前缀之间可以进行结合，以提供比单一前缀更多的信息。例如，指向整型数的指针可以用前缀 pi 表示，指向指针的指针可以用前缀 pp 表示。

关于范围信息的其他类型通常放在类型前缀的前面。成员变量用 m_标识，整型成员变量可标识为 m_iSomeVar。全局变量（不是必须的）标识为 g_，还有些表示静态变量的变量标识为 s_（这并不常见）。正规的匈牙利标注法比较复杂，许多公司都采用它的简化版本。表 1.1.1 是一个常用的正规匈牙利标注法变形版本。你可以在别的书（如[Petzold96]）中，或在 Simonyi 的原始论文（万维网的很多地方都可以找到）找到其他的描述。

表 1.1.1 匈牙利标注法举例

类型	描述
I	Integer
F	Float
D	Double (float)
L	Long (nteger)
C	Character
B	Boolean
Dw	Double word
W	Word
by 或 byte	Byte
Sz	C-style (null-terminated) string
常用扩展名	**描述**
Str	C++ 字符串对象
H	句柄 (自定义类型)
V	向量 (自定义类)
Rgb	RGB 三元组 (自定义结构或类型)
变址（Modifier）	**描述**
P	指针
R	索引
U	非符号
a 或 ary	数组
范围	**描述**
m_	成员变量
g_	全局变量
s_	静态变量

表中列出了最常用的标注类型。除了几个常用的类以外（如那些表示 3D 向量和指针的类），对象通常不加任何前缀。你的公司可能对其他常用工具类也使用了约定。要注意，绝大多数描述标记都很有逻辑性，不需要在表里进行查找。

你所采用语法的精确并不如所有人遵守它的一致来得重要。如果公司所有的代码看上去都很相似，程序员可能就不需要写，而是轻松地在原有代码上进行工作了。

有一句话要注意——不要在代码规范上做得太过。一两页纸就应该包含所有的公司风格要求描述。如果程序员需要查找变量如何命名，他们会很不愿意使用这种标准。对于严格遵循 Simonyi 的原始系统，我持保留态度。对于日常的操作来说，它太复杂了；况且现在可读性比类型安全更为重要，简化的版本就可以达到同样的效果，没有必要创建难以阅读的代码。

类名的设计也应该易于阅读和维护。使用类前缀来表示通用的设计意图是 Windows 程序员中使用较广泛的约定。以字母 C 打头的类表示具体类（concrete class），或有特定用途和实现的类；以字母 I 打头的类是接口类，或意图用作设计模板的类。这些类并不直接用于应用程序，而是用于派生其他的类。

用功能作为类前缀也是很有用的，它可以补充或代替上述前缀。例如，所有处理用户界面（User Interface，UI）系统的类可以在前面加上 UI。这对于对类按工程字母进行排序了的编程环境或工具尤为有用。

1.1.2 类设计

C++的类提供了设计的无限灵活性，这既是好事又是坏事。除了构造函数（constructor）与构析函数（destructor）外，C++的类中没有命名要求。不过你可能想要自己建立一个类命名标准，以下是一个简单的示例：

```
class Sample
{
public:
    Sample()          { Clear(); }
    ~Sample()         { Destroy(); }

    void Clear();

    bool Create();
    void Update();
    void Destroy();
};
```

这个类中你应注意的第一件事是不太起眼的构造函数。基于很多理由，最好以这种方式实现类。一开始，C++构造函数没有返回值，因此在这里最好不要做任何可能出错的事情。我们在此简单调用 Clear()，这个函数用于清空所有内部成员变量。在独立的函数中清除变量的好处是，你可以在任何时候清除类的变量。稍后你就会看到为什么这点尤为重要。

有些时候，你不想在类刚创建的时候就“激活”它，这经常在封装那些是自己的成员的类时发生。最后是关于效率的问题。将对象的实际创建点与构造函数分离，你就可以动态创

建对象一次，但重复调用 Create()和 Destroy()成员函数来复用同一个对象的内存。动态分配内存的开销很大，所以只要可能，都应尽量避免。就如前面所提的，无论表示什么对象，Create()和 Destroy()成员函数真正完成创建和销毁对象的工作。Create()函数用一个简单的布尔值表示成功或失败，这个值既直观又容易实现。另一种流行的返回值选择是使用标准的错误代码类型，通常为符号整型（signed integer）。布尔类型易于使用，但如果不能返回代码的话，则需要额外的错误查询机制。异常处理虽然理论上优于值返回，但它往往既会损耗运行性能又容易被程序员忽视。此外，错误代码和值返回都在头文件中，而异常处理是不能自文档化（self-documenting）的。

Destroy()函数也有很重要的一点需要注意。因为我们既希望有自动清空的便利，又希望有“按需创建和删除”的灵活性，所以我们需要确保 Destry()函数能安全地多次调用，或在没有调用 Create()函数的情况下也能安全调用。记住在销毁函数的最后，一定要调用 Clear()函数来将所有对象重置回初始状态。

游戏编程常常意味着编写实时系统，而不是大多数商业应用中更常见的基于事件的编程模型。我们可能想在类的设计上认识这种差别。类中最后的部分是 Updata()函数，这是个“步进”函数，也就是每一帧调用一次的函数。给这个函数一个通用名称非常有用。根据具体的类的要求，你可以选择是否让这个 Update()函数返回一个布尔值以在这个步进函数中检验运行时的错误。

1.1.3 类层次结构设计

如何通过继承来复用类，是面向对象编程的一大要素。关于对象间关系的完整讨论以及如何实现超出了本篇文章的范围，但是有一条设计原则非常重要，需要简单地提一下。

扩展类之间的合作主要有两种方法：继承（inheritance）和分层（layering）。继承就是从一个类派生出另一个类；分层则是指一个对象作为成员包含于另一个对象，也称作组合、容器与嵌套。

有一个简单的原则：如果两个对象之间是“是”的关系，采用公有继承（public inheritance）；如果是“有”的关系，则采用分层。到底什么算“是”，什么算“有”呢？这主要根据听起来哪种更顺。如果我们把它们放在句子里意思会清楚一些。

快艇类（class Corvette）是车类（class Car）的一种。

快艇类（class Corvette）有一种收音机类（class Radio）。

大声把这两种关系读出来往往能帮助你判断类之间的关系。一般来说，听起来正确的就是正确的。

1.1.4 设计模式

在给普通的编程问题创建一个解决方案的时候，大多数程序员都会不自觉地想到他们曾解决过的类似问题，并从以前的解决方案中推出新的解决方案。设计模式就是关于形式化这些通用软件解决方案，为日常工程任务的讨论提供可供参考的通用框架的。其他书籍对很多

设计模式都有更详细的描述，这里我们只讨论游戏开发者常使用或涉及的模式。

1. Singleton（单件）模式

当大量的类和/或模块需要访问一个全局对象时，使用 singleton 模式。该模式只用创建一个非本地的静态对象就可以了，但是在实践上存在很多内在问题，其中一个就是对象应该在何时真正创建，这和其他的全局范围对象的问题相同。singleton 模式对这个问题的解决方法是：强迫通过一个类来进行访问，这个类中存储有一个内部静态对象。以下是大致的基本实现形式：

```
class Singleton1
{
public:
    Singleton1& Instance()
    {
        static Singleton Obj;
        return Obj;
    }
private:
    Singleton1();
};
```

这个简单的代码优雅地解决了问题，但如果要从它派生出同样优雅的新的类，可不是件容易事。通过改变设计，提供在对象创建到销毁期间更多的具体干预，我们扩展了基本的 singleton 概念，使得我们的原始类具有较好的可扩充性。

```
class SingletonBase
{
public:
    SingletonBase()
    { cout << "SingletonBase created!" << endl; }
    virtual ~SingletonBase()
    { cout << "SingletonBase destroyed!" << endl; }
    virtual void Access()
    { cout << "SingletonBase accessed!" << endl; }
    static SingletonBase* GetObj()
    { return m_pObj; }
    static void SetObj(SingletonBase* pObj)
    { m_pObj = pObj; }
protected:
    static SingletonBase* m_pObj;
};

SingletonBase* SingletonBase::m_pObj;

inline SingletonBase* Base()
{
    assert(SingletonBase::GetObj());
    return SingletonBase::GetObj();
```

```
}

// 创建一个派生的 singleton 类型 (singleton-type) 类
class SingletonDerived : public SingletonBase
{
public:
    SingletonDerived()
    { cout << "SingletonDerived created!" << endl; }
    virtual ~SingletonDerived()
    { cout << "SingletonDerived destroyed" << endl; }
    virtual void Access()
    { cout << "SingletonDerived accessed!" << endl; }
protected:
};

inline SingletonDerived* Derived()
{
    assert(SingletonDerived::GetObj());
    return (SingletonDerived*)SingletonDerived::GetObj();
}

// 使用代码……
// 复杂的 singleton 的使用需要做更多工作，而这个更加灵活。
// 它还允许对对象创建进行更多的控制，这有时候非常有用。
SingletonDerived::SetObj(new SingletonDerived);

// 注意这个方法的功能被新类重载了
// 即使通过原有方法访问也是如此。
Base()->Access();
Derived()->Access();

// 很不幸，singleton 上的这个变量要求显示地创建和销毁。
delete SingletonDerived::GetObj();
```

这个对 singleton 类的修改并不仅在创建与销毁的层面上，还在于全局访问上，保留可访问性正是 singleton 的基本点。另外，从使用者的角度来看，增加的内联访问函数也使得代码更容易阅读了。

singleton 模式适用于你通常会采用全局变量或指向单个类实例的指针的地方。例如管理员类型的类，这个类只需要一个实例（模式的名字）。可以成为 singleton 类型类的有：管理应用程序声音的类、用户界面类、图形类，甚至可以是该游戏或应用程序类。

那么，为什么要费这么多事来使用 singleton，而不是简单地创建一个全局对象或指针呢？有几个很重要的原因。首先，如果你准备创建全局变量，通过一个简单的函数来访问变量要比在所有的文件中“extern”全局变量简单，而且你还可以对变量初始化的时间进行控制。其次，如果使用指针而不用对象的话，你可以通过 C++对对象进行完全地访问，也就是说你获得了更好的访问控制，这样你就可以利用这种控制能力做许多事情，如对类每一次的访问进行监控。最后，你可以使用本文中的技术创建你自己的 singleton 类，然后再派生类。你可以在保持和现有基类兼容的条件下，扩充你自己的基类。让我们来检验一下这种做法的效果如何。

为了充分体现这种可扩展性，请想象如下情况：库 A 使用一个如前所描述的 singleton 类。库 B 的运行必须使用到库 A，所以它依赖那些类并把他们包括在头文件中。应用程序 C 同时使用 A 和 B，但需要针对一些具体游戏项对 A 进行一些改动。应用程序 C 并不重新创建一个新版本的类（也不通过所谓并行项目对原有类进行任何改进），而是简单地从库 A 派生一个新类（类 D）。如果作为我们 singleton 的约定，由应用程序负责分配对象，则我们可以用派生出类 D 来代替类 A。通过用新的名字创建一个新的访问函数，并将其返回指针指向类 D 而不是类 A，我们就可以访问 D 的所有新功能了。但是类 B 还是继续使用老的访问函数，其返回指针指向 A，并希望其函数同以前一样运行。要注意，虚拟函数可以被覆盖，但要保证新的功能兼容于老的，以保证向下兼容。

以这种方式，singleton 模式可以让你创建一套简单库版本策略。你可以看到一个简单的技术是如何形成强大的机制来对代码进行组织和复用的。本书中 Scott Bilas 所写的《一种自动的 singleton 工具》文章中提出了对 singleton 模式的另一种变体，他以一种优雅的方式使用模板和公有继承来自动创建 singleton 类。

2. Façade 界面模式

从 singleton 继续到我们下一个研究的模式——Façade 模式。这种模式一般用于通常意义上的管理类，这种类提供一个到大量相关类的接口，通常是某种子系统接口。这些类常常被设计成 singleton，因为一般一个管理对象对应一种子系统才有意义，例如，你只需要单个对象来管理到声音或图形用户界面子系统的访问。

为了使类之间的相关性（也称作耦合）达到最小，有必要使用 Façade 或管理器。让我们来设想最坏的情况，项目中的每个类都“知道”并要求显式地访问其他每个类，如图 1.1.1 所示。类之间最大的关联性为$(n-1)^2$，其中 n 为项目中的类数目。

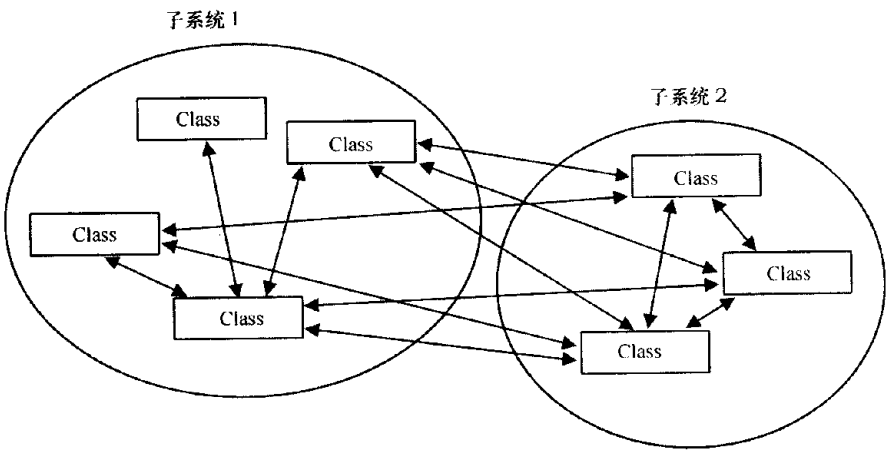


图 1.1.1 坏耦合示例

当整个子系统需要进行大的改动或是替换的时候，这种相关性就会产生问题。面向对象程序设计在类的范围内对实现改动进行了保护，但更广泛的改动保护则需要一种新的范式。Façade 模式在更大规模上解决了面向对象编程所保护的同类问题。

实现 Façade 类的首要原则是：尽量避免子系统内部类对外的暴露。这一点并不是总能完全实现，但是通过巧妙的编码和函数封装，你可以在很大程度上减少这些类的暴露，如图 1.1.2 所示。你每隐藏了一个类，就表明在下一次子系统重实现时所需做的工作减少了一点。

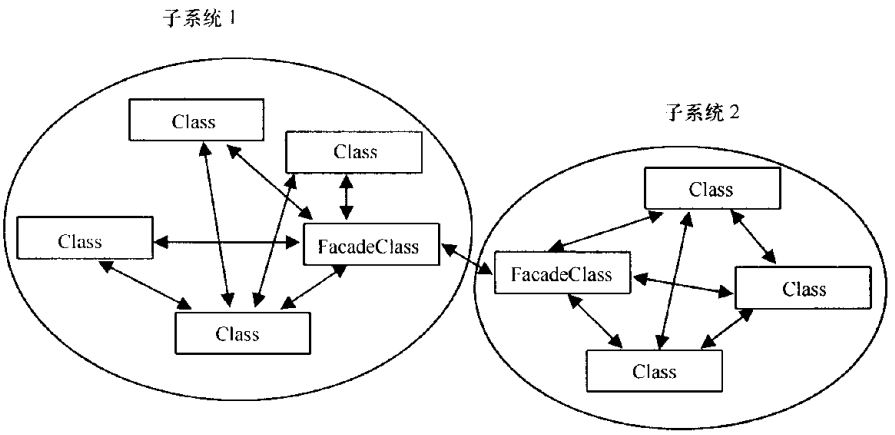


图 1.1.2 使用 Façade 类减少相关性

3. State（状态）模式

几乎所有的游戏程序员都需要处理游戏状态不断改变的实时追踪问题。状态在刚开始的时候通常是简单的枚举，其行为实现基于 switch ... case 结构的选择。然而当状态的数量增大，函数需要被更多的状态所共享时，问题就来了。剪切—粘贴的恶梦随之而来，程序员要费力地找出共享了代码的所有状态，并确保如果其中一个状态进行改变，所有这些状态都进行了相应的处理。

简单的使用对象来表示逻辑状态是一种更为优雅的面向对象解决方案。使用对象的好处是：状态得到了更好的封装，状态可以在它们的基类中逻辑地共享代码，并且新的状态可以通过继承轻易地从现有状态中派生出来。这些好处减少了在离散的状态间进行剪切、粘贴代码所带来的典型问题，如图 1.1.3 所示。

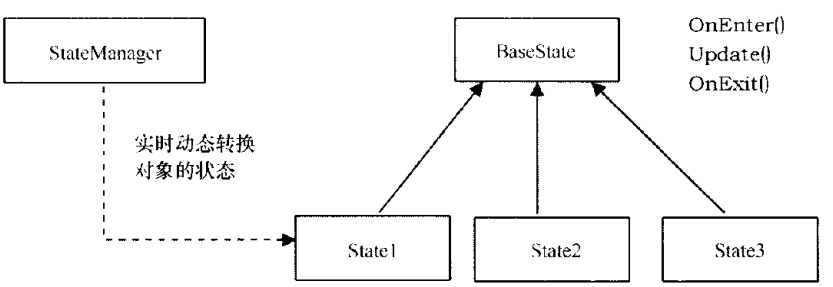


图 1.1.3

虽然这种模式没有指明如何进行状态的过渡，但用一个中央管理器来处理这些类的过渡通常很有效。采用这种方式可以避免内部对象之间的关联性，只需要管理器知道所有不同的

状态对象就可以了。更好的方法是将状态进行枚举，并通过使用 factory 对象来创建，下一小节对此进行了讲解。

State 模式并不是只能用于表示离散的游戏状态，它还可以用于 AI 系统，甚至可用于在一个游戏中表示不同的游戏模式类型。使用不同的对象来表示每一种游戏模式，你就能够通过使用动态连接库（DLL）或其他向现有代码动态添加对象的方法，灵活地在初始发布以后再添加新的行为。

4. Factory（工厂）模式

Factory 模式用于组织对象的创建，它的一种形式定义为一种方法，允许抽象接口类指定何时创建具体的派生实现类。在应用程序框架或其他类似的类层次结构中通常需要这种方法。但是游戏程序员通常只使用 Factory 模式的一个特定子集——使用 Factory 对象进行位于一个中央类的枚举对象的创建，通常通过一个单成员函数实现。

在英文中，这表明单个对象要负责创建大量的其他对象，这些对象常常有一个公共的基类。这个基类接受某种类 ID 并返回一个分配的对象。把对象分配聚集到一个位置的好处是游戏程序员尤其值得注意的：

- 因为动态内存分配很昂贵，所以你会想要对分配进行仔细地监控，在一个中央区域分配所有的对象使得分配监控变得更为容易。
- 通常，全体对象的公共初始化或创建方法应该在一个类层次结构中。如果把对全体对象的分配放到一个中央区域，就使得基于全体对象的操作（如把它们插入资源管理器中）变得容易了。
- 工厂具有可扩充性，它允许新的对象从现有工厂中派生出来。通过传递新的类 ID，你就可以在运行时扩充新的类而不用改变现有的基代码。

最后一点强调了工厂模式具有可扩充性的优点，因而应该避免创建简单函数或静态类，因为它们不能派生新的类。

```
BaseClass* ClassFactory::CreateObject(int id)
{
    BaseClass* pClass = 0;
    switch(id)
    {
    case 1:
        pClass = new Class1;
        break;
    case 2:
        pClass = new Class2;
        break;
    case 3:
        pClass = new Class3;
        break;
    default:
        assert(!"Error! Invalid class ID passed to factory!");
    };
```

```
    // 也许需要进行一些一般性初始化工作
    pClass->Init();

    return pClass;
}
```

从以上程序你可以看到工厂创建方法在技术上一点也不复杂，但是将这些对象的创建集中起来就能够提供有效的组织和可扩充的机制。

当在一个对象层次结构中的大量对象需要在运行时动态创建的时候，就可以使用工厂模式，这些对象可以包括 AI 对象、资源（如纹理和声音）或者更抽象的对象（如游戏状态，这在前面讨论过）。

1.1.5 总结

开发出好的面向对象技术并不仅仅在于技术本身，它应该遍布于你代码的方方面面。从长远看来，这将节省你的时间、减少你的麻烦。编写优秀的代码比过程代码更灵活、更具可维护性和可扩充性。游戏程序员不应该因为其个人爱好而采用一种复杂的新语言或编码风格，总有方法能满足他们。

本文最后列出了一些参考资料，它们是进一步学习通用面向对象编程和 C++语言用法的必不可少的工具。

1.1.6 参考资料

[Gamma94] Gamma, et. al., *Design Patterns,* Addison-Wesley Longman, Inc., 1994.

[Meyers98] Meyers, Scott, *Effective C++,* second edition, Addison-Wesley Longman, Inc., 1998.

[Meyers96] Meyers, Scott, *More Effective C++,* Addison-Wesley Longman, Inc., 1996.

[Petzold96] Petzold, Charles, *Programming Windows 95,* Microsoft Press, Inc., 1996.

.2 使用模板元编程的快速数学方法

Pete Isensee

当程序员想到 C++模板时，他们通常会联想到 STL（标准模板库）、通用容器（generic container）和类型安全宏（type-safe macro）。大多程序员并不清楚模板可以作为虚拟编译程序，可以快速大量地创建优化的代码。模板的这种能力最早是 Erwin Unruh 在 1994 年注意到的，他向 C++标准委员会展示了一个模板程序，该程序没有编译，而是强制编译程序在它的错误消息中生成一个素数列表[Veldhuizen99]。

这一发现使得许多程序语言专家开始关注于将模板用作预编译程序。Todd Veldhuizen 和 David Vandevoorde 极大地扩展了模板的这种能力，他们指出，实际上任何算法都能被模板化，算法的输入参数在编译期提供。只要有好的编译程序，中间代码可以完全优化掉，以达到极高的效率。

让我们通过简单的例子来了解模板的这种作用。

1.2.1 斐波纳契数

斐波纳契序列形如：0，1，1，2，3，5，8，13，……。该序列的通用方程是：$Fib(n) = Fib(n-1) + Fib(n-2)$，递归生成斐波纳契数的典型函数如下：

```
unsigned RecursiveFib( unsigned n )
{
  if( n <= 1 )
    return n;
  return RecursiveFib( n-1 ) + RecursiveFib( n-2 );
}
```

信不信由你，这个简单函数的时间复杂度是指数级的，它非常低效，根本不能用于产品代码中。该函数距离其模板化版本仅一步之遥：

```
template< unsigned N > struct Fib
{
  enum
  {
    // 递归定义
    Val = Fib< N-1 >::Val + Fib< N-2 >::Val
  };
};
```

```
// 基本情况的模板特化
// (结束条件)
template <> struct Fib< 0 > { enum { Val = 0 }; };
template <> struct Fib< 1 > { enum { Val = 1 }; };

// 让该模板形如函数
#define FibT( n ) Fib< n >::Val
```

通过#define“调用”此模板：

```
std::cout << FibT( 4 ); // Fib< 4 >::Val
```

关于此模板化版本需要注意以下几点：

- 模板函数并不是真正的函数——它是叫作 Val 的枚举整数，在编译期递归生成。语句 Val = Fib< N-1 >::Val + Fib< N-2 >::Val 不常见，但是合法。
- Fib 被定义为结构，以简化标记（notation）。在默认情况下结构数据是公用的，这也正是我们所需要的。在后面的代码列表中也采用了类似的标记。
- 模板参数 N 用于指定函数的输入。这种模板参数的使用并不常见，但完全可行。例如，std::bitset<N>用 N 的数值作为它的模板参数来定义表示过的位数。数字参数必须在编译期被获知，如果当 i 还是可变的变量时调用 FibT(i)，则会产生编译错误。
- 要中止递归，需要正确地处理结束条件。对于斐波纳契数来说，结束条件就是当 N 为 0 和 1 时。在模板中处理基本情况的方法是使用模板特化（template specialization）。标记了 template <>的，表示为模板特化。当 N 为 0 和 1 时，Val = N。

请考虑编译程序如何计算 FibT(4)：

```
= Fib<4>::Val
= Fib<3>::Val + Fib<2>::Val
= Fib<2>::Val + Fib<1>::Val + Fib<1>::Val + Fib<0>::Val
= Fib<1>::Val + Fib<0>::Val + 1 + 1 + 0
= 1 + 0 + 1 + 1 + 0
= 3
```

由于所有的输入在编译期都确定了，所以编译程序可以将 FibT(N)换算为常量。换句话来说，编译程序可以生成与以下完全等价的代码：

```
std::cout << 3; // Fib(4)
```

该方法可以成为你 C++工具包中的有用工具。你很可能会有指数级运行时间不能降为常数级运行时间的情况，通过使用模板元编程，就可以通过增加额外的编译时间来降低程序的执行时间。对于游戏来说，执行时间通常比编译时间更重要，所以这项技术也将非常有用。

1.2.2 阶乘

这是另一个将标准函数转成模板化版本的举例。

首先，标准的 C++版本如下：

```
unsigned RecursiveFact( unsigned n )
{
  return ((n <= 1) ? 1 : (n * RecursiveFact(n - 1)));
}
```

模板元编程的版本如下：

```
// 模板化阶乘 (n)
template< unsigned N > struct Fact
{
  enum { Val = N * Fact< N - 1 >::Val };
};

// 基本情况的模板特化
template <> struct Fact< 1 > ·
{
  enum { Val = 1 };
};

// 让模板形如函数
#define FactT( n ) Fact< n >::Val
```

就如在斐波纳契例子中一样，编译程序可以换算调用，如将 FactT(4)换算为常数 24。我们已经从线性级运行时间降为了常数级运行时间，这便是使用元编程的好处。但也有两个缺点：编译时间上的损失，这一点通常并不重要；代码可读性上的损失，这可以通过良好的宏定义来避免，如 FactT(n)。

让我们退一步来说。模板元编程的确引人瞩目，且不可否认很高效，但是并没有很多游戏需要生成斐波纳契序列或数字的阶乘。就算有游戏需要，代码在编译期能确定输入参数的可能性也不大。这仅是单纯的 C++小技巧，还是有实用价值的技术呢？

1.2.3 三角学

现在来看一个更复杂的例子——正弦函数。许多游戏使用正弦表或类似的方法来进行快速三角计算。如果我们要让编译程序读懂 $x = sin(1.234)$并产生一个移动指令，该怎么做呢？模板元编程可以做到。

生成标准三角函数需要采用级数展开。正弦的展开式如下：

$$\sin(x) = x - (x^3 / 3!) + (x^5 / 5!) - (x^7 / 7!) + (x^9 / 9!) - \cdots$$

其中 x 是弧度，$0 <= x < 2\pi$。为了有效地进行计算，将展开式改写为：

$$\sin(x) = x * \text{term}(0)$$

其中 $term(n)$递归计算如下：

$$\text{term}(n) = 1 - x^2 / (2n + 2) / (2n + 3) * \text{term}(n+1)$$

不使用模板，可以将上述展开式写为如下代码：

```
double Sine( double fRad )
{
  const int iMaxTerms = 10;
  return fRad * SineSeries( fRad, 0, iMaxTerms );
}

double SineSeries( double fRad, int i, int iMaxTerms )
{
  if( i > iMaxTerms )
    return 1.0;

  return 1.0 - ( fRad * fRad / (2.0 * i + 2.0) / (2.0 * i + 3.0)
              * SineSeries( fRad, i + 1, iMaxTerms ) );
}
```

加大 iMaxTerms 可以提高精确度，但会损失运行速度。将它转成模板化版本很容易，具体解决方案参见程序清单 1.2.1。该方案使用了两个模板对象：Sine<R>，计算 *R* * *term*(*0*)；Series<R,I>递归计算 *term*(*I*)，递归次数由 Max Terms 指定。使用模板元编程版本：

```
double x = SineT( 1.234 );
```

理论上，编译程序可以生成和以下等价的代码：

```
double x = 0.94381820937463368;
```

sin(1.234)的实际值是 0.94381820937463370…，采用 10 次递归的模板版本计算的精确度达到了小数点后 15 位！不用多费力气，我们就得到了这个解决方案，它通过使用编译程序使我们能够随时（或定时）获取正弦值。因为编译程序可以在编译期生成（且仅生成）所需的正弦表入口，所以在运行时不用计算正弦表，也不用在可执行文件中嵌入正弦表。

1.2.4 实际世界中的编译程序

模板元编程有一个潜在的重要问题。当今（2000 年）许多编译程序不能换算递归以及那些包含了基于模板的复杂算法的数学方法。在正弦计算的例子中，按级数展开 10 项计算，每往下走一个或两个指令就需要编译程序换算近 20 次浮点乘法、50 次整数乘法、20 次浮点除法、10 次浮点减法以及 10 次递归调用。编译程序“能”做到吗？当然可以。编译程序“应”做到吗？如果给予足够的资源（RAM 和时间）则是应该可以的。那么，编译程序“会”做到吗？这就需要看具体情况了。

我用 Microsoft Visual C++ 6.0 测试了前面的例子。VC6 在斐波纳契和阶乘模板中有很优秀的表现，而在正弦模板中却表现不佳，生成的代码甚至不如 C 运行时的 sin()函数。在默认情况下，VC6 将递归展开 8 层，且对计算不进行任何优化。通过指定#pragmas inline_depth(255)和 inline_recursion(on)，VC6 对递归完全展开，并优化所有计算，从而还是幸运地得到了好的结果。

以上说明，所有的优化方法都需要经过实际的证明。应该检验编译程序生成的代码，并计算引入模板之前和之后的性能。你可能需要转变几个编译程序的标志以获取希望的结果。

希望将来开发编译程序的程序员能对模板优化以及模板元编程本身给予更多的关注。

1.2.5 重访三角学

如果说 C++编译程序在处理模板方面技术还不成熟，那么我们能做些什么来帮助编译程序进行正确的处理呢？程序清单 1.2.2 展示了对正弦函数的另一种尝试，取消了递归，级数展开到 10 项。这样计算复杂性下降到了 12 次乘法、21 次除法和 10 次减法。同时，模板特化因不再需要而得以消除。

1.2.6 模板和标准 C++

并非所有编译程序都能完全与 C++标准兼容，尤其是在它们处理模板的时候更是如此。模板是如此灵活而且强大，以至于为了让它们能够正确运行，编译程序的开发者需要付出大量辛苦的劳动。这种困难在 Visual C++的开发中得以体现，其对模板的支持能力和对标准的遵循进展缓慢。例如 VC6 不支持模板偏特化，并使很多模板函数的特化版本更加复杂，且无法达到它们应有的通用性。

然而更重要的在于其是否提供对浮点模板参数的支持。在程序清单 1.2.1 和 1.2.2 中的 sine 模板函数使用浮点模板参数作为输入的弧度值。而在 C++标准中，“一个非类型模板参数不应被声明为浮点类型”。也就是说，一个遵循标准的编译器中，

```
template< double R > struct Sine // 编译器错误
```

将产生一个错误信息。解决方法是使用引用参数：

```
template< double& R > struct Sine // 正确
```

有趣并且不幸的是，Visual C++ 6.0 支持浮点类型作为模板参数，但不支持引用参数。它完全颠倒了标准的定义！为了在符合标准的编译器中使用 sine 模板代码，必须将 double R 改为 double& R。

1.2.7 矩阵

模板元编程的优势所在是处理矩阵运算。三维游戏中大量使用矩阵。将关键函数模板化能够显著地提高速度。在下面的章节中，我们将使用模板来改进矩阵初始化、矩阵变换和矩阵乘法。

1. 单位矩阵

单位矩阵（identity matrix）包括的元素除了对角线上为 1 外都为 0。我们首先看一个常规的实现。请注意内嵌的 for 循环：

```
matrix33& matrix33::identity()
{
```

```
    for (unsigned c = 0; c < 3; c++)
      for (unsigned r = 0; r < 3; r++)
        col[ c ][ r ] = ( c == r ) ? 1.0 : 0.0;
    return *this;
}
```

模板元编程版本的代码在程序清单 1.2.3 中。模板版本的参数包括矩阵 Mtx、矩阵大小 N（假设是矩形矩阵）、当前行 R 和列 C。在每个迭代中，我们计算下一个矩阵元素。

需要注意的关键点是我们循环遍历每列和每行的方法。在算法的每一步，我们知道简单的从 0 到 N 递增的 I。通过给定的 I，我们可以用 I 对 N 取模计算当前行。例如 N 为 3 时，行序列将是：0, 1, 2, 0, 1, 2, 0, 1, 2, 0。当前列是 I 除 N 后对 N 取模。如果 N 为 3，列序列将是：0, 0, 0, 1, 1, 1, 2, 2, 2, 0。模板指定算法在 I 等于 N 时终止。现在我们可以将原有版本替换为：

```
matrix33& matrix33::identity()
{
  IdentityMtxT( matrix33, *this, 3 );
  return *this;
}
```

编译器将会把新版本展开为：

```
matrix33& matrix33::identity()
{
  col[ 0 ][ 0 ] = 1.0;
  col[ 0 ][ 1 ] = 0.0;
  // ...
  col[ 2 ][ 1 ] = 0.0;
  col[ 2 ][ 2 ] = 1.0;
  return *this;
}
```

换句话说，编译器将完全展开循环。当然我们也可以自己手工展开循环，但模板版本是更加通用化的解决方法。它也能被用于任意大小的矩形矩阵（通过你的特化操作指定）。例如，4×4 矩阵的代码如下：

```
matrix44& matrix44::identity()
{
  IdentityMtxT( matrix44, *this, 4 );
  return *this;
}
```

2．矩阵初始化

我们可以通过在生成单位矩阵中使用过的相同技术来创建模板化的初始化代码。事实上，我们在程序清单 1.2.3 中，惟一需要改变的是决定每个矩阵元素值的行：

```
mtx[ C ][ R ] = ( C == R ) ? 1.0 : 0.0; // 单位矩阵
```

将被替换为：

```
mtx[ C ][ R ] = 0.0; // 零矩阵
```

或者更通用的：

```
mtx[ C ][ R ] = static_cast< F >( Init ); // 初始化矩阵
```

这里的类型 F 是存储在每个元素中值的类型，而 Init 是缺省为 0 的数字模板参数。这个通用解决方案允许你简单地初始化矩阵元素为任意常量值。

3．矩阵变换

以对角线为轴对矩阵进行翻转变换：

```
matrix33& matrix33::transpose()
{
  for (unsigned c = 0; c < 3; c++)
    for (unsigned r = c + 1; r < 3; r++)
      std::swap( col[ c ][ r ], col[ r ][ c ] );
  return *this;
}
```

这个算法将很大程度上损害性能，因为它实际上只完成了很少的实际工作。对 3×3 矩阵来说，实际上只有 3 次交换；对 4×4 矩阵来说，实际上只有 6 次交换。程序清单 1.2.4 是模板实现代码。我们可以将原有代码替换为：

```
matrix33& matrix33::transpose()
{
  TransMtxT( matrix33, *this, 3 );
  return *this;
}
```

上述代码将被编译器展开为：

```
matrix33& matrix33::transpose()
{
  std::swap( col[0][1], col[1][0] );
  std::swap( col[0][2], col[2][0] );
  std::swap( col[1][2], col[2][1] );
  return *this;
}
```

内嵌的 for 循环将被优化掉，只留下交换操作。Swap 本身也是一个内联函数，因此我们将只剩下 9 条内存移动指令。没什么能够做得更好了。

4．矩阵乘法

我们最后看看元编程如何将矩阵乘法模板化。一个常规的非模板实现类似如下代码：

```
matrix33& matrix33::operator *= (const matrix33& m)
```

```
{
  matrix33 t;
  for (unsigned r = 0; r < 3; r++)
  {
    for (unsigned c = 0; c < 3; c++)
    {
      t[c][r] = 0.0;
      for (unsigned k = 0; k < 3; k++)
        t[c][r] += col[k][r] * m[c][k];
    }
  }
  *this = t;
  return *this;
}
```

相应的模板元编程版本代码在程序清单 1.2.5 中。矩阵乘法的内层循环变成模板参数 K。不像单位矩阵和矩阵变换算法，其扩展为 N 维矩阵迭代器，矩阵乘法将扩展为 N 维立方体迭代器（注意特化将“N 维立方体”作为参数）。现在我们可以将原有版本替换为：

```
matrix33& matrix33::operator *= (const matrix33& m)
{
  matrix33 t;
  ZeroMtxT( matrix33, t, 3 );
  MultMtxT( matrix33, t, *this, m, 3 );
  *this = t;
  return *this;
}
```

我们首先初始化结果矩阵为空（以 0 填充），以使 MultMtxImpl 模板中的+=操作符能够正常工作。编译器将把新的乘法操作符展开为以下形式：

```
matrix33& matrix33::operator *= (const matrix33& m)
{
  matrix33 t;

  // ZeroMtxT
  t[0][0] = 0.0;
  // ...
  t[2][2] = 0.0;

  // MultMtxT
  t[0][0] += col[0][0] * m[0][0];
  t[0][0] += col[1][0] * m[0][1];
  // ...
  t[3][3] += col[2][0] * m[0][2];
  t[3][3] += col[3][0] * m[0][3];

  *this = t;
  return *this;
}
```

乘法模板也有一个用于处理任意矩形矩阵的通用解决方案。为处理任意 N×N 矩阵的代码如下：

```
matrixNN& matrixNN::operator *= (const matrixNN& m)
{
  matrixNN t;
  ZeroMtxT( matrixNN, t, N );
  MultMtxT( matrixNN, t, *this, m, N );
  *this = t;
  return *this;
}
```

5. 实际矩阵性能

这些模板化的矩阵操作到底效率如何？一般来说这要依赖于编译器的质量和所选择的编译器选项。

我在 VC6 下进行测试，使用选项包括：完整优化、为速度优化、所有合适（any suitable）内联一级#pragmas inline_depth(255)和 inline_recursion(on)。我对每个操作测试一亿次，结果如表 1.2.1 中所示。所有时间的单位都是 ms。展开列指出编译器是否能够展开递归。

表 1.2.1　　矩阵操作测试结果

操作	非模板化（ms）	模板化（ms）	展开
matrix33::zero	33 992	29 330	完整
matrix44::zero	36 063	30 292	完整
matrix33::identity	45 827	29 526	完整
matrix44::identity	46 845	29 905	完整
matrix33::transpose	35 338	29 955	完整
matrix44::transpose	90 638	30 245	部分
matrix33::op *=	62 904	50 352	部分
matrix44::op *=	326 890	792 901	部分

为查看性能的相关性，我将这些结果以图形方式表现出来，以展示模板化版本与非模板版本相比有多大的性能提高（图 1.2.1）。

除了 4×4 矩阵乘法的异常情况外，模板化版本都有显著的性能提升。意料之内的是矩阵变换操作取得了最好的性能提升。在简单算法的情况下，编译器可以完全展开模板递归。然而 4×4 矩阵乘法（4^3 次运算）因为编译器只能部分展开递归，并且递归函数调用的负荷远远超过任何内联函数获得的性能提升，所以结果中模板化版本反而比原有版本慢。这个结果简单地展示了优化的重要性。

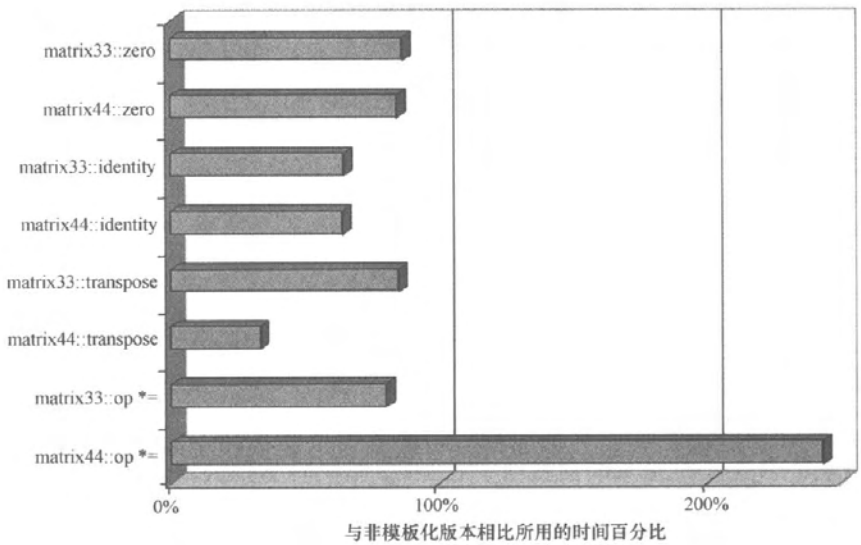


图 1.2.1 模板函数的相对性能

1.2.8 总结

模板是以指令流方式直接生成算法的最有效途径。以通用方式减少和展开代码的能力是一项非常强大的编程技术。模板元编程的概念在一开始可能不同寻常，但它实际上并不比展开标准递归调用的练习更困难。宏可以被用来隐藏调用代码的细节。

元编程技术可以被扩展用于用途广泛的函数，包括平方根计算、最大公约数、矩阵求逆或排序。任何在编译期能够获知部分输入参数的算法，都可以从元编程中获益。

当前版本的编译器在处理模板时仍有很多限制，特别是处理递归模板。编译器模板错误信息从含义模糊到难以辨认的都有。对任何优化来说，都是在时间和空间上求得平衡。对大多数情况，模板元编程可以生成最好的结果：最小并且最快的代码。对其他情况，展开代码将比原有代码显著增大，这将降低或抵消带来的速度优势。不过可以预期，模板元编程将在C++库和游戏开发中扮演重要的角色。

程序清单 1.2.1

```
// sin(R) 级数展开
// 对符合标准的编译器，将 double R 改为 double& R
template< double R > struct Sine
{
   enum { MaxTerms = 10 }; // 增加精确性
   static inline double sin()
   {
      return R * Series< R, 0, MaxTerms >::val();
   }
};
```

```
template< double R, int I, int MaxTerms >
struct Series
{
   enum
   {
      // Continue 为true，知道我们已经计算了M 项
      Continue = I + 1 != MaxTerms,
      NxtI = ( I + 1 ) * Continue,
      NxtMaxTerms = MaxTerms * Continue
   };

   // 递归定义，为每项调用一次
   static inline double val()
   {
      return 1 - R * R / (2.0 * I + 2.0) /
            (2.0 * I + 3.0) * Series< R * Continue, NxtI, NxtMaxTerms >::val();
   }
};

// 用于终止循环的特化
template <> struct Series< 0.0, 0, 0 >
{
   static inline double val() { return 1.0; }
};

// 使模板能够以类似函数的方式使用
#define SineT( r ) Sine< r >::sin()
```

程序清单 1.2.2

```
//  sin(R) 级数展开
// 对符合标准的编译器，将double R改为double& R
template < double R > struct Sine
{
   // 一个合适的编译器将能够把所有在编译期知道的值缩减为一个常量
   static inline double sin()
   {
      double Rsqr = R * R;
      return R * ( 1.0 - Rsqr / 2.0  / 3.0
             * ( 1.0 - Rsqr / 4.0  / 5.0
             * ( 1.0 - Rsqr / 6.0  / 7.0
             * ( 1.0 - Rsqr / 8.0  / 9.0
             * ( 1.0 - Rsqr / 10.0 / 11.0
             * ( 1.0 - Rsqr / 12.0 / 13.0
             * ( 1.0 - Rsqr / 14.0 / 15.0
             * ( 1.0 - Rsqr / 16.0 / 17.0
```

```
            * ( 1.0 - Rsqr / 18.0 / 19.0
            * ( 1.0 - Rsqr / 20.0 / 21.0
            ) ) ) ) ) ) ) ) );
   }
};

// 使模板能够以类似函数的方式使用
#define SineT( r ) Sine< r >::sin()
```

程序清单 1.2.3

```
// 模板化的单位矩阵；N 是矩阵大小
template< class Mtx, unsigned N > struct IdMtx
{
   static inline void eval( Mtx& mtx )
   {
      IdMtxImpl< Mtx, N, 0, 0, 0 >::eval( mtx );
   }
};

// 对矩阵每个元素赋值
template< class Mtx, unsigned N, unsigned C, unsigned R, unsigned I >
struct IdMtxImpl
{
   enum
   {
      NxtI = I + 1,           // 计数器
      NxtR = NxtI % N,        // 行（内层循环）
      NxtC = NxtI / N % N     // 列（外层循环）
   };
   static inline void eval( Mtx& mtx )
   {
      mtx[ C ][ R ] = ( C == R ) ? 1.0 : 0.0;
      IdMtxImpl< Mtx, N, NxtC, NxtR, NxtI >::eval( mtx );
   }
};

// 为 3×3 和 4×4 矩阵特化
template<> struct IdMtxImpl< matrix33, 3, 0, 0, 3*3 >
{
   static inline void eval( matrix33& ) {}
};
template<> struct IdMtxImpl< matrix44, 4, 0, 0, 4*4 >
{
   static inline void eval( matrix44& ) {}
};
```

```
// 使模板能够以类似函数的方式使用
#define IdentityMtxT( MtxType, Mtx, N ) \
      IdMtx< MtxType, N >::eval( Mtx )
```

程序清单 1.2.4

```
// 模板化的矩阵变换；N 是矩阵大小
template< class Mtx, unsigned N > struct TransMtx
{
  static inline void eval( Mtx& mtx )
  {
    TransMtxImpl< Mtx, N, 0, 1, 0 >::eval( mtx );
  }
};

template< class Mtx, unsigned N, unsigned C, unsigned R, unsigned I >
struct TransMtxImpl
{
  enum
  {
    NxtI = I + 1,
    NxtC = NxtI / N % N,
    NxtR = ( NxtI % N ) + NxtC + 1
  };
  static inline void eval( Mtx& mtx )
  {
    if( R < N )
      std::swap( mtx[ C ][ R ], mtx[ R ][ C ] );
    TransMtxImpl< Mtx, N, NxtC, NxtR, NxtI >::eval( mtx );
  }
};

// 为 3×3 和 4×4 矩阵特化
template<> struct TransMtxImpl< matrix33, 3, 0, 1, 3*3 >
{
  static inline void eval( matrix33& ) {}
};
template<> struct TransMtxImpl< matrix44, 4, 0, 1, 4*4 >
{
  static inline void eval( matrix44& ) {}
};

// 使模板能够以类似函数的方式使用
#define TransMtxT( MtxType, Mtx, N ) \
      TransMtx< MtxType, N >::eval( Mtx )
```

程序清单 1.2.5

```
// 模板化的矩阵乘法; N是矩阵大小
template< class Mtx, unsigned N > struct MultMtx
{
   static inline void eval( Mtx& r, const Mtx& a, const Mtx& b )
   {
      MultMtxImpl< Mtx, N, 0, 0, 0, 0 >::eval( r, a, b );
   }
};

template< class Mtx, unsigned N, unsigned C, unsigned R, unsigned K, unsigned I >
struct MultMtxImpl
{
   enum
   {
      NxtI = I + 1,              // 计数器
      NxtK = NxtI % N,           // 内存循环
      NxtC = NxtI / N % N,       // 列
      NxtR = NxtI / N / N % N    // 行
   };
   static inline void eval( Mtx& r, const Mtx& a, const Mtx& b )
   {
      r[ C ][ R ] += a[ K ][ R ] * b[ C ][ K ];
      MultMtxImpl< Mtx,N,NxtC,NxtR,NxtK,NxtI >::eval( r,a,b );
   }
};

// 为3×3和4×4矩阵特化
template<> struct MultMtxImpl< matrix33, 3, 0, 0, 0, 3*3*3 >
{
   static inline void eval( matrix33&, const matrix33&,
                            const matrix33& ) {}
};
template<> struct MultMtxImpl< matrix44, 4, 0, 0, 0, 4*4*4 >
{
   static inline void eval( matrix44&, const matrix44&,
                            const matrix44& ) {}
};

// 使模板能够以类似函数的方式使用
#define MultMtxT( MtxType, r, a, b, N )       \
      MultMtx< MtxType, N >::eval( r, a, b )
```

1.2.9 参考文献

[Veldhuizen99] Veldhuizen, Todd, "Techniques for Scientific C++." www.extreme.indiana.edu/~tveldhui/papers/techniques/, August 1999.

[Veldhuizen98] Veldhuizen, Todd, et al., "Blitz++ Numerical Class Library." Available www.oonumerics.org/blitz/,August 1998.

[Veldhuizen96] Veldhuizen, Todd, and Kumaraswamy Ponnambalam, "Linear Algebra with C++ Template Metaprograms," *Dr. Dobb's Journal*, August 1996.

[Veldhuizen95] Veldhuizen, Todd, "Using C++ Template Metaprograms," *C++ Report*, May 1995.

[Pescio97] Pescio, Carlo, "Binary Constants Using Template Metaprogramming," *C/C++ Users Journal*, February 1997.

[Karmesin99] Karmesin, Steve, et al., "PETE, Portable Extension Template Engine," www.acl.lanl.gov/pete/, February 1999.

1.3 一种自动的 Singleton 工具

Scott Bilas

本文提出了一种方便且安全的方法，提供到C++类 singleton 的访问，同时保留当其实例化和销毁时的完全控制。

1.3.1 定义

singleton 是一种对象，它在一个系统中的任何时候只有一个实例。在游戏中，singleton 的一些常见例子是纹理贴图、文件或用户界面的管理程序。它们每一个都是一个子系统，通常假定从游戏开始时可用，一直持续到游戏关闭。

有一些此类子系统可以通过使用全局函数和静态变量来实现，例如内存管理程序的 malloc()和 free()函数。这种子系统并不是 singleton，因为它们并没有将其功能封装到类中，且不能通过使用单个类的实例来进行表示。这种内存管理程序也可以转换成类，并用作 singleton，但这种做法并不常见。

singleton 的一个典型例子是纹理贴图管理程序。可把它叫作 TextureMgr，有如 GetTexture()和 UseTexture()的方法。它的目标是在存档文件中找到纹理贴图，将其转换为系统图形对象，使其对光栅化程序可用，并一直拥有这些纹理贴图直到不再需要它们时将其删除。在系统中只需要 TextureMgr 的一个实例，所以该类很自然地可以用作 singleton。

1.3.2 优点

singleton 的优点是什么？首先，由于标号非常重要，singleton 提供了概念上的透明。以 singleton 调用类并遵循命名约定（如-Mgr、-Api、Global-等），关系到我们希望该类如何被使用的重要细节。

singleton 还提供了书写的便利性。在 C++系统中，每个对象都必须属于某事物。这些对象的从属模式依赖于游戏，但通常是多层层次模式，即每个高层拥有一个子对象集，而每个子对象同样又可以有其自己的子对象。每个对象公开一个函数集，让它的子对象访问。例如，要得到 TextureMgr 的实例，可能需要调用一系列函数，如 GetApp() -> GetServices() -> GetGui() -> GetTextureMgr()，其中每个函数返回一个指向所请求子对象的指针。这种多层解引用使得系统使用不便而且也不高效。singleton 被视为

全局对象，因而可以解决这一问题。

1.3.3 问题

那么，为什么不直接使用全局对象呢？全局对象的使用当然很方便，TextureMgr 对象可以通过声明为外部连接全局范围（或名字空间）的 g_TextureMgr 对象引用来访问，也可以通过返回对象引用的函数来访问。但是全局对象的创建和销毁次序取决于执行时的情况，然而在可移植的方式中这通常是不可预计的。

对这些问题有很多变通办法，但我们真正需要的是既拥有把 singleton 当作全局对象来对待的方便性，又不会在它创建和销毁时失去对它的控制。

1.3.4 传统的解决方法

在教科书中，对 singleton 的管理代码通常类似如下：

```
TextureMgr& GetTextureMgr( void )
{
    static T s_Singleton;
    return ( s_Singleton );
}
```

为了书写方便，许多变化使用模板和宏，但效果是一样的。这种解决方法允许 singleton 按需实例化——在首次函数调用时。该方法便于使用，但是它把 singleton 的销毁留给了编译程序，要求在应用程序关闭的时候进行销毁。我们需要比这更为有力的控制。销毁次序在游戏中非常重要，因为有些子系统需要在其他子系统之前关闭和销毁。另外，如果我们需要在保持游戏运行的同时，关闭游戏的一部分呢？使用这个方法是办不到的。

1.3.5 较好的方法

我们所需要的无非就是追踪 singleton 的能力，为此我们所需要的是指向它的一个指针。如果我们按照如下所做会怎样：

```
class TextureMgr
{
    static TextureMgr* ms_Singleton;

public:
    TextureMgr( void ) { ms_Singleton = this; /*...*/ }
   ~TextureMgr( void ) { ms_Singleton = 0;    /*...*/ }

    // ...

    TextureMgr& GetSingleton( void ) { return ( *ms_Singleton ); }
};
```

为了安全的目的增加几个断言，这个方法确实有效。现在，我们可以在任何时候创建和销毁 TextureMgr，并且对该 singleton 的访问就像调用 TextureMgr::GetSingleton()一样简单。但此解决方法还有一点不方便，那就是相同的代码（用于追踪 singleton 指针）需要加到每个 singleton 类中。

1.3.6 更好的方法

一种更通用的解决方法是使用模板来自动定义 singleton 指针，并完成指针设置、查询和清除的工作。它还可以检查（通过 assert()）确保没有将 singleton 实例化多次。最重要的一点是，我们可以免费获得所有这些功能，只需要从以下这个简单的类派生就可以：

```
#include <cassert>

template <typename T> class Singleton
{
   static T* ms_Singleton;

public:
   Singleton( void )
   {
      assert( !ms_Singleton );
      int offset = (int)(T*)1 - (int)(Singleton <T>*)(T*)1;
      ms_Singleton = (T*)((int)this + offset);
   }
  ~Singleton( void )
      { assert( ms_Singleton );  ms_Singleton = 0;  }
   static T& GetSingleton( void )
      { assert( ms_Singleton );  return ( *ms_Singleton );  }
   static T* GetSingletonPtr( void )
      { return ( ms_Singleton );  }
};

template <typename T> T* Singleton <T>::ms_Singleton = 0;
```

将任何类转换成 singleton，只需要按以下 3 个简单步骤来做：

（1）从 Singleton <MyClass>公开派生你的类 MyClass。

（2）确保使用前在系统中创建了 MyClass 的实例。如何实例化并不重要，可以通过将它设为全局或局部静态，让编译程序来做；也可以通过一个属主类，用 new 和 delete 来自己做。不论实例在何时如何创建，它将被追踪，并通过公共接口为系统其他部分用作 singleton。

（3）在系统的任何地方调用 MyClass::GetSingleton()来使用对象。如果你像我一样懒，也可以#define g_MyClass 为 MyClass::GetSingleton()，并将其作为全局对象来对待，以方便书写。

以下是使用该类的例子：

```
class TextureMgr : public Singleton <TextureMgr>
{
public:
```

```
    Texture* GetTexture( const char* name );
    // ...
};

#define g_TextureMgr TextureMgr::GetSingleton()

void SomeFunction( void )
{
    Texture* stone1 = TextureMgr::GetSingleton().GetTexture( "stone1" );
    Texture* wood6 = g_TextureMgr.GetTexture( "wood6" );
    // ...
}
```

这个 Singleton 类的惟一目的是在它的派生（MyClass）类型的任何实例创建和销毁时自动注册和注销它们。我们从 MyClass 派生就是为了继承这种便利的功能。这样做不会从任何方面影响到类的大小，只会增加一些自动的函数调用。

那么，它是如何工作的？所有重要的工作在 Singleton 的构造函数中完成，在此它计算出派生实例的相对位置，并将结果存储到 singleton 指针（ms_Singleton）。注意，派生类可能不仅仅从 Singleton 派生，在这种情况下 MyClass 的“this”可能与 Singleton 的“this”不同。这种解决方法假设一个不存在的对象在内存的 0x1 位置上，将此对象强制转换为两种类型，并得到其偏移量的差值。这个差值可以有效地作为 Singleton<MyClass>和它派生类型 MyClass 的距离，可用于计算 singleton 指针。

1.3.7 参考文献

Meyers, Scott, *More Effective C++*, Addison-Wesley Publishing Co., 1995.

1.4 在游戏编程中使用STL

James Boer

经过了9年的历程，C++终于在1997年正式标准化了。不仅定义了正式的语言规范，它还以标准C++库的形式向C++程序员提供了大量工具集。这个库的很大一部分是标准模板库，或叫作STL。STL是容器（数据）类的集合，包括从向量到平衡二叉树。除了基本的容器外，STL还提供了大量操作这些基本容器的各类算法。

使用STL的常见顾虑是，是否会降低代码效率。事实上，STL的设计是以速度作为首要考虑因素的，例如：向量没有绑定检查；在访问容器前迭代器也不会被激活。其结果就是，STL向量生成的代码的性能与单纯动态分配数组是相当的。其他容器的性能也是如此。STL是为高性能C++应用设计的，将它们大量用于代码中，你一点也用不着担心。

1.4.1 STL的类型和术语

STL是标准C++库中的一大部分，且较为复杂。有效使用STL，需要了解其基本组件和它们的工作方式。

1. 容器

STL容器表示经典的数据抽象和组织方案，如向量、链表、队列和映射表。但是我们应该区分特定的容器类型并了解它们是如何实现的。

STL容器中的向量、链表和双队列是隐式数据类型，它们既描述了抽象数据类型也暗示了特定的实现方式。向量（vector）是动态变长的数组；链表（list）以双链表（double-linked list）实现；双队列（duque），或双端队列（double-ended queue），以如下方式实现，它允许元素在可随机访问的数组类型结构的任意一端以常数时间插入或删除。双队列按顺序存储数据集，也叫顺序容器（sequence container），也就是说数据插入的顺序影响它们存储的顺序。

栈、队列和优先队列容器是较高级别的抽象，它们描述容器的行为，但允许不同的底层实现。例如，队列可以分别用向量、链表和双队列实现。它们也叫作容器适配器。容器适配器依据作为底层数据的容器序列的不同分为不同的类。

其他的容器，如映射表（map）、集合（set）、多重映射表（multimap）和多重集（multiset）都由红黑树（平衡二叉树）实现，但提供的容器行为

不同。因为它们数据插入的顺序依照特定的排序准则，所以也叫作关联容器（associative container）。

2. 迭代器

迭代器（iterator）可以看作指向容器内元素的指针。实际上 STL 甚至用指针符号来遍历和访问容器数据。例如，++ 运算符将迭代器移至容器中的下一个元素，这就像将指向数组元素的指针递增一样。此外，和指针一样，可以对迭代器使用 * 运算符来访问它所指向的实际数据。

3. 算法

出乎意料，操作 STL 类的算法并没有以容器类成员函数的形式出现，而是以单独的函数存在，操作迭代器。为什么 STL 的设计者选择这种看似非面向对象的方式呢？

通过从算法中分离出数据，设计者极大地减少了专用算法的组合数量。由于每个不同容器都有相似类型的迭代器，因而每种算法只用编写一次，而不用为每个容器都编写一次。缺点是有时会得到次优解。然而在大多数情况下，专用成员函数就可以完成容器的大多数基本操作。

1.4.2 STL 概念

使用 STL 时有几个基本概念很重要。首先，应该了解使用容器时用于确定范围的方法。begin()和 end()是对所有容器通用的两个方法，它们返回容器的整个范围。如图 1.4.1 所示，begin()返回容器中的第一个元素，而 end()返回的位置是最后一个可用元素的后面。

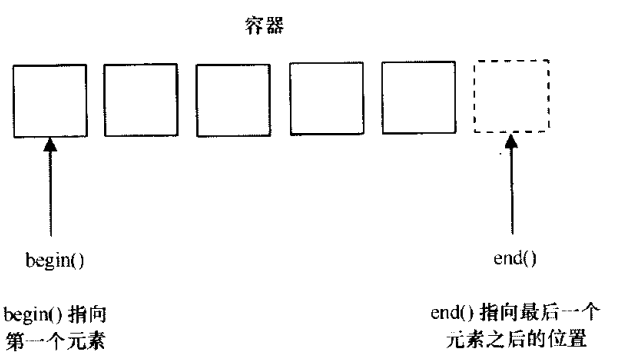


图 1.4.1 end()指向最后一个可用元素之后

以这种方式组织范围有几个优点：首先，消除了空链表的特例代码；其次，容器中的迭代有一个简单的结束准则——到 end()为止。这种系统的缺点是不太直观，并且反向迭代需要专门的成员和迭代器。

一定要记住不要去取指向 end()的迭代器所指向的数据，这种行为将导致不可预计的结果。

当使用函数指定范围时，STL 中的函数通常将参数看作迭代器，一个指定开始元素，另

一个指定结束元素。为了与 begin()和 end()有效匹配，这些函数假定包括指定的第一个元素，不包括指定的最后一个元素。在数学中，以下写法标示出了这种范围：

```
range [begin...end)
```

STL 设计的另一个方面也应该注意。STL 容器按值，而不是按地址传送信息。这意味着，当处理小的数据类型时，允许容器复制数据的做法是可取的；当处理较大的数据结构和类时，传进这些对象或结构的指针的做法则更好，否则每一次的插入或访问都要调用构造函数。

1.4.3 向量（Vector）

STL 向量在本质上就是变长数组。注意，虽然正规的 C++标准并没有规定用于容器的底层数据结构，但性能和接口的要求使得实际的实现没有多少变化的余地。所有版本的 STL 都很相似，只有实现的细节有些细微区别。

向量的行为与标准 C 数组几乎完全一致，只有一个主要的差别：它们是动态变长的。了解这个变长的特性很重要。

向量用数组来实现，这些数组需要周期性地重新分配内存并传送数据到新的数组。这对开发人员来说意味着两件事情。第一，根据可能增长的需要，向量可以分配比当前所需更多的内存。第二，向向量末端增加元素被描述为以常数时间进行——要记住，这里的常数时间指的是平均为常数时间。换句话来说，一些增长的函数在分配新内存、将现有数组复制到新内存中并删除旧内存时，会需要大量资源；不过它们并不是在任何时候都需要这些额外资源。根据具体的实现，向量在空间溢出时可以分配现有已分配内存两倍的内存空间。

了解向量何时重新分配内存也是至关重要的，因为在重新分配内存时会使所有当前指向向量中元素的迭代器失效。在读完以下代码后，让我们来检验这些函数以帮助我们更精确地管理向量的内部内存：

```
#include <vector>
#include <iostream>

using namespace std;

// 为了提高可读性，定义容器和迭代器的名字
typedef vector<int> IntVector;
typedef IntVector::iterator IntVectItor;

void main()
{
    // 创建一个整数向量对象
    IntVector c;
    // 保留 10 个整数的空间
    c.reserve(10);

    // 将 3 个不同的元素填入向量中
    c.push_back(3);
    c.push_back(99);
```

```
    c.push_back(42);

    // 现在循环并打出所有元素值
    for(IntVectItor itor = c.begin(); itor != c.end(); ++itor)
        cout << "element value = " << (*itor) << endl;

// 创建了元素，我们可以向在一般数组中那样访问或替换它们。
    c[0] = 12;
    c[1] = 32;
    c[2] = 999;
    for(int i = 0; i < c.size(); i++)
        cout << "element value = " << c[i] << endl;
}
```

这个例子展示了开始使用 STL 容器时所需知道的大部分基本原则。请注意程序清单顶端所包括的该程序所需的头文件，以及名字空间 std 的用法。就像 C++库的其他部分一样，STL 也是 std 名字空间的一部分，并且需要在程序中声明。

接下来，typedef 定义了我们希望在程序中使用的容器和迭代器的表示写法。这种做法很常见，它不仅使程序更易读，还在需要时使得底层数据结构的改变更容易（下面我们将看到这有多么容易）。

下一段代码创建了向量容器对象 c，并进而调用保留了 10 个整数所需的空间的向量函数。然后代码向向量后面 push_back()了 3 个整数。由于预先分配了足够的内存，因此不需要额外的内存分配。

当你需要紧密地监视和控制向量的内存分配时，有几个例程会有所帮助。如程序所示，可以通过调用 reserve()并传递一个大小参数来在向量中保留一个缓冲区。这个值可以通过调用 capacity()检索到。如果 capacity()==reserve()且又有一个元素插入进数组，则会进行一次内存分配且当前所有的迭代器都将失效。要决定可分配给单个向量的最大内存数量，可以使用 max_size()函数。

push_front()、push_back()、pop_front()和 pop_back()函数对所有的基本顺序容器（向量、链表和双队列）都通用。这些函数从容器的前或后加入并移动元素。依据向量的实现，应该尽量避免在此类容器上使用 push_front()或 pop_front()函数，因为这些操作具有 O(n)的复杂度，只有你确实需要此功能时才应该使用它们。

例子代码的最后一部分示范了 STL 用法中最常用的组件——迭代循环。我们对一个迭代器使用了 for 循环来确定当前位置。初始化位置设置到 begin()，迭代器使用前缀++运算符来递增，直到迭代器等于 end()为止，此时满足了循环的出口条件。每个有可访问的迭代器的容器都可以用这种方式循环遍历。

由于迭代器是向量中跟踪当前位置的惟一项，所以我们需要用它来提取所需的信息。参照指针的概念，我们可以简单地访问迭代器所指向的数据。

在标准迭代器循环后是像使用典型数组一样使用向量的一个例子。需要注意，数组下标不能用于向链表中插入元素——只能用于访问已有元素。

1.4.4 链表（List）

STL 链表可能是使用得最广泛的基本 STL 结构了。它用双向链表实现，因而任何插入和删除元素的操作都可以在真正的常数时间内完成。此能力的代价是损失了随机访问，而这是向量和双队列所允许的。

使用 STL 容器的一大好处是，有统一的命名约定以及能通用于全库的方法。当你学会了某一类容器的基本操作后，你就基本通晓了所有容器的使用方法。

链表的使用比向量的使用还要容易。push_front()和 push_back()函数的作用不出所料，对链表的迭代遍历也和向量例子中的完全一样。在本代码中，我们将看到许多与向量类中相同的技术。

```
#include <list>
#include <iostream>

using namespace std;

class Foo
{
public:
    Foo(int i)                { m_iData = i;    }
    void SetData(int i)       { m_iData = i; }
    int GetData()             { return m_iData; }
private:
    int m_iData;
};

// 为了提高可读性，定义容器和迭代器的名字
typedef list<Foo*> FooList;
typedef FooList::iterator FooListItor;

void main()
{
    // 创建一个整数链表容器
    FooList c;

    // 将 3 个不同的元素填入链表中
    c.push_back(new Foo(1));
    c.push_back(new Foo(2));
    c.push_back(new Foo(3));

    // 迭代遍历该链表
    for(FooListItor itor = c.begin(); itor != c.end();)
    {
        if((*itor)->GetData() == 2)
            // 示范从链表中删除一个元素的正确方法
        {
```

```
            delete (*itor);
            itor = c.erase(itor);
        }
        else
            ++itor;
    }

    // 确保所有的对象都被删除
    // 链表销毁并不会自动完成该项工作
    for(FooListItor itor2 = c.begin(); itor2 != c.end();
        ++itor2)
        delete (*itor2);
}
```

我们在该例中看到了相同的容器操作基本类型，不过我们在此使用了用户定义的对象而不是内部数据类型，这更符合通常的使用场合。我们通过按值插入数据来检验它在实践中的差别。

STL 容器并不对传递进的数据进行操作，它们只是复制所接收和分发的数据。为了消除在内存中复制大的数据结构所造成的开销，可以传递指针，让这些指针指向较大的动态分配的对象。很自然，我们用于示例用途的对象非常小，但如果进行大量复制，也足以对性能造成严重影响。

使用指针指向动态分配的结构或对象时，有几件事情需要记住。首先，你要负责在使用完对象后释放所有分配的内存。容器并不知道将使用何种类型的数据，因而它们不可能帮你自动释放内存。

其次，许多运算可能会失败，这是因为它们直接对对象或结构的指针进行操作，而不是对对象或结构本身。以链表的 sort()函数为例，它使用 < 运算符来比较值并依此结果进行排序。就算运算符对于类 Foo 是合法的，但链表的排序仍是按照指针的实际值而不是对象中数据的值。

因而，有必要设计自己的比较运算符，在比较之前先对指针解引用。这种比较很容易做到，读者可以参看 James Boer 的文章《资源与内存管理》中的实例代码。

第三点对所有常规指针操作都适用，但在 STL 环境中还是值得一提。记住，在复制容器时，复制的仅仅是指针而不是对象。如果产生了重复的指针，将极难确定哪一个对象需要删除。对此只有两种解决办法：对对象使用智能指针（smart pointer），或避开容器间复制元素的 STL 的例程或算法。

在迭代遍历链表的时候，对从链表中删除元素的做法须非常小心。因为删除当前指向的元素将导致迭代器失效，所以你必须确保正确使用 erase()函数的返回值，它是这个函数将检索出的容器中的下一个合法位置。通过将这个返回值分配给老的迭代器，我们就跳过了非法位置。但是这又给我们带来另一个问题。当 for 循环在循环结束处试图对迭代器递增时，由于我们已经对迭代器递增让其指向了下一个位置，因而就会出现问题。为了解决这个问题，我们将递增运算从 for 循环体移到了循环中的条件选择语句内，在元素没有删除时才进行递增。

一般来说，最好使用算法来从容器中删除元素，而不是手工迭代来做，如算法 remove_if()

就能安全有效地进行该项操作。这些算法（以及如何创建你自己的算法）的完整列表和描述将有一整本书的篇幅之多，在此我只向你推荐本文最后列出的资源，以供进一步学习之用。

1.4.5 双队列（Deque）

双队列，或双端队列，是为需要在容器的两端都能插入和删除元素，但不需要（或不经常需要）在中间插入和删除元素的场合设计的。和向量一样，双队列可以平均在常数时间内完成容器前端或后端的插入和删除，在中间插入和删除元素则比较慢。双队列也允许随机访问，但由于其内部数据较为复杂（以链接的内存块系列来组织的），双队列随机访问的效率不如向量。和向量不同的是，双队列没有决定何时进行追加内存分配的机制。

```
#include <deque>
#include <iostream>

using namespace std;

// 为了提高可读性，定义容器和迭代器的名字
typedef deque<int> IntDeque;
typedef IntDeque::reverse_iterator IntDequeRItor;

void main()
{
    // 创建一个整数双队列容器
    IntDeque c;

    // Fill the deque with 3 different elements
    c.push_front(3);
    c.push_front(2);
    c.push_front(1);
    c.push_back(3);
    c.push_back(2);
    c.push_back(1);

    // 向后循环该链表
    // 需要使用专用的迭代器和写法
    for(IntDequeRItor ritor = c.rbegin(); ritor != c.rend();
++ritor)
        cout << "Value = " << (*ritor) << endl;

    // 删除第一个和最后一个元素
    c.pop_front();
    c.pop_back();

    // 直接访问元素——如果需要，查看双队列是否非空
    // 访问不存在的元素将导致不可预计的行为，很可能产生访问违例
    if(!c.empty())
    {
        cout << "Front = " << c.front() << endl;
```

```
        cout << "Back = " << c.back() << endl;
    }
}
```

在上述程序清单中我们可以看到与 STL 用法相似的代码，不过有一些新的变化。首先，让我们来介绍反向迭代器。你可能注意到了，到此为止我们所有的迭代都是正向的。虽然有双向迭代器，但创建一个专门的反向迭代器，像标准迭代器那样使用，通常会更简单。

我们需要反向迭代器的原因是：由于容器的约束条件（见图 1.4.1），我们不能简单的向后迭代并希望能用相同的条件（itor != begin()）来进行检查，这将使得容器中第一个元素在迭代循环之外。我们利用一个反向迭代器配合 rbegin()和 rend()函数来代替普通迭代器。这些函数与它们正向查找的版本类似，只是 rbegin()实际指向最后一个元素，而 rend()指向第一个有效入口的前面。这是这些函数的正向版本的精确复制。因为你可以使用完全相同的语法，使用反向迭代器的加操作进行反向遍历。

在这个例子中，我们引入对应于 push_front()和 push_back()的函数对 pop_front()和 pop_back()。这些函数分别从容器的前端或后端简单地删除一个元素。注意，元素的值不会被返回。你应该使用我们在这个例子中引入的其他几个函数访问前端和后端的值：front()和 back()。这些函数返回容器前端或后端的元素值。在这个例子中，我们在尝试访问这些元素之前，使用 empty()函数检测并确保容器不为空。试图访问空列表中的元素的结果是“未定义行为”，你可以预期这种行为将造成某些类型的访问异常：pop_front()和 pop_back()在对空容器进行操作时没有意义。

1.4.6 映射表（Map）

STL 映射表或许是使用起来最复杂，而适用性最广的简单容器。这里我们将演示如何使用映射表代替其他基于树的结构：集合、多重集和多重映射表。通过学习映射表的于原理，你可以很容易学会其他类型容器的使用，因此我们将这个研究工作留给你自己。

映射表本质上是一个值对（value-pairing）容器。两个任意类型的数据以键/值结构配对并插入到容器中。通过键查找值可以获得 O(log *n*)的时间复杂度。虽然没有 hash 表那么高效，但差别基本上可以忽略不计，同时还能获得插入数据时自动排序的优点。这种存储方法的直接好处是，允许迭代访问复杂的排序数据（一个平衡二叉树，又被称为红黑树）。

```
#pragma warning(disable:4786)
#include <map>
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

// 这个函数对象允许我们比较映射表容器
template <class F, class S>
class value_equals
{
```

```
private:
    S second;
public:
    value_equals(const S& s) : second(s)
    {}
    bool operator() (pair<const F, S> elem)
    { return elem.second == second; }
};


// 为获得更好可读性而进行的容器和迭代子的类型定义
typedef map<int, string> isMap;
typedef isMap::value_type isValType;
typedef isMap::iterator isMapItor;

void main()
{
    isMap c;

    // 插入键/值对
    c.insert(isValType(100, "One Hundred"));
    c.insert(isValType(3, "Three"));
    c.insert(isValType(150, "One Hundred Fifty"));
    c.insert(isValType(99, "Ninety Nine"));

    // 显示所有键和值
    for(isMapItor itor = c.begin(); itor != c.end(); ++itor)
        cout << "Key = " << (*itor).first << ", Value = "
             << (*itor).second << endl;

    // 你也可以通过关联数组方式访问映射表
    cout << "Key 3 displays value " << c[3].c_str() << endl;

    // 或者以关联数组方式插入键/值对
    c[123] = "One Hundred Twenty Three";

    // 基于键找到并删除一个特定值
    isMapItor pos = c.find(123);
    if(pos != c.end())
        // 删除一个元素将使指向它的所有迭代器无效
        // 现在调用 pos++ 结果将是未定义行为
        c.erase(pos);

    // 基于值找到并删除元素
    pos = find_if(c.begin(), c.end(), value_equals<int, string>("Ninety Nine"));
    if(pos != c.end())
        c.erase(pos);

    // 如果你需要在遍历列表时删除元素……
    for(isMapItor itr = c.begin(); itr != c.end(); )
```

```
        {
            if(itr->second == "Three")
                c.erase(itr++);
            else
                ++itr;
        }
    }
```

我们已经在这个例子中引入一个新的中间数据类型，value_type，它表示容器中每个元素所使用的键/值对。为方便使用，我们将此类型连同其他常用类型一起定义类型。

像其他容器一样，可以使用 insert()函数联合插入一个键/值对，惟一不同的是你必须插入一个 map::value_type 类型。映射表在每个条目被插入时将之排序，因此在任意时间容器中的内容都是按键进行排序的。我们可以在使用迭代器遍历映射表并显示所有键及其关联值的时候看到这个排序关系。

通过迭代器访问键和值时需要使用一个额外的结构来遍历。取迭代器指向值时将返回 value_type 结构，它有两个成员数据：first 和 second。访问 first 可以获得键的数据；访问 second 可以获得值数据。

除了可以通过迭代器访问以外，映射表还提供了通过它们的键值随机访问的接口。映射表可通过类似关联（或稀疏)数组的方式使用。可以使用 index()运算符访问或插入元素。然而这种运算符在使用时需要小心。如果你试图访问一个索引值并不存在的元素，映射表将使用元素类型的缺省构造函数创建一个新的元素并插入到其中。这往往不是使用者所希望获得的结果，需要小心防止这种情况产生。

我们可以看到 find()函数能够简单地基于键查找元素。因为键已经被排序，这个函数能够获得 O(log *n*)时间复杂度的响应。

如果要基于值查找一个元素，我们需要做一些额外的工作。最好情况下，这些工作将在线性时间内完成，因为数据是以键而不是值为依据进行排序的。这个问题的解决方法是我们前面提到的泛型 STL 算法。我们使用 find_if()算法解决这个特定问题。这个函数需要三个参数：一个迭代器指向开始位置；一个迭代器指向结束位置；一个函数判断算法何时返回真值。迭代器是自说明的，但函数对象，或说是 functor，则需要一些额外的详细解释。

在 STL 中，类和重载的函数运算符（知道你可以这样做吗？）被用于代替函数。这种替代使得泛型编程问题能够同时获得封装性和类型安全。这个例子中的函数对象简单地比较 value_pair 中的 second 值并且返回结果。使用我们希望搜索的结果值初始化函数对象，提供了一个清晰完整的封装性良好的解决方案。注意，对大多数解决方案来说，STL 提供随时可用的函数对象，只需要简单地将之插入你的代码。请查看全面的 STL 书籍来了解那些已经可用的不同算法和函数对象。

前面的图描述了在容器中搜索值的首选方法，但如果你需要迭代遍历并手工在映射表中删除对象，我们还需要向你展示一种完成类似工作的合适方法。在迭代遍历一个映射表时删除对象会造成一个特殊的速度方面的问题，STL 设计者没有像在其他容器那样提供一个 erase()函数，来删除特定元素并返回下一个有效位置。不幸的是，因为这个疏忽，我们不能使用类似第二个代码片断中那样简单的方法来删除元素。相反，我们不得不在删除后重新排序，以使我们的迭代器不会无效。

在这个例子中，与在 for 循环语句中步进迭代器不同，我们在循环体内以条件语句方式完成步进迭代器工作。注意，当一个元素需要被删除，我们在将迭代器作为参数传递给 erase()后使用后置递增运算符步进此迭代器；但如果此元素不需要被删除，我们使用标准前置递增运算符。因为操作发生的顺序不同，这个方法允许在不使用临时迭代器进行重新排序的情况下进行安全的迭代操作。不幸的是，较之由设计者提供更快速的 erase()函数，创建这类代码的需求更容易引入错误。幸运的是，标准委员会已经考虑在未来修订这个函数，以避免这种类型的潜在问题。

现在可以回答你可能要问的问题，“为什么你总是在迭代循环中使用前置递增运算符？”

答案是效率。后置递增运算符需要返回一个当前旧值的拷贝，因此它需要使用一个临时对象。两个不同解决方法以同样的方式工作，但除非你有类似前面例子中的特殊原因要使用后置递增（或后置递减）运算符，否则都应该使用前置递增和前置递减运算符。

1.4.7 堆栈(Stack)，队列(Queue)和优先队列(Priority Queue)

我们将堆栈、队列和优先队列放在一起介绍，因为它们的使用方法非常简单并且只需要做一点额外的解释。这些容器是真实的容器适配器（container adapter）的例子，因为它们是使用上述现有容器的一个受限接口实现的。

1. 堆栈

STL 堆栈类提供三个主要成员函数（push()、pop()和 top()），从容器中增加和删除元素。这些成员函数分别用于将元素入栈，从栈中弹出元素，或获取栈顶元素。此外还提供了 size()和 empty()函数用于检查栈的当前状态。

缺省情况下，栈使用双队列实现，但也允许在构造函数中改变实现容器。

```
// 使用双队列作为底层容器类型实现一个栈
stack<int> c;

// 使用向量作为底层容器类型实现一个栈
stack<int, vector<int> > c;
```

注意，使用向量也许并不是一个很好的选择，因为 push()、pop()和 top()实际映射到 push_back()、 pop_back()和 back()。任何支持这些函数的容器都可以被用于堆栈类的底层实现。注意上述例子中的第二行代码，我们需要确保两个大于运算符之间有一个空格。否则他们将被错误地解析为一个单独的流操作符“>>”。

值得重视的是，堆栈类和其他 STL 容器一样，通过牺牲安全性来获得高性能。因此，类假设当你调用 pop()或 top()时，至少有一个实际值在堆栈中存在。因此在执行这些操作之前，总是需要通过 size()或 empty()来验证堆栈不为空。队列和优先队列设计上也遵循相同风格，因此该警告也适用于它们。

2. 队列

队列类工作方式与堆栈很类似，区别在于队列中元素可以从后端插入并从前端弹出。下

列成员函数被队列类定义用来维护元素：push()、 pop()、 front()、 back()。back()指向元素插入的位置，而 front()指向元素弹出的位置。

与堆栈类似，队列也定义了 size()和 emtpy()来管理容器大小。与堆栈类相同，你也可以指定除了缺省的双队列以外的容器作为队列的底层实现容器。与堆栈不同的是，使用向量作为队列的底层实现容器是一种非常低效的选择，因为在向量头部插入元素的性能非常低。然而列表就在很多情况下适用。

3. 优先队列

优先队列以与队列相同的方式工作，只是有一个重要的区别：索引插入元素使用小于(<)运算符立即以降序排序。因为有排序功能，优先队列的构造函数可以通过一个额外的第三个参数，允许你使用自己的函数重载缺省的<运算符。这个能力在你希望插入对象指针而非传入对象值的时候格外有用。通过编写一个函数对象类，以提供<运算符来比较指针指向对象，可以避免优先队列使用缺省比较运算符对指针的值而非对象的值进行排序。在《资源和内存管理》一文中提供了一个示例代码演示了这个函数对象。

1.4.8 总结

STL 对 C++开发人员来说是一组强有力的工具。在同时了解它的作用和限制后，可以在不危及你代码的性能和完整性的情况下，使用它的大部分可用功能。

解释如何使用 STL 的内容可以写一整本书。仅阅读这篇文章不可能覆盖到 STL 库中各种主要函数的使用。如果你希望完整利用 STL 的强大功能，没什么比一本好的参考书更重要。本文最后列出了一些非常优秀的教程和参考书籍。

1.4.9 参考文献

[Nicolai99] Josuttis, Nicolai M., *The C++ Standard Library: A Tutorial and Reference*, Addison Wesley Longman, Inc., 1999.

[Stroustrup97] Stroustrup, Bjarne, *The C++ Programming Language*, third edition, Addison Wesley Longman, Inc., 1997.

[Breymann98] Breymann, Ulrich, *Designing Components with the C++ STL*, Addison Wesley Longman, Inc., 1998.

1.5 一个通用的函数绑定接口

Scott Bilas

脚本引擎（scripting engine）和网络消息传送（network messaging）有一个很重要的共同点，那就是它们都要求以类型安全、有效且便利的方式与游戏的功能接口。本文提出了一种方法，可以导出函数并在运行时进行动态绑定，而且不会牺牲运行速度。

1.5.1 要求

对脚本引擎的基本要求是，可以调用函数并传递其参数。为此，我们需要知道这个函数的名称、它在内存中的位置以及它带有的参数。这些参数的类型，作为语言的一部分，必须是脚本引擎直接支持的类型。我们假定支持的类型有 bool、float、int、string 和 void。

对网络 RPC（远程过程调用）的基本要求是，可以从远程机器上调用函数并传递其参数。考虑到我们的机器很可能在不同的内存地址上运行代码，我们不能直接在网络上传递函数指针，而必须将它们转换成一种两边都能识别的标记。我们使用序列 ID 来表示，这种序列 ID 可以快速地与实际的函数指针进行互换。另外，我们还需要知道如何识别出参数中的字符串和内存指针，这样才能将它们指向的数据包裹进 RPC 块的末端，以便在网络上进行传送。

为了方便使用，我们应能简单地调用具有 RPC 功能的函数，而不用在调用端的代码中显式地打包进任何参数。如果调用需要在其他的机器上执行，调用函数应该自动将它的参数和序列 ID 发送到网络进行传输；如果调用需要在本地执行，则调用函数直接执行代码。远程机器上的调度程序会依据序列 ID 查找函数，并在解出函数指针后直接调用他们。

1.5.2 关于平台

在此需要指出，本文中的范例代码只针对以下特定平台：x86 机，Win32 操作系统，Visual C++ 6.0 环境。具体为：

（1）有一些汇编代码是 x86 专用的。

（2）使用的名称 mangle 与 unmangle，以及调用约定是特定于 Visual C++ 6.0 的。

（3）使用到了 Win32 的映像（DLL/EXE）导出方式。

虽然实现上有一些差别，但至少本文中的概念对其他平台还是适用的。所有 x86 汇编代码都可以转换到任何其他的指令集上，当然你需要知道目标平台的调用约定。动态连接库（DLL）并不是 Win32 特有的，它所需要的只是一个从导出函数名称到内存地址的映射表。最后，你应该知道其他的编译器（尤其是开源编译器，如 GCC）是如何 mangle 与 unmangle 名称的。

1.5.3 第一次尝试

让我们先回头看看目标任务是什么。我们的目标是寻找到一种通用的方式导出游戏的功能，做到可以从脚本调用或作为 RPC 在网络上传送。以下是一个非常简单的解决方案：

```
void Foo( void );
void Bar( void );
// ...

enum eFunction
{
   FUNCTION_FOO,
   FUNCTION_BAR,
   // ...
};

struct Function
{
   typedef void (*Proc)( void );

   const char* m_Name;
   Proc        m_Proc;
   eFunction   m_Function;
};

Function g_Functions[] =
{
   { "Foo", Foo, FUNCTION_FOO, },
   { "Bar", Bar, FUNCTION_BAR, },
   // ...
};
```

eFuncion 枚举提供了序列 ID 表，为所有可用函数提供惟一的 ID。Function 结构将文本名称映射到函数指针与惟一的 ID 上。最后，g_Function 数组是系统中所有发布函数的集合。范例函数导出的当然就是 Foo 和 Bar。

我们设想的脚本引擎可以在编译脚本时查找 g_Function 数组，通过名称来找出调用，找到后直接调用该过程。这项查找最好通过索引进行，以便提高速度。我们设想的网络消息传送系统可以将函数调用转换成它们的 eFunction ID，并使用这些 ID 来找出其他机器的 RPC 调用。这实现起来很简单，也很容易。

这个解决方案的效果很好，但有一个致命缺点：所有的函数必须一样——它们必须都不

带参数，且返回 void。我们可以改变 Function::Proc 类型，以使得函数可以至少返回一个值并可以带有参数。不过，这并不是个可取的解决方案，因为要求所有发布的函数都有统一的特征是不大可能的。此外，现代游戏所需的函数集数量极多，而且形式多样，这种限制非常不方便。

解决这个问题的方法是将参数从实际类型到 Function::Proc 所要求的通用类型来回转换。例如，我们可以让函数传递两三个无符号整数，并将实际的参数打包进这些整数中。这就是应用程序接口（API）用于回调（如窗口过程）的常用的有效技术。但是这种方法并不安全，而且基于一般用途的脚本语言也不能很好地支持它。此外，这种方法也不能区分哪些类属参数是指针。这个缺点使得在网络上为 RPC 传递参数非常困难。让我们再来试试其他的方法。

1.5.4 第二次尝试

通常，对“第一次尝试”中问题的部分修改是提供一个封装类来在内部缓冲区存储参数，并提供加入和取出的方法用于数据进出对象。

```
struct Parameters
{
    std::vector <unsigned char> m_Data;

    bool        ExtractBool  ( void );
    int         ExtractInt   ( void );
    float       ExtractFloat ( void );
    const char* ExtractString( void );

    void AddBool  ( bool );
    void AddInt   ( int );
    void AddFloat ( float );
    void AddString( const char* );
};

void Foo( Parameters& params )
{
    int   param1 = params.ExtractInt  ();
    float param2 = params.ExtractFloat();
    // 使用 param1, param2...
}

void Bar( Parameters& params );
// ...

enum eFunction
{
    FUNCTION_FOO,
    FUNCTION_BAR,
};
```

```
struct Function
{
   typedef void (*Proc)( Parameters& );

   std::string m_Name;
   Proc        m_Proc;
   eFunction   m_Function;
};

Function g_Functions[] =
{
   { "Foo", Foo, FUNCTION_FOO, },
   { "Bar", Bar, FUNCTION_BAR, },
   // ...
};
```

现在我们就可以向任意函数传递一般性的参数了，这是一项重要的进步！但是这个方法也有它自身的一系列缺点，有些和前面的方法一样。

首先，这个解决方案本质上是非类型安全且危险的，这源于其加入/取出功能。因为不知道进入 Parameter 对象的应该是什么，在编译的时候 C++编译器不能对类型进行检查；就其本身的定义而言，进入 Parameter 对象的可以是任意类型。为了提供基本的运行时检查，我们能做的就是在每次调用 Add 方法时存储类型，然后每次调用 Extract 方法时，在调用函数中检查类型。这样做效率不高而且容易出错。此外，一旦函数参数改变，就必须找出并更新所有对该函数的调用。编译器不能检测出此类改变，手工查找替换的动作又增加了错误发生的可能性。如果不小心漏了一个调用的改变，将导致难以查找的潜在 bug。

1.5.5 部分解决方法

让我们从后往前来考虑。在此，我们真正想要的是一个函数说明表，它提供以完全通用的方式调用一个特定函数所需要的全部信息。我们要能用一块内存来建立栈（如 push 参数），能直接跳到所调用的函数，然后检索返回值以传回给原始的调用函数。为此，我们需要知道该函数的名称、在内存中的位置、返回类型、参数类型以及调用约定。

```
// 函数说明
struct Function
{
   // 简单变量说明
   enum eVarType
   {
      VAR_VOID, VAR_BOOL, VAR_INT, VAR_FLOAT, VAR_STRING,
   };

   // 可能的调用约定
   enum eCallType
   {
      CALL_CDECL, CALL_FASTCALL, CALL_STDCALL, CALL_THISCALL,
```

```
    };

    typedef std::vector <eVarType> ParamVec;

    std::string  m_Name;
    void*        m_Proc;
    unsigned int m_SerialID;
    eVarType     m_ReturnType;
    ParamVec     m_ParamTypes;
    eCallType    m_CallType;
};

typedef std::vector <Function> FunctionVec;

// 全局输出函数规范集
FunctionVec g_Functions;
```

目前假定有方法把所有输出函数的说明填入 g_Function（后面会解释如何做）。现在，我们该如何使用这些信息来实际调用函数呢？首先，需要知道我们平台的各种调用约定是如何作用的。

1.5.6 调用约定

可以查看编译器的文档看看其调用约定是如何作用的。在 x86 Win32 Visual C++上，所用函数调用都有以下共同点：

（1）栈的开口方向向下，所有参数按照从右到左的方向压入栈。实际上，在栈中按照内存地址的升序，参数的顺序是从左到右的。

（2）栈指针（esp）总是指向栈的最低内存地址，而不是像它的名字“top”。它必须按 dword（4Byte）对准，所以每个推入的参数都必须同样地调整为 dword。push 指令先递减 esp 再存储数据；pop 指令先装数据再递增 esp。

（3）按值传递的参数是完整地推入栈中的。双精度（8Byte）或自定义类型被复制到栈中，索引和指针所包含的内存地址直接推入栈中。

（4）简单非浮点返回值（如整数和指针）储存在 eax 寄存器中。8 字节结构返回到 edx 和 eax 中作为一对。浮点和双精度通过 FPU 返回到 STO 中。自定义类型的返回值将其地址最后压入栈中，不过它们也将返回到 eax 中。

以下是我们要支持的两个调用约定：

- __cdecl。调用函数清空栈，这意味着在调用完成后调用函数负责从栈中弹出它自己的 argument。对可变 argument 的函数要求使用这条约定，因为被调用函数不一定有弹出正确数量 argument 所需的信息。在 C 和 C++中，这条约定对静态和全局函数是默认的。
- __stdcall。被调用函数清空栈。这是用于 Win32 API 调用的标准约定，其原因是，对于客户代码大小来说它更高效。

对其他三个调用约定（__fastcall 和两个 thiscall 变体）的支持超出了本文的范围。根据应用的需要，那三个调用可能也值得研究并进行支持。

现在我们有了足够的信息用这两个约定进行通用函数调用。我们还需要有函数在 FPU STO 寄存器中检索浮点值。以下是干这项“苦活”的一些函数。

```
DWORD Call_cdecl( const void* args, size_t sz, DWORD func )
{
   DWORD rc;                // 这是我们的返回值...
   __asm
   {
      mov   ecx, sz         // 获得缓冲区大小
      mov   esi, args       // 获得缓冲区
      sub   esp, ecx        // 分配栈空间
      mov   edi, esp        // 目标栈帧的起始地址
      shr   ecx, 2          // 以双字为单位操作
      rep   movsd           // 复制参数到真正的栈中
      call  [func]          // 调用函数
      mov   rc,  eax        // 保存返回值
      add   esp, sz         // 恢复栈指针
   }
   return ( rc );
}

DWORD Call_stdcall( const void* args, size_t sz, DWORD func )
{
   DWORD rc;                // 这是我们的返回值...
   __asm
   {
      mov   ecx, sz         //获得缓冲区大小
      mov   esi, args       //获得缓冲区
      sub   esp, ecx        //分配栈空间
      mov   edi, esp        //目标栈帧的起始地址
      shr   ecx, 2          //以双字为单位操作
      rep   movsd           //复制参数到真正的栈中
      call  [func]          //调用函数
      mov   rc,  eax        //恢复栈指针
   }
   return ( rc );
}

__declspec ( naked ) DWORD GetST0( void )
{
   DWORD f;                // 临时变量
   __asm
   {
      fstp dword ptr [f]      // 弹出 ST0 到 f
      mov eax, dword ptr [f]  // 复制到 eax
      ret                     // 完成
   }
}
```

现在，有了函数的地址和存储在内存缓冲区中的参数，我们就可以以几乎完全通用的方式调用函数。

1.5.7 调用函数

在进行实际调用之前，客户子系统（脚本引擎、网络 RPC 等）需要做一些预备工作。首先，要查出在 g_Function 中对应于要调用函数的 Function 结构的实例。对脚本引擎来说，要验证此函数说明是否与预期相符：检查并在必要时转换参数；如果不符合则给出错误信息。这个过程的开销很大，应该在脚本解释的时候进行，不应实时进行。

为网络 RPC 查找 Function 实例要复杂一些。一种好的办法是从函数中截取出将要在网上进行的调用。在 g_Function 中用小于当前指令指针（eip）的最高 m_Proc 值，找出当前正被调用函数的 Function 实例。示例如下：

```
__declspec ( naked ) DWORD GetEIP( void )
{
    __asm
    {
        mov eax, dword ptr [esp]
        ret
    }
}

// 示例的 RPC 调用函数
void NetFoo( bool send, int i )
{
    // FindFunction() 将从 g_Functions 中查找最高的 'm_Proc'
    // 小于'ip' 并返回之
    static const Function* sFunction = FindFunction( GetEIP() );
    if ( send )
    {
        // RouteFunction() 应该将参数打包并通过网络发送请求。
        RouteFunction( sFunction, (BYTE*)&send + 4 );
        return;
    }

    // ... 普通的 NetFoo 执行代码
    printf( "i = %d\n", i );
}
```

下一步构造传递给函数的参数缓冲。对建立在虚拟机上的脚本引擎来说，这很容易；所有的参数都已经按照 dwrod 对齐，放在了虚拟的栈中。我们只用获取参数的起始位置，顺着传递下去即可。网络 RPC 则要复杂一些。指针不能直接在网上传输，但是可以采用专用的字符串，因此要分析 VAR_STRING 类型的 m_ParamTypes，并将字符串内容加到向网络进行传输的缓冲区的末端。在接收端，解析指向附加数据的指针，将数据块的开始作为参数缓冲的起始处。

现在，我们有了 Function 实例和参数缓冲，可以根据不同的 m_CallType 调用 Call_cdecl() 或 Call_stdcall()，传入参数缓冲和 m_Proc。如果 m_ReturnType 是 float 或 double，就可以使

用返回值或调用 GetSTO()去获取。这就是用通用的方式调用函数的全部。

1.5.8 完备解决方案

到目前为止，我们一直假定 g_Function 数组已经建立。让我们返回去把这点补充完整。有几种方式建立 g_Function 数组，使用宏或函数建立可能是实现起来最简单但是使用的时候最不安全的。

```
float Foo( int, const char* );
int Bar( void );

void SetupFunctionExports( void )
{
   {
      Function function;

      function.m_Name       = "Foo";
      function.m_Proc       = Foo;
      function.m_SerialID   = g_Functions.size();
      function.m_ReturnType = Function.eVarType::VAR_FLOAT;
      function.m_ParamTypes . push_back( Function.eVarType::VAR_INT );
      function.m_ParamTypes . push_back( Function.eVarType::VAR_STRING );
      function.m_CallType   = Function.eCallType::CALL_CDECL;

      g_Functions.push_back( function );
   }

   {
      Function function;

      function.m_Name       = "Bar";
      function.m_Proc       = Bar;
      function.m_SerialID   = g_Functions.size();
      function.m_ReturnType = Function.eVarType::VAR_INT;
      function.m_CallType   = Function.eCallType::CALL_CDECL;

      g_Functions.push_back( function );
   }
}
```

以上示例只是作为演示之用，并不是最优的。可以用一些帮助函数和宏简化表中新函数的添加来改进上述示例。不过，它终归不是很安全，也不方便。向表中添加新函数意味着有人要写代码来规定其类型、名称和调用约定。改变函数（如增加参数）而没有更新表，将导致难以调试的问题。要保持函数说明与其实际原型同步需要耗费很大的工作量。

需要找到一种方法自动建立表并安全地消除这些隐患。很幸运，C++编译器便有我们所需要的全部信息。在分析函数原型的时候，编译器会建立一个内部的函数表示（其返回类型、参数、调用约定等），正是建立函数说明所需要的。但不幸的是，在代码中无法访问这些信息；

此外，当连接程序建立最终的 EXE 程序时，这些信息会被删除。可以寻找一种方式使用 PDB（调试符号库）来查询所需信息，但是不能要游戏附带调试符号。此外，并没有简单的方法分辨哪些是输出函数。

结合 Win32 映像文件输出表的功能和 C++语言名称 mangle 的功能，可以获取我们所需的信息。如果使用__declspec(dllexport)关键字来标记输出函数，则此函数的名称和地址将出现在 EXE（或 DLL）输出表中。此外，由于是 C++应用程序，这些名称将被 mangle 以支持类型安全和重载名字定位（overloaded name resolution）。Mangle 后的名称编码了我们所需的所有信息，因此所要做的只是将名称解码成我们可以理解的形式，然后用它建立 Function 表项，添加到 g_Functions 中。

mangle 名称的形式与其特定的实现有关，而且也没有文档化，它甚至在不同版本的 Visual C++中都不一样，因此最好不要对其进行逆向工程。其实也没有必要，Microsoft 在 ImageHlp.dll 和 DbgHelp.dll 中都输出有一个叫作 UnDecorateSymbolName()的名称解 mangle 函数。如果我们从最后的实例中获取 Foo()函数并 DLL 输出，表项?Foo@@YAMHPBD@Z 将出现在 EXE 的输出表中。如果 unmangle 了其名称，所得到的是 float_cdecl Foo(int, char const *)。这就能够很方便地分析并转换成添加到 g_Functions 表的 Function 表项。

因此，现在我们建立 g_Functions 的过程如下：

（1）遍历 EXE 输出表中的所有表项，并检索每个函数的地址和 mangle 名称。

（2）unmangle 每个名称获得文本形式的函数原型。

（3）分析函数原型，检索名称、类型以及调用约定信息。

（4）将结果存储在新的 g_Functions 表项中。重复每个输出。

遍历整个输出以获取函数地址和 mangle 名称需要具备 Win32PE 格式文件二进制格式的知识。此格式的说明文档可以在 Microsoft Developer Network Library（http://msdn.microsoft.com）中找到。在“Microsoft Portable Executable and Common Object File Format Specification”中查找“.edata”可以找到 Win32 输出表的结构。

最后还有一点细节。输出表中的表项指向的是一个中间表，该中间表才指向真正的函数。如果你所关心的只是绑定到函数并以通用的方式进行调用，这一点并不重要。但是如果你需要反查找并从被调用函数转换 eip 以找到它的函数实例（RPC 中需要，如前所述），就需要获取实际的地址来比较，而不能是中间表表项的地址。这很简单：解引用 DLL 输出表项给出的地址，找到中间表表项。第 1 个字节应该是 0xE9（jmp），接下来 4 字节是到函数实际进入点的偏移。将 DLL 输出表项给出的地址加 5 得到完整 jmp 指令，加上那个 4 字节偏移，便得到函数的入口点地址。这个地址就可以用于从 g_Functions 寻找 Function 实例的反向查找。

1.5.9 结论

以上便是以完全通用的方式调用函数的全部。为了在系统中发布函数并允许其他子系统（如脚本引擎和网络 RPC）绑定到它，用_declspec（dllexport）对其进行标记即可（这个繁琐的标记符号最好用宏来表示，以使程序更整洁）。在运行时，函数绑定发布程序遍历整个 Win32 输出表并从每个表项提取出名称、类型和调用约定信息。其他子系统可以通过内存地址、名称或序列 ID 查找函数，并使用_cdecl()或_stdcall()调用方式进行调用。

这些工作看上去好像并不必要。对于小输出集的小型项目来说，可能的确是。但稍大型的项目很可能要经常改变。一旦基础工作做好了，向系统添加新函数就如同标记它们为输出一样简单，并且这些新函数能立即可用。这个过程不仅本身值得去做，它还可以成为开发团队各方面能力的强大推进力量。当与基于一般用途的脚本引擎结合的时候，除了能够依照本来编写的意图服务于特定内容需求以外，它还可以成为一个有用的调试工具。

出于篇幅限制和简洁性的考虑，我们略过很多相关特性。通用函数绑定的概念在很多方面都可以再延伸。可以很容易地对它进行改进，让它支持指针和索引、可变参函数、非字符串的网络传送。在 RPC 封装中支持自定义类型，可以通过串行接口实现，在后处理 RPC 参数缓冲到网络输出缓冲时进行检测和直接调用。此外，支持调用类成员函数非常有用，而且也容易实现。最后，还有个不一定必要的功能，那就是对 EXE 进行后处理，把输出表脱离出来，并将其转换成本地数据格式以便直接导入到 g_Functions 中。这项功能可以增强安全性能，但它需要游戏附带 DbgHelp.dll。

1.5.10 参考文献

Microsoft Developer Network Library, http://msdn.microsoft.com.

1.6 通用的基于句柄的资源管理器

Scott Bilas

所有的计算机应用都是数据库。它们大部分时间都在处理数据资源——创建、销毁、高速缓存、查询、保存以及恢复各种类型的对象。游戏通常有多种类型的数据库，每种数据库常常将各种不同的情况硬编码，以保证速度。游戏数据库的例子有：文件系统、纹理管理器、字体管理器、游戏角色管理器。在此之上有大量基于不同游戏风格和内容的专用数据库。

建立在所有 C++游戏中的一个资源数据库就是基本对象内存管理器（basic object memory manager）。程序员调用 new 创建新对象并向四处传递其指针，以便其他对象能向它传递消息。当该对象不再需要的时候将被删除，其资源返回给系统。通常情况下这种方法很有用，但如果需要考虑共享资源则不行。对此我们需要一个更专业的数据库。

让我们以字体对象为例。字体至少由一个位图和一系列说明组成，如其字符单元定位于 *X*,*Y*（或 *U*,*V*）处，这样图形系统才能将其渲染到屏幕上。对内存的使用和创建的时间而言，这种对象责任重大。游戏中的不同系统，如开发控制台和 GUI 中的文本控制，要使用字体对象，但是不能让每个系统都创建自己的本地字体对象拷贝。显然，那样做的运行速度很慢而且会消耗大量内存。要解决这个问题，我们需要以某种方式来共享字体对象。我们的解决方案是一个叫作 FontMgr 的对象，它的方法有：获取字体指针，在系统空闲时进行加载，将其高速缓存直到不再需要为止。FontMgr 在全局范围有效（可作为 singleton，参见 1.3 节），负责系统中的所有字体对象。

在此我们讨论的是专业数据库。FontMgr 负责处理数据资源，并且由于被作为 API 使用，它承担了“字体中心”的职责。FontMgr 被告知要删除一个字体以释放资源，而游戏中的一些系统仍拥有该字体的指针，该怎么办呢？我们怎样才能既保证系统的安全又不牺牲性能呢？是否应该在建立 MousePointerMgr 的时候再次拷贝粘贴这段代码？ 本文提出了一种简单、安全、通用且高效的方式管理资源对象。

1.6.1 方法

资源管理器的工作是按需创建资源，将它们提供给需要使用的对象，并最终将其删除。用简单指针的方式提供出这些资源显然很容易也很方便，但并不安全。指针会变成空悬指针：即系统的某部分可让资源管理器删除某资源，而该动作马上使得仍指向该资源的所有指针无效。没有好办法阻

止空悬指针问题发生，只有在游戏崩溃的时候才会发现有的对象试图使用已经被删除的资源。问题就在于，由于客户可以随意复制指针而不用告知管理器，没有办法知道到底有多少无效索引。

另一个问题是，使用指针的话，底层的数据组织方式不能改变。任何对缓冲区的再分配都会使得所有相应的指针无效。这个问题在将游戏保存在磁盘上的时候尤其突出。指针不能保存在磁盘上，因为下一次游戏载入时，系统内存的设置很可能不同，而且甚至有可能根本不在同一台机器上。指针必须转换成一种可还原的形式，如偏移量或到某序列的惟一标示符。对该问题的处理需要在客户端代码上做很多工作来支持。

因此坦白说，用指针的方式提供资源，对安全、灵活的资源管理器来说并不是个好主意。我们不使用指针，也不试图编写高智能、过于复杂的“智能指针”，而是加一层抽象，使用句柄，把重任转交给管理器类。句柄是一个古老的编程概念，在 API 上已经成功运用了几十年。Win32 文件系统中 CreateFile()调用返回的 HANDLE 类型就是句柄。表示打开的文件系统对象的文件句柄，通过 CreateFile()调用创建，传递给其他函数（如 ReadFile()、SetFilePointer()）进行操纵，最后由 CloseHandle()关闭。使用无效句柄或已关闭句柄调用函数，会返回错误代码，而不会导致崩溃。这种方法高效、安全且易于理解。

句柄很适合单 CPU 寄存器，因为它能以集合方式高效存储，并且能作为参数传递给函数。它们的有效性易于检查，而且由于不是直接指向资源，可以改变底层的数据组织方式而不会产生无效句柄。这显然优于指针。句柄还易于存盘，因为它们所指向的数据结构可以在游戏恢复时以相同的顺序重建。句柄本身就是惟一标识的，因而可以直接存储，而不需进行转换。

1.6.2 Handle 类

用由两个比特域组成的无符号整数表示句柄（参见程序清单 1.6.1）是一种快速、安全的方式。第一部分（m_Index）是惟一标识符，用于快速地找出指向它的句柄管理器的数据库。句柄管理器可以随意使用这个数字，但最有效的使用方式是将其看作到 std::vector 的简单索引。第二部分（m_Magic）是“魔术数”，用于使句柄有效。通过解引用，句柄管理器可以检查以确保句柄的魔术数部分与它对应的数据库项目相匹配。

Handle 类很简单，基本上只需要管理魔术数。调用 Init()将分配给句柄下一个魔术数（该数自动递增并在必要时回绕）。请注意，魔术数并不是 GUID，它的目的只是为了进行快速简单的有效性检查，并且其前提是不会产生两个对象有着相同的索引和魔术数（需要考虑到回绕）的情况。魔术数为零表示句柄数据为零的“空句柄”。默认的 Handle 创建函数会把自己设为空，在 IsNull()查询时返回 true。这用于错误情况很方便，当函数创建对象并返回指向该对象的指针时，可以返回空指针以表示有错误发生。

Handle 类通常是只读的无符号整数。虽然可以安全地将其置回空来重置句柄，但句柄一旦创建就不应再修改。注意，Handle 是参数化的类，需要一个 TAG 类型来完整定义。模板参数 TAG 除了区分句柄类型外不起任何作用；TAG 类型的对象在系统任何地方都不会被用到。这么做的原因是为了保障类型安全。使用参数化的 Handle 类型可以确保，如果向期望获取某种类型句柄的函数传递另外一种类型的句柄，则不会被编译器编译通过。因此为了保障类型安全，我们创建一个新的句柄类型，作为一个惟一符号并被用作 Handle 类型的参数。TAG

类型可以是任何独立的类型，例如下面的材质句柄：

```
struct tagTexture  {  };
typedef Handle <tagTexture> HTexture;
```

现在我们需要一个句柄管理器用来允许高层所有者通过句柄来分配、解引用和释放对象。

1.6.3 HandleMgr 类

HandleMgr 类型是由三个主要元素组成的参数化类型：数据存储、魔术数存储和空闲队列（这个类在程序清单 1.6.2 中）。数据存储是一个简单的 DATA 类型对象矩阵（或者任何可以随机访问的集合）。DATA 类型是 HandleMgr 的第一个类型参数，应该是一个非常简单的类，包含它控制的资源的上下文信息。例如，在 HandleMgr 中管理文件，DATA 类型可以只是一个文件句柄和文件名：

```
struct FileEntry
{
    std::string m_FileName;
    HANDLE      m_FileHandle;  // 操作系统文件句柄
};

struct tagFile  {  };
typedef Handle <tagFile> HFile;
typedef HandleMgr <FileEntry, HFile> FileHandleMgr;
```

这个简单的句柄管理器维护对应于所有已知打开文件的一组上下文对象。FileHandleMgr 类型不应该被客户程序直接使用，而应该通过对其封装的其他类型（如 FileMgr）来将其句柄抽象到问题域（这就是 DATA 类型的目的所在）。这个类可以这样编写：

```
class FileMgr
{
    FileHandleMgr m_Mgr;

public:
    HFile OpenFile ( const char* name );
    bool  ReadFile ( HFile file, void* out, size_t bytes );
    bool  CloseFile( HFile file );

    // ...
};
```

在调用上述方法的时候，FileMgr 使用 m_Mgr，通过句柄解引用得到实际的 FileEntry 对象。在验证解引用成功后（对无效句柄解引用可能失败），FileMgr 执行实际操作。

对我们的 HandleMgr 类型来说，每个句柄惟一地引用一个对象存储中的元素，加上在魔术数存储中对应的元素。解引用句柄得到实际的 FileEntry 对象的过程非常简单，只需要将句柄的 m_Index 作为对象存储的索引（一个非常快速的操作）。

当解引用句柄时，代码也检查魔术数存储中相同索引的数字是否与 m_Magic 相同，以验

证句柄的有效性。当句柄被释放或分配时魔术数存储中的相应入口被更新为新句柄的魔术数。这个过程基本上能够确保，已释放对象的悬挂句柄，不会因为后续的句柄分配操作将此入口填充，而被引用到一个未知的对象上；相反此操作会导致失败并返回错误代码。很明显，魔术数存储和对象存储应该有相同的数量。

当对象被释放时，句柄管理器将入口的索引添加到空闲队列中。这避免在分配句柄时去使用复杂度为 O(n)的线性搜索算法来获取一个可用入口。值得注意的是 DATA 类型不是传统意义上的 C++类。它不应该具有完成任何重要操作的构造函数和析构函数，例如分配和释放局部资源。对象存储中包括的对象需要被 vector 类自己构造、析构和复制。注意在相同示例 FileEntry 类中使用的 std::string 足够简单并满足我们的需要；它使用引用计数来最小化构造和析构操作的代价，其在 vector 复制时的代价几乎可以忽略不计。

当需要从对象存储中分配一个对象时，我们通常喜欢重用一个在空闲列表中的索引指向的已经构造好但不再被使用的对象。这个对象在可以被使用前需要其成员函数对其重新初始化，因为不能再调用构造函数来初始化它。但对象从对象存储中被释放，它不会被析构；而是将其索引加入到空闲列表中，并手工释放其资源。这个小限制的原因是，我们将 DATA 类型直接嵌入到 vector 中，而不是在每次句柄分配和释放时使用 new 和 delete 操作指针来构造和析构对象。这样做的主要好处是速度快，对象不必每次都完整地构造和析构。为简便起见，初始化和停用（shutdown）代码可以被移至成员函数中以便于 HandleMgr 简单回调。

句柄验证的需求可以根据应用程序通过 HandleMgr 的附加模板参数来选择。例如，测试一个无效句柄的操作可能被认为是无用的并可直接去掉（虽然调试断言总是保留）。对一个更强壮的系统来说错误处理是非常重要的，代码可以检测无效句柄，设置错误条件并中断函数调用。

1.6.4 使用示例

程序清单 1.6.3 提供了一个材质管理器类的例子。这个类允许使用者通过名字在需要的时候构造并获得材质。它自动在删除后卸载材质并提供一组查询函数来使用材质。材质以名字为索引来加速查找，并确保相同的材质不会被重复加入到存储中。为其添加引用技术来使之更安全的工作，作为一个练习留给读者，读者需要以 ReleaseTexture()替换 DeleteTexture()。

另一个更完整的使用文件句柄的例子，可以参考我的 GDC 2000 演讲“*It's Still Loading?Designing an Efficient File System*”，可通过 www.aa.net/ ~scottb/gdc/在线访问。

1.6.5 注意

HandleMgr 类非常简单，是用以阐明一些基本概念的，但它可以通过很多方式被扩展：

- 建立一个能够处理较大 DATA 对象的 HandleMgr 类，通过指针保存对象。这也允许向使用者隐藏数据结构。
- 为标准功能增加自动引用技术，而不是将之留给 HandleMgr 的所有者。
- 为基于嵌入链表的稀疏对象存储提供常量时间的访问方法。使用 STL 风格的迭代子名称和操作以确保一致性。

- 很多数据库，如字体管理器或材质管理器，需要通过名字来访问对象并获取句柄。作为一种标准特性将之添加到类或单独子类中。
- HandleMgr 系统在与 Singleton 模式结合时非常有效（参考书中另一篇文章《一种自动的 Singleton 工具》）。很多游戏中的数据库天然就符合 Singleton 模式。
- 对 Singleton 模式做一些改进，以使通过 TAG 类型参数化的 Handle 类型，能够通过->操作符，在其被管理的 Signleton 模式管理器中解引用自己。
- 保存游戏的功能应该能很容易被添加，但需要定制你的游戏架构。句柄可以直接被保存；只需要确保 HandleMgr 将对象数据和对象索引一起存储，然后在恢复时，索引句柄都将保持有效。

程序清单 1.6.1

```
#include <cassert>

template <typename TAG>
class Handle
{
   union
   {
      enum
      {
         // 使用位域(bit fields)的大小
         MAX_BITS_INDEX = 16,
         MAX_BITS_MAGIC = 16,

         // 解引用断言中需要比较的大小
         MAX_INDEX = ( 1 << MAX_BITS_INDEX) - 1,
         MAX_MAGIC = ( 1 << MAX_BITS_MAGIC) - 1,
      };

      struct
      {
         unsigned m_Index : MAX_BITS_INDEX;  // 资源数组的索引
         unsigned m_Magic : MAX_BITS_MAGIC;  // 需要检查的魔术数
      };
      unsigned int m_Handle;
   };

public:

// 生命期.

   Handle( void ) : m_Handle( 0 ) { }

   void Init( unsigned int index );

// 查询.
```

```
    unsigned int GetIndex ( void ) const  {  return (  m_Index  );  }
    unsigned int GetMagic ( void ) const  {  return (  m_Magic  );  }
    unsigned int GetHandle( void ) const  {  return (  m_Handle );  }
    bool         IsNull   ( void ) const  {  return ( !m_Handle );  }

    operator unsigned int ( void ) const  {  return (  m_Handle );  }
};

template <typename TAG>
void Handle <TAG> :: Init( unsigned int index )
{
    assert( IsNull() );              // 不允许重新赋值
    assert( index <= MAX_INDEX );    // 有效范围验证

    static unsigned int s_AutoMagic = 0;
    if ( ++s_AutoMagic > MAX_MAGIC )
    {
        s_AutoMagic = 1;    // 0 用于 "空句柄"
    }

    m_Index = index;
    m_Magic = s_AutoMagic;
}

template <typename TAG>
inline bool operator != ( Handle <TAG> l, Handle <TAG> r )
{  return ( l.GetHandle() != r.GetHandle() );  }

template <typename TAG>
inline bool operator == ( Handle <TAG> l, Handle <TAG> r )
{  return ( l.GetHandle() == r.GetHandle() );  }
```

程序清单 1.6.2

```
#include <vector>
#include <cassert>

template <typename DATA, typename HANDLE>
class HandleMgr
{
private:
    // 私有类型
    typedef std::vector <DATA>         UserVec;
    typedef std::vector <unsigned int> MagicVec;
    typedef std::vector <unsigned int> FreeVec;

    // 私有数据
    UserVec  m_UserData;       // 我们需要获取的数据
    MagicVec m_MagicNumbers;   // 相应的魔术数
    FreeVec  m_FreeSlots;      // 数据库中需要跟踪的空闲列表
```

```
public:

// 生命期

    HandleMgr( void ) { }
   ~HandleMgr( void ) { }

// 句柄方法

    // 分配
    DATA* Acquire( HANDLE& handle );
    void  Release( HANDLE  handle );

    // 解引用
    DATA*       Dereference( HANDLE handle );
    const DATA* Dereference( HANDLE handle ) const;

    // 其他查询
    unsigned int GetUsedHandleCount( void ) const
    { return ( m_MagicNumbers.size() - m_FreeSlots.size() ); }
    bool HasUsedHandles( void ) const
    { return ( !!GetUsedHandleCount() ); }
};

template <typename DATA, typename HANDLE>
DATA* HandleMgr <DATA, HANDLE> :: Acquire( HANDLE& handle )
{
    // 如果空闲列表为空，则新增一个，否则使用第一个可用表项

    unsigned int index;
    if ( m_FreeSlots.empty() )
    {
        index = m_MagicNumbers.size();
        handle.Init( index );
        m_UserData.push_back( DATA() );
        m_MagicNumbers.push_back( handle.GetMagic() );
    }
    else
    {
        index = m_FreeSlots.back();
        handle.Init( index );
        m_FreeSlots.pop_back();
        m_MagicNumbers[ index ] = handle.GetMagic();
    }
    return ( m_UserData.begin() + index );
}

template <typename DATA, typename HANDLE>
void HandleMgr <DATA, HANDLE> :: Release( HANDLE handle )
{
```

```
    // 哪一个?
    unsigned int index = handle.GetIndex();

    // 确认其有效性
    assert( index < m_UserData.size() );
    assert( m_MagicNumbers[ index ] == handle.GetMagic() );

    // 可以删除了——标记其没有使用并加入到空闲列表
    m_MagicNumbers[ index ] = 0;
    m_FreeSlots.push_back( index );
}

template <typename DATA, typename HANDLE>
inline DATA* HandleMgr <DATA, HANDLE>
:: Dereference( HANDLE handle )
{
    if ( handle.IsNull() ) return ( 0 );

    // 检查句柄有效性——为提供性能可以去掉这个检查
    // 如果你假设索引句柄解引用都总是有效
    unsigned int index = handle.GetIndex();
    if (   ( index >= m_UserData.size() )
        || ( m_MagicNumbers[ index ] != handle.GetMagic() ) )
    {
        // 无效句柄=客户程序错误
        assert( 0 );
        return ( 0 );
    }

    return ( m_UserData.begin() + index );
}

template <typename DATA, typename HANDLE>
inline const DATA* HandleMgr <DATA, HANDLE>
:: Dereference( HANDLE handle ) const
{
    typedef HandleMgr <DATA, HANDLE> ThisType;
    return ( const_cast <ThisType*> ( this )->Dereference( handle ) );
}
```

程序清单 1.6.3

```
#include <vector>
#include <map>
#include <cassert>

// ... [ 平台相关的句柄类型 ]
typedef LPDIRECTDRAWSURFACE7 OsHandle;

struct tagTexture  {  };
typedef Handle <tagTexture> HTexture;
```

```
class TextureMgr
{

// 材质对象数据

    struct Texture
    {
        typedef std::vector <OsHandle> HandleVec;

        std::string  m_Name;        // 重构造时使用
        unsigned int m_Width;       // 宽度
        unsigned int m_Height;      // 高度
        HandleVec    m_Handles;     // surface 句柄

        OsHandle GetOsHandle( unsigned int mip ) const
        {
            assert( mip < m_Handles.size() );
            return ( m_Handles[ mip ] );
        }

        bool Load  ( const std::string& name );
        void Unload( void );
    };

    typedef HandleMgr <Texture, HTexture> HTextureMgr;

// 数据库以名字为索引

    // 大小写不敏感的字符串比较谓词
    struct istring_less
    {
        bool operator () ( const std::string& l, const std::string& r ) const
            { return ( ::stricmp( l.c_str(), r.c_str() ) < 0 ); }
    };

    typedef std::map <std::string, HTexture, istring_less > NameIndex;
    typedef std::pair <NameIndex::iterator, bool> NameIndexInsertRc;

// 私有数据

    HTextureMgr m_Textures;
    NameIndex   m_NameIndex;

public:

// 生命期

    TextureMgr( void ) { /* ... */ }
   ~TextureMgr( void );
```

```
// 材质管理

   HTexture GetTexture   ( const char* name );
   void     DeleteTexture( HTexture htex );

// 材质查询

   const std::string& GetName( HTexture htex ) const
      { return ( m_Textures.Dereference( htex )->m_Name ); }
   int GetWidth( HTexture htex ) const
      { return ( m_Textures.Dereference( htex )->m_Width ); }
   int GetHeight( HTexture htex ) const
      { return ( m_Textures.Dereference( htex )->m_Height ); }
   OsHandle GetTexture( HTexture htex, unsigned int mip = 0 ) const
      { return ( m_Textures.Dereference( htex )->GetOsHandle( mip ) ); }
};

TextureMgr :: ~TextureMgr( void )
{
   // 在析构前释放索引剩余材质
   NameIndex::iterator i, begin = m_NameIndex.begin(), end = m_NameIndex.end();
   for ( i = begin ; i != end ; ++i )
   {
      m_Textures.Dereference( i->second )->Unload();
   }
}

HTexture TextureMgr :: GetTexture( const char* name )
{
   // 插入和查找
   NameIndexInsertRc rc =
      m_NameIndex.insert( std::make_pair( name, HTexture() ) );
   if ( rc.second )
   {
      // 这个一个新的插入操作
      Texture* tex = m_Textures.Acquire( rc.first->second );
      if ( !tex->Load( rc.first->first ) )
      {
         DeleteTexture( rc.first->second );
         rc.first->second = HTexture();
      }
   }
   return ( rc.first->second );
}

void TextureMgr :: DeleteTexture( HTexture htex )
{
   Texture* tex = m_Textures.Dereference( htex );
   if ( tex != 0 )
```

```
    {
        // 使用索引删除
        m_NameIndex.erase( m_NameIndex.find( tex->m_Name ) );

        // 从数据库删除
        tex->Unload();
        m_Textures.Release( htex );
    }
}

bool TextureMgr::Texture :: Load( const std::string& name )
{
    m_Name = name;
    // ... [ 从文件系统载入材质，失败时返回 false ]
    return ( true /* 错误时返回 false */ );
}

void TextureMgr::Texture :: Unload( void )
{
    m_Name.erase();
    // ... [ 释放 surfaces ]
    m_Handles.clear();
}
```

1.6.6 参考文献

[Bilas00] Bilas, Scott, GDC 2000 Talk, It's Still Loading? Designing an Efficient File System, www.aa.net/~scottb/gdc/.

Meyers, Scott, *More Effective C++*, Addison-Wesley Longman, Inc. , 1995.

1.7 资源和内存管理

James Boer

和其他类型的软件相比，计算机游戏和视频游戏需要更多地处理大量的媒体资源，如图形、音效、音乐、视频、模型、动画以及其他类型的内存消耗性数据。要在处理这些大型数据的同时还维持合理的内存布局，这并不是件容易的事情。在本文中，我们将考察一个简单的资源管理器的运行情况，并讨论如何在实际应用中使用它以及如何对它进行扩展。

首先，让我们把问题清晰地定义出来，并给出期望的解决方式。如果给定的时间不足以显示一个加载屏幕或中断某动作，我们希望能在这时使用大于内存容量的数据。假设我们有这样的介质，能在游戏运行时动态地加载数据。在游戏机系统上，很可能是 CD 或 DVD；在 PC 上，则可能是硬盘。

我们的解决方案需要创建资源对象，这些对象能够自动加载、废除，并能基于其使用的模式重新加载数据。我们还将创建一个管理器来协调可用资源并控制对资源对象的访问。这可以通过句柄来实现，句柄在本质上就是惟一标识符。

1.7.1 资源类

首先，让我们考察一下资源类。

```
class BaseResource
{
public:

    enum PriorityType
    {
        RES_LOW_PRIORITY = 0,
        RES_MED_PRIORITY,
        RES_HIGH_PRIORITY
    };

    BaseResource()              { Clear(); }
    virtual ~BaseResource()     { Destroy(); }

    // 清除类的数据
    virtual void Clear();
```

```
    // 完成构造和析构功能的函数
    // 注意继承类的 Create()函数不必完全匹配基类
    // 不应该作这样的关于参数的假设
    virtual bool Create() { return false; }
    virtual void Destroy()    {}

    // 释放并重建必须可放弃，并且支持不使用附加参数来重建类包含的数据
    virtual bool Recreate() = 0;
    virtual void Dispose() = 0;

    // GetSize()必须返回类中数据的大小，IsDisposed()使管理者能知道数据是否存在
    virtual size_t GetSize() = 0;
    virtual bool IsDisposed() = 0;

    // 这些函数设置参数决定哪些类型的资源将被丢弃
    inline void SetPriority(PriorityType priority)
    { m_Priority = priority; }
    inline PriorityType GetPriority()
    { return m_Priority; }

    inline void SetReferenceCount(UINT nCount)
    { m_nRefCount = nCount; }
    inline UINT GetReferenceCount()
    { return m_nRefCount; }
    inline bool IsLocked()
    { return (m_nRefCount > 0) ? true : false; }

    inline void SetLastAccess(time_t LastAccess)
    { m_LastAccess = LastAccess; }
    inline time_t GetLastAccess()
    { return m_LastAccess; }

    // 小于操作符定义资源如何在丢弃时排序
    virtual bool operator < (BaseResource& container);

protected:
    PriorityType m_Priority;
    UINT         m_nRefCount;
    time_t       m_LastAccess;

};
```

BaseResource 类用作模板，其他资源容器类必须从它派生。基类必须覆盖几个成员函数，这几个成员函数对系统的工作方式至关重要。

初始 Create()函数需要能够从磁盘或甚至从内存的其他地方加载一些资源数据。在类中保留必要的数据非常关键，这样才能在 Recreate()函数中重复这项操作任意次数。例如，该数据可能是存储一个待加载位图的路径和文件信息。应用程序必须覆盖 Dispose()和 Recreat()函数，以使得资源管理器能以合适的方式将资源在内存中换入换出。请记住，需要换出的不是“所有”的类数据，而只是资源中最大的那部分（如位图数据、声音缓冲等）。

为了让系统能够正确工作，还需要正确覆盖 GetSize()和 IsDisposed()函数。GetSize()很容易理解，它将返回当前可换出数据的大小；如果数据已被换出，函数将返回大小 0。通常，可以通过加上其他的数据成员计算出对象的实际大小，但实践表明，这样做并不值得。如果数据被废除，IsDisposed()将返回 true，否则返回 false。该类并没有给出如何判断此状态的方法，需要派生类提供必要的数据成员在需要的时候追踪状态。常用的做法是直接检查指针是否为 null，而不是添加数据成员。

有一些数据访问函数，提供了对数据成员 m_Priority、m_nPrfCount 和 m_LastAccess 的访问。m_Priority 是一个枚举，定义了资源的大致优先级（高、中、低）。高优先级的项将在内存保存较长时间，低优先级将先被换出。m_nRefCount 函数表示资源第几次被上锁。稍后我们再来考察这个函数。m_LastAccess 函数是资源最近被访问的时间。

小于运算符（<）是用来决定废除资源的优先顺序的。缺省函数如下：

```
bool BaseResource::operator < (BaseResource& container)
{
    if(GetPriority() < container.GetPriority())
        return true;
    else if(GetPriority() > container.GetPriority())
        return false;
    else
    {
        if(m_LastAccess < container.GetLastAccess())
            return true;
        else if(m_LastAccess > container.GetLastAccess())
            return false;
        else
        {
            if(GetSize() < container.GetSize())
                return true;
            else
                return false;
        }
    }
    return false;
}
```

从这个函数可以看出，资源首先根据优先级，再根据访问时间，最后根据大小来排序。虽然这个算法相当简单，但它对很多情况都能很好地处理。如果需要一个不同的或者更复杂的算法，可以修改基类，也可以在派生类中使用新的排序运算。

1.7.2 资源管理类

管理资源问题的另一半是使用一个管理器，让它组织所有的存储资源，按需提供访问，并在内存预算范围内处理资源的动态配置与重分配。让我们来考察一下 ResManger 类，看看它是如何工作的。

```
class ResManager
{

public:

    ResManager()      { Clear(); }
    virtual ~ResManager() { Destroy(); }

    void Clear();

    bool Create(UINT nMaxSize);
    void Destroy();

    // ------------------------------------------------------------------------
    // 资源映射表迭代

    // 循环访问每个子项的函数
    // 跨越 DLL 边界直接访问映射表或迭代子将导致一个栈指针错误
    // 但通过这个包装来访问将是安全的
    inline void GotoBegin()
    { m_CurrentResource = m_ResourceMap.begin(); }
    inline BaseResource* GetCurrentResource()
    { return (*m_CurrentResource).second; }
    inline bool GotoNext()
    { m_CurrentResource++; return IsValid(); }
    inline bool IsValid()
    { return (m_CurrentResource != m_ResourceMap.end()) ? true : false; }


    // ------------------------------------------------------
    // 一般资源访问

    // 允许资源管理器在不超过最大允许内存的情况下预先保留一些内存
    // 以便在以后插入资源时可以直接使用
    bool ReserveMemory(size_t nMem);

    // 如果你提供资源句柄变量的地址，资源管理器将提供给你惟一句柄
    bool InsertResource(RHANDLE* rhUniqueID, BaseResource* pResource);
    bool InsertResource(RHANDLE rhUniqueID, BaseResource* pResource);

    // 从管理器中完全删除一个对象
    bool RemoveResource(BaseResource* pResource);
    bool RemoveResource(RHANDLE rhUniqueID);

    // 析构一个对象并释放其内存
    bool DestroyResource(BaseResource* pResource);
    bool DestroyResource(RHANDLE rhUniqueID);

    // 使用 GetResource 函数告诉管理器你希望访问对象
    // 如果资源已经被释放，它将在被返回之前重建
```

```
    BaseResource* GetResource(RHANDLE rhUniqueID);

    // 锁定资源以确保资源不会被资源管理器管理
    // 你可以使用这个函数来确保一个资源的实例不被交换出内存
    // 资源包括一个引用计数器来确保能安全的保存锁的数量
    BaseResource* Lock(RHANDLE rhUniqueID);

    // 解锁一个对象使资源管理器知道你不再需要一个排他性访问
    // 当所有的锁都被释放(引用计数为0),
    // 对象将被认为可以被安全管理并可以被交互出内存
    // 对象可以通过句柄或对象指针被引用
    int Unlock(RHANDLE rhUniqueID);
    int Unlock(BaseResource* pResource);

    // 基于资源指针获取存储句柄
    // 注意这里假设没有重复指针,并将返回第一个匹配资源
    RHANDLE FindResourceHandle(BaseResource* pResource);

protected:

    // 内部函数
    inline void AddMemory(UINT nMem)
    { m_nCurrentUsedMemory += nMem; }
    inline void RemoveMemory(UINT nMem)
    { m_nCurrentUsedMemory -= nMem; }
    UINT GetNextResHandle()
    { return --m_rhNextResHandle; }

    // 当你希望管理器检查已释放资源时需要调用此函数
    // 资源只有在达到最大可分配限制后才会被交互出去,
    // 并且将按照优先级从低到高丢弃资源,
    // 优先级由类的 < 运算符决定。
    // 如果请求内存不能被释放则函数将失败
    bool CheckForOverallocation();

protected:
    RHANDLE      m_rhNextResHandle;
    UINT         m_nCurrentUsedMemory;
    UINT         m_nMaximumMemory;
    ResMapItor   m_CurrentResource;
    ResMap       m_ResourceMap;
};
```

资源管理器的核心是数据成员 m_ResourceMap。这是个 STL 映射表，它表明每个惟一的资源句柄（即一个简单的无符号 int）对应一个指向资源对象的指针。

句柄可以被预先赋值（可以硬编码或从脚本文件读取）或被资源管理器自己动态赋值。需要注意的是，当前实现是一个非常简陋的实现，它只是简单的从最大句柄值开始并向下工作。如果你的用户定义 ID 从相对低的值开始，则结合这两种方法能获得较好效果。这个方法在你使用完空间之前能够给你几十亿可用句柄值。如果你计划使用很多资源，你需要实现

一个更完善的句柄分配策略。

一旦资源管理器对象调用 Create()函数并传入目标内存限制，管理器就可以被使用了。简单地调用 InsertResource()函数来插入资源到管理器。如果你传入句柄地址而非其值，函数为你填充这个句柄值。在例子中，我们创建一个工厂类，自动分配、创建和插入资源对象到管理器。

了解资源管理器的特性是非常重要的。当你指定内存目标，InsertResource()函数允许内存目标暂时超过当前资源总数。管理器将把资源换出内存，以使当前使用内存低于指定的限制。虽然这种方法可以在某些情况下被采用，例如你的资源从一个通用内存池中分配或者你工作在一个有真实虚拟内存的环境下；当它也可能造成问题，例如你工作在一个固定数量的特殊内存环境下，如音效和材质内存。

要求管理器保留你希望载入资源大小的内存可以解决这个问题。ReserveMemory()函数使用一个标准的 size_type 参数。函数在能够释放请求内存大小的时候返回真值。在函数成功返回之后，你就可以安全地调用 InsertResource()函数。一般情况下，ReserveMemory()函数会在资源类的 Create()函数中被调用，在载入一些资源头信息后，资源类能够知道载入完整资源需要多少内存。一旦内存被管理器保留，Create()函数可以完成数据载入工作并将资源插入到管理器中。要优化这个过程，你可以预先载入这个信息并将之保存到全局可访问表中。

1.7.3 句柄如何工作

这个系统通过使用句柄防止使用者直接操作对象，这允许管理器自由决定何时将资源换出内存。要访问资源，使用者必须调用成员函数并将句柄传入，以获得指向资源的句柄。下面是使用示例：

```
SomeResource* pRes = (SomeResource*)resmgr.GetResource(hResHandle);
if(!pRes)
    return Error;

// 现在资源可以被安全地使用，直到调用管理器其他的函数
```

需要牢记的是，资源指针只在下一次调用管理器其他的函数之前有效。访问其他资源可能导致资源管理器将你先前准备访问的资源换出内存。你一般还需要在资源类代码中加入断言，确保这些成员函数不会在资源已经被释放后还被调用。

如果你希望获得并保留一个资源指针，有一种机制能够做到：Lock()函数。锁定一个资源将递增对象的引用计数，这将防止资源管理器在调用 Unlock()函数解锁对象之前，将此资源交换出内存。需要牢记的是，最后必须解锁你锁定的对象，否则资源管理器假设此资源在程序结束后还不允许被释放，导致内存泄露的问题出现。因为资源管理器有全部资源的索引，当其析构函数被调用时，会自动释放所有相关资源。

1.7.4 可能的扩展和改进

使用资源管理器对高效管理大量资源是非常有益的。虽然在访问资源时会增加稍许困

难，但这个困难将被简单的自动内存管理的优点所抵消。

如果你的应用程序的完整数据集已经在资源管理器中被索引管理，则可以使用现有功能来实现一个预缓存系统。要载入一个被判定为可选预缓存的资源，需要使用 GetResource()函数访问资源并提高优先级。这种方法强制这些资源中任意的已交换出去的数据被重新载入，并且可以直接访问，又可通过提升的优先级防止此资源再次被交换出内存。对不再需要使用的资源，只需要简单地降低其优先级，则当需要更多内存处理其他数据时它会被自动放弃。

在这些扩展之外，使用者可能还希望构造更全面的报告功能。你可以建立一个反馈机制来报告超过平均情况的资源释放操作，并可以通过调节优先级来最小化此类问题发生的可能性。通过动态有效地设定优先级，使用者可以动态改进管理器的性能。

其他相关技术在本书的另一篇 Scott Bilas 的《通用的基于句柄的资源管理器》中有详细介绍。不仅仅是将管理器用作虚拟内存系统，这个资源管理器还关注于诸如使用模板和其他智能、类型安全句柄的技术。

1.7.5 结论

随着现代游戏中需要维护数据量的增加，处理海量数据的技术也在发展。建立一个有效并高效的资源管理器，可以减少程序员对内存限制和内存漏洞的担心，同时可以提供更强大的资源使用情况监视工具。

1.8 快速数据载入技巧

John Olsen

如何变得更快是游戏编程中永恒的挑战。无论何时，一旦你让某人凝视屏幕并等待，你就中断了信息流并有流失玩家的风险。但随着游戏关卡变得越来越大，你所需的载入时间也变得越来越长。本节的技巧可以减少你的游戏关卡载入时间。

1.8.1 预处理你的数据

在所有你能做的事情里面，最重要的是尽你所能去预处理关卡数据(level data)。这可以通过使用独立的工具程序，如用来编辑游戏数据的独立关卡编辑器，或者在游戏开发中自定义的数据，打包代码。我曾经针对同一游戏的不同部分，分别使用了这两种方法，都获得很好的结果。

为了最终获得更短的数据载入时间，需要将你的数据预处理为游戏中最终使用的格式。只需要做些计划，你就可以让你的C++类或C结构的布局更适合高速载入。所有需要被存储的数据都必须是非静态成员变量，并且不能将指针保存到数据文件中。

如果你的数据里面需要用到指针，则必须确保在它们被完全载入并正确设置以后才使用，因为指针在载入后通常都指向无效数据。另外一种解决方法是将指针改为句柄或索引。请参见Scott Bilas的文章《通用的基于句柄的资源管理器》。

因为C++使用虚函数表，你还必须确保不在类里面使用任何虚函数，否则在你用无效数据覆盖虚函数表后，虚函数会调用内存中任意位置的代码。如果你想真正安全，可以尝试将所有的访问函数变成静态，以保证它们不会被包含在你的数据里。

1.8.2 保存你的数据

无论是在游戏中还是在独立的预处理工具中，一旦数据被填充到结构中，你就可以将它们写到硬盘上。对C++来说，你可以使用this指针和sizeof()来处理类。对C来说，只需要结构指针和sizeof()来处理结构。需要确保你没有使用 sizeof(this)，否则获取的不是结构的大小而是指针的大小。这个大小是你的类的所有非静态成员数据和编译器自己为这个类加上的一些东西。

下面的例子代码演示如何在C++中对有成员函数的数据类进行载入和保存。

```
#include <stdio.h>

class GameData
{
public:
    bool Save(char *fileName);
    bool Load(char *fileName);
    bool BufferedLoad(char *fileName);
    // Add accessors to get to your game data.
private:
    // 一次只打开一个文件
    static FILE *fileDescriptor;
    // 游戏数据
    int data[1000]; // 使用你自己的数据格式替换这个字段
};

bool GameData::Save(char *fileName)
{
    fileDescriptor = fopen(fileName, "wb");
    if(fileDescriptor)
    {
        fwrite(this, sizeof(GameData), 1, fileDescriptor);
        fclose(fileDescriptor);
        // 报告写文件成功
        return TRUE;
    }
    else
    {
        // 报告写文件出错
        return FALSE;
    }
}
```

1.8.3 使用简单方法载入你的数据

使用上述方法保存数据，可使应用程序在以后需要载入关卡时，非常轻松地读回它们。只需要将数据读回到游戏中和写入它们时相同的结构或类中。

```
bool GameData::Load(char *fileName)
{
    // 以读模式打开文件
    fileDescriptor = fopen(fileName, "rb");
    if(fileDescriptor)
    {
        fread(this, sizeof(GameData), 1, fileDescriptor);
        fclose(fileDescriptor);
        // 报告读取文件成功
```

```
        return TRUE;
    }
    else
    {
        // 报告读取文件错误
        return FALSE;
    }
}
```

1.8.4 更安全地载入你的数据

对于某些游戏控制台硬件，至少有一件非常重要的事情需要注意。一些系统总是要读完当前磁盘扇区。例如，Sony PlayStation 从 CD-ROM 读取的数据一定是 2048 字节的倍数。也就是说，如果要直接读取数据到你的结构中，而且结构的长度不是 2048 字节的倍数，那你就破坏了紧接在那个结构之后的内存数据。我们称之为“memory stomp”。

为避免这种 memory stomp，需要有一个足够大的临时缓冲区来保存填充到 2K 边界的数据文件。如果你需要读多个文件，不要每次都分配和释放一个新的缓冲区。而是改用一个最大缓冲区，只分配一次，并且在所有的读取操作中重用它。在完成所有的读取操作后再释放它。我将在下面演示这个较为简单的读取方法。

如果你使用的系统只有很少量的内存，也许所有的内存都已有安排，同时动态内存也被一起禁止。在这种情况下，你需要事先在内存中某处的有一个缓冲区，这样在你需要读取文件时它不会已被使用。使用它作为临时缓冲区而避免使用动态内存分配，如下：

```
// 检查你的硬件中一次读取块的大小
// 将值放在这个定义中
#define READ_GRANULARITY 2048

bool GameData::BufferedLoad(char *fileName)
{
    // 确认读缓冲区有足够空间
    //如果使用 READ_GRANULARITY 的倍数，使用的空间会小一些；但下面的方法会更快一点
    char *tempBuffer = new char[sizeof(GameData) + READ_GRANULARITY];
    if(!tempBuffer)
    {
        // 不能分配缓冲区，返回错误代码
        return FALSE;
    }
    fileDescriptor = fopen(fileName, "rb");
    if(fileDescriptor)
    {
        fread(tempBuffer, sizeof(GameData), 1, fileDescriptor);
        fclose(fileDescriptor);
        memcpy(this, tempBuffer, sizeof(GameData));
        delete tempBuffer;
        // 报告读文件成功
        return TRUE;
```

```
        }
        else
        {
            delete tempBuffer;
            // 报告读文件错误
            return FALSE;
        }
    }
```

现在你已经掌握了高度优化的关卡载入。通过预处理数据，你将节省下将数据转换为可用格式的 CPU 时间，并且减少了需要读取数据的总量。既快又小才是最好的优化。

1.9 基于帧的内存分配

Steven Ranck

本文将介绍一个简单而且极快速的内存分配系统，以防止在游戏关卡之间出现内存碎片。它可以在游戏关卡载入时被广泛使用。此外这个系统在分配和释放内存时都非常快速，并可用于从游戏机到 PC 到街机的任何类型的平台上。

1.9.1 常规内存分配的挑战

包括 malloc() 和 new 在内的标准内存分配系统的最大问题在于，内存会变成碎片，从而导致游戏性能下降，并且可能导致无法分配一个可用的大内存块。但一个应用程序请求分配一块内存时，成熟的操作系统，如 UNIX 和 Microsoft Windows，会使用高级内存管理系统将物理内存块在逻辑上组合到一起，以提供请求的连续内存块。但是这个组合需要消耗很多本可用于游戏的 CPU 指令周期。而对于只有很小的库函数集的家用游戏机来说，并没有成熟的内存管理器可用。

1.9.2 介绍基于帧的内存

解决这些传统内存分配难题的一种办法是，采用基于帧的内存。基于帧的内存可以消除内存碎片的问题并且使用起来非常快速。但是它不能用来作为类似 malloc()和 new 的基于一般目的的内存分配系统。基于帧的内存最适合用于游戏和关卡初始化模块。

如图 1.9.1 所示，基于帧的内存以类似堆栈的方式工作。在初始化时，游戏从操作系统分配一个单独的内存块，并交由帧内存系统使用和管理。这个内存块在游戏的整个生命周期中只分配一次，直至游戏结束才被操作系统收回。在图 1.9.1 中，帧内存系统掌握着 Memory Block 所指的整块内存。通过 Memory Block 的指针，我们可以计算出基地址（Base）和顶地址（Cap）内存指针，并将之对齐到符合应用程序设计用于的特定系统的内存边界。基指针指向我们内存块中最低的已对齐内存地址，顶指针指向我们内存块中最高的已对齐内存地址的下一个地址(如图 1.9.1 中箭头位置所示)。内存块、基指针和顶指针在游戏的生命周期中保持不变。最后，低堆帧和高堆帧的指针被分配设置为等于基指针和顶指针。我们等会儿可以看到在游戏过程中这两个指针是如何随着内存的分配和释放而改变的。下面的代码将完成初始化帧内存系统的工作。

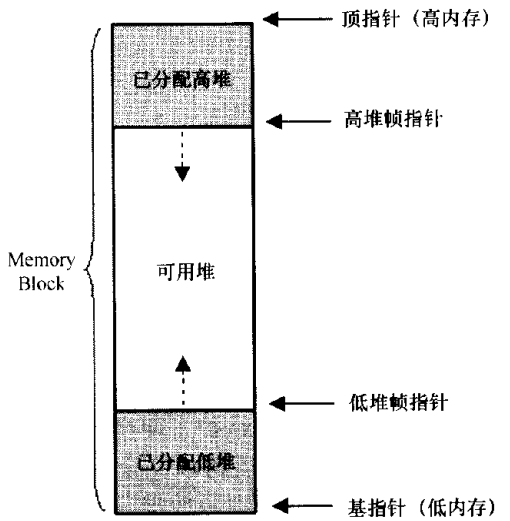


图 1.9.1

```
typedef unsigned char u8;
typedef unsigned int uint;

#define ALIGNUP( nAddress, nBytes ) ( (((uint)nAddress) + \
    (nBytes)-1) & (~((nBytes)-1)) )

static int _nByteAlignment;     // 内存对齐的字节数
static u8 *_pMemoryBlock;       // malloc()返回值
static u8 *_apBaseAndCap[2];    // [0]=基指针, [1]=顶指针
static u8 *_apFrame[2];         // [0]=低帧指针, [1]=高帧指针
// 必须在游戏初始化时调用一次且最多一次
// nByteAlignment 必须是 2 的幂
// 成功时返回 0，错误发生时返回 1
int InitFrameMemorySystem( int nSizeInBytes, int nByteAlignment ) {
    // 必须保证 nSizeInBytes 是 nByteAlignment 的整数倍:
    nSizeInBytes = ALIGNUP( nSizeInBytes, nByteAlignment );

    // 首先分配我们的内存块:
    pMemoryBlock = (u8 *)malloc( nSizeInBytes + nByteAlignment );
    if( _pMemoryBlock == 0 ) {
        // 没有足够内存，返回错误标志:
        return 1;
    }

    _nByteAlignment = nByteAlignment;

    // 设置基指针:
    _apBaseAndCap[0] = (u8 *)ALIGNUP( _pMemoryBlock, nByteAlignment );
    // 设置顶指针:
    _apBaseAndCap[1] = (u8 *)ALIGNUP( _pMemoryBlock + nSizeInBytes, nByteAlignment );

    // 最后，初始化低帧指针和高帧指针:
    _apFrame[0] = _apBaseAndCap[0];
    _apFrame[1] = _apBaseAndCap[1];
```

```
    // 成功!
    return 0;
}
```

停止帧内存系统：

```
void ShutdownFrameMemorySystem( void ) {
    free( _pMemoryBlock );
}
```

函数 InitFrameMemorySystem()在游戏初始化时被调用一次且最多一次，传入参数包括帧内存系统管理的最大内存数和字节对齐数。所有通过帧内存系统进行的分配操作都必须保证内存对齐。注意，ALIGNUP()宏需要 nBytes 参数值为 2 的幂。

这样的话，帧内存系统就可以使用了。它维护了两个独立的堆：分配时向上增长的低堆和分配时向下增长的高堆，如图 1.9.1 所示。这就可以完全由游戏决定如何使用每个堆。例如，可将高堆用于存储 3D 图形数据，而低堆用于声音数据。在这个例子中，图形和声音模块的内存分配是完全无关的，因此也不会造成内存碎片（因为这两个堆是物理上分离的）。

1.9.3 分配和释放内存

分配帧内存的工作类似堆栈操作。一个系统调用从一个或两个堆中请求一块内存。如果指定了低堆，低堆帧指针将根据分配空间大小而增长，改变前的值被返回。低堆帧指针总是指向下一个可分配的内存字节。另一方面，如果指定了高堆，高堆帧指针将根据分配空间大小而减小，改变后的新值被返回。这是因为高堆帧指针总是指向最后分配的内存字节。如果两个帧指针彼此交错，则没有足够内存满足需求。下面的函数将完成分配操作：

```
// 返回内存块的基指针，或者在没有足够内存时返回 0
// nHeapNum 是堆编号：0=低，1=高。
void *AllocFrameMemory( int nBytes, int nHeapNum ) {
    u8 *pMem;

    // 首先，将请求大小内存对齐：
    nBytes = ALIGNUP( nBytes, _nByteAlignment );

    // 检查可用内存：
    if( _apFrame[0]+nBytes > _apFrame[1] ) {
        // 没有足够内存：
        return 0;
    }
    // 现在完成内存分配操作：
    if( nHeapNum ) {
        // 从高堆向下分配：
        _apFrame[1] -= nBytes;
        pMem = _apFrame[1];

    } else {
        // 从低堆向上分配：
        pMem = _apFrame[0];
        _apFrame[0] += nBytes;
```

```
    }

    return (void *)pMem;
}
```

这个函数能非常快速地完成基于帧的内存分配。既然帧内存通过类似堆栈的方式分配，它也需要使用同样的方式释放。这就是帧被引入的原因。帧是游戏从内存系统获取用来释放内存的句柄。内存只能通过帧来释放。帧起到类似系统分配的内存页中书签的作用。当一个帧被释放后，在此帧被获取后分配的所有内存都将被释放。图 1.9.2 演示了帧的使用。

图 1.9.2 的(b)和(c)展示了两个使用 AllocFrameMemory()函数分配的独立内存。在(d)中，游戏从内存系统获取一个帧。这个帧只是一个游戏以后用来释放内存的句柄。在(e)和(f)中，游戏分配了另外两个内存块。在(g)中，游戏要释放所有在帧获取以后分配的内存。下面的函数从高堆或低堆获取一个帧。

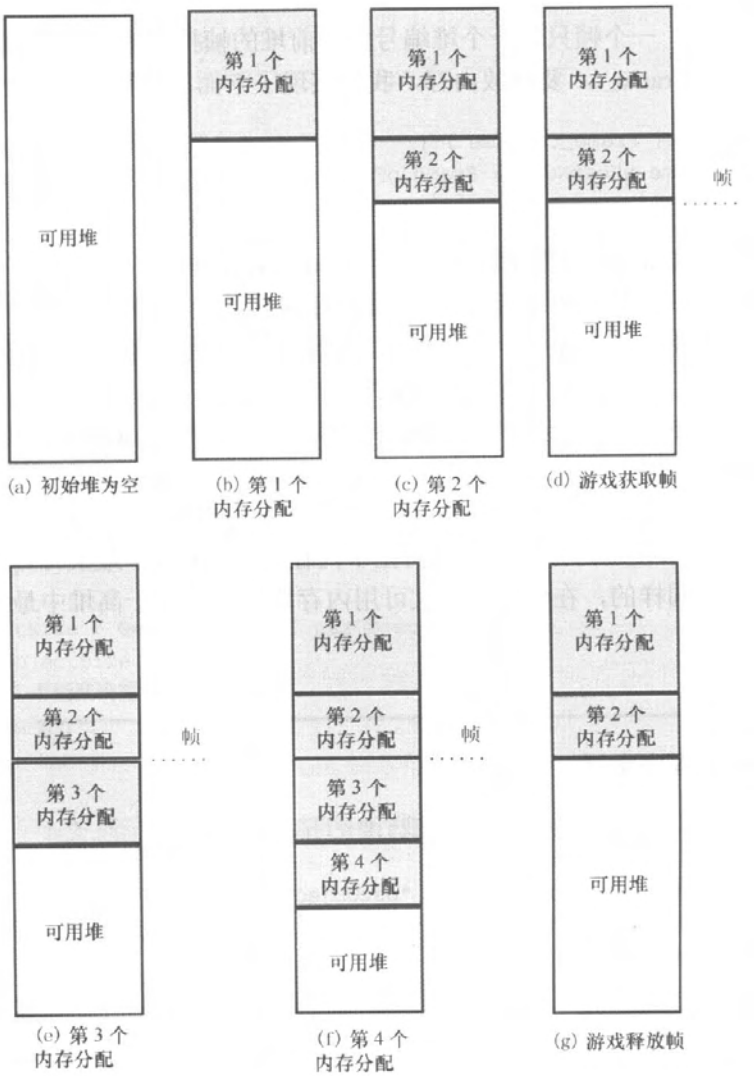


图 1.9.2 内存的分配和释放

```
typedef struct {
    u8 *pFrame;
    int nHeapNum;
} Frame_t;

// 返回一个可以在以后用于释放内存的帧句柄
// nHeapNum 是堆的编号：0=低，1=高.
Frame_t GetFrame( int nHeapNum ) {
    Frame_t Frame;

    Frame.pFrame = _apFrame[nHeapNum];
    Frame.nHeapNum = nHeapNum;

    return Frame;
}
```

对内存系统来说，一个帧只是一个堆编号和当前堆的帧指针的拷贝。但对于游戏，它只是一个简单的句柄，Frame_t。要释放内存，我们实现了下面的函数：

```
void ReleaseFrame( Frame_t Frame ) {
    _apFrame[Frame.nHeapNum] = Frame.pFrame;
}
```

游戏调用 ReleaseFrame()函数来释放从调用 GetFrame()函数获取帧以来分配的所有内存。倘若释放帧的顺序和获取帧的顺序相反，那么游戏同时可以分配多少个帧并没有限制。无论如何，内存系统并不要求帧被释放。举例来说，如果获取了帧 1、帧 2 和帧 3，释放帧 3 后直接释放帧 1，并且永远不释放帧 2 是有效的操作。

两个相互独立的堆有很多好处。我们前一个例子中，高堆被 3D 图形使用，而低堆被用于存储声音数据。假设在两个堆之间的分配操作按如下步骤执行：分配 3D 内存块，分配声音内存块，分配另外一个 3D 内存块，分配另外一个声音内存块。当 3D 内存被释放时（两块都被释放），不会出现未使用内存的空洞且防止了内存碎片的产生。最大可用内存块总是等于释放堆的最大大小。同样的，在低堆中最大可用内存块也总是等于高堆中最大可用内存。

1.9.4 例子

考虑下面的应用程序实例：

```
#define _HEAPNUM 1    // 随你的便，这里我们将使用高堆（1）。

extern int GetObjectSize( const char *pszObjectName );
extern int LoadFromDisk( const char *pszObjectName,
void *pLoadAddress );

// 我们的 CopCar 将被读到这里
static void *_pObject1;
// RobberCar 将被读到这里。
static void *_pObject2;
```

```
// 把 CopCar 和 RobberCar 对象从磁盘读到_HEAPNUM 中
// 如果成功传回 0，否则传回 1
int LoadCarObjects( void ) {
    Frame_t Frame;

    // 得到一个帧柄：
    Frame = GetFrame( _HEAPNUM );

    // 尝试读取 CopCar 对象：
    _pObject1 = LoadMyObject( "CopCar" );
    if( _pObject1 == 0 ) {
        // 不能读取对象。释放内存：
        ReleaseFrame( Frame );
        return 1;
    }

    //尝试读取 RobberCar 对象：
    _pObject2 = LoadMyObject( "RobberCar" );
    if( _pObject2 == 0 ) {
        //不能读取对象。释放内存：
        ReleaseFrame( Frame );
        return 1;
    }

    // 对象读取成功。保持已分配的内存：
    return 0;
}

// 从_HEAPNUM 分配内存，同时把指定的对象从磁盘读到已分配的内存中。
// 如果成功，传回对象指针，否则传回 0。
void *LoadMyObject( const char *pszObjectName ) {
    int nObjectSize;
    void *pObject;

    nObjectSize = GetObjectSize( pszObjectName );
    if( nObjectSize == 0 ) {
        // 不能准确得到对象大小：
        return 0;
    }

    pObject = AllocFrameMemory( nObjectSize, _HEAPNUM );
    if( pObject == 0 ) {
        // 内存不足：
        return 0;
    }

    if( LoadFromDisk( pszObjectName, pObject ) ) {
        // 不能从磁盘上读取对象：
        return 0;
    }
```

```
    // 对象读取成功:
    return pObject;
}
```

在前面的例子中，函数 LoadCarObjects()获取了一个帧，但只在载入对象出现问题时释放它。如果两个对象都被正常载入，帧不会被释放并且函数返回正常的内存。高级函数可以获取它自己的帧以封装包括 LoadCarObjects()以内的所有对象载入函数。当需要释放所有对象内存时，高层函数只需要简单地使用其获取的帧调用 ReleaseFrames()函数，即可完成所有内存释放操作。

1.9.5 结论

既然基于帧的内存以类似堆栈的方式工作，它就需要帧被以与获取顺序相反的顺序释放；否则内存可能被破坏。检测释放违反这个条件的方法很简单。可以使用以下函数替换释放帧的函数：

```
void ReleaseFrame( Frame_t Frame ) {
    // 检测低堆中释放操作的有效性:
    assert( Frame.nHeapNum==1 || (uint)Frame.pFrame<=(uint)_apFrame[0] );

    // 检测高堆中释放操作的有效性:
    assert( Frame.nHeapNum==0 || (uint)Frame.pFrame>=(uint)_apFrame[1] );

    // 释放帧:
    _apFrame[Frame.nHeapNum] = Frame.pFrame;
}
```

这段代码可以在游戏的调试版本中检测出以不正确的顺序释放帧的企图。还可以加入更多的断言以检测其他参数的有效性问题。

最后需要说的是，对于使用多种无关的内存类型（主内、声音、贴图、图形等）的游戏平台，对于每一种内存类型，我们可以很容易地用一个基于帧的内存系统来实现，然后使用主帧将他们联系到一起。回想一下前面的例子。假设 LoadFromDisk()为特定模型载入图形、贴图和声音，图形被放在系统内存，贴图放入贴图内存，而声音放入声音内存。在这种情况下，需要有三个独立帧内存系统通过主帧联系到一起：

```
typedef struct {
    Frame_t SysmemFrame;        // 系统内存帧
    Frame_t TexmemFrame;        // 贴图内存帧
    Frame_t SoundmemFrame;      // 声音内存帧
} MasterFrame_t;
```

1.10 简单快速的位数组

Andrew Kirmse

我们喜欢位操作，因为它们速度快，且能够高效地组织数据；但我们也讨厌位操作，因为它们容易出错且跟机器字长相关。我们真正需要的是一种抽象的位操作，能够带给我们位操作的优点，但隐藏繁琐的底层细节。

1.10.1 概述

本文通过三个C++类实现位数组。基类BitArray是一个简单的一维位数组。它的子类BitArray2D是一个二维位数组，而TwoBitArray是一个值为0~3的整数的数组。所有的类都可以通过类似英语的语法或者通过常用的操作符进行维护。这些类具有清楚的语法、可移植性、范围检测，正确地使用了const并具有很高的性能。

C++标准模板库（STL）在位集头文件中包含一个一维位数组。尽管具有丰富的特性，但大多数实现被隐藏，并且难以扩充或修改（实现示例可参考[SGI98]）。一些STL实现也依赖于部分没有被某些编译器实现的C++标准，如成员模板和名字空间。本文中的实现非常直观，易于与现有代码集成，而且还提供了在游戏开发中较为实用的附加特性。

1.10.2 位数组

基类BitArray使用起来跟普通C++中的bool类型数组很相似，但当然你也可以将位看作值为0或1的整数。这些位存储在字节序无关（endian-independent）的long类型缓冲区中。语法上你可以将BitArray以与标准C++数组相同的方式使用，但它还同时具有动态数组的优点和附加的操作符。例如，你可以像如下代码一样操作：

```
BitArray bits(num_bits), other_bits(num_bits);
bits.Clear();
bits[10] = true;
if ((bits & other_bits).AllBitsFalse()) {}
```

该类实现了标准位操作运算符 &、|、^、&=、|=、^=和~。但没有实现移位运算符。

为保障高性能，BitArray在创建时不做任何初始化工作。Clear()方法

设置所有位为 false。作为优化特性之一，小的 BitArray 可以存储在一个机器字中而无需分配额外的内存。使得这个类同样适用于很小的标志（flag）集合。语法：

```
flags[FLAG_INDEX] = true;
```

比传统的语法更加清晰

```
flags |= 1 << FLAG_INDEX;
```

BitArray 和其他类在数组索引越界时调用 assert。在实时游戏中这样比 STL 抛出异常更适合，因为抛出异常需要很大的负载。

BitArray 类 C++中通过代理类（Proxy Class）模式（参见[Meyers95]了解关于代理类模式的更多信息）实现数组下标运算符。代理类 BitProxy 表示 BitArray 中一个单独的位。表达式：

```
a[10] = true;
```

我们使用=运算符向 BitProxy a[10]赋值，而表达式：

```
bool val = a[10];
```

我们通过 bool 运算符只读取一位的值。BitProxy 类允许我们延迟计算表达式 a[10]，直到我们使用它时（读取或写入）。这是一个在其他位数组类中也常被使用的非常便捷的技巧。

注意 BitProxy 对象必须以值形式返回，因此它可能会导致实例化和删除临时对象的代价。在一般情况下，位的值会被立即读取或赋值，编译器可以优化掉这个临时对象。

1.10.3 其他数组

BitArray2D 类似一个二维的 BitArray。它的大多数行为跟其他的二维数组一样。

```
BitArray2D bits(10, 20);
bits[5][4] = true;
```

注意 BitArray2D 数组下标运算符返回一个 ArrayProxy，这是一个代理类，表示数组中单独的一行。这个机制被用来模拟 C++中双重下标的语法。

BitArray2D 不将其下标元素视为一维的 BitArrays 数组。

```
bits2d[5] = bit_array;     // 非法!
bits2d[5].FlipAllBits();   // 非法!
```

虽然的确可以允许这样的操作，但这会严重增加 ArrayProxy 类的复杂度，而且这样使用很少见。

BitArray2D 实现上就是一个包含二维数组所需位数的独立 BitArray 类。但因为 BitArray2D 不提供和 BitArray 相同的公共接口，所以它私有集成 BitArray 类。类似 BitArray，其所有函数都短小并内联（inlined）。

最后一个类，TwoBitArray，提供一个二位值数组。（可以很容易扩展该类，让每个元素包含更多位的数组。）它的实现也是简单地使用包含两倍于 TwoBitArray 元素数量的 BitArray。

尽管非常简单，这个类提供了一个将状态信息压缩到最小空间的简单途径。

1.10.4 应用

即便是实现像位数组这样的传统的底层 C 结构，也可以受益于 C++。为使用 C++代码获取易读性和易理解性而带来的性能代价通常是可以忽略的。对游戏来说，这意味着很多程序现在只需耗费更短的开发时间和更少的代码调试工作。

如果你在游戏的主要数据结构中使用这些类（例如在二维贴图（tile）数组里的每个元素上附加状态位），可以增加实用的运算符用于从流中读写数组。如果你这样做，而且移植性很重要的话，请注意在处理数组内容时的字节序问题。

示例代码包括一个测试程序举例说明了这些位数组类的使用方法。

1.10.5 参考文献

[SGI98] SGI, "STL header bit set," www.sgi.com/Technology/STL/bitset, 1998.

[Meyers95] Meyers, Scott, *More Effective C++*, Addison-Wesley Longman, Inc., 1995.

1.11 在线游戏的网络协议

Andrew Kirmse

大多数加密方案假定一个可信的发送者和一个可信的接收者要通过一个不可信的通道进行通信。假设发送者会故意去欺骗接收者听起来似乎很可笑，但这恰恰是在线游戏设计者所要面对的问题。一些玩家是不能信任的，或者更糟，他们完全可以访问加密算法和通过客户端程序进行的所有通信。在这种情况下，我们不能期望提供完全安全的通信，但我们可以让攻击者的麻烦大于其获得。本文提供了一些为在线游戏构造应用层通信协议的实用技术。

1.11.1 定义

协议设计主要针对于有一个或多个不可信客户端与一个可信中央服务器的客户/服务器类型在线游戏。（端对端游戏中的作弊的确也是一个问题，但因为在这种游戏中没有一个可信实体（entry），所以这种情况下完全防止作弊是不可能的。）对客户/服务器模式的游戏来说作弊的结果很严重，因为作为惟一可信实体的服务器维护着游戏状态并验证所有客户端命令。如果游戏状态被长久地保持住，一个成功的欺骗行为会使数以千计的玩家受到影响。

我们在客户/服务器类型系统下考虑协议安全特性。客户端和服务器通过网络通道发送可靠（通常是TCP）或不可靠（UDP）的报文来进行通信。虽然客户端也可以与其他客户端直接进行通信（通常用于聊天或语音），我们假设只有在客户端和服务器之间的通信数据才需要得到保护。

每个报文包括两个部分：包含管理信息的包头（header），和包含我们要进行通信的实际数据的有效负载（payload）。网络协议的目标是将发送者的原始有效负载发送到接收者。任何对发送者的有效负载序列进行的修改都应该被检测到。我们只处理有效负载的发送，将报文的顺序和可靠性问题交给低层协议栈去解决。

1.11.2 篡改报文

大多数针对协议的黑客攻击都是偶发的：他们尝试更改报文的字节看看会发生什么。针对此类攻击的第一线防御是一个简单的校验和（checksum）。校验和是通过组合报文中每个字节得到的一个短数字。发送

者计算报文的校验和并且将之与报文一起发送给接收者。接收者根据收到的报文重新计算校验和；如果计算得到的校验和与发送者的校验和不匹配，则报文被破坏并且应该被丢弃。校验和的计算范围必须包含包括包头在内的整个报文，以使接收者可以像检测有效负荷一样检测包头的有效性。

一个完美的校验和算法能对任意修改过的报文计算出不同的值。当然如果这个完美的校验和算法太长则会变得根本不实用。哈希（hash）函数具有相同的设计目标并可以构造极好的校验和。特别是实用的单向哈希函数，它可以将输入不规则地映射到一个很宽的范围，以致通过哈希值来重新构造输入的任意一个部分在实际中不可能办得到。MD5 算法就是一个经过广泛测试，可以公开使用的单向哈希函数，并且对于游戏它的速度也足够快。公开领域的实现可以在网上找到[Plumb93]。

这种简单的校验和机制有两个弱点。首先，因为客户端程序包含校验和计算代码，攻击者可通过逆向工程获取校验和算法，然后对任何消息计算有效校验和。其次，攻击者可以捕获有效包并在稍后重发，这种攻击被称为报文重放（packet replay）攻击。

1.11.3 报文重放

在报文重放攻击中，恶意用户从客户端捕获报文（通常通过报文监听）然后多次发送。通常使用报文重放以超过游戏允许的速度来执行命令，即使客户端有时间检测。例如客户端可以使用一个计时器每秒向服务器发送一个特定命令，不管玩家以何等频率执行此命令。使用报文重放，一个单独用户可以每秒发送相同的命令几百次。

系统设计者可以通过在服务器端也设置一个类似的每秒一次的计时器来阻止这种攻击。但因为可变的网络延迟，这种防范措施实际上是不可行的。虽然它能检测大部分报文重放攻击，但变化的网络延迟可能导致报文同时抵达服务器端，导致合法的命令序列被拒绝。我们当然不希望我们的安全机制将合法玩家当成欺骗者。

要预防报文重放，每个报文需要包含一些状态信息，因此即使相同的有效负荷也要有不同的位模式（bit pattern）。一个随着每个报文发送而累加的计数器之类的方法就可以做到，尽管这种策略使攻击者能够很容易的预期。一个较好的方法是使用一个状态机为连续的报文生成连续的识别号。一个快速并且足够复杂的计算方法是常用于系统库中的线性叠加随机数生成器。这样的生成器以下列方式操作：

```
State = (State + a) * b;
```

这里的 a 和 b 是仔细挑选出来的整数（对于这种生成器的讨论可参考[Knuth98]）

发送者和接收者都为他们的连接保持一个线性叠加随机数生成器。当发送一个报文时，发送者生产一个随机数并将之添加到报文中，同时步进随机数生成器。接收者使用自己的生成器检查收到报文中的随机数，如果数字不匹配则表示报文已经被篡改；如果数字匹配，则接收者也步进随机数生成器以准备接受下一个报文。

这个策略有两个复杂之处。首先是发送者和接收者如何初始化并同步他们的状态机。他们可以使用相同的固定种子启动各自的状态机，但这样一来初始报文流的位模式总是一样，因而会成为可被分析的漏洞。替代办法是，由服务器使用随机生产的种子值初始化其状态机，

并在其第一个消息中将之发送给发送者。

第二个复杂之处是如何在通信中保持状态机的同步。在一个可信连接中，包永远不会丢失，因此同步是可以有保障的。当报文被丢弃或重新排序，无论如何，情况将变得更加复杂。如果消息被丢失，发送者的状态机将比接收者的状态机多步进一次；后来的报文即使合法也将都被拒绝。一个简单的解决方案是使用一个在每个报文中发送的真实序列号（大多数游戏总是包含这个序列号，以通过不可信传输层协议提供可信连接）。通过这个序列号，接收者可以决定需要步进它的状态机多少次以适合当前报文。如果应用程序允许无序发送，较老的状态机状态必须被保存以便在被打乱次序的报文抵达后使用。

大多数运行时库中的 rand 函数因为精度不够而不适合被用于状态机实现（大多数实现都只有 15 位），但通常可以被作为随机数来源。一个快速、高精度的随机数实现可参见[Booth97]。

1.11.4 其他技术

理想情况下，为了阻扰对有效负载的分析，两个具有同样有效负载的报文在其位模式上应该有尽可能少的相关性。一个简单的消除两个集合之间相关性的方法是将其数据与一系列随机位进行异或（XOR）操作。假设在前面描述的报文重发预防中，发送者和接收者已经同步了随机数生成器。因此发送者可以为每个报文生成一个随机数序列，并将之与报文有效负载进行异或操作；接收者生成相同的数字序列并以相同的方式获取原始数据。

事实上两个具有相同长度的报文可能给攻击者一个报文编码相似数据的线索。要进一步干扰攻击者，每个报文可以包含一些可变长度的随机垃圾数据，其仅用来改变报文长度。垃圾数据长度也由另外一个同步状态机决定。发送者检测其状态机决定需要生成并插入多少字节的随机数作为垃圾数据到发送的报文中。接收者只需要忽略垃圾数据。增加垃圾数据总量可以进一步隐藏有效负荷，但需要消耗额外的带宽。一般来说应用程序的带宽都是有限的，因此垃圾数据的平均长度应该大大小于有效负荷的平均长度。

1.11.5 逆向工程

客户端包括完整的加密算法，总是可以进行逆向工程；这是最难解决的问题，也是任何阻止协议篡改的机制的根本弱点。你可以采用以下的一些步骤增大逆向工程的难度：

- 当公开发布时删除所有代码中的符号和调试信息。
- 不要将缓冲区加密和解密放在一个独立的函数中；而要将之与其他网络代码合并到一起。这是一个值得以可维护性换取安全性的地方。
- 运行时计算“魔术数”（例如初始化随机数种子），而不是将其值直接保存在可执行文件中。
- 在每个版本的客户端中包含一个好的加密机制，甚至包括早期测试版。如果任意一个客户端版本缺乏加密，用户就可以记录从该客户端发出的未加密报文流，并可以使用其知识攻击后续版本的加密机制。
- 牢记你的目标是让作弊的成本最大化，而非完全禁止作弊。

1.11.6 实现

本文所提供的实现代码包括一个使用到上述所有技术的 C++类 SecureTransport。SecureTransport 对象封装了在发送者和接收者之间的一个双向连接。对每个方向，对象维护 4 个线性叠加随机数生成器作为协议状态机。它们被初始化为静态值，服务器在其第一个报文中将随机数种子发送给客户端。该类将通过以下机制使用状态机：

（1）异或报文头部的长度字段。（如果底层协议如 UDP 提供了报文长度则不需要。）

（2）一个防止报文重放攻击的消息序列号。

（3）决定每个报文中垃圾数据的长度。

（4）生成对有效负载进行异或操作的随机位。

一个单独的随机数生成器生成实际的垃圾数据。在调试期间，则可以将垃圾数据设置为固定常数以便于调试。

1.11.7 参考文献

[Booth97] Booth, Rick, *Inner Loops*, Addison-Wesley Developers Press, 1997.

[Knuth98] Knuth, Donald, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, third edition,Addison-Wesley Longman, Inc., 1998.

[Plumb93] Plumb, Colin, "md5.c,"http://src.openresources.com/debian/src/admin/HTML/S/rpm_2.4.12.orig%20rpm-2.4.12%20lib%20md5.c.html, 1993.

1.12 最大限度地利用 Assert

Steve Rabin

几乎所有人都使用过断言 assert，但不是所有人都能做到最大限度地使用它。本文包括 7 个能够最大限度“榨取”assert 功能的技巧。如果你已经熟悉 assert，可以直接从“Assert 技巧#1”开始阅读。

1.12.1 Assert 基础

每个程序员都应该“虔诚”地使用 assert 宏。Assert 宏是一个简单的，无需额外代价的、针对你的假设进行双重检测的工具，它时时刻刻都在保护着你。通过给 assert 宏一个条件进行计算，你就断言这个条件应该为真（TRUE）。如果条件为假（FALSE），assert 将弹出一个对话框告诉你发生了什么问题。你可以选择忽略这个断言并继续执行你的代码、中断程序，或者直接中断到断言失败的代码。

Assert 宏使你的程序有所防备。如果你知道一个指针必须为空，你就断言它为空。如果能养成在代码中使用断言的习惯，你将可以在错误给你带来麻烦之前捕获它们。

向量规一化是一个非常需要使用断言的例子。在下列函数中，三个假设必须都满足才能让程序执行无误。src 和 dst 向量指针必须都有效，并且 src 向量的长度必须不为零。

```
void VectorNormalize( Vec* src, Vec* dst )
{
   float length;

   assert( src != 0 );   // 检查 src 向量必须不为空 (NULL)
   assert( dst != 0 );   // 检查 dst 向量必须不为空 (NULL)

   length = sqrt( (src->x*src->x) + (src->y*src->y) +
(src->z*src->z) );

   assert( length != 0 );   // 检查长度不能为零 (避免被零除异常)

   dst->x = src->x / length;
   dst->y = src->y / length;
   dst->z = src->z / length;
}
```

因为 VectorNormalize 函数需要尽可能地快，我们不能在发布版本中浪费时间去检测假设是否成立。可是当游戏在开发期间，我们又需要知道所有可能发生的问题。这就是断言能够大有作为的地方了。asssert 宏不会被编译到最终发布版本中，因此假设可以在开发期间被测试并在编译发布版本时自动被删除。这种机制允许你在代码中大范围使用断言，而无需担心在游戏发布前需要将其删除。

既然 assert 宏不会被编译到发布版本中，在断言中不会修改你的程序状态就至关重要。作为一个法则，不要在断言中调用函数或修改任何变量；否则你的调试和发行版本将有不同的行为，导致难以预料的严重问题。

为什么不干脆让 VectorNormalize 函数直接拥有错误检测功能呢？因为这个函数太底层，所以它不能提供如何补救问题的线索。较好的方法是在所有调用 VectorNormalize 函数的代码片断中检测这些假设，因为这样可以直接处理问题。如果一个代码片断检测这些假设失败了，assert 将在 VectorNormalize 函数中被引发，程序员就能知道如何修复导致问题的实际代码了。

1.12.2 Assert 技巧 #1：嵌入更多信息

传统 assert 宏的一个缺点是无法告诉你较多信息。如果 assert(src != 0)失败了，它只会在断言对话框中提示字符串“src != 0”。不幸的是，这不能给你足够信息来处理问题。除非你在调试器中运行游戏，否则并不会清楚问题发生在哪儿。有一项技术是，嵌入更多信息到你的条件中。思考一下下面的 VectorNormalize 函数：

```
void VectorNormalize( Vec* src, Vec* dst )
{
   float length;

   assert( src != 0 && "VectorNormalize: src vector pointer is NULL" );
   assert( dst != 0 && "VectorNormalize: dst vector pointer is NULL" );

   length = sqrt( (src->x*src->x) + (src->y*src->y) + (src->z*src->z) );

   assert( length != 0 && "VectorNormalize: src vector is zero length" );

   dst->x = src->x / length;
   dst->y = src->y / length;
   dst->z = src->z / length;
}
```

当这段代码中第一个断言失败时，断言对话框将显示字符串“src != 0 && "VectorNormalize: src vector is zero length"”。即使是你的测试者正在运行游戏，它们也可以告诉你断言失败的函数名，以及相关原因。

1.12.3 Assert 技巧 #2：嵌入更多更多信息

有时程序员会在程序运行到一个无法预料的位置时简单地输入 assert(0)。你可以使用相同

的技巧，通过简单的取反字符串导致失败来插入描述字符串。例如：

```
assert( !"VectorNormalize: The code should never get here" );
```

这一行完成的作用与 assert(0)相同，并可以给你更多的调试信息。

1.12.4 Assert 技巧 #3：使之更好用一些

前面两个技巧可以通过写一个简单的宏合并到一起。这个宏接受两个参数，第一个参数是需要计算的条件；第二个则是描述字符串。它模拟了前两个技巧，但便于输入和阅读：

```
#define Assert(a,b)   assert( a && b )
```

下面两行使用了新的宏：

```
assert( src != 0, "VectorNormalize: src vector pointer is NULL" );
assert( 0, "VectorNormalize: The code should never get here" );
```

1.12.5 Assert 技巧 #4：编写自己的 assert 宏

最终，所有人都会使用被真正定制后的 assert 宏。通过编写自己的断言对话框代码可以获得更多的控制并可新增特性。

标准 C 的断言有一个非常让人讨厌的问题：它会在调试器里中断代码到 assert.c 文件，而不是你程序中断言出现的行。通过编写自己的 assert 宏，调试器可以直接中断到输入断言的行。这就避免了为找到你实际感兴趣的代码而所做的毫无意义的堆栈跟踪。下面是一个自定义 assert 宏的例子：

```
#if defined( _DEBUG )
extern bool CustomAssertFunction( bool, char*, int, char* );

#define Assert( exp, description ) \
  if( CustomAssertFunction( (int)(exp), description, __LINE__, __FILE__ ) ) \
  { _asm { int 3 } } //this will cause the debugger to break here on PC's

#else
#define Assert( exp, description )
#endif
```

上面的宏调用 CustomAssertFunction 函数（需要由你自己编写）。CustomAssertFunction 函数应该弹出一个显示断言信息的对话框，允许使用者继续或中断程序运行。如果使用者选择中断，CustomAssertFunction 函数应该返回 TRUE，调试器则中断到断言所在行（这就是 int 3 指令在 PC 上所做的）。否则，函数应该返回 FALSE，程序将继续运行。

1.12.6 Assert 技巧 #5：无价之宝

一旦有一个自定义的 assert 宏，你就可以为你的断言对话框增加一个“总是忽略”选项。这是一个令人惊异的特性，它允许你在忽略一个断言后从此再不提示此断言。这在某个断言对每一帧都会失败的情况下特别有用，它使得你能够在继续运行程序时避免上百万次地点击断言对话框。要实现这个特性，每个断言都必须跟踪自己是否被忽略的状态，并禁止自己在此后被激发。

实现“总是忽略”的方法是在 assert 宏中使用一个静态布尔变量。这个布尔变量保存此断言是否被忽略。初始时此布尔值设置为 FALSE。当代码执行，它在计算断言条件前检测这个布尔值。如果布尔值为 FALSE，则将执行此布尔值的指针作为参数来调用 CustomAssertFunction 函数。如果断言条件失败并且使用者在断言对话框中选择“总是忽略”，它就设置此布尔值为 TRUE。下列代码实现了这个宏：

```
#if defined( _DEBUG )
extern bool CustomAssertFunction( bool, char*, int, char*, bool* );

#define Assert( exp, description ) \
  {  static bool ignoreAlways = false; \
    if( !ignoreAlways ) { \
      if( CustomAssertFunction( (int)(exp), description, \
                                __LINE__, __FILE__, &ignoreAlways ) ) \
      { _asm { int 3 } } \
    } \
  }

#else
#define Assert( exp, description )
#endif
```

1.12.7 Assert 技巧 #6：给“超级铁杆”

关于在 VectorNormalize 函数中清晰显示的断言，有一个罗嗦的问题。问题在于，错误的来源并不在 VectorNormalize 函数内。真正的错误只会在调用 VectorNormalize 函数的函数中发生，这将范围缩小到只剩几百个过程！如果断言失败时没有运行调试器，断言实际上并没有起任何作用。令人惊讶的是，这种情况非常普遍，因为测试者几乎都不会通过调试器来运行游戏。

一个简单的解决方法是在断言对话框中提供堆栈信息！*Microsoft System Journal* 杂志的专栏作家 John Robbins 铸造了 superassert 这个词来形容这个方法。在他的专栏“BugSlayer”里，他提供了一个 Windows 平台的例子，包括完整的源程序，可以在后面的参考文献[Robbins99]中找到。

1.12.8 Assert 技巧 #7：让它更简单——复制和粘贴

如果一个简单的技巧就能简化大量要做的工作，那实在是太酷了！本技巧就可以。如果断言提供了大量重要调试信息，为什么不直接让它简单易懂，以使测试者就能正确转告呢？

在 Windows 环境，你可以在断言对话框中提供一个按钮完成将信息复制到剪贴板的工作。只需要轻松地点击几下鼠标，任何人都能轻松复制和粘贴 assert 到电子邮件或错误报告中。这个简单而功能强大的办法能帮助你的测试者将精确、有意义的信息传达给程序员。

下面的代码复制任意字符串到剪贴板中。你可简单地修改它，并将之加入到你的 CustomAssertFunction 函数中去。

```
if( OpenClipboard( NULL ) )
{
   HGLOBAL hMem;
   char szAssert[256];
   char *pMem;

   sprintf( szAssert, "Put assert info here" );
   hMem = GlobalAlloc( GHND|GMEM_DDESHARE, strlen( szAssert )+1 );

   if( hMem ) {
      pMem = (char*)GlobalLock( hMem );
      strcpy( pMem, szAssert );
      GlobalUnlock( hMem );
      EmptyClipboard();
      SetClipboardData( CF_TEXT, hMem );
   }

   CloseClipboard();
}
```

1.12.9 参考文献

[McConnell93] McConnell, Steve, *Code Complete*, Microsoft Press, 1993.

[Robbins00] Robbins, John, *Debugging Applications*, Microsoft Press, 2000.

[Robbins99] Robbins, John, *Microsoft Systems Journal*: BugSlayer, www.microsoft.com/msj/defaulttop.asp? page=/msj/0299/bugslayer/bugslayer0299top.htm (code available at www.microsoft.com/msj/0299/code/Feb99BugSlayer.zip), February 1999.

[Saltzman99] Saltzman, Marc. *Game Design: Secrets of the Sages*, Brady Publishing, 1999.

1.13 Stats：实时统计和游戏内调试

John Olsen

每个人实际调试的时间都超过了自己当初的预期时间。通过开发一个实时调试和数据编辑系统（Stats）可以缩减大量时间。这个系统可以使开发人员通过在运行于目标平台的实时系统中更简单地进行调试和数据追踪来提升工作效率。这一技术已经被应用于面向 PC 和游戏机的商业产品。系统的名字是从统计（statistics）延伸来的，因为这个想法的最初目的是在游戏中单独显示供调试使用的数字统计信息。

文中描述的工具集非常易于实现，具有很强的可扩展性，并可针对游戏软件的设计和测试阶段的多个方面，具有很强实用性。也就是说你可以很容易地为你的需求定制此系统，并能够在以后的项目中适应变化且沿用下去。

1.13.1 Why：需求驱动的技术

PC 和游戏机系统在调试全屏实时程序时都存在很多问题。在 PC 上，你必须借助基于网络的远程调试、多屏幕系统，或者在调试会话期间在窗口模式和全屏模式之间切换。有些时候调试环境在系统运行时不允许访问部分或所有数据，而且在实时循环中错误的断点还可能导致系统锁死。例如在视频游戏机（此后简称游戏机）上没有键盘，因此一些类似下拉窗口被用来实现命令行界面，这通常用于第一人称视角射击游戏，是一种非常实用的调试工具。我们尝试修复调试环境的缺陷，但如果能够减少编辑/编译/运行的例行工作将更好。

每位在实时环境下工作的程序员肯定都碰到过使用普通调试方法几乎永远无法发现的 bug。有时代码在你单步跟踪时运行结果不同。有时你需要调试一个实时网络游戏来重现一个问题；或者更麻烦，需要同时调试一个网络游戏的两端。

游戏机在处理有限的存储路径时有额外的困难。在一些游戏机开发环境中，可以访问开发 PC 的文件系统来读写文件，但一旦你使用 CD 或游戏卡，原本用于 PC 上的数据路径就非常受限了。惟一的选择是将数据存储到记忆卡上，并在开发系统需要访问 PC 文件系统时读取记忆卡。

游戏机的游戏系统，以及通用的嵌入式系统，有一些 PC 不会碰到的特殊问题。游戏机系统的调试器已经越来越好，但仍有很多需要改进的地方。此外，不是所以的调试器都可以在你将游戏刻录到 CD 后继续工作。

从某种角度来说，基本上每个项目都需要一个帧率（frame-rate）计数器在屏幕上显示一些实时系统的数据。其他的一些常用项目，如多边形数、选择效率、基于视角内容的一般执行时间等，难以从调试器中获取，但易于在实时系统中追踪，这使得你需要构造一个能够显示任意数字列表的系统。这正是我要做的事情，最终我也做了一些关于这方面的工作。

在游戏内显示的思想还可以扩展到数据编辑，这在很多方面都能带来好处。首先，调试器通常不能读写数据文件。其次，有了游戏内编辑器，无论是基于键盘还是基于控制器，都能在游戏运行时编辑实际的游戏数据。再次，编辑的数据可以载入或存储在 PC 机或游戏机开发系统上。

1.13.2 How：一个进化过程

本章中包括的代码经过了多年的发展，从一个简单的不可编辑的显示数字列表到一个包含文本标签和只读数字的列表，最终到当前版本，支持可在游戏运行时进行编辑的多页数据。这个基于文本的显示只是一个简单的显示层，浮在游戏显示层的上方。

对很多项目来说，通过可编辑版本的 Stats 比编写特殊调试代码更简单。Stats 可以用来开关局部特性，并且可以将一组相关特性显示在一页中。通过这些设置，你可以在系统仍处于运行时，获取一些关于系统行为的详细数据。

在游戏机上因为调试较困难，可以通过一些不依赖于游戏机生产厂商提供的工具集的附加方法，获得部分收益。

1.13.3 What：一个基于 C++类的系统

所有类及其成员函数的完整代码可以在随书附带的光盘中找到。在继续下面内容的同时你可能需要用到它们。

使用 Stat 基类和继承类型为每个显示数据类型，完成类似 Stats 系统的实现只需要不到一周时间来进行设计。系统包括一个基本的作为所有东西的容器的 Stat 类，和一些包括多项需要显示内容的单独页。

使用什么方式将 Stats 显示到屏幕上需要进行权衡。如果你在 Stat 类的子类中放入自己的打印代码将减少可移植性，但可以做到任何你想做的事情，如用柱状图显示统计信息。如果你如提供的实例中那样简单地将统计信息作为字符串输出到显示设备，你的实现将具有很好的可移植性并易于在跨平台开发中使用。

系统的基础是一个包含页链表和其他 Stats 系统全局信息的类。Initialize()函数取代构造函数的功能，因为这时一个完全静态的类并不能通过构造函数来构造实例。基类有一个 Print()函数用于调用当前页的 Print()函数。

基于 Windows 的示例代码还包含键盘输入处理，将按键状态的布尔数组作为输入。此数组由 Windows 事件处理循环通过对 WM_KEYDOWN 和 WM_KEYUP 消息的跟踪来构造。在游戏机上，传入的是一个包含游戏手柄状态的数组。实时循环每轮需要调用 CheckInput()和 Print()函数。页面通过其构造函数调用基类的 AddPage()函数自动加入到列表中，这使得程序员无需手工单独增加每个页面。

初始化惟一的功能是保证基类没有指向可能被解释为页的随机数据。同时也设置了满屏所能显示的最大行数。

基类所有链表中的每一页都包括零或多项。无论是否显示，链表中总有一页是当前页。项目使用通用项父类加入到页中。项目的构造函数自动调用 AddEntry()函数，类似页构造函数调用 AddPage()函数。

每页包含一个项目链表。项目构造函数被用来设置统计的初试值，包括显示的页，文本标签，和与项目相关的优先级（也可以理解为行号）。继承类型也有初试值，需要由继承类型初始化为一个特定值。每个继承类型需要重载缺省类型 CStatEntry 的虚函数，以使父系统可以通过通用接口访问统计类型。

增加一个新类只需要复制从 CStatEntry 继承的类的代码，将其改名并增加专用变量。一旦完成这些，你需要重写部分成员函数来处理数据类型。

存储在各种统计中的实际数据依赖于统计类型并包含在继承出的统计类型中。每个数据类型有自己的 Print()函数来替换基类的虚函数。这个函数非常有用，因为基类可以在无需关心具体统计类型的情况下，通过遍历一个通用统计指针列表来要求每个统计项进行打印操作。所有统计项从一个通用父类继承出来使得浏览更加简单。使用作为通用统计指针的父类类型将便于组织和控制。支持多页并且每页都有当前项保持跟踪，这可能会造成一些其他方面的困扰。

一个典型的 Stats 应用就是将定点数统计值转换为等价的可打印浮点数。对较大的页来说统计可能需要损耗一些资源，但可以大大减少开发时间。值 1.75 比需要心算 56/32 来获取对应的 6 位浮点值来说更易读。

向一个应用程序增加 Stats 系统需要三步：

（1）在启动时调用 CStatBase::Initialize()。

（2）在实时循环的每步调用 CStatBase::CheckInput()来更新键盘或游戏机手柄的状态。

（3）在屏幕渲染后调用 CStatBase::Print()在你的实时图像上显示统计信息。

实现一个新的统计类型所需的时间依赖于其复杂程度，但可以在 10 分钟内增加一个简单的数字类型。使用枚举、向量、矩阵、或更复杂类型的统计类型需要更长时间，但也很简单。

在示例代码中的继承统计类型 CStatIntPtr 需要一些额外的解释，因为它的接口稍有不同并能够用来做一些有趣的事情。存储在统计中的值是一个指向整数的指针。当它被打印时，其指向的整数值被显示。这个特性允许你声明一个统计，并能够在值被改变时自动更新显示，而不必在单独在每帧中设置此统计值。它也允许你直接编辑其指向的数字的值，这是此系统的一个强大的特性。你可以通过在调试代码中增加一行统计实例声明，在你的运行代码中直接访问变量。

在 Sony PlayStation 平台上，Stats 系统显示覆盖整个屏幕的统计信息，通常所需的执行时间是每帧 2ms。相对来说，每秒 30 帧情况下显示一帧需要 33.3ms，因此 Stats 系统使用小于 6%的处理器就能显示一个完整的数据页。而一旦你决定在编译你的发布版本时不包括 Stats 系统，这个时间将归还给其他程序。在 PC 系统上所需的处理时间百分比应该更少。

关于这个 Stats 实现中有一点需要值得注意。此系统不是用来让页和项显示或消失的。所有的页和项必须静态声明。虽然动态构造和统计删除的实现并不困难，但静态统计基本上已

经足够了，因此不用增加多余的特性。你可以通过修改类的析构函数，在每个函数中增加断言来确认统计不会被析构或超出有效范围。

1.13.4 Where：可用性

一个 Stats 系统的显著应用是用户界面原型，在还没有真实屏幕可用之前可在此设置屏幕流程。一个典型的菜单可以在几分钟内设置好，并允许你绕过还没有完成的用户界面代码。你可以避免不必要的设计，因为基于文本的界面已经在 Stats 系统中实现了。只需简单地编写临时 Stats 系统，你就不必等待其他人完成核心代码块才能继续工作。

你可以将 Stats 系统合并到游戏中的目录和路径编辑中去。Stats 可以在游戏运行中用于设置载入、移动和保存指令。它也可以被用来编辑 AI 属性并载入和保存任意类型的数据文件，包括在内存 dump 类中的批处理脚本文件（参考“1.8 快速数据载入技场”获取详细信息）。它也可以被用来触发构造基于世界位置的帧计算时间统计图表来显示高负载区域。

Stats 系统也可以给出方便的方法来跳到任意游戏级别。在游戏中任何你希望重设、打开或关闭的任意多选项都可以使用 Stats 系统处理。例如一些高亮冲突、调整摄像机行为，指定玩家速度、设置环境光源，等等。它可以通过编辑应用程序中摄像机位置和可视范围来从极大程度上简化模型构造者的工作。这些数字可以被简单的插入到模型软件中，而不必再经历耗时的构造－测试－构造－测试的流程。

1.13.5 小结

在我所参与的使用了 Stats 系统的项目中，它不仅为软件设计者，也为美工和关卡设计者节省了大量的时间。无论是对 PC 还是游戏机，Stats 都是一个简单的为应用程序增加调试接口的方法。也可以在开发过程中为现有应用程序适当地翻新 Stats 系统，来解决难以处理的问题。

1.14 实时的游戏内建剖析

Steve Rabin

剖析（profile）代码在大部分的软件开发中都是一个常规性的步骤，但在游戏开发中则往往是至关重要的步骤。因为游戏经常性需要监视性能瓶颈和其他可能导致帧率（frame rate）下降的愚蠢错误。当帧率下降时，不通过有效实际测量是无法猜出问题发生在哪儿的。是因为昨天晚上最后调整的人工智能（AI）代码，还是因为今天早上修改的碰撞检测代码？或许更糟，没准是因为一部分这周来都没有触及的代码，在处理新数据时导致问题。确认问题的惟一方法就是剖析程序。

本文向你展示如何直接将剖析代码加入到游戏中。你不但能通过它快速找到代码中的瓶颈所在，而且任何人，包括其他程序员、产品经理、设计师、美工和测试人员，都可以从屏幕上直接获取这些信息。这些信息使得作为一个可访问工具的剖析程序能够从本质上帮助你调整代码和寻找错误。例如，如果帧率在每次大规模开火时下降，则导致问题的可能是图形复杂性或碰撞检测逻辑。如果剖析功能随时可用，简单按一个键就能告诉你答案。

有些人坚持在接近完成模块代码之前都不加入剖析代码。他们认为："为什么在代码的正常功能还没有实现的时候就要去考虑速度问题？"虽然这种观点也有正确的地方，只有代码实际上能够工作时才值得去剖析；但很多时候我之所以要在开发中期就剖析一个模块，只是为了发现我认为应该被调用的函数为什么没有被调用或被调用了两次。显然，剖析作为一种工具能够在开发的任何阶段辅助调试工作。

1.14.1 开始考虑细节

这个实时剖析器允许你监控你感兴趣的任意代码点或代码段。你只需要在希望剖析的区域开始和结束的时候调用一个函数即可。每个样本中都由一个 ProfileBegin 和一个 ProfileEnd 组成，并由一个你选择的标识符来区分。使用提供的代码，你可以使用 ProfileBegin("InsertSampleNameHere") 和 ProfileEnd("InsertSampleNameHere") 将想要查看的代码包括进去。

剖析的代价

整体上来说，剖析器用来保持对你样本跟踪的时间总量可以忽略不计，特别当你只要每次监视少量几个点的时候。不幸的是将结果显示在屏幕上

的功能，根据实现方法以及文本数量，或许会稍微降低一些你的帧率。如果你屏幕显示文本的实现方式不好（或者根本就没有实现），你可以总是以图形方式在屏幕上显示，或者将其单独存储到文本文件中去。而且与其他调试代码一样，你可以对剖析代码进行包装，使之在编译发布版本时根本不被包括进去。然而直到临近发布，你肯定会需要在优化版本中能够激活剖析器。

虽然监视代码不会使用很多时间，但还是不要监视太小的代码片断，尤其是那些在一帧内会被执行几百次的代码。否则监视代码可能会使用比目标代码片断更长的时间，并导致实际上更糟的视觉效果。一个例子是去监视类似 VectorNormalize()这样在一帧中被调用几百次的函数。剖析器能够精确地告诉你函数被调用了多少次，但时间信息将没有作用。在这种情况下，应该去求助于更专业的剖析器。

值得重视的是这个实时剖析器不应该取代传统的剖析器。一个真正的剖析器可以提供给你无法被这个技术所取代的更有价值的信息。也就是说，这个实时剖析器将增强你常常做的剖析工作。可以将这个剖析器看作一种快速但并不优雅的获取有用信息的途径。当你准备获得更精确测量信息时，应该换用更专业的剖析器。

1.14.2 剖析器将告诉你什么?

剖析器将在每一帧结束时给你下列信息。你通常会希望将这些信息打印到屏幕或其他的输出设备。本文不会帮助你绘制屏幕文本（参考“6.1 文本工具库”），但会给你以下重要数据。

（1）样本点的惟一名称。

（2）在此样本上耗费的平均、最小和最大帧时间比例。

（3）每帧中此样本被调用的次数。

（4）此样本点与其他样本点的关系（父/子）。

剖析器视图智能处理样本并保持对其父子关系的跟踪。例如，如果在你的游戏主循环和游戏主循环的图形绘制例程中都有样本，则父子关系将被自动建立。

结果如表 1.14.1 所示：

表 1.14.1 采样结果 1

平均值	最小值	最大值	#	剖析名
14.3	11.9	34.9	1	游戏主循环
85.7	65.1	88.1	1	图形绘制例程

在表 1.14.1 中显示的结果说明了一些问题

（1）图形绘制例程占用了 85.7%的帧率。

（2）除了图形绘制例程之外的其他部分占用了 14.3%的帧率。

（3）图形绘制例程在游戏主循环中被调用（通过缩进标示）。

（4）游戏主循环应该使用 100%的帧率，但因为图形绘制例程在游戏主循环中被剖析，

因此游戏主循环的帧率要减去图形绘制例程的帧率。

（5）游戏主循环最高使用了 34.9%，而平均值只有很低的 14.3%，这说明游戏主循环中的一些代码周期性影响帧时间。这些代码也许是人工智能代码或碰撞检测代码，需要增加更多的样本来使用剖析器定位。

（6）游戏主循环和游戏图形例程在每帧中只被调用一次（通过#列标示）。

在增加更多剖析器样本来定位峰值问题后，结果如表 1.14.2 所示：

表 1.14.2 采样结果 2

平均值	最小值	最大值	#	剖析名
2.4	1.8	2.8	1	**游戏主循环**
2.2	1.9	2.3	1	**游戏对象更新**
7.6	6.5	27.4	32	**AI 更新**
1.1	0.8	1.3	1	**碰撞检测**
1.0	0.9	1.1	1	**物理**
85.7	65.1	88.1	1	**图形绘制例程**

从表 1.14.2 所示结果中，我们可以发现人工智能修正样本被调用了 32 次（或许有 32 个游戏对象需要人工智能）。可以清楚地看出在一些帧中占用 27.4%的帧率的人工智能修正是造成峰值的样本。在人工智能修正中应该有一些代码被周期性地调用。也许是那些代码的工作可以被多帧使用，因此造成间隔性的峰值出现。继续增加剖析样本直到确定的代码段被辨别出。使用剖析器能够很容易地跟踪到问题。

1.14.3 增加剖析器调用

如前面所述，必须将你希望进行剖析的代码使用 ProfileBegin 和 ProfileEnd 包裹起来。你总是应该将你程序的主循环包裹起来，并且在循环末尾调用 ProfileDumpOutputToBuffer。ProfileDumpOutputToBuffer 将剖析信息格式化为文本，并存储到一个文本缓冲区，以使你能在主循环的某处将之显示到屏幕上。下列代码是包裹游戏循环的一个例子：

```
void main {
  //在这里初始化代码

  ProfileInit();   //你必须在主循环之前调用此函数

  while( !ExitGame ) {
    ProfileBegin( "Main Loop" );

    ReadInput();
    UpdateGameLogic();

    ProfileBegin( "Graphics Draw Routine" );
    RenderScene();
    RenderProfileTextBuffer();     // 输出最后一帧的剖析信息文本
```

```
        ProfileEnd( "Graphics Draw Routine" );

        ProfileEnd( "Main Loop" );
        ProfileDumpOutputToBuffer();   // 将在下一帧中显示的缓冲区
    }
}
```

1.14.4 剖析器的实现

在一个给定帧中，每个剖析样本需要下列信息：

```
typedef struct {
    bool bValid;                // 数据是否有效
    uint iProfileInstances;     // ProfileBegin 调用次数
    int iOpenProfiles;          // 没有相匹配 ProfileEnd 调用的 ProfileBegin 调用次数
    char szName[256];           // 样本名称
    float fStartTime;           // 当前样本开始时间
    float fAccumulator;         // 帧内所有样本总计
    float fChildrenSampleTime;  // 所有子样本耗时
    uint iNumParents;           // 父样本数
} ProfileSample;
```

我们需要跨越多帧保持的样本历史信息，将在下列结果中被存储：

```
typedef struct {
    bool bValid;        // 数据是否有效
    char szName[256];   // 样本名称
    float fAve;         // 每帧的平均时间（百分比）
    float fMin;         // 每帧的最小时间（百分比）
    float fMax;         // 每帧的最大时间（百分比）
} ProfileSampleHistory;
```

为简单和速度考虑，将预分配 ProfileSample(s)和 ProfileSampleHistory(s)数组。预分配数据使我们在每次采样时不必为分配和释放内存而耗费时间。在任何样本被采样之前，调用 ProfileInit 来初始化两个数组并记录开始时间。

两个函数被用来获取时间：GetTime 和 GetElapsedTime。GetTime 将以秒的形式返回系统时间（调用时的精确时间）。GetElapsedTime 将返回自上一帧完成以来耗费的所有时间（通过 1/current_frame_rate 计算）。

```
#define NUM_PROFILE_SAMPLES 50
ProfileSample g_samples[NUM_PROFILE_SAMPLES];
ProfileSampleHistory g_history[NUM_PROFILE_SAMPLES];
float g_startProfile = 0.0f;
float g_endProfile = 0.0f;

void ProfileInit( void )
{
    uint i;
```

```
    for( i=0; i<NUM_PROFILE_SAMPLES; i++ ) {
      g_samples[i].bValid = false;
      g_history[i].bValid = false;
    }

    g_startProfile = GetTime();
  }
```

1.14.5 ProfileBegin 的细节

我们现在已经可以开始记录样本了，因此让我们看看程序列表 1.14.1 中的 ProfileBegin 函数。当此函数被调用时，它首先检查是否已经有一个同名样本存在。如果找到，则说明此样本已经在此帧之前被调用过。在这种情况下，我们需要递增 iOpenProfiles、iProfileInstances 并标记 fStartTime。

变量 iOpenProfiles 被 ProfileBegin 递增并被 ProfileEnd 递减。事实上它被用于跟踪有多少样本开始但没有结束。注意这个实现完全不处理递归调用（一个样本在结束前开始多次）。因为这个原因，这里有一个断言用于捕获此问题。

变量 iProfileInstance 被 ProfileBegin 递增以统计样本在帧中被调用的次数。你可能记得，这是显示输出的剖析信息中一个关键部分。如果样本从没在此帧中被调用过，代码从数组中找到一个未被使用的样本数据结构并将之初始化。

1.14.6 ProfileEnd 的细节

ProfileBegin 的功能很容易理解，实际的工作则是在 ProfileEnd 中完成的。这个函数计算结果并评估父子关系。

ProfileEnd 第一步先在数组中找到样本。一旦找到样本，结束时间将被记录下列并且会递减 iOpenProfiles。代码循环遍历所有样本，统计有多少样本的剖析没有结束（父节点），并将最近的一个为结束样本的索引记录下来（直接父节点）。父节点数量将被记录到 iNumParents。如果有父节点，则样本时间将被在直接父节点的结构中（稍后被用来计算直接父节点的样本时间）。

既然这个样本可能在此帧中再次被开放，样本时间被保存到 fAccumulator 中，以使其他样本可以使用 fStartTime。代码列表 1.14.2 展示了 ProfileEnd 函数。

1.14.7 处理剖析数据的细节

所有剩下的工作将是处理、格式化并将数据导出到文本缓冲区。这些工作在游戏主循环的最后由 ProfileDumpOutputToBuffer 函数完成。代码列表 1.14.3 展示了此函数。注意两个函数 ClearTextBuffer 和 PutTextBuffer 被用于将文本到输出缓冲区。你必须提供这些函数。PutTextBuffer 将你提供的文本字符串放入一个垂直可滚动的缓冲区，以使每次成功的调用会在缓冲区尾部

插入一个新的文本字符串。ClearTextBuffer 则简单地清除垂直可滚动缓冲区。

另外两个函数 StoreProfileInHistory 和 GetProfileFromHistory 也被提及。他们在代码列表 1.14.4 中被列出。这些函数被用于跟踪每个样本的平均、最小和最大帧率百分比。通过调用 StoreProfileInHistory，你将根据当前测量结果和以前的每次采样结果来计算平均值。GetProfileFromHistory 获得新平均值用于显示。

1.14.8 后期增强

或许你已经意识到这个剖析器完全以 C 语言编写（除 C++风格的注释以外）。通过将之转换到 C++语言，可以消除 ProfileEnd 函数的显式调用。诀窍是使用类的构造和析构函数。注意下面的示例代码，它剖析一个 for 循环：

```
{
  ProfileInstance profile_instance( "Timing the For Loop" );

  for( int i=0; i<10000; i++ );
}
```

在代码中 ProfileInstance 对象被声明并使用一个描述字符串初始化。当其构造函数被调用时，字符串连同系统时间将被记录下来。这些信息将以类似 ProfileBegin 的方式记录到全局数据中。

因为 profile_instance 对象在花括号内，所以会在程序执行到此范围外时被销毁。因此，构造函数将在 for 循环结束后被立即调用。析构函数将以 ProfileEnd 完全相同的方式记录系统时间并保存信息到全局数据结构中。

既然剖析代码行现在有一些难以使用，我们可以通过一个简单的宏将之简化：

```
#define Profile(a)     ProfileInstance profile_instance(a)
```

使用此宏的剖析代码示例如下：

```
{
  Profile( "Timing the For Loop" );

  for( int i=0; i<10000; i++ );
}
```

这个增强功能的美妙之处在于，剖析代码现在只占一行。更妙的是，不再需要一个包含完全匹配字符串的结束语句了！剖析已经简单的不能再简单了！

1.14.9 将它们组合起来

通过将以上函数组合起来，我们能够获得一个相当强大的剖析器。非常明确的是，实际使用是最困难的部分。在本书的附带光盘中，你可以找到完整的剖析器实现代码。这个简单的剖析器可以立刻成为你最重要的调试工具。

程序清单 1.14.1：ProfileBegin

```
void ProfileBegin( char* name )
{
  uint i = 0;

  while( i < NUM_PROFILE_SAMPLES && g_samples[i].bValid == true ) {
    if( strcmp( g_samples[i].szName, name ) == 0 ) {
      // 找到一个样本
      g_samples[i].iOpenProfiles++;
      g_samples[i].iProfileInstances++;
      g_samples[i].fStartTime = OSGetTime();
      assert( g_samples[i].iOpenProfiles == 1 ); // 每次的最大值为 1
      return;
    }
    i++;
  }

  if( i >= NUM_PROFILE_SAMPLES ) {
    assert( !"Exceeded Max Available Profile Samples" );
    return;
  }

  strcpy( g_samples[i].szName, name );
  g_samples[i].bValid = TRUE;
  g_samples[i].iOpenProfiles = 1;
  g_samples[i].iProfileInstances = 1;
  g_samples[i].fAccumulator = 0.0f;
  g_samples[i].fStartTime = GetTime();
  g_samples[i].fChildrenSampleTime = 0.0f;
}
```

程序清单 1.14.2：ProfileEnd

```
void ProfileEnd( char* name )
{
  uint i = 0;
  uint numParents = 0;

  while( i < NUM_PROFILE_SAMPLES && g_samples[i].bValid == true )
  {
    if( strcmp( g_samples[i].szName, name ) == 0 )
    { // 找到一个样本
      uint inner = 0;
      int parent = -1;
      float fEndTime = GetTime();
      g_samples[i].iOpenProfiles--;

      // 统计所有父节点并找到直接父节点
      while( g_samples[inner].bValid == true ) {
```

```
            if( g_samples[inner].iOpenProfiles > 0 )
            {  // 找到一个父节点（任何开放的节点都是父节点）
               numParents++;
               if( parent < 0 )
               {  // 替换无效父节点（索引）
                  parent = inner;
               }
               else if( g_samples[inner].fStartTime >=
                        g_samples[parent].fStartTime )
               {  // 替换最直接的父节点
                  parent = inner;
               }
            }
            inner++;
         }

         // 记录样本父节点当前数量
         g_samples[i].iNumParents = numParents;

         if( parent >= 0 )
         {  // 记录时间到 fChildrenSampleTime（累加）
            g_samples[parent].fChildrenSampleTime += fEndTime -
               g_samples[i].fStartTime;
         }

         // 保存累及样本时间
         g_samples[i].fAccumulator += fEndTime - g_samples[i].fStartTime;
         return;
      }
      i++;
   }
}
```

程序清单 1.14.3：ProfileDumpOutputToBuffer

```
void ProfileDumpOutputToBuffer( void )
{
   uint i = 0;

   g_endProfile = GetTime();
   ClearTextBuffer();

   PutTextBuffer( "  Ave :   Min :   Max :   # : Profile Name\n" );
   PutTextBuffer( "-----------------------------------------\n" );

   while( i < NUM_PROFILE_SAMPLES && g_samples[i].bValid == TRUE ) {
      uint indent = 0;
      float sampleTime, percentTime, aveTime, minTime, maxTime;
      char line[256], name[256], indentedName[256];
      char ave[16], min[16], max[16], num[16];
```

```
    if( g_samples[i].iOpenProfiles < 0 ) {
      assert( !"ProfileEnd() called without a ProfileBegin()" );
    }
    else if( g_samples[i].iOpenProfiles > 0 ) {
      assert( !"ProfileBegin() called without a ProfileEnd()" );
    }

    sampleTime = g_samples[i].fAccumulator - g_samples[i].fChildrenSampleTime;
    percentTime = ( sampleTime / (g_endProfile - g_startProfile ) ) * 100.0f;

    aveTime = minTime = maxTime = percentTime;

    // 在历史记录中增加新的测量值，并获取平均、最小和最大值
    StoreProfileInHistory( g_samples[i].szName, percentTime );
    GetProfileFromHistory( g_samples[i].szName, &aveTime, &minTime, &maxTime );

    // 格式化数据
    sprintf( ave, "%3.1f", aveTime );
    sprintf( min, "%3.1f", minTime );
    sprintf( max, "%3.1f", maxTime );
    sprintf( num, "%3d", g_samples[i].iProfileInstances );

    strcpy( indentedName, g_samples[i].szName );
    for( indent=0; indent<g_samples[i].iNumParents; indent++ ) {
      sprintf( name, "  %s", indentedName );
      strcpy( indentedName, name );
    }

    sprintf(line,"%5s : %5s : %5s : %3s : %s\n", ave, min, max,
      num, indentedName);
    PutTextBuffer( line );    // 将当前行写入文本缓冲区
    i++;
  }

  { // 为下一帧重置样本数据
    u32I i;
    for( i=0; i<NUM_PROFILE_SAMPLES; i++ ) {
      g_samples[i].bValid = FALSE;
    }
    g_startProfile = GetTime();
  }
}
```

程序清单 1.14.4：StoreProfileInHistory、GetProfileFromHistory

```
void StoreProfileInHistory( char* name, f32 percent )
{
  uint i = 0;
  float oldRatio;
  float newRatio = 0.8f * GetElapsedTime();
  if( newRatio > 1.0f ) {
```

```
        newRatio = 1.0f;
    }
    oldRatio = 1.0f - newFraction;

    while( i < NUM_PROFILE_SAMPLES && g_history[i].bValid == TRUE ) {
        if( strcmp( g_history[i].szName, name ) == 0 )
        { // 找到一个样本
            g_history[i].fAve = (g_history[i].fAve*oldRatio) + (percent*newRatio);
            if( percent < g_history[i].fMin ) {
                g_history[i].fMin = percent;
            }
            else {
                g_history[i].fMin = (g_history[i].fMin*oldRatio) + (percent*newRatio);
            }

            if( percent > g_history[i].fMax ) {
                g_history[i].fMax = percent;
            }
            else {
                g_history[i].fMax = (g_history[i].fMax*oldRatio) + (percent*newRatio);
            }
            return;
        }
        i++;
    }

    if( i < NUM_PROFILE_SAMPLES )
    { // 添加到历史数据
        strcpy( g_history[i].szName, name );
        g_history[i].bValid = TRUE;
        g_history[i].fAve = g_history[i].fMin = g_history[i].fMax = percent;
    }
    else {
        assert( !"Exceeded Max Available Profile Samples!");
    }
}

void GetProfileFromHistory( char* name, f32* ave, f32* min, f32* max )
{
    uint i = 0;
    while( i < NUM_PROFILE_SAMPLES && g_history[i].bValid == TRUE ) {
        if( strcmp( g_history[i].szName, name ) == 0 )
        { // 找到一个样本
            *ave = g_history[i].fAve;
            *min = g_history[i].fMin;
            *max = g_history[i].fMax;
            return;
        }
        i++;
    }
```

```
        *ave = *min = *max = 0.0f;
    }
```

1.14.10 参考文献

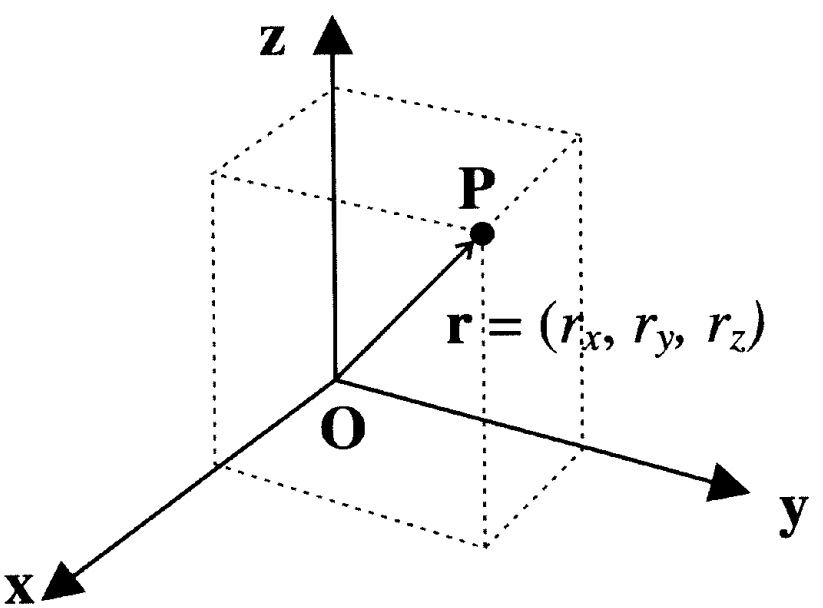
[Abrash97] Abrash, Michael, *Michael Abrash's Graphics Programming Black Book*, The Coriolis Group 97

[McConnell93] McConnell, Steve, *Code Complete*, Microsoft Press, 1993

[Meyers95] Meyers, Scott, *More Effective C++*, Addison-Wesley Longman, Inc., 1996

第 2 章

数学技巧



一点，从玩家的观点来看，我们需要能够实时生成游戏世界。不仅如此，每次游戏重新开始或者在不同的平台或机器上玩的时候，这个游戏世界还必须具有同样的外观。

简单起见，我们可以假定每一个游戏世界中的对象都可以用它在游戏世界中的方位和一个特性集来表示，这些特性表明了它将如何出现在玩家面前。每个对象还可以与玩家交互，这一点我们将随后讨论。

举个例子，让我们考虑一个游戏世界的简化模型。就是说，我们有一个容器对象叫星系，它包含了一定数目的恒星。在游戏进行时，我们将使用一种可重复的（在玩家看来它总是一样的）并容纳了足够多行星的方式来填充星系，以提供一种合理的无限幻觉。

为了达到这个目的，我们构造了一系列的恒星对象并为每个对象设置方位属性，这样它们就存在于星系中的给定点上了。假定方位属性由坐标 x 和 y 组成，显然我们需要生成一些数值来赋给它们。这就是用到可预测随机数序列的地方，我们可以用一系列生成的随机数来给出每个恒星的坐标。

ANSI C 规范提供了一种简单的方法，它给出了下面两个函数：

```
srand (seed)
rand()
```

每当我们需要一个数字序列的时候，需要一个二级操作。首先给定发生器一个种子数，这是用来繁殖序列的。每当我们给出同一种子数时总可以生成同一个序列。这样序列就是可重复的，但我们不必在任何地方都存储它（随机序列），因为它可以极快地生成。这样我们就可以在游戏中完整地生成一个近乎无限的星系，极大地减少了静态存储的需求。

以下非常简单的伪代码将这一概念作用于一个给定的有限规模的星系。假定一个可能的栅格为100×100，我们就有 10 000 个可能的空间来放置恒星。注意每一序列执行播种一次且仅一次；在以下情形，我们在星系上播种数 1：

```
srand (1)
for galaxy_x = 1 to 100
  for galaxy_y = 1 to 100
  probability = rand() % 100
  if probability > 70 then
        universe (galaxy_x,galaxy_y) = star
  else
        universe(galaxy_x, galaxy_y) = no_star
```

细心的读者也许已经注意到了一个可能的缺点：我们仍然需要在某个地方存储每个恒星的位置，这需要大约 10 000 字节的实时空间。这个大小永远不可能适合于安装到一个只有 16KB RAM 的机器内。即使可以，也不可能扩展到程序规模扩大 10 倍的情况下。

因此，我们应该稍微改进一下这个技术。从前述例子当中，我们看到一个恒星存在于一个给定的点 (x, y) 的机会有 70%。决定这一百分比的数字是从一个基于简单种子 1 生成的序列得来的。由于我们想得到一个接近无限的总体，最好是按需进行计算，正如以下代码所示。

```
int StarAt ( int nGalaxy, int nX, int nY)
{
    int x, y, nReturn;
```

2.0 可预测随机数

Guy W. Lecky-Thompson

当今市场上，细节丰富的背景已成为成功游戏的重要因素，游戏中的各种动作正是在此背景之上发生的。不仅如此，背景还在玩家与游戏的交互中扮演着积极的角色。为了得到这种效果，传统的方法是仅将手工制作的关卡数据提取出来，然后将其保存在一个相对比较复杂，而且占用空间较大的关卡文件里面去，以备实时重放。

然而有时候，游戏的深度可能会低于玩家的期望值。引起这种失望的原因有时仅仅是因为游戏不够大而已；战役一旦打响，立刻血流满地，这不免让玩家觉得高潮太短，虎头蛇尾。

即使拥有强劲的资源配置，对于规模宏大而复杂的游戏而言，其设计者也常常缺乏足够的空间进行调配，尤其如果游戏对图像使用了照片级真实感渲染，而且音响效果极好，还夹杂了重击音乐和爆炸声。这些都是重要的因素，但是游戏设计者可能常感觉他们是以牺牲了一些其他的东西为代价的。在资源有限的环境中，如在掌上电脑和控制台市场，这一点尤为明显。

一个巨大而复杂的游戏例子是 *Elite*。这个游戏最初是由 David Braben 和 Ian Bell 于 1980 年为英国广播公司（BBC）的 B 型计算机所写。自那时起它被移植为各种个人电脑格式，直至今天仍被常玩不衰。最初的游戏是在一个有 32KB 存储器（16KB ROM 和 16KB RAM）的机器中运行的，但它仍以拥有难以匹敌的游戏深度而著称：近乎无限个行星，每一个都有各自的名字和特征。

本文研究的是一种能用于提供给玩家应得的游戏深度的技术，即使是在如最初的 *Elite* 所面对的那样资源有限的环境中，也用不着牺牲组成完美游戏的任何其他重要成份。

2.0.1 可预测随机数

这一技术最重要的原则就是，为了在一个游戏世界中给出无限空间的幻觉，我们需要满足两个分解条件。可以把它们称为宏无限（macro-infinite）和微无限（micro-infinite）。第一个条件涉及到问题的空间规模，或者说离散的实体数目。第二个条件则表明了每一个对象所支持的细节级别。我们将在本文中看到如何使用同一种重要的技术同时满足这两个条件。

存储大量的关卡数据将永难满足我们所确定的分解要求，为了避免这

一点，从玩家的观点来看，我们需要能够实时生成游戏世界。不仅如此，每次游戏重新开始或者在不同的平台或机器上玩的时候，这个游戏世界还必须具有同样的外观。

简单起见，我们可以假定每一个游戏世界中的对象都可以用它在游戏世界中的方位和一个特性集来表示，这些特性表明了它将如何出现在玩家面前。每个对象还可以与玩家交互，这一点我们将随后讨论。

举个例子，让我们考虑一个游戏世界的简化模型。就是说，我们有一个容器对象叫星系，它包含了一定数目的恒星。在游戏进行时，我们将使用一种可重复的（在玩家看来它总是一样的）并容纳了足够多行星的方式来填充星系，以提供一种合理的无限幻觉。

为了达到这个目的，我们构造了一系列的恒星对象并为每个对象设置方位属性，这样它们就存在于星系中的给定点上了。假定方位属性由坐标 x 和 y 组成，显然我们需要生成一些数值来赋给它们。这就是用到可预测随机数序列的地方，我们可以用一系列生成的随机数来给出每个恒星的坐标。

ANSI C 规范提供了一种简单的方法，它给出了下面两个函数：

```
srand (seed)
rand()
```

每当我们需要一个数字序列的时候，需要一个二级操作。首先给定发生器一个种子数，这是用来繁殖序列的。每当我们给出同一种子数时总可以生成同一个序列。这样序列就是可重复的，但我们不必在任何地方都存储它（随机序列），因为它可以极快地生成。这样我们就可以在游戏中完整地生成一个近乎无限的星系，极大地减少了静态存储的需求。

以下非常简单的伪代码将这一概念作用于一个给定的有限规模的星系。假定一个可能的栅格为100×100，我们就有 10 000 个可能的空间来放置恒星。注意每一序列执行播种一次且仅一次；在以下情形，我们在星系上播种数 1：

```
srand (1)
for galaxy_x = 1 to 100
  for galaxy_y = 1 to 100
  probability = rand() % 100
  if probability > 70 then
        universe.(galaxy_x,galaxy_y) = star
  else
        universe(galaxy_x, galaxy_y) = no_star
```

细心的读者也许已经注意到了一个可能的缺点：我们仍然需要在某个地方存储每个恒星的位置，这需要大约 10 000 字节的实时空间。这个大小永远不可能适合于安装到一个只有 16KB RAM 的机器内。即使可以，也不可能扩展到程序规模扩大 10 倍的情况下。

因此，我们应该稍微改进一下这个技术。从前述例子当中，我们看到一个恒星存在于一个给定的点 (x, y) 的机会有 70%。决定这一百分比的数字是从一个基于简单种子 1 生成的序列得来的。由于我们想得到一个接近无限的总体，最好是按需进行计算，正如以下代码所示。

```
int StarAt ( int nGalaxy, int nX, int nY)
{
    int x, y, nReturn;
```

```
srand (nGalaxy);
    for (y = 0; y <= nY; y++)
    {
        for (x = 0; x < nX; x++)
        {
            nReturn = rand() % MAXIMUM_VALUE;
        }
    }
    return nReturn;
}
```

2.0.2 替换算法

许多各种各样的无序或半无序的算法都能产生我们在此使用的这种效果。这不是一篇数学论文，而且对于部分高级算法来说，执行其所需要的处理能力过高，目前还无法达到，另外还要顾及游戏运行时所需要执行的其他操作，因此我们将仅仅局限于下面讨论的这种较为简单的方法。

此方法可用如下的伪码来表示：

```
Choose two large integers, Gen1 and Gen2, such that one
    is double the other
Choose a seed value that is between 1 and the smaller
    of the large integers
Choose a value Max that represents the highest number
    that is to be returned
For each iteration,
    Multiply Gen1 by the seed, and add Gen2
    The new seed is remainder of this value divided by Max
Return seed as the random value
```

上述代码的工作机制如下：首先我们得到一个可重复的数字序列，它由一个给定的模式开始（一个乘法和一个加法），然后打破这一模式，由一个除法得到余数。然后再使用得到的结果生成一个具有同一基本方程式的新序列。这样，一个相当随机的序列就建成了。源代码见后文。这是从伪随机类得到的，在本书所附光盘中给出了源代码。

```
PseudoRandomizer::PseudoRandomizer(unsigned long ulGen1,
                                   unsigned long ulSeed,
                                   unsigned long ulMax)
{
    this->ulGen1 = ulGen1;
    this->ulGen2 = ulGen1 * 2;
    this->ulSeed = ulSeed;
    this->ulMax  = ulMax;
}

unsigned long PseudoRandomizer::PseudoRandom()
```

```
{
        unsigned long ulNewSeed;

        ulNewSeed = (this->ulGen1 * this->ulSeed) + this->ulGen2;

        // Use modulo operator to ensure < ulMax
        ulNewSeed = ulNewSeed % this->ulMax;

        this->ulSeed = ulNewSeed;
        return this->ulSeed;
}
```

还有最后一个限制。用这种系统能够生成的最大数字是 4 294 967 295。从前面的算法描述中可知在 4 294 967 295 次迭代之后此序列会重复生成。这是由使用的数据类型限定的，即 32 位的无符号数。如果需要更大的数字，就必须改用不同的表示法了。

2.0.3 无限宇宙算法

到此为止，我们已经看到了怎样针对特定的游戏中的区域来确定某一个给定的特性。我们已经解决了主要问题的一半，即宏无限分解。现在我们将把注意力转向如何使用伪随机数来处理游戏对象的特征和细节——即微无限分解。

本质上来讲，微无限分解对应于用户在某一点进行放大以查看那个地方有什么东西。举我们现在的例子，我们可能会说每个恒星被零个或多个行星环绕运行。这些行星的轨道距这个恒星有一个确定的距离。我们稍后考查它们的其他特性，现在确定它们的位置就足够了。

为了要确定这些项，我们可以为恒星类添加一个属性，该属性给出了行星的数目，并且进行计算。下列代码片断展示了我们如何能计算出恒星类的一个属性——行星的数目。

```
/* Taken from header file */

class star {
private :
    …
    int x_position, y_position;
    int number_of_planets;
    …
public :
    …
    void SetNumberOfPlanets();
    …
};

/* Taken from implementation file */

void star::SetNumberOfPlanets() {
    pseudorandom->seed(this->x_position + (this->x_position
        * this->y_position));
    this->number_of_planets = pseudorandom->generate() % 20;
}
```

我们已经由这些代码轻松地跳到一个稍微高点的层次了，有些东西得解释一下。首先，我们的种子数是源自一个恒星的惟一值，基于由给定恒星对象的坐标属性 x 和 y 所决定的位置。这意味着对于每一颗恒星，我们可以为具体的对象播种生成值。这样我们就减少了在循环中有两个同样的恒星对象的可能性。

第二点要注意的是我们对生成的数字进行了取模运算以限制可能绕恒星运行的行星数目。为了引入更多的真实性（一个更大的微无限分解），这些代码可以基于恒星的其他属性，如大小、强度或接近度，进行修改。

有了这些信息，我们现在可以考虑怎样使用一个类似的技术来为给定的行星产生距离值了。正如已经提到过的，重要的出发点是用来决定对象特征的种子。记住，每一个可能行星的种子应该是惟一的，我们必须设法结合其所属恒星与此行星的位置来得到它。下列代码片断表明了一种可能的方式：

```
/* Taken from header file */

class planet {
private :
    …
    int distance_from_star;
    …
public :
    …
    void SetDistanceFromStar(int planet_number,
         int star_x_position, int star_y_position);
    …
};

/* Taken from implementation file */

void planet::SetDistanceFromStar   (int planet_number
                                    int star_x_position,
                                    int star_y_position) {

    pseudorandom->seed(planet_number + (star_x_position +
        (star_x_position *this->star_y_position)));
    this->distance_from_star = pseudorandom->generate() % 20;
}
```

于是，就确定位置来说，我们已经有效地从宇宙“放大”到恒星，再到行星。为了要完成背景幕（以及我们关于微无限分解的讨论），我们应该开始使用更多的可预测随机数序列来增加特性。

下一阶段的关键是对于任何一个宇宙中给定的对象，我们要能分离出一组能描述此对象的属性。在此之后，我们还需要决定这些属性的表现形式，其前提是要有一组恒定的随机数可用。

反过来看，每个属性本身可能又是一个对象（像行星对恒星系对宇宙），具有它自己类似的能被设置的性质，借由一个惟一的基准来播种。在下面的例子当中，我们假定需要一个行

星对象的地图，并且这个地图是由简单的栅格表示的，我们可以将其他对象放入其中。所有关于宇宙、星系、行星和地图对象的完整代码包含在本书的随书光盘中。伪码如下：

```
// Define the map size (side x side)
map->grid_side = pseudorandom->generate() % 100

// Place an object on the map for a given position x,y
pseudorandom->seed((map->grid_side * y) + x)
map->grid_square(x,y) = pseudorandom->generate() % 2
```

（例如，模取为 2 可能意为 0 是水，1 是陆地）

在前述的例子中，播种已经被省略了。正如在给出的代码示例中所看到的那样，种子是在一个对象基于对象的基础上的实例化对象上生成的。事实上，给出的代码也存储了一些用来生成种子的属性，为了简洁起见我们在这儿也省略了。

通过观察 ANSI 的 srand 和 rand 函数共同工作可以证明，有比开始时显而易见的更多的对种子的选择。我们在上文中曾提出：

```
srand (x_position + (x_dimension * y_position))
```

这里生成器基于目标容器的维数，根据惟一的基准为点（x, y）播种。然而，重复运行 srand 和 rand 表明这种方法产生的结果相当有规律。（请参看“4.16 实时真实地形生成”，它包含了一个图表说明了一个模式，该模式是使用 ANSI 生成器技术的结果。）

随书光盘中包含有伪随机类的完整代码，但是下面的算法需要在此重复一下，它能仅使用 ANSI 函数，或与伪随机类自身一起使用，来为生成器使用正确播种。

```
srand ( y_position)
x = x_position
while x > 0
    rand()
    x = x - 1
```

下一次调用 rand 会产生所需的数字，它可以被看作是后继序列中的第一个。

2.0.4 结论与展望

在本文中，我们已经看到了如何使用宏无限和微无限分解技术、通过伪随机数序列进行繁殖，以及在一个惟一对象的基准上进行播种。这些技术使我们能够在一个资源有限的环境约束下创造近乎无限的游戏世界，并且在运行时生成。

开发者可以“照搬”使用这一技术，也可进一步改进它。例如，事件也可以用同样的方法来处理，可以基于游戏的历史或实时播种。这一技术将确保如果在客栈里的一个给定点开火（可能与玩家的行为相关），我们能够预知它的发生时间，以确保它在每一场游戏中都会在同一时间出现。

成功运用可预测随机数的关键就在于如何能正确地在对象属性与游戏状态上同时使用好种子数。

2.0.5 参考文献

Lecky-Thompson, Guy W., "Algorithms for an Infinite Universe," *Gamasutra,* www.gamasutra.com/features/19990917/infinite_01.htm, September 17, 1999.

2.1 插值方法

John Olsen

你是否曾想让程序把某物体从一个位置逐渐移动到另一个位置？有成打的具有不同适应度与不同 CPU 需求的方法能够做到这一点。本文讨论了其中的 4 种方法，并给出了每种方法如何工作的详细信息：

- 使用浮点数学的帧率相关的 ease-out （Frame-rate-dependent ease-out using floating-point math）；
- 使用整型数学的帧速相关的 ease-out （Frame-rate-dependent ease-out using integer math）；
- 帧速无关的线性内插（Frame-rate-independent linear interpolation）；
- 帧速无关的 ease-in 和 ease-out（Frame-rate-independent ease-in and ease-out）。

所有这些方法有一些共同的基础。假定想从一个特殊的点出发到达另外的一个点，你心里可能有也可能没有对于花费多长时间能够到达那里的时间限制。起点和终点可以是任何数值或是数值组合。例如，它们可以是温度、高度、3D 位置、方向或速度矢量，或者有关事物的任何数值。插值的意思仅仅是从一个值沿着某种平滑的路径到达另一个值。

例如，如果你想要在一个矢量上执行这些插值方法，可以分别把算法用于矢量的每个组成部分。

2.1.1 使用浮点数学的帧速相关 ease-out

这种方法表现为一种帧速率相关的方式，因而每秒钟调用 10 帧与每秒调用 20 帧，其表现是不同的。这意味着只有当精确性你首要关心的问题不是精确时，才能使用这种方法。

这个方法的幕后思想是：在计算当前值和期望值的一个加权平均值时，当前值上有一个较大的权重。这能够由公式 2.1.1 做到。新的 x 值等于原始值 x_0 乘以一个权重因子，加上最终的目标 x 值，所得的和再除以总权重（以适当地保持数值范围）。得到的结果 x 作为下一次通过该公式计算的 x_0。

$$x = \left(x_0 \times (weight - 1) + x_f\right) / weight \tag{2.1.1}$$

权重必须是比 1 大的值才能从这个公式中得到所期望的行为。更高的权重会使到达目标位置花费的时间更长。这生成了一条平滑曲线，如图 2.1.1 所示，该图显示了一个值怎样在开始时快速变化，然后在接近目标值

时慢慢地稳定下来，这称为 *ease-out*。

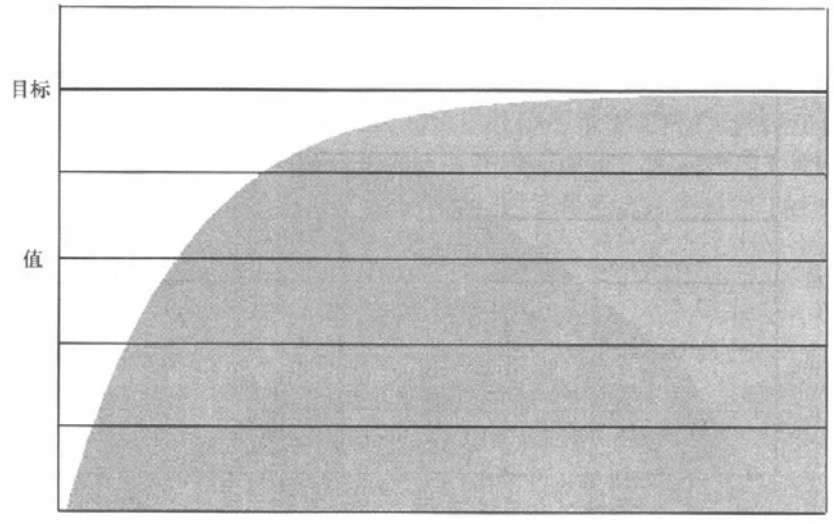


图 2.1.1 浮点 ease-out

被用于浮点 ease-out 插值的 C++类实例称为 CEaseOutDivideInterpolation（见程序清单 2.1.1）。当开始一个插值的时候调用 Setup( )，传递初值、终值以及一个控制插值发生速度的比例因子。每次实时循环调用 Interpolate( )来完成插值工作，然后调用 GetValue( )重新得到当前插入的值。当插入的值不再改变时，返回 TRUE，表明已非常接近到达目标。这些浮点代码可能会花很长的时间才能到达一个稳定的状态。

2.1.2 使用整型数学的帧速相关 ease-out

这个方法消耗的 CPU 资源非常少，因为它不使用除法。该方法对于控制台系统或是老式的具有有限浮点支持的硬件更为重要。它工作得非常快，但即使与前述方法相比，也有一些适应性方面的限制。

使用整数运算建立加权平均值的过程会带来一些有趣的副作用。在插值过程中，变化率趋向于固定在一个特定的水平，由于舍入误差，你可能永难逼近目标点。公式 2.1.2 给出了修正的方法。 (2^n-1) 和 n 的值是你想要的硬编码的速度，它给出了一个与公式 2.1.3 类似的形式，这里“>>”表示一个移位算子，如同在 C 语言中一样。这一计算会带来一些舍入上的误差，如图 2.1.2，它与图 2.1.1 及浮点方法相比平滑性要差得多。

$$x=(x_0\times(2^n-1)+x_f)>>n \tag{2.1.2}$$

$$x=(x_0\times 7+x_f)>>3 \tag{2.1.3}$$

即使此方法得到的曲线不太平滑，它对于较大的数值也是非常有用的。对于定点数学而

言，它工作得很好，在此，一些整数位被定义为小数部分。例如，一个 32 位的数可能被认为有 20 位整数位和 12 位小数位。此时从一个整型表示转换为定点表示，要除以 4 096。按那样的方法增大比例能带来更好的平滑性，得到的曲线更接近于图 2.1.1。

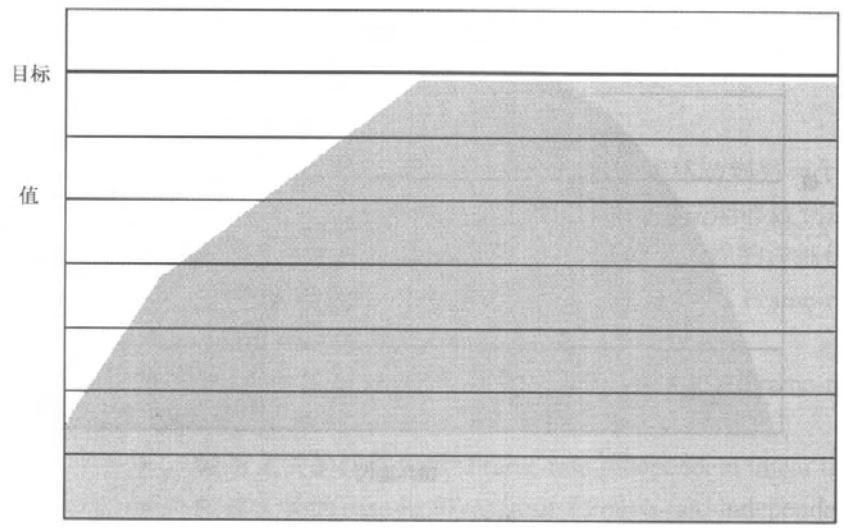


图 2.1.2 整型 ease-out

程序清单 2.1.2 包含了一个叫做 CEaseOutShiftInterpolation 的 C++类的实例，能被用来进行基于整数的 ease-out 插值。当你开始一个插值的时候，调用 Setup()，传递初值、终值及一个控制插值发生速度的移位因子。在每一遍实时循环中，调用 Interpolate()去做插值工作，然后调用 GetValue()重新得到当前的插值。当插入的值不再改变则返回 TRUE，表明你离目标已经非常接近了。由于在此处使用的是基于整数的代码，因此这种（收敛）情况在实际到达目标之前就很可能会发生了。

2.1.3 帧速无关线性内插

在线性内插的情形，我们想在运动一开始就计算出一个理想的速度，并在每一帧直接使用这个值就行了。制图时这种方法给出了一条直线，如图 2.1.3 所示。一点入门水平的物理知识就能说明如何得到这一速度（见公式 2.1.4）。如果 x 的单位是英尺，而且 t 的单位是秒，它给出了一个以英尺/秒为单位的速度。

$$v = (x_f - x_0)/t \tag{2.1.4}$$

下一步需要对每一帧应用该速度。要正确地使用这种方法，以达到帧率无关的效果，我们首先需要知道每帧花费的时间。一旦有了每帧的时间，就可以用已经计算出的速度来计算位置的改变，然后把它加到原始值之上，如公式 2.1.5 所示。如果你想走捷径，计算速度的单位用每帧的距离而不用每秒的距离，就失去了帧速无关的特征，但是可以因避免了公式 2.1.5

中的乘法而节约一点儿时间。

$$x = x_0 + t_f \times v \tag{2.1.5}$$

程序清单 2.1.3 中的能被用作线性内插的 C++类的实例称为 CLinearInterpolation。当你开始一个插值的时候，调用 Setup( )，传递初值、终值以及你所希望花费的到达目的的时间长度。在每一遍实时循环中，用你想处理的时间长度调用 Interpolate( )，然后调用 GetValue( )重新得到当前的插值。当指定的时间已经期满并且你已经到达了目标点，则返回 TRUE。

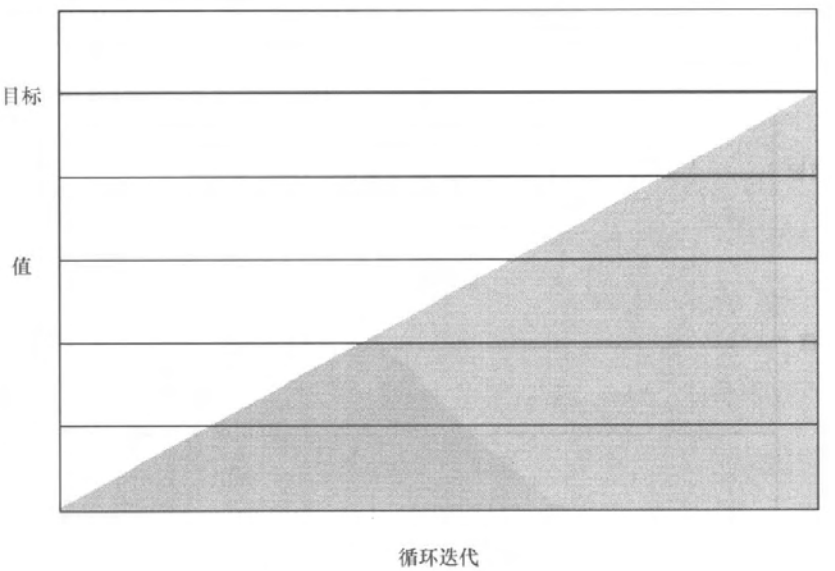


图 2.1.3 线性内插

2.1.4 帧速无关 ease-in 和 ease-out

现在我们需要一些物理背景知识。为了产生正确的 ease-in 和 ease-out，我们需要从零速度开始，以恒定加速度加速，在中间点达到最大速度，然后慢慢地减速，当到达目标点的时候下降到零速度。这一过程如图 2.1.4 所示。

第一步是计算需要的加速度。在后半程加速度被简单地取反。任何一本物理课本都能给出你需要的数学方程，如方程 2.1.6 所示。我们需要解出方程中的加速度。我们想从零速度开始，用 x 代表起点与终点的平均值而不是最终的目标，则可以导出公式 2.1.7。

$$x = x_0 + v_0 t + \frac{1}{2} a t^2 \tag{2.1.6}$$

$$a = \frac{2(x - x_0)}{t^2} \tag{2.1.7}$$

一旦知道了加速度，我们就需要将其应用于每一帧。这种方法与在线性内插中的做法类似，你必须考虑每帧的时间。开始时速度是零。在每一帧，必须首先知道是在前半程还是在后半程，这样你才能知道是加速还是减速。应用公式 2.1.8 对速度进行加减，然后应用于位置

上，如公式 2.1.5 所示，这与线性内插方法中一样，只是速度在每一帧都改变。当到达预期的终点时，这一速度应该非常接近于零，但是由于舍入误差，它可能不是完全精确地等于零。

$$v = v_0 + at \tag{2.1.8}$$

最后一个 C++类的实例在程序清单 2.1.4 给出，它能被用于 ease-in 和 ease-out 插值，叫做 CEaseInOutInterpolation。它的接口与线性内插的相同，可以在测试时很方便地从一个转换为另一个。当你开始一个插值的时候，调用 Setup( )，传递初值、终值及希望花费的时间长度。在每一遍实时循环中，用你想要处理的时间长度调用 Interpolate( )去做插值工作，然后调用 GetValue( )重新得到当前的插值。

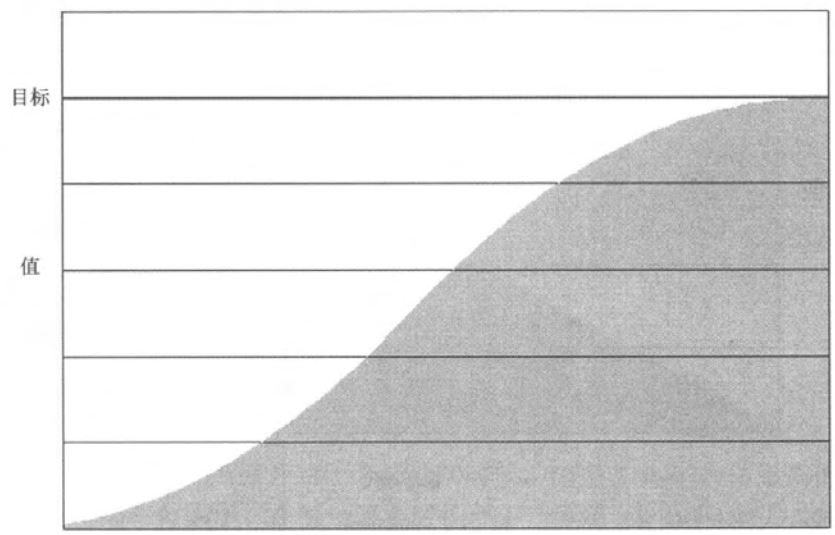


图 2.1.4 ease-in 和 ease-out

2.1.5 危险地带

当你在软件的不同部分应用插值的时候，有一些东西是必须小心对待的。角度是个问题，因为一个不经考虑的算法可能会告诉你 1°和 359°中间角度是 180°，而正确的结果或许可能是 0 时，正确答案依赖于你想让插值如何进行。当你对角度、四元数，或者任何一种有多种表示方法数值进行插值之前，首先需要小心，以确保数字在正确的范围之内。

此外，当参考示例代码的时候，你需要抓住代码的思想，并为满足特殊需要优化它们，而不是照搬这些代码。

程序清单 2.1.1 CEaseOutDivideInterpolation

```
class CEaseOutDivideInterpolation : CEaseOutShiftInterpolation
{
public:
    bool Setup(float from, float to, float divisor)
```

```
    {
        if(divisor <= 0)
        {
            return false;
        }
        _value = from;
        _target = to;
        _divisor = divisor;
        return true;
    }
    bool Interpolate() // Note: Not time dependent.
    {
        float oldValue = _value;
        if(_divisor > 0)
        {
            _value = (_value * (_divisor-1.0f) +
_target)/_divisor;
        }
        // Not likely to be true very often.
        return (_value == oldValue);
    }
    float GetValue()
    {
        return _value;
    }
private:
    float _value;
    float _target;
    float _divisor;
};
```

程序清单 2.1.2 CEaseOutShiftInterpolation 定义

```
class CEaseOutShiftInterpolation
{
public:
    bool Setup(int from, int to, int shift)
    {
        if(shift <= 0)
        {
            return false;
        }
        _value = from;
        _target = to;
        _shift = shift;
        return true;
    }
    bool Interpolate() // Note: Not time dependent.
    {
        int oldValue = _value;
        if(_shift > 0)
```

```
        {
            _value = (_value * ((1 << _shift) - 1) +
_target) >> _shift;
        }
        // lots more likely to be true than with float version.
        return (_value == oldValue);
    }
    int GetValue()
    {
        return _value;
    }
private:
    int _value;
    int _target;
    int _shift;
};
```

程序清单 2.1.3 CLinearInterpolation 定义

```
class CLinearInterpolation
{
public:
    bool Setup(float from, float to, float time)
    {
        if(time < 0)
        {
            return false;
        }
        _remainingTime = time;
        _value = from;
        _step = (to-from)/time; // Calculate distance per second.
        return true;
    }
    // Return TRUE when the target has been reached or passed.
    bool Interpolate(float deltaTime)
    {
        _remainingTime -= deltaTime;
        _value += _step*deltaTime;
        return (_remainingTime <= 0);
    }
    float GetValue()
    {
        return _value;
    }
private:
    float _value;
    float _step;
    float _remainingTime;
};
```

程序清单 2.1.4 CEaseInOutInterpolation 定义

```
class CEaseInOutInterpolation
{
public:
    bool Setup(float from, float to, float time)
    {
        if(time <= 0)
        {
            return false;
        }
        _value = from;
        _target = to;
        _speed = 0.0f;
        // derived from x=x0 + v0*t + a*t*t/2
        _acceleration = (to-from)/(time*time/4);
        _remainingTime = _totalTime = time;
        return true;
    }
    bool Interpolate(float deltaTime)
    {
        _remainingTime -= deltaTime;
        if(_remainingTime < _totalTime/2)
        {
            // Deceleration
            _speed -= _acceleration * deltaTime;
        }
        else
        {
            // Acceleration
            _speed += _acceleration * deltaTime;
        }
        _value += _speed*deltaTime;
        return (_remainingTime <= 0);
    }
    float GetValue()
    {
        return _value;
    }
private:
    float _value;
    float _target;
    float _remainingTime;
    float _totalTime;
    float _speed;
    float _acceleration;
};
```

2.2 求刚体运动方程的积分

Miguel Gomez

本文试图给出一个在理论上和实践上模拟刚体运动的指南。文中导出了牛顿－欧拉方程，给出了一些简单的数值积分方法。本文假定读者学过下列课程本科水平的导论：经典力学、线性代数、微积分学、矢量分析，以及微分方程理论。

2.2.1 运动学：平移和旋转

为了导出描述刚体运动的微分方程，需要先奠定一些基础。让我们马上给出一个固定的坐标系，使动态变量能够关于该坐标系被表示出来。一个固定的坐标系意为三个线性无关的矢量（基）和原始位置（原点），它不平移也不旋转（一个惯性坐标系）。让我们简单地用相互正交的单位矢量作为基（规范化正交基（orthonormal basis））。这一固定的坐标系称为为世界坐标系（world frame），或世界空间（world space）。空间中的任何一个点可以由这一坐标系的三个坐标分量来描述，如图 2.2.1 所示。

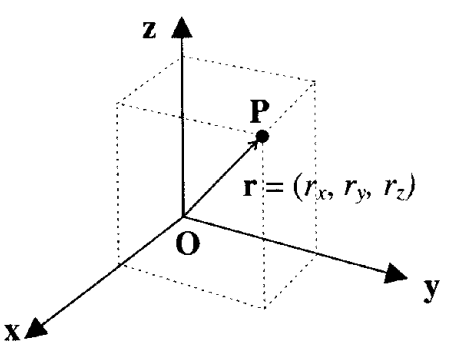


图 2.2.1 矢量 r 从原点 O 指向点 P

一个独立的体积可以忽略的物体称为质点（particle）；当需要考虑物体的体积时，则称之为物体（body）。质点或物体中所含的物质的数量称为它的质量（mass），每单位体积的质量数称为密度（density）。通常，一个物体可以有任意形状甚至于可以随时间的变化而变形。如果一个物体中质量的分布是不均衡的，则它为非均匀密度（non-uniform density）。无论一个物体的质量如何分布，在任何情况下任何时候都有一个空间中的点是物体的质心（center of mass），记为 $\mathbf{r}_{cm}$。质心的位置是用物体中每一质元 m_i 的一个加权和来计算的：

$$\mathbf{r}_{cm} = \frac{\sum \mathbf{r}_i m_i}{\sum m_i} = \frac{\sum \mathbf{r}_i m_i}{M}$$

当质量在体积中处处连续分布时，这一加权和就变为了积分：

$$\mathbf{r}_{cm} = \frac{\int \mathbf{r}\rho(r)dV}{\int \rho(r)dV} = \frac{\int \mathbf{r}\rho(r)dV}{M}$$

在这种情形下，每个质元通过一个三维密度函数$\rho(\gamma)$乘上一个体积微元 dV 来计算：

$$m_i = \rho(\mathbf{r})dV$$

将本地坐标系与物体联系起来可以帮助我们进行下面的工作。为了达到此目的，原点最好选为 $\mathbf{r}_{cm}$，我们可以拿它来表示物体的位置。规范化正交基 $\mathbf{R} = \{\mathbf{R}^0, \mathbf{R}^1, \mathbf{R}^2\}$对应于物体本地的 X 轴、Y 轴和 Z 轴（见图 2.2.2）。

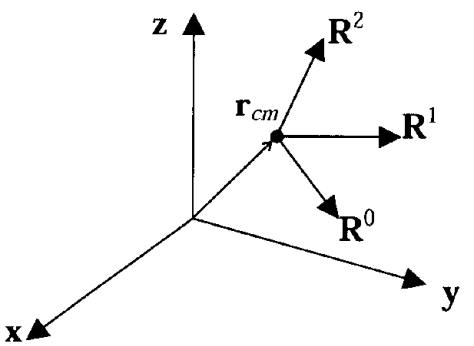


图 2.2.2 向量 R^0, R^1, R^2 定义了本地物体坐标系的 x 轴、 y 轴和 z 轴

把 $\boldsymbol{R}^0$、$\boldsymbol{R}^1$、和 $\boldsymbol{R}^2$ 看作矩阵 R 的列很方便，即

$$\mathbf{R} = \begin{bmatrix} R_x^0 & R_x^1 & R_x^2 \\ R_y^0 & R_y^1 & R_y^2 \\ R_z^0 & R_z^1 & R_z^2 \end{bmatrix}$$

这使得可以容易地将一个矢量从一个物体的本地空间转换到世界空间，反之亦然：

$v_{world} = Rv_{local}$ 且 $v_{local} = R^T v_{world}$ （因为对于规范正交基来说，$R^T = R^{-1}$）

转换一个点也同样简单：

$r_{world} = Rr_{local} + r_{cm}$ 且 $r_{local} = R^T(r_{world} - r_{cm})$

如果在时刻 t_1 一个物体的位置在 $\mathbf{r}_1$ 上，在时刻 t_2 它的位置在 $\mathbf{r}_2$ 上，（图 2.2.3），那么它在 t_1 与 t_2 间的平均速度为：

$$\mathbf{v}_{ave} = \frac{\mathbf{r}_2 - \mathbf{r}_1}{t_2 - t_1} = \frac{\Delta \mathbf{r}}{\Delta t}$$

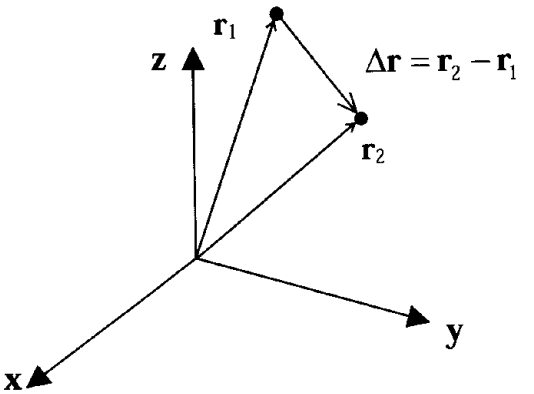


图 2.2.3 一个物体经过时间 $\Delta t = t_2 - t_1$ 从 $\mathbf{r}_1$ 运动到 $\mathbf{r}_2$

当我们以越来越小的时间间隔采样物体的轨迹时，就接近了质点的瞬时速度（instantaneous velocity），它是质点在任何时刻 t 的真实速度，并且等于位置对时间的导数：

$$v = \lim_{\Delta t \to 0} \frac{\Delta r}{\Delta t} = \frac{dr}{dt}$$

类似地，如果它的速度从一个时刻到另一个时刻变化了，就说它被加速了，由下式给出其加速度：

$$\mathbf{a} = \lim_{\Delta t \to 0} \frac{\Delta \mathbf{v}}{\Delta t} = \frac{d\mathbf{v}}{dt}$$

除了平移以外，物体还可以旋转。每单位时间物体经过的旋度的数量称为角速度（angular velocity），也称为旋转速度（rotational velocity），由下式给出：

$$\lim_{\Delta t \to 0} \frac{\Delta \boldsymbol{\theta}}{\Delta t} = \boldsymbol{\omega}$$

这里 $d\theta$ 是一个非常微小的旋度（以弧度为单位），且 ω 是质心的角速度。

用一个矢量表示一个有限旋转是一种欺骗。严格说来，有限的旋转，无论有多小，都不能被看作矢量，因为它们不遵守交换律（commutative）。这意味着如果一个物体先以第一个角度绕第一个轴旋转，然后以第二个角度绕第二个轴旋转，与颠倒操作次序得到的方位相比，结果未必相同。另一方面，无穷小（infinitesimal）旋转（如果你觉得此概念可信的话）是与顺序无关的，故它们可以被看作矢量。这就是为什么角速度能被认为是一个矢量的原因（参见[Chow95]）。

如果一个矢量 $\mathbf{r}$ 正在以一个恒角速度旋转，那么它在世界坐标系中对时间的导数为：

$$\frac{d\mathbf{r}}{dt} = \frac{\partial \mathbf{r}}{\partial t} + \boldsymbol{\omega} \times \mathbf{r}$$

如果 $\mathbf{r}$ 的长度不改变，则导数简化为：

$$\frac{d\mathbf{r}}{dt} = \boldsymbol{\omega} \times \mathbf{r}$$

根据这种关系，**R** 对时间的导数为：

$$\frac{d\mathbf{R}}{dt} = \boldsymbol{\omega}^* \mathbf{R}$$

这里反对称矩阵：

$$\boldsymbol{\omega}^* = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

代替了叉积。

2.2.2 动力学：力与旋转力矩（torque）

牛顿第一运动定律表明，如果一个物体没有受到外力的作用，它将保持静止或匀速运动。这也称为线动量（linear momentum）守恒律。一个物体的线动量 $\boldsymbol{p}$ 由它的体积 v 与质量 m 的乘积来计算：

$$\mathbf{p} = m\mathbf{v}$$

一个动量关于时间的改变率等于所有作用于物体上的外力的和（合力（net force））：

$$\mathbf{F}_{net} = \sum \mathbf{F}_i = \frac{d\mathbf{p}}{dt} = m\frac{d\mathbf{v}}{dt} = m\mathbf{a}$$

当一个物体相对于一个参照点运动并且它的运动不是径直地指向或者离开参照点，就说它有对于参照点的角动量（angular momentum）。角动量矢量 **L** 定义为方位矢量 **r** 与线动量 **p** 的叉积。因此，矢量 **L** 与 **r** 和 **p** 都是正交的。

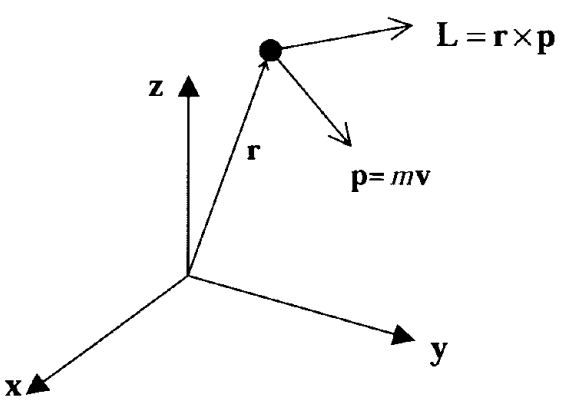


图 2.2.4 角动量 **L** 与 **r** 和 **p** 都正交

当一个外力作用改变角动量时，就说产生了一个旋转力矩（torque）。角动量对时间的导数等于物体上的净旋转力矩：

$$\mathbf{N}_{net} = \sum \mathbf{N}_i = \frac{d\mathbf{L}}{dt} = \mathbf{r} \times \frac{d\mathbf{p}}{dt} = \mathbf{r} \times \mathbf{F}$$

2.2.3 刚体的特性

如果一个物体中的每一个质元对于该物体中任何其他的质元而言都是不可平移或旋转的，该物体就称为一个刚体（真正的刚体在自然界中是不存在的；任何一个物体，无论有多么硬，当它被扰动或旋转的时候，都会稍微有些变形。变形重新分配质量并改变惯性张量（inertia tensor），使运动变得更复杂）。

刚体有两个特性使得它们的运动更易于处理，一个就是它们的质心是固定的。当一个刚体旋转的时候，其内的每一个小质元 m_i 都有关于质心 $\mathbf{r}_{cm}$ 的角动量。该物体相对于它的质心的总角动量（在世界空间中）是所有这些无穷小部分的和：

$$\mathbf{L}_{cm} = \sum \mathbf{r}_i \times \mathbf{p}_i = \sum \mathbf{r}_i \times (m_i \mathbf{v}_i)$$

这里 $\mathbf{r}_i$（也是在世界空间中）是从 $\mathbf{r}_{cm}$ 到 m_i 的矢量。m_i 的速度由下式给出：

$$\mathbf{v}_i = \boldsymbol{\omega} \times \mathbf{r}_i$$

我们可以写为：

$$\mathbf{L}_{cm} = \sum m_i \mathbf{r}_i \times (\boldsymbol{\omega} \times \mathbf{r}_i) = -\sum m_i \mathbf{r}_i \times (\mathbf{r}_i \times \boldsymbol{\omega}) = -\sum m_i \mathbf{r}_i^* \mathbf{r}_i^* \boldsymbol{\omega}$$

这里：

$$\mathbf{r}_i^* = \begin{bmatrix} 0 & -r_{iz} & r_{iy} \\ r_{iz} & 0 & -r_{ix} \\ -r_{iy} & r_{ix} & 0 \end{bmatrix}$$

把它代入并相乘得：

$$\mathbf{L}_{cm} = \sum \begin{bmatrix} m_i(r_{iy}^2 + r_{iz}^2) & -m_i r_{ix} r_{iy} & -m_i r_{ix} r_{iz} \\ -m_i r_{ix} r_{iy} & m_i(r_{ix}^2 + r_{iz}^2) & -m_i r_{iy} r_{iz} \\ -m_i r_{ix} r_{iz} & -m_i r_{iy} r_{iz} & m_i(r_{ix}^2 + r_{iy}^2) \end{bmatrix} \boldsymbol{\omega}$$

$$= \begin{bmatrix} \sum m_i(r_{iy}^2 + r_{iz}^2) & \sum -m_i r_{ix} r_{iy} & \sum -m_i r_{ix} r_{iz} \\ \sum -m_i r_{ix} r_{iy} & \sum m_i(r_{ix}^2 + r_{iz}^2) & \sum -m_i r_{iy} r_{iz} \\ \sum -m_i r_{ix} r_{iz} & \sum -m_i r_{iy} r_{iz} & \sum m_i(r_{ix}^2 + r_{iy}^2) \end{bmatrix} \boldsymbol{\omega}$$

这个由和组成的对称矩阵称为惯量张量 $\mathbf{I}$，这里

$$\mathbf{I}=\begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

对角线元素称为旋转惯量（moments of inertia），非对角线元素称为惯量积（products of inertia）。对具有连续质量分布的刚体，该和可以转化为积分：

$$I_{xx}=\lim_{m_i\to 0}\sum m_i(r_{iy}^2+r_{iz}^2)=\int(r_y^2+r_z^2)\rho(\mathbf{r})dV$$

$$I_{xy}=\lim_{m_i\to 0}\sum -m_i r_{ix}r_{iy}=-\int r_x r_y\rho(\mathbf{r})dV$$

等等。关于质心的角动量现在能够根据惯量张量给出：

$$\mathbf{L}_{cm}=\mathbf{I}\boldsymbol{\omega}$$

关于这一点的事实是，矢量 $\mathbf{r}$ 被指定为关于世界坐标意味着惯性张量依赖于物体的方位，每当物体旋转时必须重新计算。然而，我们可以通过对角化（diagonalizing）惯量张量来避免在每次旋转后不得不对积分重新求值。矩阵的对角化包括将其转换为非对角线元素都为零的一个基。该基是惟一的，并且由矩阵的特征向量（eigenvectors）组成。关于这个基的对角线元素称为矩阵的特征值（eigenvalues）。

通常，矩阵的特征值和特征向量不一定是惟一的，也不一定是实型的。幸运的是，对称矩阵的特征值和特征向量总是实型的并且相互正交（参见[Lang87]）。惯量张量的规一化特征向量称为刚体的主轴（principal axes），并且特征值称为主旋转惯量（principal moments of inertia）。对于物体的主轴，惯量张量简化为：

$$\mathbf{I}_p=\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

而且积分简化为：

$$I_{xx}=\iiint(y^2+z^2)\rho(x,y,z)dxdydz$$

$$I_{yy}=\iiint(x^2+z^2)\rho(x,y,z)dxdydz$$

$$I_{zz}=\iiint(x^2+y^2)\rho(x,y,z)dxdydz$$

这里 x、y 和 z 是物体的本地坐标。对于刚体，这些积分仅需计算一次，并且世界空间中的惯量张量由下式给出：

$I=RI_pR^T$ （参见[Baraff97a]）

I 的逆为：

$$I^{-1}=RI_p^{-1}R^T$$

这里：

$$I_p^{-1}=\begin{bmatrix}\frac{1}{I_{xx}} & 0 & 0\\ 0 & \frac{1}{I_{yy}} & 0\\ 0 & 0 & \frac{1}{I_{zz}}\end{bmatrix}$$

对于一个具有恒密度的矩形盒子形刚体，通常能很精确地近似主旋转惯量。盒子的主轴与它的边是平行的，于是主旋转惯量变为：

$$I_{xx}=\frac{M}{12}(d_y^2+d_z^2),\quad I_{yy}=\frac{M}{12}(d_x^2+d_z^2),\quad 且I_{zz}=\frac{M}{12}(d_x^2+d_y^2)$$

这里 d_x，d_y 和 d_z 分别是盒子在 x、y 和 z 上的大小[Baraff97a]。关于不规则形状物体惯量张量的计算的更完整的讨论见[Mirtich96]。

为了计算一个刚体的旋转运动，我们需要知道角速度怎样关于时间变化。对角速度与角动量的原始关系求导数，有（参见[Baraff97b]）：

$$\mathbf{N}_{net}=\frac{d\mathbf{L}}{dt}=\frac{d}{dt}(\mathbf{I}\boldsymbol{\omega})=\frac{d\mathbf{I}}{dt}\boldsymbol{\omega}+\mathbf{I}\frac{d\boldsymbol{\omega}}{dt}=\boldsymbol{\omega}\times(\mathbf{I}\boldsymbol{\omega})+\mathbf{I}\frac{d\boldsymbol{\omega}}{dt}$$

它最终给出：

$$\frac{d\boldsymbol{\omega}}{dt}=\mathbf{I}^{-1}\left[\mathbf{N}_{net}-\boldsymbol{\omega}\times(\mathbf{I}\boldsymbol{\omega})\right]$$

现在我们可以来讨论描述一个刚体的平移和旋转运动的微分方程了。关于质心的平移运动满足下面的关系：

$$\frac{d\mathbf{r}}{dt}=\mathbf{v}$$

$$\frac{d\mathbf{v}}{dt}=\frac{1}{m}\mathbf{F}_{net}$$

而旋转运动可描述为：

$$\frac{d\mathbf{R}}{dt}=\boldsymbol{\omega}^*\mathbf{R}$$

$$\frac{d\boldsymbol{\omega}}{dt}=\mathbf{I}^{-1}\left[\mathbf{N}_{net}-\boldsymbol{\omega}\times(\mathbf{I}\boldsymbol{\omega})\right]$$

这些方程合起来称为刚体运动的牛顿—欧拉方程。现在有了这些方程，让我们探讨一些求它们的积分的简单方法。

2.2.4 求运动方程的积分

假定物体的初始位置是 $\mathbf{r}_0$，下一个位置 $\mathbf{r}_1$ 能通过下面的关系近似：

$$\mathbf{v} = \frac{d\mathbf{r}}{dt} \approx \frac{\mathbf{r}_1 - \mathbf{r}_0}{\Delta t}$$

解出 $\mathbf{r}_1$ 可得：

$$\mathbf{r}_1 = \mathbf{r}_0 + \mathbf{v}\Delta t$$

这一方法称为欧拉积分（Euler integration），是对初值问题的积分解法中最简单的方法。同样的技术可以用来求余下的动态变量的积分：

$$\mathbf{v}_1 = \mathbf{v}_0 + \frac{1}{m}\mathbf{F}_{net}\Delta t$$

$$\mathbf{R}_1 = \mathbf{R}_0 + \boldsymbol{\omega}^*\mathbf{R}_0\Delta t$$

$$\boldsymbol{\omega}_1 = \boldsymbol{\omega}_0 + \mathbf{I}^{-1}\left[\mathbf{T}_{net} - \boldsymbol{\omega}\times(\mathbf{I}\boldsymbol{\omega})\right]\Delta t$$

不幸的是，用这种方法求方位的积分生产了错误，而且 $\mathbf{R}$ 必须在每帧被重新正交化。此外，对于较大的角速度，这一积分是很不精确的。一个更好的求方位积分的方法是寻找一个由角速度与时间间隔相乘得到的旋转“矢量”：

$$\Delta\boldsymbol{\theta} = \boldsymbol{\omega}_0\Delta t$$

旋转基经过的角度为：

$$\theta = |\Delta\boldsymbol{\theta}|$$

旋转基所绕的轴为：

$$\hat{\mathbf{u}} = \frac{\Delta\boldsymbol{\theta}}{\theta}$$

则基 $\mathbf{R}$ 可被矩阵与它相乘来旋转：

$$\mathbf{M}_r = \begin{bmatrix} 1-2(y^2+z^2) & 2xy-2sz & 2xz+2sy \\ 2xy+2sz & 1-2(x^2+z^2) & 2yz-2sx \\ 2xz-2sy & 2yz+2sx & 1-2(x^2+y^2) \end{bmatrix}$$

这里 $s = \cos(\frac{\theta}{2})$，且 $(x, y, z) = \hat{u}\sin(\frac{\theta}{2})$，故 $\mathbf{R}_1 = \mathbf{M}_r\mathbf{R}_0$（参见[Watt 2000]）。

若方位存储在一个单位四元数 $\mathbf{q}$ 中，则

$$\frac{d\mathbf{q}}{dt} = \frac{1}{2}\boldsymbol{\omega}\mathbf{q}$$

这里 $\boldsymbol{\omega}$ 是纯四元数 $\omega_x\mathbf{i} + \omega_y\mathbf{j} + \omega_z\mathbf{k}$。方位现在能够由下面公式求积分得到：

$$\mathbf{q}_1 = \mathbf{q}_0 + \frac{1}{2}\boldsymbol{\omega}\mathbf{q}\Delta t$$

在该方案中的每一步后规一化 $\mathbf{q}$ 对于预防“偏差”是很重要的（参见[Baraff97a]）。

虽然欧拉方法是求微分方程积分的最简单方法，但也是最不精确和最不稳定的方法。如果角速度变得太大，角速度将以指数规律增大到无穷大。一种简单的避免爆炸的削减方法是，每一帧都用一个比1稍微小一点（可能是0.999）的比例因子乘以角速度。这种方法是有效的，但是它有一些使人恼火的结果，那就是它将使旋转减速趋于终止。更先进的积分技术的讨论参见[Derrick97]、[Gerald99]和[Hairer93]。

2.2.5 参考文献

[Baraff97a] Baraff, David, “An Introduction to Physically Based Modeling: Rigid Body Simulation I—Unconstrained Rigid Body Dynamics,” www.cs.cmu.edu/~baraff/pbm/pbm.html, 1997.

[Baraff97b] Baraff, David, “An Introduction to Physically Based Modeling: Rigid Body Simulation II—Nonpenetration Constraints,” www.cs.cmu.edu/~baraff/pbm/pbm.html, 1997.

[Chow95] Chow, Tai L., *Classical Mechanics*, John Wiley & Sons, Inc., 1995.

[Derrick97] Derrick, William R., and Grossman, Stanley I., *A First Course in Differential Equations with Applications*, third edition, West Publishing Company, 1987.

[Gerald99] Gerald, Curtis F., and Wheatley, Patrick O., *Applied Numerical Analysis,* sixth edition, Addison Wesley Longman, Inc., 1999.

[Hairer93] Hairer, E., Norsett, S. P., and Wanner, G., *Solving Ordinary Differential Equations I: Nonstiff Problems,* second edition, Springer-Verlag, 1993.

[Lang87] Lang, Serge, *Linear Algebra*, third edition, Springer-Verlag, 1987.

[Mirtich96] Mirtich. Brian, “Fast and Accurate Computation of Polyhedral Mass Properties,” *Journal of Graphics Tools* (vol. 1, no. 2): pp. 31–50, 1996.

[Watt2000] Watt, Alan, *3D Computer Graphics*, third edition, Addison-Wesley, 2000.

2.3 三角函数的多项式逼近

Eddie Edwards

随着硬件的发展，我们解决特殊问题的方法也在变化。例如三角函数sin、cos 和 arctg，在过去要快速地计算它们我们想都不敢想；我们通常会去查表。使用一个表有其自身的缺陷（量化误差是主要问题），但是它很快，或者至少，它曾经很快。

现在，CPU 的速度增长得好像比 RAM 的速度快得多。实际的随机存取特别慢，因为许多类型的 RAM 是为超高速缓冲存储器线补充（cache line refill）而不是为个别的字的存取（RDRAM 即为一例）优化的。当前，在某些体系结构中，CPU 做一个浮点乘法所用的时间已经从超过 10 个周期减少到了仅仅 1 个周期。现在 CPU 远比 RAM 快，因此我们可以来重新估价一下查表法的前提。在访问一个单独的存储单元的时间内，我们通常能完成大量的计算工作，而且相比任何接近同一程度的算法，这些计算免受量化误差之苦。也许我们应该考虑计算出正弦函数和余弦函数的值，而不仅仅在一个表中进行查找了。

这一建议提出了一个很有趣的问题："怎样做？"常用的计算复杂函数的解决方案是通过多项式逼近；即寻找可以逼近我们想要的函数的一个多项式，然后将 x 的值代入这个多项式并得到一个该函数的近似值。由于现在乘法的代价非常低廉，这好像能成为一种快速的函数求值方法。本文的第一部分中，详细地描述了多项式逼近是如何工作的，以及怎样为你自己的用途来操纵多项式。

计算函数的问题现在转化为寻找一个好的多项式以用于逼近的问题了。这其实是对能给出好结果的"魔术"数——多项式系数的寻找。

获得这些数值的最有名的方法是查看函数的泰勒级数（Taylor series）。泰勒级数是一个与该函数相等的无穷多项式（对于 x 的某一范围）。如果我们截断泰勒级数，可以得到一个有限多项式，假定它为函数的一个好的逼近。本文稍后将用实例说明泰勒级数如何工作并讨论它们的局限性。

有一些名气不大的泰勒级数的替代方法。本文的主要目的是阐明其中的一个技术：拉格朗日级数。与泰勒级数相比，它有一些特殊的优势。拉格朗日级数可以完全排除逼近中的某些误差，平均起来给出的结果的精确性提高了很多倍。

2.3.1 多项式

多项式就是关于变量（x）的幂次项的和，每一个幂被一个系数相乘。多项式的标准写法如下：

$$a[0]+a[1]*x+a[2]*x*x+a[3]*x*x*x+\cdots a[d]*pow(x,d)$$

数 $a[]$称为多项式的系数，数字 d 是多项式的次数。我们可以用 C++语言说明这些成分：

```
float Poly::Evaluate(float x)
{
  float  powx = 1;
  float  sum = 0;

  for (int n = 0; n <= d; n++)
  {
    sum += a[n] * powx;
    powx *= x;
  }

  return sum;
}
```

这是最显然和最直接的用程序来计算多项式的方法，除此之外还有许多其他替代方法。这些可替代方法在某些方面依赖于多项式的因式分解。在此不推荐替代方法，有下面两个原因：

- 分解因式常导致对每个系数做除法，这意味着系数对舍入误差是敏感的（尤其是在单精度浮点数中）。
- 用因式分解计算有一个包含所有乘法的关键路径，而用简单方法计算有一个只包含一个乘法和一个加法的关键路径。故简单方法在 CPU 上流水线处理更好，这是很重要的。

例如，下面这样的因式分解是一个不需要除法的替代方法：

$$a[0]+x*(a[1]+x*(a[2]+x*(a[3]\cdots)))\cdots)$$

这一方法的求值过程如下面 C++代码所示：

```
float Poly::Evaluate(float x)
{
  float  sum = a[d];

  for (int n = d - 1; n >= 0; n--)
  {
    sum = sum * x + a[n];
  }

  return sum;
}
```

注意每次迭代乘法都以前一次的迭代为基础，故循环不能被中止和流水线处理。这一方法的确有一个优势，即只需要一个累加寄存器，所以它对于 x86 体系结构的标量 FPU 可能是一个好的选择。

2.3.2 定义域和值域

一个函数的定义域是它能被调用的范围。值域是在它的定义域上的返回值的范围。你可能会认为这些多项式的定义域是无穷的，并非如此！

近似多项式以下形式的项的和：

$$a[n]*pow(x,n)$$

这些项中的每一项本身必须是一个浮点数，所以一定在一个浮点数的范围内，即大约在 0 到 2^127（忽略符号，且假定采用 IEEE 单精度格式）。

如果我们取该多项式的 2 基对数，得到：

$$\log 2(a[n]*pow(x,n)) = \log 2(a[n]) + n*\log 2(x) < 127$$

它给出了一系列的 *log2*(*x*)必须服从的不等式。非常明显，它不是无穷的！例如，如果多项式的次数为 10 并且所有系数的值都为 1，我们有：

$$\log 2(x) < 12.7$$

得到 *x* 的一个有效的定义域为−6000 到 6000。

一个三角函数的值域通常是众所周知的。例如，正弦函数的值域为−1.0 到 1.0。另外一些函数的值域是无穷的，例如当自变量趋近于 90° 时，正切函数值会变得越来越大。

我们现在遇到了一个典型的数值精确性问题。假定想用一个 10 次多项式来计算 *sin*(6000.0)。从定义域分析知，浮点数项将变得非常大：一直到刚好小于 2^127，但是我们知道最后的结果应该在−1.0 和 1.0 之间。最后的结果取决于最后的加法的次序，而且因为每个加数非常大，它们必须刚好互相抵消以给出期望得到的较小的数字。

不幸的是，单精度浮点数格式只存储 23 位精度，这意味着对于像 2^127 这样大的数，根本就没有在 0.0 到 1.0 之间的精度。事实上，它的最低位的值是 2^104！因此对一个数量级为 1.0 的数值来说，其计算误差可以达到 2^100，实际上相当于说我们对答案一无所知！（事实上，这就像说，“虽然知道键盘就在我的桌子上，计算结果却告诉我它竟然是在猎户星座附近的一颗行星上”。）

可以证明这对于正弦和余弦函数并不成问题，因为它们是周期函数。如果你想知道 *sin*(6000.0)，可以重复减去 2**pi* 直到它落在一个较好的范围内（−*pi* 到+*pi*），然后再应用多项式。一个快速的做法如下：

（1）用 65 536/2**pi* 乘 *x* 的值。

（2）将 *x* 转换为整型。

（3）左移 16 位（在一个 32 位处理器上）。

（4）算术右移16位（以扩展符号）。

（5）将 x 转换回浮点型。

（6）用 2**pi* /65 536 乘 x 的值。

这一方法给出了 x 的一个在−*pi* 到+*pi* 之间的16位校正值。

你也许想只需要使用第3步结束后的数值就够了，在此时所有高位都在一个计算机字的高位字节上。这一格式是一种表示转数的定点格式，具有32位小数和0位整数，因此如果将两个角度相加在一起就可以得到所期望的结果了，而不需要任何取模运算（实际上，计算机的有限字长替你做了取模运算）。

这一度量单位称为度（*rotation*），以我的经验，不同的公司给了它的名字也不同，可说是最易混淆的术语！由于该数字是旋转的数目，作为一个无符号32位定点小数存储，名称"度"看来最为贴切。

当你需要用一个弧度且需在剩下的时间内用度的时候，在弧度和度之间转换通常是最高效的。

1. 改变定义域

前面的例子展示了怎样能够改变函数的变量的度量单位。例如，正弦函数取弧度，但是通过预乘 2**pi*/65 536，可以使其把度作为一个带符号16位小数（s.15, 1:0:15）。

事实上我们可以通过改变多项式的系数内在地改变我们的函数和单位。假设我们有一个取弧度为单位的函数，但是想让它取度数。只要这样做就可以了：

```
float mult = (2.0 * pi) / 360.0;
return SinePoly::Evaluate(x * mult);
```

或者我们可以改写 Evaluate 函数：

```
float Poly::Evaluate(float x)
{
  float mult = (2.0 * pi) / 360.0;
  float powx = 1;
  float powmult = 1;
  float sum = 0;

  for (int n = 0; n <= d; n++)
  {
    sum += a[n] * powx * powmult;
    powx *= x;
    powmult *= mult;
  }
}
```

下面是一个小技巧：根据这一代码，用 *a*[*n*] * *powmult* 替换每一个 *a*[*n*]，然后能够使用原始的代码来求具有新单位的函数的值。所以我们如下做：

```
void Poly::ChangeUnits(float old_units, float new_units)
{
```

```
    float mult = old_units / new_units;
    float powmult = 1;

    for (int n = 0; n <= d; n++)
    {
      a[n] *= powmult;
      powmult *= mult;
    }
  }
```

当然这一方法只有当 *new_units* 不为零的时候才起作用。

使用这一技术时要非常小心。如果新的值域太大（可能是 360），我们就会遇到与上一节讨论过的同样的指数问题。这个方法不能用来产生一个直接取 16 位小数的度数为单位的多项式，因为函数值将会上溢。然而，它能通过一个 4.0 左右的因子（依赖于多项式的次数）来稍微地改变一个多项式的值域。

我们也可以在定义域中增加一个偏移量，但是这种处理更为棘手。它包括用（x + *off*）的值代替 x 代入多项式中。

例如，以$(x + 2)$代替 x 代入多项式$(1 + x*x)$中，得到$(1 + (x+2)*(x+2))$，它简化为$(5 + 4*x + x*x)$。因此，在 $x = 1$ 处多项式的值为 10，它是旧的多项式在 $x = 1 + 2 = 3$ 处的值。这一代换如果用手工来做将很繁琐，随书光盘中的 **OffsetDomain( )**方法能用来机械地做这项工作。

如果你确实想改变定义域，建议在调用多项式计算方法之前先对 x 简单地用手工做一下。该方法增加了一两个周期，但非常健壮而且很简单。

2. 改变值域

与定义域相比，值域的处理是很容易的，而且结果也会如愿以偿。为了改变一个多项式的输出，仅需像下面这样用因子乘每一个系数：

```
void Poly::ChangeOutputUnits(float old_units, float new_units)
{
  float mult = new_units / old_units;

  for (int n = 0; n <= d; n++)
  {
    a[n] *= mult;
  }
}
```

这个办法很有效，因为：

$mult*(a[0] + a[1]*x + a[2]*x*x + ...) = mult*a[0] + mult*a[1]*x + mult*a[2]*x*x + ...$

为了偏移输出，可简单地将偏移量加到 $a[0]$上，因为：

$offset + a[0] + a[1]*x + ... = (offset + a[0]) + a[1]*x + ...$

所以，你可以内在地改变一个多项式的值域，并且对于最终的计算程序也没有多余的代价。

2.3.3 偶多项式和奇多项式

当你终于在计算机上处理多项式的时候，将发现许多系数变得非常小——例如，1.546*2^-80。你有可能问："这是正确的系数吗？或者系数实际上应该为零？"这一问题的答案可以从偶多项式与奇多项式的分析得来。

一个偶函数满足：

$f(-x) = f(x)$

换句话说，它的图像是关于直线 $x = 0$ 对称的。一个奇函数满足：

$f(-x) = -f(x)$

即，它的图像是反对称的。在三角学中，正弦函数是一个奇函数，而余弦函数是一个偶函数。

并非所有的函数都是偶函数或是奇函数。例如（$x+1$）就既不是偶函数也不是奇函数。

一个多项式是由函数 1、x、$x*x$、$x*x*x$ 等的和构成的。这些函数中的每一个都或是奇函数或是偶函数。

$1 = 1$ 是偶函数

$x = -(-x)$ 是奇函数

$x*x = (-x*-x)$ 是偶函数

$x*x*x = -(-x*-x*-x)$ 是奇函数

所以：

如果 n 是偶数，x^n 是偶函数；如果 n 是奇数，x^n 是奇函数。

很显然，如果一个多项式包含的都是 x 的偶次幂，则多项式本身是偶函数；如果一个多项式包含的都是 x 的奇次幂，则多项式本身是奇函数（这就是最初术语奇函数和偶函数的由来）。如果一个多项式既包含偶次幂，又包含奇次幂，则它既不是奇函数也不是偶函数。

现在的重点是：如果一个多项式逼近于一个偶函数，则多项式本身应该为偶多项式。如果你的多项式表明 $sin(x)$与$-sin(-x)$不相等，则它毫无用处，因为它们是相等的！

如果能够通过分析说明一个系数应该为零（即使一个程序告诉你系数是 1.546*2^-80），就应该将其设置为零。这是校正值，而且伪非零系数是浮点数的舍入误差引起的结果。把不正确的值保留下来将导致难以预料的结果。

这隐含了如果函数是偶函数，当 n 为奇数时 $a[n]$为零；如果函数是奇函数，当 n 为偶数时 $a[n]$为零。如果你假定的 $sin(x)$的逼近多项式的 $a[2]$不是零，这个逼近就是错的。

2.3.4 泰勒级数

泰勒级数来源于一个非常简单的过程，可以用来复制多项式，基于简单的数学计算和求导数操作。我们已经看到了计算，现在应当来看看求导数了。幸运的是，多项式的求导非常简单。

当我们求一个多项式的导数时，可以得到：

$a[1] + a[2]*2*x + a[3]*3*x*x + a[4]*4*x*x*x + ... + a[d]*d*pow(x, d-1)$

每一个系数都乘以它所在项的幂，然后将幂的值减 1。这一过程可描述为：

```
void Poly::Differentiate()
{
  for (int n = 1; n <= d; n++)
  {
    a[n-1] = a[n] * n;
  }
  a[d] = 0;
  if (d > 0) d--;
}
```

注意多项式的导数中 *a[0]*（次数为 0）的值是 0（次数也为零），它说明了前述例子代码中的“特殊情况”。

让我们再次求导，得：

$a[2]*2 + a[3]*3*2*x + a[4]*4*3*x*x + ...$

每当我们求导一次，多项式的次数就降低 1 次。现在让我们看看每一步中 *a[n][0]*的值。（这里，第一个数组下标表示求导的阶数，从零阶导数开始。）

$a[0][0] = a[0]$

$a[1][0] = a[1]$

$a[2][0] = 2*a[2]$

$a[3][0] = 3*2*a[3]$

$a[4][0] = 4*3*2*a[4]$

$a[n][0] = n! * a[n]$

每个系数轮流变为 a[0]，并且被 n！相乘。

通过调用 Evaluate(0) 可以从多项式对象获得 *a[0]*；通过调用 Differentiate( )，可以将每个系数轮流变为 *a[0]*。所以，通过调用这两个函数，我们可以分离出多项式的系数：

```
void Poly::CopyPoly(Poly* p)
{
  float nfact = 1;

  d = 0;
  a[0] = p->Evaluate(0);
  p->Differentiate();

  while (!p->IsZero())
  {
     d++;
     nfact *= d;
     a[d] = p->Evaluate(0) / nfact;
     p->Differentiate();
  }
}
```

IsZero 是一个函数，该函数告诉我们多项式是否处处为零（即对所有的 *x*，Evaluate(x) =

0.0)：

```
bool Poly::IsZero()
{
  return ((d == 0) && (a[0] == 0));
}
```

需要注意的重要事项是，我们由老的多项式仅仅通过操作 IsZero( )，Evalute( )和 Differentiate( )建立新的多项式，而从不寻找多项式的次数或它的任何系数。

这一点是很有趣的，因为在数学上，这三个操作对于一个更广的对象类比仅对于多项式更有意义。这个更广的对象类在学术上通称为无限微分函数（infinitely differentiable functions），并且包含了所有你很可能熟悉的函数。事实上只有奇异函数，如真正的分形函数和 Dirac delta 函数（它包含了一个在 x=0 处的无穷大）在实践中会有麻烦。另一个有问题的函数类是那些有不连续性和锐角转角的函数类（等价于导函数中的不连续性），但我们将在稍后看一些处理它们的方法。

用面向对象术语说，Poly 类本身是 DifferentiableFunction 的一个子类，它有虚方法 IsZero( )，Evaluate( )和 Differentiate( )。

这就是泰勒级数如何计算的——或者更恰当地说，泰勒级数的称为“关于 x =0 的泰勒展式”的子集是如何计算的。下面的函数计算泰勒级数：

```
void Poly::MakeTaylorSeries(DifferentiableFunction* f, float pt)
{
  float nfact = 1;

  d = 0;
  a[0] = f->Evaluate(pt);
  f->Differentiate();

  while (!f->IsZero())
  {
    d++;
    nfact *= d;
    a[d] = f->Evaluate(pt) / nfact;
    f->Differentiate();
  }

  OffsetDomain(pt);
}
```

该函数计算 $f(x-pt)$的级数，然后偏置定义域与之相对应。你可能会问为什么要使用 pt 的值而不是 0，答案是一些函数在 x = 0 没有意义明确的值（即，$1/x$ 在那儿没有定义），所以我们避开该特殊点以预防出现无穷大问题。

实例：正弦函数和余弦函数

sin 函数的导函数是 cos 函数，而 cos 函数的导函数是−sin，所以我们可以用函数指针来

定义 TrigFunction 类，它的行为和这两个函数类似：

```
class TrigFunction : public DifferentiableFunction
{
public:
  TrigFunction()                { fptr = sin; sign = 1; }

  bool   IsZero()               { return false; }
  float  Evaluate(float x)      { return sign * float(fptr(double(x))); }
  float  Differentiate()
  {
    if (fptr == sin)
    {
      fptr = cos;
    }
    else
    {
      fptr = sin;
      sign = -sign;
    }
  }

private:
  double (*fptr)(double); // stdlib math function sin or cos
  float   sign;           // sign of function -1 or +1
};
```

不幸的是，虽然 MakeTaylorSeries 函数接受一个该类的对象，但却没有返回值，因为导数永远也不会是零。我们遇到了一个问题！

围绕这种进退两难的局面，有几种方法：

- 指定 MakeTaylorSeries 将返回的多项式的最高次数。
- 指定最小系数的值。系数“很可能”变得越来越小，因为 $n!$ 变得越来越大，一旦最后的系数太小的时候，我们可以终止这一程序。

第一种方法保证有效，而第二种仍有可能失败。为什么呢？因为系数有可能并不是变得越来越小。这一概念的数学分析超出了本文的讨论范围，但是可以确信无疑的是，你将处理的大多数函数系数将会变得越来越小。不过，你应该总是设置一个次数的最大限制，以防万一。

2.3.5 截断的泰勒级数

正弦函数的泰勒级数是一个无限次数的多项式。可惜我们只能计算（乃至存储）一个有限次的多项式。事实上，对于一个游戏而言，一个小的次数才非常理想。

当我们截断该级数的时候，会产生一个误差（除了通常的浮点误差之外），因为我们忽略的那些项确实对最终的结果有影响。可以证明，当 x 增加的时候，误差也增加，这也是限制多项式定义域的另外一个好的理由。

如果你喜欢，可以在数学上分析该误差，但是在此我们不准备深入这一点。事实上最好是做一些实际实验：

（1）在你的估计中取样误差，使用双精度 *sin*( )和 *cos*( )函数来比较。得到平均误差和最大绝对误差。作为一个单凭经验的方法，得到小于 *1/p* 的误差，这里 *p* 是屏幕对角线的像素数目。对于 640×480 的控制台游戏，*p* 大约是 800；对于 1 280×1 024 的 PC 游戏，*p* 可以大于 1 600。

（2）建立一个大的，慢速旋转的充满屏幕的子图形。用 sin 函数来旋转它，观察如抖动或尺寸的缩放等现象。有些人也许会嘲讽道“嚯，你可以旋转一个子图形了！”，别理这些刺耳的嘲笑。

对于单精度浮点数和一个*−pi/2* 到*+pi/2* 的值域而言，取 *sin* 和 *cos* 的前 5 个系数就能做得很好。此方法在当前硬件水平上也很快——事实上，当前的矢量 FPU（浮点部件）芯片是用微编码在单片上实现这一级数的。然而，采用这一值域是一种费力的做法，因为在级数被使用之前，需要对角度做一些相当繁琐的处理——硬件设计者似乎遗忘了这一点。

如果将 *sin* 和 *cos* 函数的 5 系数级数扩充到*−pi* 到*+pi*，可以得到一个大约 1/300 的误差。不幸的是将得到下面这些值：

sin(0) = 0.0

sin(pi) = 0.003

sin(−pi) = −0.003（显然，因为 sin 是奇函数）

当角度从*+pi* 变化到*−pi* 时，这一问题在旋转子图形的测试中作为“抖动”暴露了出来。误差是 1/300，但是当越过边界的时候，误差会加倍，所以旋转矢量从 *y = −0.003* 跳到 y = +0.003——一个 0.006 的跳变。在 640×480 分辨率下，这一现象相当显著。

关于泰勒级数，除了增加多项式的次数以外，做什么都不能改善这个问题。另一方面，与仅使用泰勒级数相比，还有更多的多项式逼近方法。

2.3.6 拉格朗日级数

拉格朗日级数是我为一类使用拉格朗日公式得到的近似级数而起的名字。与泰勒级数不同，对于一个特定的函数，并非只有一个惟一的拉格朗日级数，而是有一个级数族。我们可以根据想让哪些结果完全正确，以及哪些结果允许与正确值有一定的偏离而从中进行选择。

例如，我们可以要求 sin 值在以下常用点的值是精确的：

sin(−pi) = 0.00000

sin(−pi/2) = −1.00000

sin(0) = 0.00000

sin(pi/2) = 1.00000

sin(pi) = 0.00000

如果我们对这些点的逼近是完全正确的，将可以知道：

- 当对象旋转 90、180 或 270° 的时候它的大小将不会增加，完全与原始大小相等。
- 当正弦函数越过 360° 边界的时候，它将会经过 0，因此在该点将保持连续，且不表现出任何抖动。

拉格朗日证明的一个定理说明，对于给定的 N 个点，有惟一的 $N-1$ 次多项式能够刚好通过所有的点。如果我们计算这个多项式，就得到一个具有所列出特性的逼近。

假定想找到一个正弦函数的 9 次多项式。我们要求该多项式是惟一的，于是必须选择 10 个点，在这些点正弦函数的值必须完全正确。很遗憾，这点无法满足，因为我们应该有一个关于 $x=0$ 对称的点的分布。由于有一个点在 $x=0$，必须有奇数个的点才能达到目的。如果我们在一侧放置一个多余的点，就不能满足另一侧有与之对应的点了。

为了得到一个奇数的点数，我们使用一个 10 次多项式。幸运地，我们知道正弦函数是奇函数，所以知道 $a[10]=0$，所以 10 次的多项式与 9 次的多项式相同。

点的选择方案有许多（无限多！），所以我们选择那些众所周知的点。也可以代之以选择均匀分布的点。仅有的临界点在−180° 和+180°，因为这些点确保了在边界的连续性并因此无抖动。

注意通过选择点，我们假定对于函数没有先验知识。例如，我们不假设它是可导的。这可以成为一个有用的特色，例如用一个多项式逼近一个抽样波形；但是它也可能成为一个大问题，因为我们可能遗漏函数图像的重要的“特征”。也就是说，在选择你的数据点之前，要确信你知道你的函数的外表特征（换句话说，图像本身）是什么。

1. 计算拉格朗日级数

拉格朗日级数是作为一个多项式（每个点有一个不同的多项式）的和得到的。在本节的说明中，我们的数据点是 $x[0]$，……，$x[d]$，并且在这些点想得到的函数值为 $y[0]$，……，$y[d]$。

首先，考虑简单多项式：

$(x-x[n])$

每个多项式都是一次的，一共有（$d+1$）个。当 $x=x[n]$ 时，第 n 个多项式为零，并且当 x 取其他值时，第 n 个多项式都非零。

如果我们将这些多项式都乘在一起，就得到一个（$d+1$）次的多项式。这个多项式在每个数据点的值都为零，因为在该点有一个因式为零。例如，它在 $x[2]$ 为零，因为其中的一个因式为 $(x-x[2])$。

现在，如果我们改为将除了一个之外的多项式乘在一起，就得到了一个 d 次的多项式，该多项式除了在我们省略的那点外的所有点的值都为零。这是因为对于数据点 $x[m]$ 来说，它没有因式（$x-x[m]$）。在该数据点的值由在乘积多项式中以 $x[m]$ 取代 x 给出。

$$c[m]=(x[m]-x[0])(x[m]-x[1])...(x[m]-x[m-1])(x[m]-x[m+1])...(x[m]-x[d])$$

下一步用 $y[m]/c[m]$ 乘以乘积多项式，是一个称为规一化的过程。它的结果是得到一个新的多项式，该多项式在数据点 $x[m]$ 等于 $y[m]$，但是在其他数据点为 0（注意用 $y[m]/c[m]$ 乘以多项式等价于用因子改变输出单位，正如在前面所讨论的）。

这一概念给出了拉格朗日级数的组成部件。为了得到拉格朗日级数本身，需要产生所有的（$d+1$）个项并将它们加在一起。在每一个数据点 $x[n]$，该多项式中除了一项的值为 $y[n]$ 之外，其余的项的值都为零。我们因此构造了一个多项式，如我们所愿，该多项式在每一个数据点 $x[n]$ 的值都等于 $y[n]$。

例如，假设有一个简单的三点的情形：

$x[0] = -1$

$y[0] = 4$

$x[1] = 0$

$y[1] = 2$

$x[2] = 1$

$y[2] = 4$

简单多项式为 $(x-x[0])$、$(x-x[1])$ 和 $(x-x[2])$，它们分别与 $(x+1)$、$(x-0)$ 和 $(x-1)$ 相等。让我们考虑数据点 $x[0]$，将除了 $(x-x[0])$ 之外的所有的简单多项式相乘，得到：

$(x-0)*(x-1) = x*x-x$

计算它在 $x[0]$的值，得：

$c[0] = -1*-1-(-1) = 2$

我们想让 $y[0]$等于 4，故用 4/2 = 2 乘以这个多项式（改变输出单位），得：

$2*x*x-2*x$

现在插入 $x[0]$、$x[1]$ 和 $x[2]$来检查一下：

$x[0] = -1 : 2*(-1)*(-1)-2*(-1) = 4$

$x[1] = 0 : 2*0*0-2*0 = 0$

$x[2] = 1 : 2*1*1-2*1 = 0$

于是的确如期望的那样，该多项式在第一个数据点的值为 4，在其余的所有数据点的值为 0。

对其他两个点完成这项工作，得到下面每个数据点的多项式：

$x[0] : 2*x*x-2*x$

$x[1] : 2-2*x*x$

$x[2] : 2*x*x+2*x$

可以验证，每一个多项式在它自己的数据点等于 $y[n]$，在其他点等于 0。

最后，我们将这三个多项式加在一起，得：

$2*x*x-2*x+2-2*x*x+2*x*x+2*x = 2*x*x+2$

可以快速检验一下，这个多项式与每一个点的数据是相匹配的。

随书光盘中的成员函数 MakeLagrangeSeries 做了这项困难的工作，所以你就不必去做了。手工构造一个 9 次拉格朗日级数将是非常繁琐的！

注意从 MakeLagrangeSeries 得到的结果中有一些误差——普通的浮点误差，如你所料。计算是非常复杂的，并且还迭代，于是误差就积累起来了。如果你正在建立一个拉格朗日级数，它必须或者是偶函数或者是奇函数，确保忽略的奇或偶系数的值必须为 0。可以通过调用成员函数 ForceOdd 或 ForceEven 做到这一点。

2. 与泰勒级数的比较

对于一个 9 次 sin 函数或一个 10 次 cos 函数，我们已经看到了在 360° 边界的连续性问题。拉格朗日级数没有这样的问题。通常，拉格朗日级数给了对逼近多项式的特征的更出色的控制，诸如连续性和必须正确的特殊点的值。

随书光盘上的程序 main.cpp 给出了一个值域 $-pi$ 到 $+pi$ 上的平均误差和最大绝对误差的比

较。泰勒级数显示出的最大误差为 1/150，平均误差为 1/1 500。把它们与拉格朗日级数相比较，拉格朗日级数有一个最大误差为 1/11000，而平均误差仅有 1/77000（注意这些误差是用双精度浮点数测定的）。

泰勒级数是对一个函数的精确匹配，为你提供无穷多的可取项。拉格朗日级数被所取的数据点的数目内在限制。这一事实似乎暗示了泰勒级数是一个更好的逼近，但是上面这些数据证明并非如此。在这种情况下，拉格朗日级数的最大误差仅为泰勒级数的平均误差的 1/7！

3. 关于数字的备忘录

当你处理用浮点符号表示的精确的数字时，printf( )的输出经常会有一些误差，仅能给出前面几位精确的十进制数值，而浮点数不是基于十进制的（这意味着一个十进制转化形式最多是对浮点值的一个近似）。在 Poly 类的 Print( )成员函数中，你能以 16 进制值打印出系数。这样确保了当你把数字移动到自己的代码中的时候，它们正好是原始的值。在 C++中做这个工作可能很棘手，但是 Print( )中包含的一个技巧能用来另辟蹊径，从 16 进制形式得到浮点数字。

还要注意为了从这些代码中得到最好的结果，应该定义 Number 类为 double 型，即使你准备在游戏中使用单精度数。当打印这些系数的时候，可以简单地转化为浮点数。

2.3.7 不连续性处理

在实践中，不连续性的出现相当普遍，而且无论是泰勒级数还是拉格朗日级数都没有很好地处理它们。多项式通常是平滑的，于是不连续点也变得平滑了，这不能令人满意。幸运的是，这些问题的外围工作还是相当容易的。

一个常见的间断函数是正切函数。在+/−90°，正切函数趋于无穷并且接着跳到负无穷。这个不连续性可以用两种基本方法来处理。首先，你可以简单地用 *sin* 除以 *cos* 来计算正切函数。这种方法完全避免了该问题的出现（只要你做一个被零除的检查！）。第二，你可以限制你的正切函数的值域正好是在连续的范围内——即，−90° ~ +90°。对于拉格朗日级数，这意味着仅在这个范围内选取点。

如果有一个函数，该函数有一个跳跃间断点，常可以通过减去一个阶梯函数来去掉不连续点。这样就得到了一个连续函数，它可以用泰勒级数或拉格朗日级数来逼近。然后把阶梯函数加回来以恢复原始函数。如果你的函数有一个尖点，它将被多项式弄平，但是你可以在做近似之前减去一个三角函数来产生一个平滑函数。

这些推断都基于对基本算法为处理特殊情形而做的简单调整。这是一个能在其中为对付各种可能情况而训练创造力的领域。

2.3.8 结论

在本文中，我们对一些十分基础的数学做了快速浏览，并看到了一些非常重要的结果。我们已经看到多项式怎样能以多种不同的方法构造，如何对它们进行复制而不用直接访问系数。根据这些方法，我们发现了泰勒级数方法。之后我们发现泰勒级数对于很多应用不太适

当，并在拉格朗日级数中看到了一个有力的替代办法。

无论是泰勒级数还是拉格朗日级数都不是万能的。如果你在自己的代码中使用这些“数字处方（numerical recipes）”，奉劝你做一下实验并查看一下结果。确信你的代码总是如你所期望的那样工作，并且提防原始函数中的不连续点。

包括这些忠告在内，你现在有了逼近更多函数的方法了。请明智地使用它！

2.4 为数字稳定性而利用隐式欧拉积分

Miguel Gomez

选择一种求初值问题积分的方法是编写交互式应用软件的重要部分。显式的欧拉积分由于易于实现，似乎是一个不错的选择。不幸的是，这一方法具有不稳定问题，其误差以指数规律增长并且解很快就变为无穷大。本文描述了隐式的欧拉方法——一个有效且高稳定性的积分器。用两个例子来说明这种技术：指数式衰减（exponential decay）和阻尼弹簧方程（damped spring equation）。最后，讨论了导出隐式解的困难之处。我们假定读者熟悉微积分学、经典力学和微分方程理论。

2.4.1 求初值问题的积分及稳定性

初值问题就是具有初始条件的微分方程。对于有些方程，能找到一个解析解并用来计算一个物体的轨迹；然而在大多数的情形下，没有现成的解析形式可用，必须采用数值积分求解。

有许多不同的数值积分方法，选择的依据是具体应用的要求。在一些系统的某些条件下，解中的误差以指数规律增殖并且解接近于无穷。这种情况称为不稳定性（instability），并且说这种积分方法在这此条件变得不稳定。例如，你可能会将一个汽车的悬吊系统作为一个附在刚体上的阻尼弹簧建模。如果对于时间间隔来说该弹簧过于僵硬，积分得到的解将变得不稳定并且很快变为无穷。

一种改善稳定性的方法是将时间间隔再细分。一种更好的改善稳定性的方法是用 Runge-Kutta 或一些类似的方法，不再细分时间间隔；然而，连这些方法在某些点也会变得不稳定。这就是为什么我们要寻求一种简单有效的方法来保证稳定性，而不考虑方程的参数或时间间隔的大小。

2.4.2 显式的欧拉方法

作为第一个例子，让我们讨论下面的初值问题：

$$\frac{dx}{dt} = -kx$$

$$x(0) = x_0$$

分析解为：

$$x(0) = x_0 e^{-kt}$$

对每个数据点计算这个指数函数很简单、精确、但效率极低。大多数 CPU 用硬件要花费大约 30 个周期来求初等函数的值。如果要 CPU 必须用软件模拟这一过程，就太倒霉了！

幸运的是，有更有效的方法来求解的值。我们可以使用一阶导数的有限差分近似（difference approximation）：

$$\frac{dx}{dt} \approx \frac{x_1 - x_0}{\Delta t}$$

这里 x_1 是经过一时间间隔 Δt 的解。解出 x_1，得：

$$x_1 = x_0 - \frac{dx}{dt}\Delta t$$

如果导数是以 x_0 处的值来计算的，我们有：

$$x_1 = x_0 - kx_0\Delta t = x_0(1 - k\Delta t)$$

这种方法称为显式欧拉积分方法。该方法仅有一个减法和两个乘法，即使一个定点处理器也能处理。

虽然显式欧拉方法非常有效，但它并不稳定。我们可以用下面的论据证明它的不稳定性：由于解是一个指数函数，它必须随时间变化衰减到零，即意味着：

$$x_1 \le x_0$$

将其代入并整理得到下面的关系：

$$k\Delta t \le 2$$

故如果 $k = 1$ 且 $\Delta t = 2$，则 $x_1 = -x_0$，且解是稳定的，虽然不十分精确。若这一条件没有被满足，则 x 很快趋于无穷，且我们正在模拟其运动的物体也消失了。正如前面提及的，这种欧拉方法能通过仅仅减少时间间隔变得更为稳定，且需要更多数据点的计算。然而，若 k 选得太大，时间间隔的再细分代价将高得惊人。

2.4.3 隐式欧拉方法

刚才描述的方法是一种显式的方法，因为它完全依赖于先前求解的值来计算导数。相反，隐式欧拉方法计算 x_1 处而不是 x_0 处的导数值（见图 2.4.1），得到：

$$x_1 = x_0 + (-kx_1)\Delta t$$

解出 x_1 得：

$$x_1 = \frac{x_0}{(1 + k\Delta t)}$$

将它代入到稳定条件得：

$$0 \le k\Delta t$$

这非常重要。我们已经找到了一个简单、有效的方法，无论 k 或Δt 多大，它都是稳定的（参见[Hairer93]）。

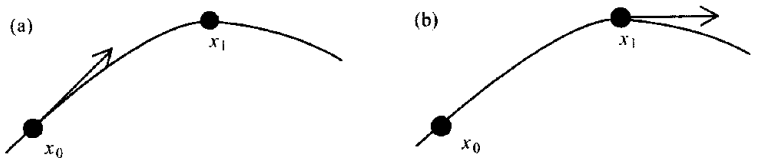


图 2.4.1 显式欧拉方法计算 x_0 处的导数(a)，隐式欧拉方法计算 x_1 处的导数(b)

让我们看一个更为复杂的例子。假设一个物块 m 通过一个弹簧连在一个固定装置上，当物块不在它的平衡位置上时，受到 k N/m 的推力或拉力。我们同样假定有一个与它的速度以常数 b 成反比的外力，致使系统失去能量。如果我们设置平衡位置在原点($x_{eq} = 0$)，那么描述它的运动的微分方程为：

$$m\frac{d^2x}{dt^2} = -kx - b\frac{dx}{dt}$$

满足初始条件：

$$x(0) = x_0$$

$$x'(0) = v(0) = 0$$

解出二阶导数，得

$$\frac{d^2x}{dt^2} = -\omega^2 x - 2\lambda\frac{dx}{dt},\quad \lambda = \frac{b}{2m}$$

为简单起见，这里我们做了代换$\omega = \sqrt{\frac{k}{m}}$ 和 $\lambda = \frac{b}{2m}$。如果条件$\omega^2 > \lambda^2$ 被满足，则解为：

$$x(t) = x_0 e^{-\lambda t}\cos(\omega t)$$ （参见[Chow95]）

这个二阶微分方程能够被改写为由两个一阶微分方程组成的系统，即：

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -\omega^2 x - \lambda v$$

计算出 x_1 和 v_1 处的导数得：

$$x_1 = x_0 + v_1\Delta t$$

$$v_1 = v_0 - \omega^2 x_1 - \lambda v_1$$

最后，求出 v_1 得：

$$v_1 = \frac{v_0 - \omega^2 x_0 \Delta t}{1 + \lambda \Delta t + (\omega \Delta t)^2}$$

它对于所有正的 k，b 和 Δt 值都是稳定的（见图 2.4.2）。

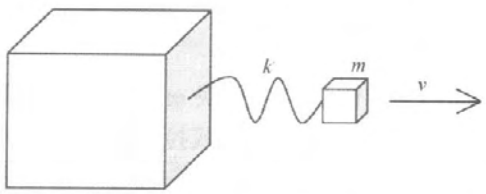


图 2.4.2 一个物块附在一个具有阻尼系数为常数 k 的弹簧上。阻尼与速度成反比

2.4.4 不准确性

隐式欧拉方法的准确性不如相对的显式欧拉方法。即使没有阻尼，解也会慢慢地失去它的全部能量。在大多数的应用中，我们更关心的是效率而不是准确性，故这通常并不是问题，而是应谨记的东西。

2.4.5 寻找隐式解

找到一个对于 x_1 的隐式计算未必容易。例如，如果弹簧的力是与 x_2 而不是与 x 成比例，我们就得到一个关于 x_1 的二次关系。对于一个不能解出 x_1 的系统，可能需要像牛顿方法那样的数值根的搜索（参见[Gerald99]）。在这样的情况下，隐式积分器对于实时应用软件是不实用的。

2.4.6 结论

当稳定性成为着重考虑的问题时，请记住可用隐式欧拉方法。有许多其他类型的隐式方法能够适宜于不同的需要。进一步深入的讨论请参阅[Gerald99]、[Hairer96]和[Hairer93]。

2.4.7 参考文献

[Chow95] Chow, Tai L., *Classical Mechanics*, John Wiley & Sons, Inc., 1995.

[Derrick97] Derrick, William R., and Grossman, Stanley I., *A First Course in Differential Equations with Applications*, third edition, West Publishing Co., 1987.

[Gerald99] Gerald, Curtis F., and Wheatley, Patrick O., *Applied Numerical Analysis*, sixth

edition, Addison Wesley Longman, Inc., 1999.

Hairer93] Hairer, E., Norsett, S. P., and Wanner, G., *Solving Ordinary Differential Equations I: Nonstiff Problems*, second edition, Springer-Verlag, 1993.

[Hairer96] Hairer, E., and Wanner, G., *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, second edition, Springer-Verlag, 1996.

2.5 小波：理论与压缩

Loïc Le Chevalier

我们通常将小波（wavelets）与压缩方法联系在一起。但是小波这个词同时涉及了数学理论、压缩方法和数据分析工具。这是一个有着许多应用的有力方法。

2.5.1 原理

无论哪一种小波应用（压缩，分析等），出发点都是一群 N 值：标量或矢量。我们由这 N 个值建立了如图 2.5.1 中所示的一棵树。在树的每一级，我们计算两个值的平均值，它成为下一级的值。这样得到的树是一棵二叉树（binary tree），该树有 $N(N+1)/2$ 个结点，与各自的平均值相对应。这棵树一旦建成，就可用来对初值进行重构。注意那些换算值是没有被压缩的。重构使得我们可以在简化形式中保持所有的信息，然而压缩常导致信息的丢失。小波的主要原理就是这样一个可逆原理：在任何时候，我们都能回溯到初值而不丢失信息。

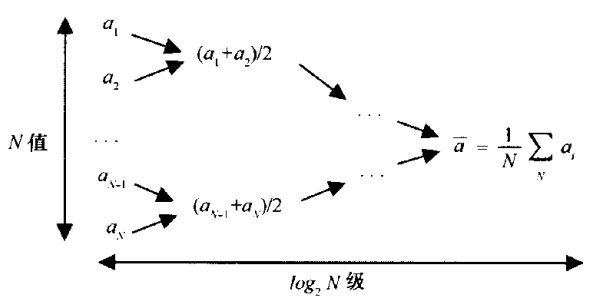


图 2.5.1 建立树

在我们建立的树的每个分支上添加一个新的值，该值等于分支的两个端点间的差值——即两个值间的距离，这两个值由一个分支相连且相差 1 级。如图 2.5.2 所示，从一个结点出发的（两个）量值大小是相反的，因为这两个值与平均值是等距的（例如，2 和 8 的平均值是 5，从 2 到 5 的距离是−3，从 8 到 5 的距离是 3）。

现在，无需换算了。正相反，从一个 N 值矢量发展到一棵有 $N(N+1)/2$ 个结点的值的树结构，为分支加上 $N(N+1)/2-1$ 个值！因而下一步包括选择一棵树的级数，即一个介于 1 和 $log_2 N$ 之间的换算级 p。这样一级就由结点的值以及由它产生的更低的分支组成（见图 2.5.3）。

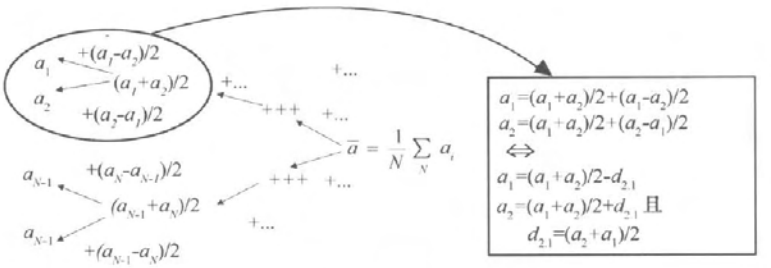


图 2.5.2 计算距离

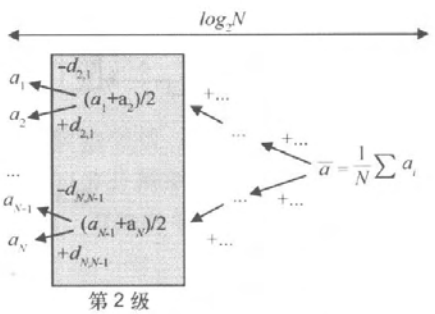


图 2.5.3 选择换算级

一旦换算级 p 选定了，由 $0<p<log_2N$，通过从第 1 级到第 p 级的顺序计算（见图 2.5.4），我们可以得到第 p 级的压缩值。这样就有了一个不丢失信息的原理，它的线性耗费（linear cost）是一个关于 N 的函数。

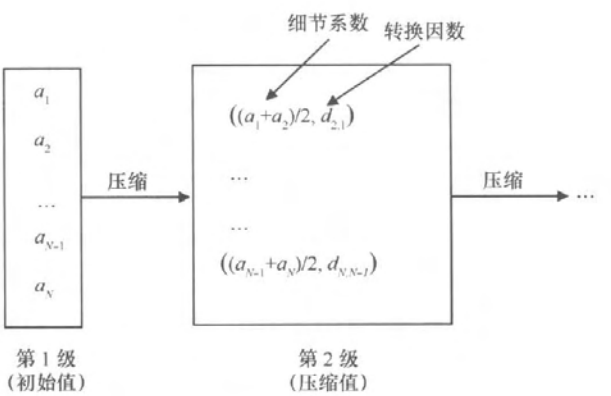


图 2.5.4 压缩值

一旦进行了换算，我们就通过逆操作恢复初值。这种恢复（restitution）操作有它自己的线性耗费 N。重要的是注意从第 p 级开始恢复初始值，我们必须保存这一级的转换因数（scale factor），以及所有从第 1 级到第 p 级的细节系数（detail coefficient）。

2.5.2 一个实例

例如，假定有一个线性编码的图像，为简单起见，设分辨率为4×4像素16灰度级。因而初值是一个有16个分量的标量向量，每个分量的值介于0到15之间。图2.5.5展示了与该图相关的树。

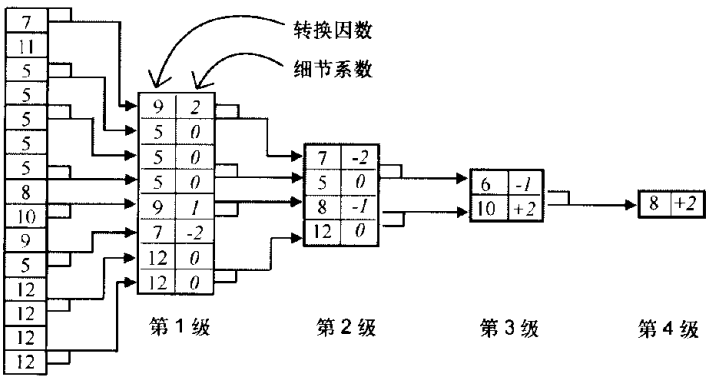


图2.5.5 一棵实例树

这样，如果我们选择换算第3级，就保持一对换算因数（6，10）以及所有第3、2、1和0级的细节系数。注意第0级没有等级系数，它们都隐含地为零。

如果我们现在想回溯到16个初值，我们通过加上或减去第3级的系数计算第2级的转换因数，等等，直到我们回溯到第0级因数，即初始值。

2.5.3 应用

前述构建原则的确常被图像金字塔（image pyramid）用来mipmap，事实上它说明了小波压缩方法。这些小波也称为哈尔小波（Haar wavelets）。它们非常适于处理离散的值，如标量向量（理论上我们对它们一无所知）。但是很多其他类型的小波对于处理连续的或其他函数是很有用的。这些类型小波的理论基础与前面描述的原则非常接近。

小波有许多应用。最早且最著名的是图像压缩（image compression）。然而，与上述原理相比，利用小波压缩和解压需要一个附加的阶段。前述原理是无损的，换句话说，整个图像可以从任何一级被重新精确组合。但是这对于压缩是毫无意义的，故我们能舍弃一些值以丢失可能的最小的信息。通过这样最小化误差，得到了非常令人信服的结果。例如，一个有几十万字节的图像能被压缩到仅有几千字节而只有极少的失真。简而言之，具有极少细节的区域被大幅度地压缩，而有很多细节的区域则压缩很少。因而，失去的信息在视觉上就几乎感觉不到了。图2.5.6是一个压缩比为116的例子：原始图像大小为6.25MB，而压缩后只有53KB。

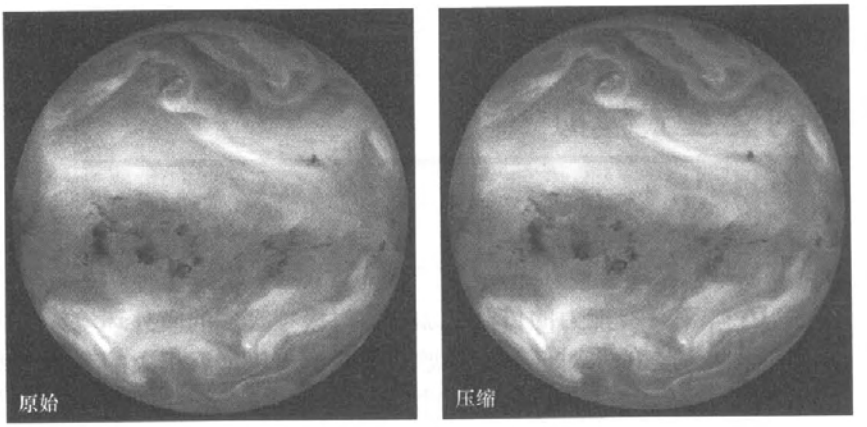


图 2.5.6 一个图像压缩的例子

小波的另一个著名的应用是数据分析（data analysis），是一种非周期函数傅立叶变换的替代方法。我们也可以引用图像的多分辨率（multiresolution）成果或 3D 模型来调整（无限放大等）图像或 3D 模型（LOD、 再细分等）的细节水平。

2.5.4 参考文献

An Introduction to Wavelets, Institute of Electrical and Electronics Engineers, www.amara.com/IEEEwave/IEEEwavelet.html.

The Wavelet Organization meta-site, www.wavelet.org.

2.6 水面的交互式模拟

Miguel Gomez

随着计算能力的日益增加，实时模拟真实的室外环境终于成为可能。动态的水对任何室外游戏场景都可以增添极大的美感。本文描述了一个简单、有效的方法来模拟真实的水面波动。通过使用二维波动方程的一个中心差分近似，每个点仅用几个算术运算就能模拟水的水平运动。文中也给出了其他方法的简要讨论，还描述了基于物理的浮力与阻力模型。最后，还讨论了渲染过程的实现与优化思想。

2.6.1 二维波动方程

一个水面可以被看作是一个绷紧伸展的弹性膜，其重力可以忽略不计。当无限小的部分移动时，它们的直接邻近点会施加线性“弹力”（表面张力）来最小化它们之间的距离。由于水平方向的力是相等的，粒子仅在 z 方向运动。关于时间和空间的垂直位置可以由偏微分方程来描述：

$$\frac{\partial^2 z}{\partial t^2} = c^2\left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2}\right)$$

这里 c 是波越过水面传播的速度。如果边界条件是齐次的（*homogeneous*）（即边缘不上下运动）且水面的初始 z 速度为零，则对于一个 $L\times L$ 大小的正方形水域的通解为：

$$z(x,y,t) = \frac{2}{L}\sum_{m=1}^{\infty}\sum_{n=1}^{\infty} A_{mn}\sin\left(\frac{m\pi x}{L}\right)\sin\left(\frac{n\pi y}{L}\right)\cos(c\omega t)$$

$$\omega = \frac{\pi}{L}\sqrt{(mx)^2 + (ny)^2}$$

系数 A_{mn} 由计算下面的积分得到：

$$A_{mn} = \frac{2}{L}\int_0^L\int_0^L f(x,y)\sin\frac{m\pi x}{L}\sin\frac{m\pi y}{L}dxdy$$

这里 $f(x, y)$是水面的初始形态（参见[Trim90]）。如果水面模拟为 z 个值（高度域）平均分隔的栅格，如图 2.6.1，前述积分就变为离散的且可以用快速傅立叶变换（*FFT*）算法计算（参见[Press92]）。

我们可以通过仅计算级数中重要的项来得到近似解，虽然这个方法直接且稳定，但是效率很低。在大多数 CPU 上，单独计算一个三角函数要

用大约 30 个周期。沿着 x 和 y 仅取前 3 个频率最大的值，每个时间间隔每一点就需要计算 9 个正弦函数，对于交互式应用中较大的栅格其代价更高得惊人。这个严重的缺点促使我们去寻找一个更加有效的数值解法。

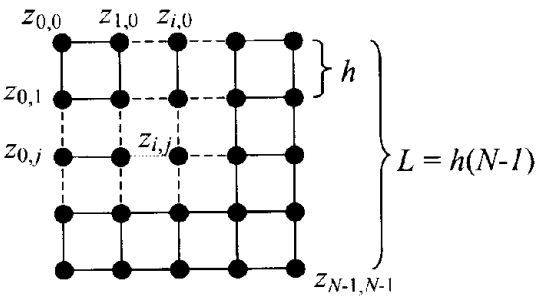


图 2.6.1 一个有 N 个点每条边都用来近似水面的 $L \times L$ 高度域

采用中心差分来模拟偏导数可以得出：

$$\frac{z_{i,j}^{n+1} - 2z_{i,j}^{n} + z_{i,j}^{n-1}}{\Delta t^2} = c^2\left(\frac{z_{i+1,j}^{n} + z_{i-1,j}^{n} + z_{i,j+1}^{n} + z_{i,j-1}^{n} - 4z_{i,j}^{n}}{h^2}\right)$$

值 $z_{i,j}^{n}$ 是第（i，j）个栅格位置在时刻 t_0 的高度。值 $z_{i,j}^{n-1}$ 和 $z_{i,j}^{n+1}$ 分别是在时刻 $t_{-1} = t_0 - \Delta t$ 和 $t_1 = t_0 + \Delta t$ 的高度。解出 $z_{i,j}^{n+1}$ 有：

$$z_{i,j}^{n+1} = \frac{c^2\Delta t^2}{h^2}\left(z_{i+1,j}^{n} + z_{i-1,j}^{n} + z_{i,j+1}^{n} + z_{i,j-1}^{n}\right) + \left(2 - \frac{4c^2\Delta t^2}{h^2}\right)z_{i,j}^{n} - z_{i,j}^{n-1}$$

这一关系简单说明了运动 z_{ij} 仅受它最邻近点的影响（图 2.6.2）。由于栅格的间距是常数，h^2 的倒数可以预先计算得到，就只剩下乘法、加法和减法了。此外，若 c 在栅格间不变，所有的系数都可以预先计算，而且后继的 z 值能仅由两个乘法和 5 个加法来计算！（且如果你实在想节省，可以让 $h^2 = 2c^2\Delta t^2$ 除去中间项。该方法对 c 还是对 h 进行限制，依赖于应用。）

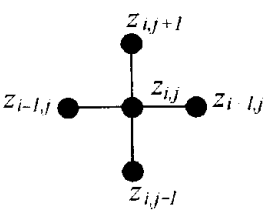


图 2.6.2 点 z_{ij} 的水平运动仅受它最邻近的点的影响

乍看起来，似乎有必要为在时刻 t_{-1}，t_0 和 t_1 的 z 值存储 3 个栅格；然而，如果 z^{n-1} 被 z^{n+1} 适当代换，只有两个栅格是必须的。在传递结束的时候，指向 z^{n+1} 和 z^{n} 的值的内存指针被交

换。在下一次迭代中，z^n 变成了 z^{n-1}，且 z^{n+1} 变成了 z^n。下面的代码片断展示了这一节省空间的诀窍是如何实现的。

```
//precalculate coefficients
const float A = (c*dt/h)*(c*dt/h);
const float B = 2 - 4*A;
long i, j;

//edges are unchanged
for( i=1 ; i<N-1 ; i++ )
{
    for( j=1 ; j<N-1 ; j++ )
    {
        //integrate, replacing z[n-1] with z[n+1] in place
        z1[i][j] = A*( z[i-1][j] + z[i+1][j] + z[i][j-1]
                  + z[i][j+1] ) + B*z[i][j] - z1[i][j];
        //apply damping coefficients
        z1[i][j] *= d[i][j];

    }
}

//swap pointers
Swap( z.pData, z1.pData );
```

2.6.2 边界条件：岛屿和海岸线

本质上，水域通常不是正方形的。河流、湖泊和海洋有着不同斜度的不规则海岸线，而且岛屿也可能存在于水域内部。如果堤岸非常陡峭甚至于是垂直的，波浪从海岸线反射回来就几乎不会失去能量；然而如果堤岸是逐渐倾斜的，波浪的反射波可能会非常弱，或根本没有。如果波浪不是垂直击向岸边的，那么就会以某个角度反射回来。

这些效果能通过用一个本地阻尼系数 $d_{i,j}$ 调节 z^{n+1} 的值来模拟（见前述代码片断）。系数为 1，表明允许在高度值上的自由运动而不损失任何能量；系数 0 则限制在那个位置的水域的所有运动。如果这些系数是根据地形特征来分配和调节的，波浪对海岸线作出的反应就更自然。例如，如果海岸是陡峭的，阻尼系数就应该从 1（水）到 0（陆地）做一个快速的转变。另一方面，如果海岸是逐渐倾斜的，阻尼系数就应该从 1 渐变到 0。实际上，将略小于 1 的阻尼系数用于“湿的”栅格比较好，可产生较少的能量损耗；否则波动会无限地延续下去。

2.6.3 实现问题

1. 不稳定性

前面描述的积分方法是一种显式方法，因为它只用到先前的和当前的 $z_{i,j}$ 值来计算 z^{n+1}。如果条件：

$$\frac{c^2\Delta t^2}{h^2} \le \frac{1}{2}$$

不满足，此积分方法将变得不稳定且后继的 z 值按指数规律增长。

另一方面，一种隐式方法能用来保证稳定性。不幸的是，隐式求解时需要为 z^{n+1} 解一组联立方程。关于隐式积分方法的更深入的讨论参见[Gerald99]、[Hairer93]和[Hairer96]。

2. 并行处理

虽然一些处理器有单指令多数据（SIMD）指令，能并行计算几个浮点值，但存储队列的需求能够减少这些指令对于求解积分的效率（或甚至禁止使用）。能够并行操作 4 个单精度浮点数的处理器通常需要 16 字节队列，故如果 N 不是 2 的幂，行必须被填充。在此不考虑一些非队列的存储器存取（图 2.6.3）。即使 CPU 允许非队列存储器存取，也通常在性能上会有所损失。

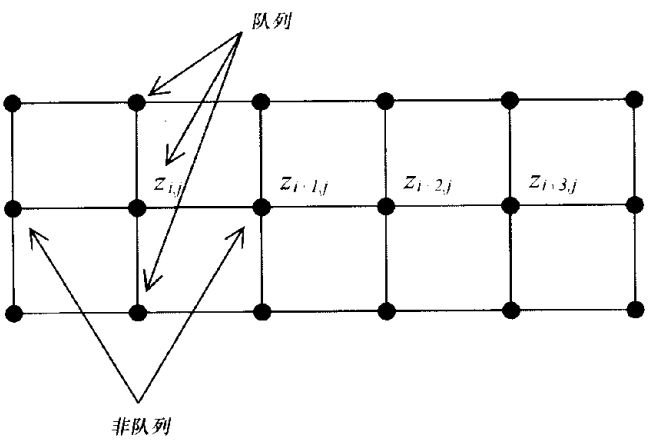


图 2.6.3 尽管栅格的行被填补到确保 16 字节队列，仍有一些数据存取是非队列的

2.6.4 与水面交互

1. 飞溅（splash）

飞溅可以通过在特定位置瞬时移动一个或几个 z 值来创建。作为解的改进，波浪从这个位置发出。这个概念阐明了显式计算通解的另一个优势。如果在 $z(t)$发生任何中断，A_{mn} 的新值必须由一个离散傅立叶变换来计算。

2. 浮力对象

如果物体不能漂浮在水面上，水还有什么用处呢？物体能够漂浮是因为它们的总密度比包围着它们的水的密度小。一个物体上的浮力等于它排出的水的重量。该力实际上是在气压梯度（pressure gradient）的方向上，但是在多数情形下，取水面的法线方向就可以了。

如果船体的外形近似于一组离散的点、法线和区域片（图 2.6.4），浮力能通过对浸没在水下部分的体积进行积分来计算。部分船体排出的水的体积为：

$$\Delta V_k = \Delta A_k (z_{water} - p_{k,z}) \hat{n}_{k,z}$$

这里 z_{water} 是在 $\mathbf{p}_k$ 的双线性插值水面高度（推荐使用双线性插值，因为其他方法可能产生初级或一级间断。它也许是对于规则栅格的最有效的插值法）。在这个位置的浮力为：

$$F_k = \rho \Delta V_k \hat{n}_{water}$$

且转动力矩为：

$$N_k = r_k \times F_k$$

这里 $\mathbf{r}_k$ 是从质心到 $\mathbf{p}_k$ 的矢量。合力与转动力矩可以通过求每一个船体顶点分量的和来计算得到。注意同样只有浸没部分对浮力有贡献。

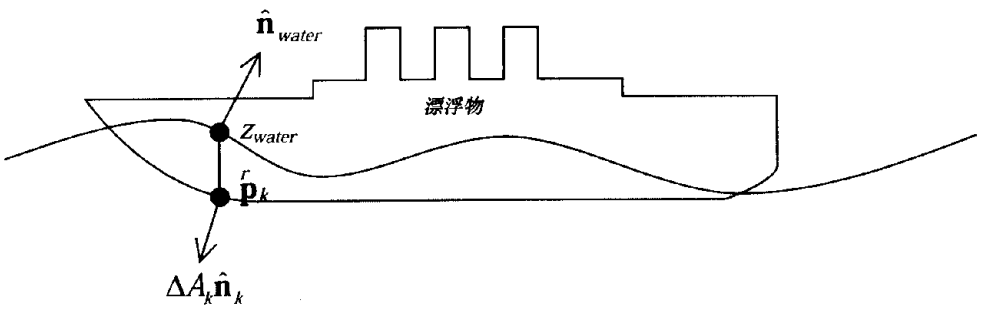


图 2.6.4 漂浮物的外形以一组点 $\mathbf{p}_i$ 近似，这些点“均匀”地分布在它的表面。单位表面法线 $\hat{\mathbf{n}}_k$ 和本地区域片 ΔA_k 也被存储

所需点的数目依赖于物体的形状及想得到的精确度。一个立方体可能需要大约 20 或 30 个点，而一棵有分枝的树可能会需要数百个点来真实地表现。

一个对任意参数化的右手螺旋三维表面的法矢量可以用下面的公式计算：

$$n = \frac{\partial S(u,v)}{\partial u} \times \frac{\partial S(u,v)}{\partial v}$$ （参见[Davis91]）

如果我们将 x 和 y 作为参数，水面可以用下面的矢量描述：

$$S(x,y)_{water} = [x, y, z(x,y,t)]$$

用中心插分近似前一个公式的导数得：

$$\frac{\partial S}{\partial x} \approx \left[1, 0, \frac{z_{i+1,j} + z_{i-1,j}}{2h}\right]$$

$$\frac{\partial S}{\partial y} \approx \left[0, 1, \frac{z_{i,j+1} + z_{i,j-1}}{2h}\right]$$

则位于第 i，j 栅格处的法矢量为：

$$n_{i,j}=\left[-\frac{z_{i+1}-z_{i-1}}{2h},-\frac{z_{j+1}-z_{j-1}}{2h},1\right]$$

将该矢量放大 $2h$ 倍不会改变它的方向，故一个等价的有效法矢量为：

$$n_{i,j}=\left[z_{i-1}-z_{i+1},z_{j-1}-z_{j+1},2h\right]$$

它必须接着规一化。

为了保持物体在水面上像冲浪板一样滑行，可以通过求每一个顶点的分量的和来计算拉力：

$$F_{drag}=\sum -bv_{k,rel}=\sum b\left[v_{water}-\left(v_{cm}+\omega_{cm}\times r_k\right)\right]$$

速度项 $\mathbf{v}_{k,rel}$ 是在 $\mathbf{r}_k$ 处船体相对水的速度。故除高度值之处，如果一个三维速度与每一个栅格区域（矢量场）相关联，水流就载着物体漂浮。

2.6.5 渲染

所有这些理论都是极好的，但是如果看不到结果，又有什么用？下面是一些实现和优化渲染过程的思想：

1. 环境帖图

当你画水的时候，能用 alpha 混合给出透明的外观。然而为了正确画出 alpha 混合三角形，你必须首先从离观众最远的地方画起，而没有 Z 缓冲器的帮助。此外，无论何时一个三角形能通过另一个可见，双倍的混合就会发生。

实际上，光线在水中并不是直线传播的。当它由一种折射率进入到另一种时就会发生弯曲。水也从它的环境反射光线。这种效果可由环境帖图（environment mapping）得到。水面环境帖图包括反射和折射从眼睛发出的光线，以及用一个环境地图表面与它们相交来计算纹理坐标（texture coordinate）（参见图 2.6.5）。于是每一个三角形由这些坐标进行纹理帖图。

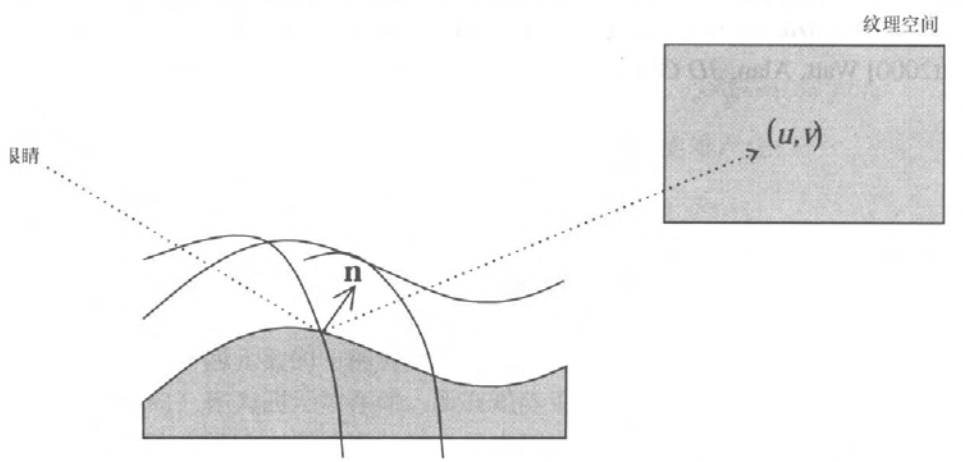


图 2.6.5 反射从眼睛（摄像机）处到环境地图的光线为每个顶点生成了一个纹理坐标（u, v）

如果在实现上非常完美的话，折射环境贴图可以带来震撼的效果；在这一点上，你可以在以后自己去体验。若是只有反射，水会显得太金属化，像液态的汞。更多的环境帖图技术和相关公式见[Watt2000]。

虽然环境帖图给出了极好的视觉效果，但是由于计算的需求，如果没有硬件支持，大规模地使用它也许是不可行的。不过可以在小规模的软件实现中验证一下，足以令人印象深刻了。

2. 细节等级（Level of Detail, LOD）处理

在一个较低分辨率下渲染高度域的远处部分，能极大地提升速度而没有视觉质量的明显降低。然而必须注意要确保顶点的法线与沿着细节等级（LOD）的变化的一致。否则，在环境帖图和照明中就会不连续。地形 LOD 处理的一个极好的自适应四叉树方法参见[Ulrich2000]。

2.6.6 参考文献

[Davis91] Davis, Harry F., and Snider, Arthur David, *Introduction to Vector Analysis*, sixth edition. William C. Brown Publishers, 1991.

[Gerald99] Gerald, Curtis F., and Wheatley, Patrick O., *Applied Numerical Analysis*, sixth edition. Addison Wesley Longman, Inc., 1999.

[Hairer93] Hairer, E., Norsett, S. P., and Wanner, G., *Solving Ordinary Differential Equations I: Nonstiff Problems*, second edition, Springer-Verlag, 1993.

[Hairer96] Hairer, E., and Wanner, G., *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, second edition, Springer-Verlag, 1996.

[Press92] Press, William H., Teukolsky, Saul A., Vetterling, William T., and Flannery, Brian P., *Numerical Recipes in C*, second edition, The Press Syndicate of the University of Cambridge, 1992.

[Trim90] Trim, D. W., *Applied Partial Differential Equations*, PWS-Kent, 1990.

[Ulrich2000] Ulrich, Thatcher, "Continuous LOD Terrain Meshing Using Adaptive Quadtrees," *Gamasutra*, www.gamasutra.com/features/20000228/ulrich_01.htm, 2000.

[Watt2000] Watt, Alan, *3D Computer Graphics*, third edition Addison-Wesley, 2000.

2.7 游戏编程四元数

Jan Svarovsky

四元数对于表现和处理点的 3D 旋转是很有用的。其应用包括骨架动画、逆向运动，以及任何通常的 3D 物理或图像引擎。本文是这样组织的：首先充分阐明四元数以使你能够在自己的 3D 游戏中运用它们，接着逐渐深入其数学基础。

2.7.1 将四元数当作矩阵替换物

完全可以在一个游戏中将所有的旋转矩阵很简单地替换为四元数。它们能够描述在 3D 空间中绕任何轴进行的任何旋转。四元数占用较少的空间，4 个数字胜于 9 个，并且许多操作（如乘法）代价低廉。一些操作如在四元数间的插值也更赏心悦目。当需要一个矩阵（如旋转一个矢量）时，你可以很容易地将四元数转换为旋转矩阵并且再转换回来。

```
// a black-box quaternion type that can replace 3x3 matrices
class Quaternion
        {
private:
    float x, y, z, w; // These will be explained later
public:
    Quaternion Inverse() const;
};
Quaternion quaternion_from_matrix(Matrix33   &mat );
Matrix33   matrix_from_quaternion(Quaternion &quat);
Quaternion interpolate(Quaternion &a, Quaternion &b,
float b_amt);
Quaternion operator *(Quaternion &a, Quaternion &b);
```

一些其他现存的函数可能对矩阵来说更难产生：

```
Vector3 Quaternion::AxisOfRotation() const;
float Quaternion::AngleOfRotation() const;
// rotation that will get you from v0 to v1
Quaternion RotationArc(Vector3 v0, Vector3 v1);
```

四元数的一种典型运用是将所有的矩阵（例如一个动画形象的骨骼走向）作为四元数存储。所有矩阵乘法都被四元数计算取代，而且仅仅是在流水线的末端，当矢量必须旋转进入世界空间或者显示在屏幕上的时候，

才将四元数转换为矩阵。

3×3 的旋转矩阵可以直接由四元数表示。一个四元数和一个平移矢量能够代表编码一个旋转和平移的 4×4 矩阵。

2.7.2 为什么不使用欧拉角

四元数可以避免万向节锁（gimbal lock）。对于一个 3 角度的（侧转(roll)、俯仰(pitch)、偏转(yaw)）系统，总存在某些方向，在这些方向上不可能使用任何简单的 3 值运算来实现（看似简单的）局部旋转。当你试图进行一个局部向左或向右的偏转时，经常会看见结果却是“俯仰”旋转了 90° 。

2.7.3 *X*、*Y*、*Z*和 *W*代表什么

四元数中的 4 个数常表示为（*x*，*y*，*z*，*w*），有一些物理意义。如果我们考虑以角度θ绕轴 *A*（X_A，Y_A，Z_A）旋转的所有旋转矩阵，如图 2.7.1 所示，四元数 *Q* 将是：

$$Q = (s X_A, s Y_A, s Z_A, c)$$

$$s = sin(\theta / 2)$$

$$c = cos(\theta / 2)$$

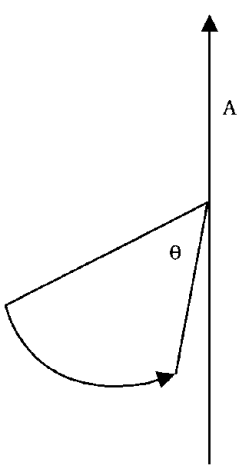


图 2.7.1 以角θ绕轴 *A* 的旋转

这将导致两个结果：很容易求出旋转角（见前述提及的方法），因为它是 *w* 项的反余弦的 2 倍；同样可以很容易地求得旋转轴。

注意每一个旋转矩阵可由两个四元数表示。如果一个普通的旋转由一个轴和一个角定义，每个旋转将有一个具有反向的角和反向的轴的等价的旋转。在四元数的项中，可以让两个四元数的θ相差为 2π（或 360° ）。由于这些项是关于（$\theta/2$）的，这就在 sin 和 cos 项中加上了π：

$$sin(\alpha + \pi) = -sin(\alpha)$$

$cos(\alpha + \pi) = cos(-\alpha)$

这样会在插值时导致问题：两个数值上差异非常大的四元数代表了非常类似的旋转。这个“转向”定位能被看作是任何四元数库实现中额外的测试。

2.7.4 源自什么数学基础

我们这里使用的四元数是通用四元数的一个子集。通用四元数是复数的扩充。复数根据 i 定义，i 是–1 的平方根（它不能用“传统的”数来表示）：

$$i * i = -1$$

虽然 i 与“实”数不同，但是我们可以将其像任何其他变量一样包含在表达式中，它具有非常有趣的特性，那就是其平方等于–1。为某个 a 和 b 构造一个“复”数（$a + bi$），i 的任何倍数和一个实数必须位于 i 的左边。例如，两个复数的乘积为：

$$(a + bi) * (c + di) = a*c + a*di + c*bi + bi*di$$
$$= a*c - b*d + (a*d + c*b)\,i$$

四元数将–1 的平方根的概念扩展到–1 的三个平方根——i、j 和 k：

$$i * i = -1$$
$$j * j = -1$$
$$k * k = -1$$

这三个元素中的每两个之间的乘积很像通常 3D 空间中三个轴的叉积：

$$i * j = -j * i = k$$
$$j * k = -k * j = i$$
$$k * i = -i * k = j$$

由此可以得到四元数的定义，与复数类似，四元数由一个实数和 i、j 和 k 项组成。由于 i、j 和 k 表现得如此像轴，四元数有时被写作一个矢量（此处是 v）和一个标量（s）或一个有 4 个项的矢量。

$$q = w + x\,i + y\,j + z\,k$$
$q = [s\ v]$　　这里 s = w 且 v = [x y z]
$q = [x\ y\ z\ w]$　　注意标量“w”在最后

四元数的加法和乘法以显而易见的方式定义。与“普通的”复数乘法非常类似，四元数乘法的结果是另外一个四元数：

$$q_1\,q_2 = (c_1 + x_1\,i + y_1\,j + z_1\,k)\,(c_2 + x_2\,i + y_2\,j + z_2\,k)$$
$$= (c_1\,c_2 - x_1\,x_2 - y_1\,y_2 - z_1\,z_2) +$$
$$(y_1\,z_2 - y_2\,z_1 + c_1\,x_2 + c_2\,x_1)\,i +$$
$$(z_1\,x_2 - z_2\,x_1 + c_1\,y_2 + c_2\,y_1)\,j +$$
$$(x_1\,y_2 - x_2\,y_1 + c_1\,z_2 + c_1\,z_2)\,k$$

在这个例子中能看到许多称性，在简化表示中更为明显：

$$q_1\,q_2 = (s_1 + v_1)\,(s_2 + v_2)$$
$$= s_1\,s_2 - v_1 \bullet v_2 + v_1 \times v_2 + s_1\,v_2 + s_2\,v_1$$

这里 $a \bullet b$ 是点积且 a×b 是叉积。

几个其他定义也是有用的。一个四元数的范数（norm）为：

$N(q) = x^2 + y^2 + z^2 + w^2$

一个四元数的共轭（conjugate）有两种表示方式：

$q^* = [-x - y - z\ w]$

$q^* = [s - v]$

四元数的乘法逆元素为：

$1/q = q^* / N(q)$

我们用来表示旋转的四元数的子集是单位四元数集，这里$|q| = 1$，或 $x^2 + y^2 + z^2 + w^2 = 1^2$。它们有一个特性，即逆元素等于其共轭。

2.7.5 四元数如何表示旋转

用单位四元数 q 表示一个矢量 $P\ [x\ y\ z]$的旋转，是由创建一个“纯”四元数 p 和其共轭 q 来做到的：

$p = x\,i + y\,j + z\,k$ “纯”意为没有标量项，$w = 0$

$Rot_q(P) = q\ p\ q^*$

为了领会它是如何工作的，要做一些推导。为了不让读者被数学推导吓住，我们首先在此给出结论：考虑以一个角度θ绕轴 A 旋转矢量 P。只要掌握了几何学，你就可以完成这项数学工作了，展开所有的项直到最后一些包含 $cos^2(\theta)$和 $sin^2(\theta)$的项出现。它们能被转化为 $cos(2\theta)$和 $sin(2\theta)$项，并且很快你就能以一个公式而告终，该公式与前述完成的四元数乘法非常类似（参见[Glassner90]）。

这种旋转的表示能用于将四元数转换为一个旋转矩阵。一个旋转矩阵能被看作是单位矩阵经一个四元数的旋转，这里 3×3 单位矩阵是三个矢量（1，0，0），（0，1，0）和（0，0，1）。一个 4×4 的旋转矩阵与 3×3 的情形等价，只是附加了一个额外的行和列，这里除了最后一行最右的元素为 1，其余额外的项都是 0。

2.7.6 参考文献

[Glassner90] Glassner et al, *Graphics Gems*, Academic Press, 1990.

[Downs] Downs, Laura, “Using Quaternions to Represent Rotation,” http://http.cs.berkeley.edu/~laura/cs184/quat/quaternion.html.

2.8 矩阵和四元数之间的转换

Jason Shankel

四元数对于表现 3D 旋转是非常方便的。四元数乘法比矩阵乘法快，而且四元数插值能产生平滑的动画。但是矩阵也有它们的用处。在实践中，对于进行顶点变换矩阵要优于四元数。此外，大多数 3D API 以矩阵形式存储它们的旋转。

本文论证了四元数到矩阵和矩阵到四元数的转换。我们采用四维矢量和矢量 / 标量两种命名法表示四元数。即：

$q = [x, y, z, w] = [w, \mathbf{v}]$

这里 $\mathbf{v} = (x, y, z)$ 是一个三维矢量，w 是一个标量。

我们也使用 q、q'和 q''指定四元数。q'和 q''是与 q 截然不同的四元数，且不应与 q 的一阶和二阶导数混淆。

2.8.1 四元数旋转

令 $q = [w, \mathbf{v}] = [\cos(\theta), \mathbf{u}\sin(\theta)]$是一个四元数，这里 $\mathbf{u}$ 是一个单位矢量。令 $q'=[w', \mathbf{v}']$是一个四元数（不一定是单位矢量），代表一个三维齐性空间的点。

操作 $qq'q^{-1}$ 使 q'以 2θ角绕轴 $\mathbf{u}$ 旋转。证明在[Shoemake94]中给出。

2.8.2 四元数到矩阵的转换

为了把四元数 q 转换到一个等价的旋转矩阵，我们必须将 $qq'q^{-1}$ 表示为一个矩阵操作。

四元数乘法采取下面的形式：

$q'' = [w,\mathbf{v}][w',\mathbf{v}'] = [ww' - \mathbf{v}\bullet\mathbf{v}', \mathbf{v}\otimes\mathbf{v}' + w\mathbf{v}' + w'\mathbf{v}]$

这里⊗是矢量的叉积而•是矢量的点积。

展开得如下的 $[x'',y'',z'',w'']$：

$x'' = yz' - zy' + wx' + xw'$

$y'' = zx' - xz' + wy' + yw'$

$z'' = xy' - yx' + wz' + zw'$

$w'' = ww' - xx' - yy' - zz'$

这个展开式能表示为一个矩阵乘法：

$$\begin{vmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{vmatrix} \begin{vmatrix} x' \\ y' \\ z' \\ w' \end{vmatrix} = L_q q'$$

乘法 $q'' = q'q$ 展开得：

$x'' = y'z - z'y + w'x + x'w$

$y'' = z'x - x'z + w'y + y'w$

$z'' = x'y - y'x + w'z + z'w$

$w'' = w'w - x'x - y'y - z'z$

或者：

$$\begin{vmatrix} w & z & -y & x \\ -z & w & x & y \\ y & -x & w & z \\ -x & -y & -z & w \end{vmatrix} \begin{vmatrix} x' \\ y' \\ z' \\ w' \end{vmatrix} = R_q q'$$

对一个四元数 $q = [\mathrm{w}, \mathbf{v}]$，$q^{-1} = [w, -\mathbf{v}]/\mathrm{N}(\mathrm{q})$。

对单位矢量，$N(q) = w^2 + x^2 + y^2 + z^2 = 1$，故 $q^{-1} = [w, -\mathbf{v}]$。

将 $q = q^{-1}$ 代入 R_q 得到：

$$\begin{vmatrix} w & -z & y & -x \\ z & w & -x & -y \\ -y & x & w & -z \\ x & y & z & w \end{vmatrix} = R_q^{-1}$$

操作 $qq'q^{-1}$ 的等价矩阵能够由连接 L_q 和 R_q^{-1} 得到：

$$\begin{vmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{vmatrix} \begin{vmatrix} w & -z & y & x \\ z & w & -x & -y \\ -y & x & w & -z \\ x & y & z & w \end{vmatrix} =$$

$$\begin{vmatrix} w^2 + x^2 - y^2 - z^2 & 2(xy - wz) & 2(wy + xz) & 0 \\ 2(xy + wz) & w^2 - x^2 + y^2 - z^2 & 2(yz - wx) & 0 \\ 2(xz - wy) & 2(yz + wx) & w^2 - x^2 - y^2 + z^2 & 0 \\ 0 & 0 & 0 & w^2 + x^2 + y^2 + z^2 \end{vmatrix}$$

$x^2 + y^2 + z^2 + w^2 = 1$，故 M 简化为：

$$\begin{vmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(wy+xz) & 0 \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) & 0 \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = M$$

2.8.3 矩阵到四元数的转换

在前面的叙述中，一个旋转矩阵和它对应的四元数的分量之间的关系作为 M 给出。

由 M，我们可以得到以下 6 个关系：

（1）$M_{1,2} + M_{2,1} = 4xy$

（2）$M_{3,2} + M_{2,3} = 4yz$

（3）$M_{1,3} + M_{3,1} = 4xz$

（4）$M_{2,3} - M_{3,2} = 4wx$

（5）$M_{3,1} - M_{1,3} = 4wy$

（6）$M_{1,2} - M_{2,1} = 4wz$

显然，如果知道一个分量，就可以由除法计算其他三个分量。因为任意但不是所有的分量能为零，所以我们要决定哪一个分量有最大的绝对值并用它计算其他的分量。一个单位四元数的最大的分量有一个至少为 1/2 的绝对值。

1. 求解 W

一个矩阵的迹是对角线分量的和。为了决定|w|，我们从计算矩阵 M 的迹出发。对于矩阵 M，它的迹为：

$\mathrm{Tr} = 4 - 4(x^2 + y^2 + z^2) = 4(1 - (x^2 + y^2 + z^2))$

回想一个单位四元数 $q = [w, \mathbf{v}] = [\cos(\theta), \mathbf{v}'\sin(\theta)]$，此处 $\mathbf{v}' = (x', y', z')$是一个单位矢量。因此 M 的迹可以表示为：

由于(x', y', z')是一个单位矢量，$x'^2 + y'^2 + z'^2 = 1$。

M 的迹简化为：

$\mathrm{Tr} = 4(1 - \sin^2(\theta)) = 4\cos^2(\theta) = 4w^2$

或者：

$|w| = \mathrm{Tr}^{1/2}/2$

所以，如果 $\mathrm{Tr} \geq 1$，我们将 $4w = \pm 2\mathrm{Tr}^{1/2}$代入公式（4）、（5）和（6）并解出 x、y 和 z 得：

$x = (M_{2,3} - M_{3,2})/2\mathrm{Tr}^{1/2}$

$y = (M_{3,1} - M_{1,3})/2\mathrm{Tr}^{1/2}$

$z = (M_{1,2} - M_{2,1})/2\mathrm{Tr}^{1/2}$

注意无论采用 Tr 的正根还是负根作为 w 的基都无关紧要，因为 q 和 $-q$ 代表同一个旋转。

2. 求解 X、Y 或 Z

如果$|w|<1/2$，我们可以通过沿着 M 的对角线检查前三个值来决定余下的分量中哪一个

是最大的。假设 $M_{2,2} > M_{1,1}$，可展为：

$$1-2x^2-2z^2 > 1-2y^2-2z^2$$

化简得：

$$-2x^2 > -2y^2$$

或者：

$$|x| < |y|$$

类似的算法适用于其他对角线元素间的比较。所以，矢量$(M_{1,1}, M_{2,2}, M_{3,3})$的最大分量对应于矢量$(x, y, z)$的最大值。

一旦我们获得了$(M_{1,1}, M_{2,2}, M_{3,3})$的最大分量，减去其他两个元素，然后方程就简化为一个单一的项。例如，假设 $M_{1,1}$ 是最大的项：

$$M_{2,2} - M_{3,3} - M_{1,1} = 1-2x^2-2z^2-(1-2y^2-2z^2)-(1-2x^2-2y^2) = 4y^2-1$$

或者：

$$y = \pm(M_{2,2} - M_{3,3} - M_{1,1} + 1)^{1/2}/2$$

一般而言，

$$v_i = (M_{ii} - M_{jj} - M_{kk} + 1)^{1/2}/2 \quad \text{这里} \quad \mathbf{v} = (x, y, z)$$

像对 w 的讨论一样，我们采用哪一个根并不重要。一旦有一个合适的 $\mathbf{v}_i$，我们就可以用代入法解出 $\mathbf{v}_j$ 、$\mathbf{v}_k$ 和 w。

$$\mathbf{v}_j = (M_{ij} + M_{ji})/(4\mathbf{v}_i)$$

$$\mathbf{v}_k = (M_{ik} + M_{ki})/(4\mathbf{v}_i)$$

$$w = (M_{jk} - M_{kj})/(4\mathbf{v}_i)$$

为了从公式（4）、（5）或（6）计算出 w，3 个矢量 $\boldsymbol{i}$，$\boldsymbol{j}$ 和 $\boldsymbol{k}$ 必须是顺序的。即，$\boldsymbol{j} = 1+(\boldsymbol{i}\%3)$ 且 $\boldsymbol{k} = 1+(\boldsymbol{j}\%3)$。

2.8.4 参考文献

[Shomake94] Shoemake, K., *Quaternions*, ftp://ftp.cis.upenn.edu/pub/graphics/shoemake/quatut.ps.Z, May 1994.

[Eberly99] Eberly, David, *Quaternion Algebra and Calculus*, www.magic-software.com/src/graphics/quat/quat.pdf, July 1999.

2.9 四元数插值

Jason Shankel

四元数是复数的四维扩展。（请参看“2.7 游戏编程四元数”中关于四元数和四元数运算的讨论。）本文介绍了4种在四元数对或四元数序列间进行插值的技术(lerp、slerp、squad以及spline)。每种技术的实际推导在本文的结尾部分详述。

2.9.1 四元数计算

在研究四元数插值之前，我们需要先定义一些四元数的计算函数：

设 $q = \cos(\theta) + \mathbf{v}\sin(\theta)$是一个单位四元数（$\mathbf{v}$ 是一个三维单位矢量）。

将复数的欧拉恒等式应用于四元数：

$$q = \cos(\theta) + \mathbf{v}\sin(\theta) = \exp(\mathbf{v}\theta)$$

由这一恒等式，我们可以定义四元数的幂函数：

$$q^t = [\cos(\theta) + \mathbf{v}\sin(\theta)]^t = \exp(\mathbf{v}t\theta) = \cos(t\theta) + \mathbf{v}\sin(t\theta)$$

也可以用这一恒等式表示一个四元数的对数：

$$\log(q) = \log(\exp(\mathbf{v}\theta)) = \mathbf{v}\theta$$

我们可以将 q^t 的导数表示为：

$$(q^t)' = q^t\log(q)$$

应用链式法则，可以表示 $q^{f(t)}$的导数为：

$$(q^{f(t)})' = f'(t)q^{f(t)}\log(q)$$

对具有两个独立变量的函数应用链式法则，我们可以表示 $q(t)^{f(t)}$的导数（为清晰起见，t 省略了）：

$$(q^f)' = f'q^f\log(q) + q'fq^{f-1}$$

2.9.2 四元数插值

四元数能用来表现3D旋转，因而我们可以使用四维矢量插值技术来生成平滑的3D动画。

设 q_0 和 q_1 是四元数。q_0 和 q_1 间的插值通用公式为：

$$q(t) = f_0(t)q_0 + f_1(t)q_1 \quad (0 \le t \le 1)$$

这里$f_0(t)$和$f_1(t)$是标量函数，如：

$$f_0(0) = 1$$

$$f_0(1) = 0$$

$f_1(0) = 0$

$f_1(1) = 1$

1. 线性插值（Linear Interpolation）

线性插值由下式给出：

$\mathbf{lerp}(t; q_0, q_1) = (1 - t)q_0 + tq_1 = t(q_1 - q_0) + q_0$

线性插值不保值，所以如果你将其用作为一个旋转，对结果进行规一化很重要。

线性插值很快，但是它不产生平滑动画。这意味着，即使 t 以常速率变化，在插值过程中，动画也会一会儿加速一会儿减速。虽然这种速度上的变化在某些应用中是可接受的，但是不理想。为了获得四元数间的平滑动画，我们必须采用球状线性插值。

2. 球状线性插值（Spherical Linear Interpolation）

正如三维单位矢量定义了一个球面上的点，单位四元数则定义了一个四维超球面上的点。通过沿着连接两个点之间大弧的插值可以得到平滑动画（见图 2.9.1）。

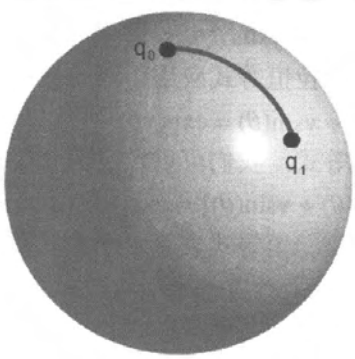


图 2.9.1 球状线性插值

球状线性插值（**slerp**）由下式给出：

$\mathbf{slerp}(t; q_0, q_1) = [q_0\sin(\theta(1 - t)) + q_1\sin(\theta t)]/\sin(\theta)$

这里 θ 是 q_0 和 q_1 间的夹角。

我们可以通过将 q_0 和 q_1 当做四维矢量并计算点积来得到 θ：

$q_0 \bullet q_1 = x_0x_1 + y_0y_1 + z_0z_1 + w_0w_1 = \cos(\theta)$

与 **lerp** 不同，**slerp** 保值，故不需要对结果进行规一化。

如果 $q_0 \bullet q_1 < 0$，则 $\theta > \pi/2$。由于 q 和 $-q$ 代表同一旋转，在这种情况下最好取反 q_0 或者 q_1，使必须沿着其进行插值的角距离最小化。取反向减少了插值过程中的不必要的自旋。

如果 $|q_0 \bullet q_1|$ 接近于 1，我们后退到 **lerp**，因为当 $|q_0 \bullet q_1|$ 趋近于 1 的时候，$\sin(\theta)$趋近于 0。

slerp 的推导见推导 2.9.1。

球状线性插值也能表示为 q_0 和 q_1 的幂函数：

$\mathbf{slerp}(t; q_0, q_1) = q_0(q_0^{-1}q_1)^t$

由此能将导数（**slerp'**）表示为：

$\mathbf{slerp'}(t; q_0, q_1) = q_0(q_0^{-1}q_1)^t\log(q_0^{-1}q_1)$

slerp 的幂函数形式和它的导数被用于导出样条插值。**Slerp** 的幂函数形式的推导见推导 2.9.2。

3. 球状三次插值（Spherical Cubic Interpolation）

slerp 产生平滑动画，但是它常沿着一个连接两个四元数的大弧进行。正如用直线连接一系列的点一样，用 **slerp** 通过一系列四元数插值产生了一个锯齿状的轨迹。在实践中，这意味着你的动画会在控制点突然改变方向。为了通过一系列四元数平滑插值，可采用样条插值（见图 2.9.2）。

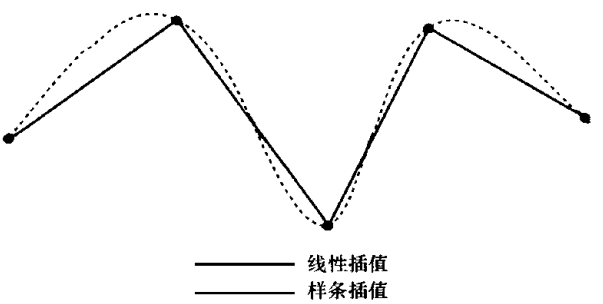


图 2.9.2 线性插值与样条插值的对比

样条插值的基础是球状三次插值，或 **squad**：

$\mathbf{squad}(t; p, q, a, b) = \mathbf{slerp}(2t(1-t); \mathbf{slerp}(t; p, q), \mathbf{slerp}(t; a, b))$

从 p 到 q 的动画并不是沿着连接 p 和 q 的大弧，而是沿着朝向大弧的连接 a 和 b 的曲线。

当两个四元数之间的夹角超过 90° 的时候，为实现 **slerp** 而转化一个输入四元数是很普通的。虽然 q 和 $-q$ 表示同一个旋转，但是是 $\mathbf{slerp}(t; p, q)$与 $\mathbf{slerp}(t; p, -q)$产生的结果并不相同。由于控制点 a 和 b 被选为对 p 和 q 起作用，而不是 $-p$ 或者 $-q$，所以最好不要取反与 **squad** 一起使用的 **slerp** 版本中的输入四元数。

4. 样条插值（Spline Interpolation）

设 $\{q_n, a_n, b_n\}_{(n=0\to N-1)}$为 N 个四元数序列。

设 $S_n(t) = \mathbf{squad}(t; q_n, q_{n+1}, a_n, b_{n+1})$

为了产生一个平滑插值序列，$\{a_n, b_n\}$假设为：

$a_n = b_n = q_n\exp[-(\log(q_n^{-1}q_{n-1}) + \log(q_n^{-1}q_{n+1}))/4]$

样条插值的推导见推导 2.9.3。

2.9.3 示例代码

随书光盘中给出了 **lerp**，**slerp**，**squad** 和 **spline** 插值的实现及四元数指数函数和对数函数。

2.9.4 推导

推导 2.9.1 推导 Slerp

slerp 保值，故两个单位四元数之间的球状线性插值产生的仍是一个单位四元数。给出基本插值函数：

$q(t) = f_0(t)q_0 + f_1(t)q_1$

我们想说明如果约束 $q(t)$ 使 $N(q(t)) = 1$，有：

$f_0(t) = \sin(\theta(1-t))/\sin(\theta)$

为使得表达式更明了，在下面省略了时间变量（$q = q(t)$，$f_0 = f_0(t)$，等）。

设 $q = [xi + yj + zk + w] = f_0q_0 + f_1q_1 = \mathbf{slerp}(t; q_0, q_1)$，设 $\theta = \cos^{-1}(q_0 \bullet q_1)$

由于 q 是一个单位矢量，$x^2+y^2+z^2+w^2=1$。展开 x^2、y^2、z^2 和 w^2 有：

$x^2 = (f_0x_0 + f_1x_1)^2 = (f_0x_0)^2 + 2f_0f_1x_0x_1 + (f_1x_1)^2$

$y^2 = (f_0y_0 + f_1y_1)^2 = (f_0y_0)^2 + 2f_0f_1y_0y_1 + (f_1y_1)^2$

$z^2 = (f_0z_0 + f_1z_1)^2 = (f_0z_0)^2 + 2f_0f_1z_0z_1 + (f_1z_1)^2$

$w^2 = (f_0w_0 + f_1w_1)^2 = (f_0w_0)^2 + 2f_0f_1w_0w_1 + (f_1w_1)^2$

将这些方程加在一起有：

$f_0^2 + 2f_0f_1(x_0x_1 + y_0y_1 + z_0z_1 + w_0w_1) + f_1^2 = f_0^2 + 2f_0f_1(q_0 \bullet q_1) + f_1^2 = 1$

我们可以将这表示为一个矩阵乘法：

$$\begin{vmatrix} f_0 & f_1 \end{vmatrix}^T \begin{vmatrix} 1 & c \\ c & 1 \end{vmatrix} \begin{vmatrix} f_0 \\ f_1 \end{vmatrix} = f^T M f = 1$$

这里 $c = q_0 \bullet q_1 = \cos(\theta)$。

矩阵 M 能被展开为：

$$M = \begin{vmatrix} 2^{1/2}/2 & -2^{1/2}/2 \\ 2^{1/2}/2 & 2^{1/2}/2 \end{vmatrix} \begin{vmatrix} 1+c & 0 \\ 0 & 1-c \end{vmatrix} \begin{vmatrix} 2^{1/2}/2 & 2^{1/2}/2 \\ -2^{1/2}/2 & 2^{1/2}/2 \end{vmatrix} = R^T C R$$

令 $\mathbf{u} = C^{1/2}Rf$：

$$\mathbf{u} = \begin{vmatrix} (1+c)^{1/2} & 0 \\ 0 & (1-c)^{1/2} \end{vmatrix} \begin{vmatrix} 2^{1/2}/2 & 2^{1/2}/2 \\ -2^{1/2}/2 & 2^{1/2}/2 \end{vmatrix} \begin{vmatrix} f_0 \\ f_1 \end{vmatrix}$$

将其乘开得：

$\mathbf{u} = [(f_1 + f_0)(2 + 2c)^{1/2}/2, (f_1 - f_0)(2-2c)^{1/2}/2]$

从现在起，我们需要显示时间变量了。$\mathbf{u}(t)$是一个二维单位矢量，故它能被写为：

$\mathbf{u}(t) = [\cos(\omega t), \sin(\omega t)]$

此处：

$\cos(\omega t) = (f_1(t) + f_0(t))(2 + 2c)^{1/2}/2$

$\sin(\omega t) = (f_1(t) - f_0(t))(2-2c)^{1/2}/2$

设 $A = (2 + 2c)^{1/2}/2$, $B = (2 - 2c)^{1/2}/2$

用 B/AB 乘 $\cos(\omega t)$且用 A/AB 乘 $\sin(\omega t)$得：

$f_1(t) + f_0(t) = B\cos(\omega t)/AB$

$f_1(t) - f_0(t) = A\sin(\omega t)/AB$

解出 f_0 和 f_1 得：

$f_0(t) = [B\cos(\omega t) - A\sin(\omega t)]/2AB$

$f_1(t) = [B\cos(\omega t)+A\sin(\omega t)]/2AB$

$2AB = (1-c^2)^{1/2} = (1-\cos^2(\theta))^{1/2} = \sin(\theta)$

$A^2+B^2 = 1$，所以对于某些相位角ψ，有 $A = \cos(\psi)$和 $B = \sin(\psi)$。

这里ψ的两个值满足 A 和 B，即ψ_0 和ψ_1。已知这一点，我们可以重新将 f_0 和 f_1 写为：

$f_0(t) = [\sin(\psi_0)\cos(\omega t) - \cos(\psi_0)\sin(\omega t)]/\sin(\theta)$

$f_1(t) = [\sin(\psi_1)\cos(\omega t) + \cos(\psi_1)\sin(\omega t)]/\sin(\theta)$

回忆三角恒等式：

$\sin(a)\cos(b) + \sin(b)\cos(a) = \sin(a + b)$

$\sin(a)\cos(b) - \sin(b)\cos(a) = \sin(a - b)$

所以有：

$f_0(t) = \sin(\psi_0-\omega t)/\sin(\theta)$

$f_1(t) = \sin(\psi_1+\omega t)/\sin(\theta)$

给出 f_0 和 f_1 的边界约束，可以解出ψ_0，ψ_1和ω：

$f_0(0) = \sin(\psi_0)/\sin(\theta) = 1 \rightarrow \psi_0 = \theta$

$f_1(0) = \sin(\psi_1)/\sin(\theta) = 0 \rightarrow \psi_1 = 0$

$f_0(1) = \sin(\psi_0 - \omega)/\sin(\theta) = 0 \rightarrow \omega = \psi_0 = \theta$

故我们可以再次改写 f_0 和 f_1：

$f_0(t) = \sin(\theta(1 - t))/\sin(\theta)$

$f_1(t) = \sin(\theta t)/\sin(\theta)$

推导 2.9.2　推导 Slerp 的幂函数形式

由 **slerp** 的定义出发：

$\mathbf{slerp}(t; q_0, q_1) = [q_0\sin(\theta(1 - t)) + q_1\sin(\theta t)]/\sin(\theta)$

q_0 是一个单位四元数（像 q_1 一样），因此

$q_0q_0^{-1} = 1$

由此可以将 q_1 改写为：

$q_1 = q_0q_0^{-1}q_1$

让我们展开 $q_0^{-1}q_1$：

$q_0^{-1}q_1 = [s_0s_1 + \mathbf{v_0}\bullet\mathbf{v_1}, -\mathbf{v_0}\otimes\mathbf{v_1} + s_0\mathbf{v_1} - s_1\mathbf{v_0}]$

注意 $q_0^{-1}q_1$ 的标量部分 $q_0^{-1}q_1 = s_0s_1 + \mathbf{v_0}\bullet\mathbf{v_1}$。这与 q_0 和 q_1 的矢量点积相同，也与 q_0 和 q_1 的夹角 θ 的余弦值相等。

由于 $q_0^{-1}q_1$ 是一个单位四元数且我们知道它的数量部分为 $\cos(\theta)$，可以将 $q_0^{-1}q_1$ 改写为：

$q_0^{-1}q_1 = \cos(\theta) + \mathbf{u}\sin(\theta)$

这里 **u** 是一个单位矢量。

好了，现在我们有：

$q_1 = q_0(\cos(\theta) + \mathbf{u}\sin(\theta))$

如果我们把它代入到上面的 **slerp** 中，得到：

$\mathbf{slerp}(t; q_0, q_1) = [q_0\sin(\theta(1-t)) + q_0(\cos(\theta) + \mathbf{u}\sin(\theta))\sin(\theta t)]/\sin(\theta)$

由三角恒等式，我们知道：

$q_0\sin(\theta(1-t)) = q_0(\sin(\theta)\cos(\theta t) - \cos(\theta)\sin(\theta t))$

把它代入 **slerp** 并化简得：

$\mathbf{slerp}(t; q_0, q_1) = q_0(\cos(\theta t) + \mathbf{u}\sin(\theta t))$

由四元数的幂函数形式，我们可以将其改写为：

$\mathbf{slerp}(t; q_0, q_1) = q_0(\cos(\theta) + \mathbf{u}\sin(\theta))^t$

哈，但是 $\cos(\theta) + \mathbf{u}\sin(\theta) = q_0^{-1}q_1$，因此再次改写得：

$\mathbf{slerp}(t; q_0, q_1) = q_0(q_0^{-1}q_1)^t$

推导 2.9.3 推导样条插值

设 $\{a_n, b_n\}$ 是一个四元数序列。

设 $S_n(t) = \mathbf{squad}(t; q_n, q_{n+1}, a_n, b_{n+1})$

为了导出样条插值，我们想找到 $\{a_n, b_n\}$ 使得 $S_n(t)$ 在控制点（$t = 0$ 和 $t = 1$）的导数是连续的。换句话说，对所有的 n，$S_n'(0) = S_{n-1}'(1)$。

为做到这一点，我们首先必须表示 **squad** 的导数。

设 $U = \mathbf{slerp}(t; p, q)$，$V = \mathbf{slerp}(t; a, b)$，$W = U^{-1}V$

已知 **slerp** 的幂函数形式，我们可以改写 **squad**：

$\mathbf{squad}(t; p, q, a, b) = U(U^{-1}V)^{2t(1-t)} = UW^{2t(1-t)}$

U、V 和 W 的导数为：

$U' = p(p^{-1}q)^t\log(p^{-1}q) = U\log(p^{-1}q)$

$V' = a(a^{-1}b)^t\log(a^{-1}b) = V\log(a^{-1}b)$

$W' = U^{-1}V' - U^{-2}U'V$

应用乘积法则，我们可以表示 **squad** 的导数：

$\mathbf{squad}'(t; p, q, a, b) = U[W^{2t(1-t)}]' + U'[W^{2t(1-t)}]$

这里 $[W^{2t(1-t)}]' = (2-4t)W^{2t(1-t)}\log(W) + 2t(1-t)W'W^{2t(1-t)-1}$

我们太幸运了，只需要对 t=0 和 t=1 计算 **squad'**：

$U(0) = p$

$V(0) = a$

$W(0) = p^{-1}a$

$U'(0) = p\log(p^{-1}q)$

$[W^{2t(1-t)}]'(0) = 2\log(p^{-1}a)$

$\mathbf{squad}'(0; p, q, a, b) = p[\log(p^{-1}q) + 2\log(p^{-1}a)]$

$U(1) = q$

$V(1) = b$

$W(1) = q^{-1}b$

$U'(1) = q\log(p^{-1}q)$

$[W^{2t(1-t)}]'(1) = -2\log(q^{-1}b)$

$\mathbf{squad}'(1; p, q, a, b) = q[\log(p^{-1}q) - 2\log(q^{-1}b)]$

代入 $S'_{n-1}(1) = S'_n(0)$得：

$q_n[\log(q_{n-1}^{-1}q_n) - 2\log(q_n^{-1}b_n)] = q_n[\log(q_n^{-1}q_{n+1}) + 2\log(q_n^{-1}a_n)]$

这给了我们一个方程和两个未知数（a_n 和 b_n）。到此为止，惟一的限制是必须经过所有的控制点并得到一个连续的导数。我们必须自己选择控制点的导数。在一个控制点为导数选择的一个合理值是两个函数的正切的平均值：

$S'_{n-1}(1) = q_nT_n = S'_n(0)$

这里

$T_n = [\log(q_n^{-1}q_{n+1}) + \log(q_{n-1}^{-1}q_n)]/2$

因而现在我们有两个方程：

$q_n[\log(q_{n-1}^{-1}q_n) - 2\log(q_n^{-1}b_n)] = q_n[\log(q_n^{-1}q_{n+1}) + \log(q_{n-1}^{-1}q_n)]/2$

$q_n[\log(q_n^{-1}q_{n+1}) + 2\log(q_n^{-1}a_n)] = q_n[\log(q_n^{-1}q_{n+1}) + \log(q_{n-1}^{-1}q_n)]/2$

解出 a_n 和 b_n：

$a_n = b_n = q_n\exp[(\log(q_{n-1}^{-1}q_n) - \log(q_n^{-1}q_{n+1}))/4]$

对一个单位四元数 q = [s,**v**]：

$q^* = q^{-1} = [s, -\mathbf{v}]$

$(pq)^* = q^*p^*$

$\log(q^{-1}) = -\log(q)$

由这些规律，我们可以说：

$\log(q_{n-1}^{-1}q_n) = -\log((q_{n-1}^{-1}q_n)^{-1}) = -\log(q_n^{-1}q_{n-1})$

将其代入到$\{a_n, b_n\}$的方程中得

$a_n = b_n = q_n\exp[-(\log(q_n^{-1}q_{n-1}) + \log(q_n^{-1}q_{n+1}))/4]$

2.10 最短弧四元数

Stan Melax

本文给出了一个称为RotationArc( )的简短程序。设有两个矢量 v_0 和 v_1，该函数返回一个四元数 q，这里 $q*v_0==v_1$。它的实现可称得上是最佳的，且避免了常见的数值不稳定性缺陷。

2.10.1 动机

你可能想知道在哪里会用到这样的函数。请考虑你视频游戏中的导弹。这是一个用到了朝向（3DOF）的物体，即使仅用到这一径向对称物体的正向（2DOF），对于其智能也是非常重要的。对于所有刚体的情形，需要一个四元数 q 来改变它的朝向，即由当前方向 v_0 改变到我们想让它具有的方向 v_1。虽然我们可以从无限多的旋转轴中进行选择，但是最好选择能最小化重取向（弧）角的那个明显的旋转轴——换句话说，该明显的轴与两个矢量都垂直。该程序对用鼠标实现一个自旋物体的“虚拟跟踪球（virtual trackball）”也是有用的（如在一个 VRML 阅读器中）。在本书所附光盘中包含有 SpinLogo 演示程序，用 `RotationArc( )` 来实现这个特征。

2.10.2 数值不稳定性

该算法可以由以下步骤很容易地做到：通过取规一化叉积得到一个旋转轴，然后取点积的反余弦函数得到矢量之间的夹角。这一旋转轴和角度能被传入一个四元数的构造器中。然而，这并不是一个好的解决办法，因为当矢量 v_0 和 v_1 很接近时，叉积（与两个矢量间的夹角的正弦值成比例）变得非常小，且当我们试图将其规一化时，它有潜在的不稳定性（见图2.10.1）。推导出角度也会有困难。取得两个平行的单位长度矢量的点积可能引起一个小的溢出（比 1 大），当推导角度时这就会成为一个问题。请试着执行一下 acos(1.00000001)。在这些情形，采用标准的四元数构造器接受一个轴和一个角度就不合适了。本解法将以一种更直接的方式来产生四元数。

我们最早是在为视频游戏 MDK2 开发导弹的过程中发现这个问题的（参见[Bioware00]）。但这并不是一个仅仅在我们的开发中暴露出来的罕见问题。它已经被其他许多人注意到，并且你也可能遇到。*Real-Time Rendering*（参见[Moller99]）简要地提到了这一问题。本文给出了一个更彻底的解释并提供了代码加入到“游戏编程精粹系列”数学库（或者你自

己的四元数库）。如果想避免讨厌的 BUG，你应该使用本文提供的代码由两个方向矢量生成四元数。

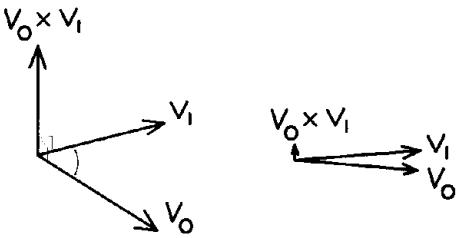


图 2.10.1 当矢量会聚时其叉积缩短

2.10.3 稳定公式的推导

为了便于讨论，假设 $c = [c_x, c_y, c_z] = \text{cross}(v_0, v_1)$，并且我们试图得到的四元数 q 具有分量 q_x，q_y，q_z，q_w。两个矢量（v_0 和 v_1）间的夹角（未知）为 t。假设 d 是点积：$d = \text{dot}(v_0, v_1)$；

一个四元数 q 的 q_x，q_y，q_z 分量具有一个长度，该长度为夹角一半（$t/2$）的正弦函数值。正如提到过的，叉积的长度为角度 t 的正弦函数值。因此：

$$[q_x, q_y, q_z] = [c_x, c_y, c_z]\frac{\sin(t/2)}{\sin(t)}$$

所以现在我们必须为 $\sin(t/2)/\sin(t)$ 项确定一个稳定的公式了。回忆半角公式：

$$\sin(t/2) = \sqrt{\frac{1-\cos(t)}{2}}$$

以及循环恒等式：

$$\sin^2 + \cos^2 = 1$$

则：

$$\frac{\sin(t/2)}{\sin(t)} = \frac{\sqrt{(1-\cos(t))/2}}{\sqrt{1-\cos^2(t)}}$$

我们知道 $\cos(t)$ 为两个矢量的点积（d）。因此，我们在公式中替换它并继续化简：

$$\frac{\sqrt{(1-d)/2}}{\sqrt{1-d^2}} = \sqrt{\frac{1-d}{2(1+d)(1-d)}} = \frac{1}{\sqrt{2(1+d)}}$$

代回得：

$$[q_x, q_y, q_z] = \frac{[c_x, c_y, c_z]}{\sqrt{2(1+d)}}$$

对 cos(*t*/2)运用半角公式，可以相当直接地得到四元数的 q_w 分量（夹角）：

$$q_w = \cos(t/2) = \sqrt{\frac{1+\cos(t)}{2}} = \sqrt{\frac{1+d}{2}}$$

为了优化我们的 C++函数使其只调用一次 sqrt( )，我们给 q_w 公式中平方根内的项乘上 2/2。结果平方根内的项现在与四元组的其他分量的相同了。换句话说，q_w 可用下面的等价公式求出：

$$q_w = \frac{\sqrt{2(1+d)}}{2}$$

当 v_0 接近于 v_1 和点积 d 接近于 1 时，这些四元数元素的新公式仍保持稳定。

2.10.4 残存不稳定性条件

注意当 v_0 趋近 $-v_1$ 时该函数仍然会变得数值不稳定。这并不奇怪，因为当 v_0 等于 $-v_1$ 时，解并不惟一；任何与 v_0 垂直的平面上的旋转轴都可以作为解。注意 v_0 和 v_1 是方向，而不是朝向。为了发现这种情况并避免被零除的可能性，可以在函数上加一个检查。然而，我们并没有这样做，因为在那种情形下不太可能会调用该函数。使用一个这样的函数的目的是朝一个目标改变物体的朝向。因而，在重复导弹跟踪智能更新后，v_0 很少会接近 $-v_1$ 而是与 v_1 会聚。

2.10.5 源代码

```
quaternion RotationArc(vector3 v0,vector3 v1) {
    quaternion q;
      v0.normalize();  // Skip if known to be unit length.
      v1.normalize();  // Do only if needed.
      vector3 c = CrossProduct(v0,v1);
      float   d = DotProduct(v0,v1);
      float   s = (float)sqrt((1+d)*2);
      q.x = c.x / s;
      q.y = c.y / s;
      q.z = c.z / s;
      q.w = s /2.0f;
      return q;
}
```

2.10.6 虚拟跟踪球

本文还奉上了实现虚拟跟踪球函数的代码。虽然此功能在你完成的游戏用户界面中可能并不必要，但是它非常易于在开发过程中抓取任何游戏对象并用鼠标控制它们旋转。

一种简单的实现物体自旋的方法是，当鼠标平行于 X 方向运动时，物体绕 Y 轴旋转，而当鼠标运动基于垂直方向（Y）时，物体绕 X 轴旋转。这种方法没有考虑关于 Z 轴的旋转（通常朝向窗体），而且给人的感觉也不太真实。

还有多种其他的用户界面方法适合于自旋对象。本文阐明了一种简单的途径。旧的和新的鼠标位置（2D [X, Y]）被转换为从视点指向窗体的光线（3D）。接着，我们确定这些光线将在何处与一个球面相交，该球面环绕用户操纵的物体。如果一条光线不与球面相交，则使用球面轮廓上最邻近的点。球面被旋转，使得从旧的鼠标发出的光线的交点与从新的鼠标发出光线的交点相符。这由传递这两个点作为 `RotationArc( )` 的输入得到（用球面的中心作为坐标系的原点）。这样返回的四元数用于调整物体的朝向。本书所附光盘上 SpinLogo 源代码中给出了虚拟跟踪球函数和从鼠标输入抽取方向矢量的源代码。

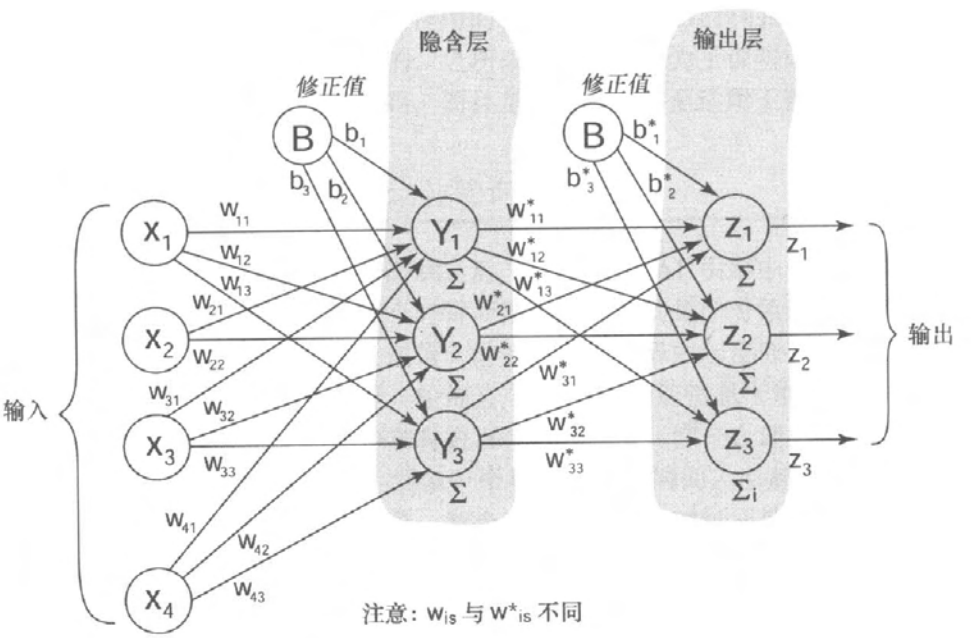
2.10.7 参考文献

[Bioware00] Bioware, Shiny, and Interplay Productions, *MDK2*, 2000.

[Moller99] Moller, Tomas, and Haines, Eric, *Real-Time Rendering*, A. K. Peters Ltd., 1999.

第 3 章

人工智能


隐含层
输出层
修正值
修正值
B
B
b_1
b_3
b_2
b^*_1
b^*_3
b^*_2
X_1
X_2
X_3
X_4
输入
W_{11}
W_{12}
W_{13}
W_{21}
W_{22}
W_{31}
W_{32}
W_{33}
W_{41}
W_{42}
W_{43}
Y_1
Y_2
Y_3
Σ
W^*_{11}
W^*_{12}
W^*_{13}
W^*_{21}
W^*_{22}
W^*_{31}
W^*_{32}
W^*_{33}
Z_1
Z_2
Z_3
$Σ_i$
z_1
z_2
z_3
输出
注意：w_{is} 与 w^*_{is} 不同

3.0 设计一个通用、健壮的 AI 引擎

Steve Rabin

历时几年开发一个耗资数百万美元的游戏是一项相当艰巨的任务。AI 引擎的底层结构将在很大程度上决定你的游戏能做什么和不能做什么。因此，与其逐渐改进一个引擎，倒不如从一开始就加入通用性和安全防范。通用性能让你的 AI 角色做你能想象出的任何事情。安全防范不但可以防止 bug 出现而且可以帮助你跟踪它们。说到底，横在你和庆功宴之间的惟一的东西就是几千个 bug。

理想的 AI 引擎能为你解决大量的问题，以下列出其中几个：

- 使游戏对象间的通信更容易。
- 提供一个实现 AI 行为的通用且易读的解决方案。
- 便于为每一个事件保存调试记录（你想捕捉所有的 bug，对吗？）。

本文包括几个部分。每一部分单独看起来可能是显而易见的、普通的，或者是疯狂的，但是把这些思想结合起来却创造出了一个非常强大的系统。所以，当你从头到尾读这篇文章的时候，要试着谨记这一宏伟蓝图以及这些概念是如何相互作用的。此外，要认识到我们这里介绍的 AI 引擎是用 C 而不是 C++实现的。之所以选择 C，是为了说明该引擎的运转不依赖于任何面向对象的代码，而且是通用的，甚至适用于控制台开发。

3.0.1 事件驱动与轮询的对比

在研究几个游戏之后，你将看到 AI 引擎中一些相当重要的模式。首先，需要大约每个时钟周期对游戏中的每个对象进行逻辑更新。其次，这些对象需要彼此通信。游戏对象在世界中起反应有两个基本的方式：通过积极地观察世界（轮询），或者通过坐等消息（事件驱动）。

由于游戏中常有数百个游戏对象，惟一合理的解决方案是尽可能采用事件驱动技术。想象一个导弹爆炸的瞬间，引起的区域破坏影响大约 15 个左右的游戏对象。每个游戏对象每个时钟周期轮询附近的爆炸，仿佛它们真能查觉到周围的环境，或者爆炸的导弹可以告诉每个游戏对象它被击中了，以及程度如何。虽然设想每个游戏对象都能感觉自己的环境并做出适当的反应更“酷”，实际上两种方法最后的结果是一样的。

由以上不那么确切的比方可知，你可能会同意事件驱动通信才是应该采用的方法！

3.0.2 消息概念

由于目标是得到事件驱动的行为，为了使其发生，需要设计一个健壮的通信系统。让我们先看一下消息的概念。消息是一个具有5个域的对象：一个描述性的名字、发送者的名字、接收者的名字、应发送的时间，以及任何相关数据。如果我传递一个消息，我应该知道所有这些必要的信息，在正确的时刻把它传送到正确的游戏对象。消息接收者得到消息，连同其中的所有附加信息，如是谁发送的消息以及任何附加的数据。

这是一个消息的实例：

```
name:damaged, from:dragon, to:knight, deliver_at_time:245.34, data:10 (破坏的数量)
```

这个概念的另一个了不起的运用是，任何游戏对象能够“收听”任何其他游戏对象的消息。因为消息中写入了预期的接收者，很容易区分是谁预订了消息。可以称它为探听、嗅探，或者只窥视，这种能力给了你解决一些棘手的逻辑问题的力量。想象一下管理器游戏对象，它拥有几个其他游戏对象。这个管理器能够探听它子节点的消息来注意关键事件，诸如成员被攻击或被破坏。

我正在描述的这个消息还有一个发送时间域。根据应在一个将来的时刻发送一个消息，我们在这个消息概念内包装一个绝妙的时钟系统。在真实世界中，人和生物常有响应时间。在稍后发送消息，依赖于模拟的事件和响应时间。毕竟当你正在玩一个游戏的时候，什么东西却突然好像锁住了脚步，你难道不恨它吗？为了交错行为变化，即使是在一个单独的游戏对象内部，也能在将来的某个时间给自己发送消息。为了增加一些非常必要的混沌，你甚至可以在一些窗体内部产生一个随机时间。

3.0.3 状态机

状态机是一个非常简单的AI概念，它可以提供许多能力，没有多少复杂性。它的基本思想是，一个游戏对象对于它所展现的行为的每一个主要部分都有一个不同的状态。目标是将一个游戏对象的行为分解为这些逻辑状态。例如在一个棒球游戏中，投手可能会有下列状态：ReadyForWindup，Windup，WaitForHit，InterceptBall，CoverBase，等等。设想一下，如果你的棒球游戏在任何时候都可以在屏幕上完全显示运动场上9个队员现在和过去的状态，它将是多么有用！换句话说，如果你遇到一个bug，可以将所有过去的状态信息与引起每一个状态变迁的因一起转储到一个文件中。例如，如果你的右翼手对被击中的球毫无反应，你可以了解他为什么不在倾听事件的正确状态。状态机不仅将行为分解为易处理的字节大小的块，而且也给了你对AI对象的动机或思想的即时访问。

3.0.4 一个使用消息的事件驱动状态机

把这三个主要概念结合起来，我们现在有了一个关于AI引擎的强大基础了。个体行为使用状态机来建立，而所有的通信和事件通告由消息来完成。注意每一个游戏对象运行一个

状态机并不排除它使用模糊逻辑、神经网络，或其他任何异乎寻常的 AI 技术。状态机仅仅提供了一个标准的普通接口，你能以任何方式开发它。

尽管状态机是简单的概念，让我们回顾一下将使一个状态机更精致和更健壮的一些重要的特征。下面是一个所需特征的清单：

（1）状态机可以有任意数目的状态；
（2）状态可以很容易地定义和设置；
（3）当一个状态被进入时，我们将能够执行任何初始化代码；
（4）当一个状态被退出时，我们将能够执行任何清除代码；
（5）可以很容易地倾听消息并执行任何响应代码；
（6）可以很容易地倾听更新信号并执行任何响应代码；
（7）可以透明地记录哪些消息被接收以及是否有响应；
（8）可以透明地记录状态改变及触发它们的消息；
（9）可以仅在某些状态内部或遍及世界状态倾听一个消息；
（10）我们可以发送消息给任何游戏对象，包括我们自己；
（11）可以在一个消息内写入一个延迟时间，使该消息能在将来的某个时间发送；
（12）运行状态机的总开销应最小。

我们的状态机应该支持所有这些特征。表 3.0.1 是所有需要结构的伪代码。

表 3.0.1　　状态机的伪代码

伪代码关键字	描述
BeginStateMachine	开始状态机定义
EndStateMachine	终止状态机定义
State(NameOfState)	指明一个特殊状态的开始
OnEnter	响应一个被进入的状态；允许初始化代码
OnExit	响应一人被退出的状态；允许清除代码
OnUpdate	响应更新游戏信号
OnMsg(NameOfMessage)	响应任何定义的消息
SetState(NameOfState)	改变状态；发送 OnExit 给旧状态并发送 OnEnter 给新状态
SendMsg()	发送消息给任何游戏对象
SendDelayedMsg()	发送一个延迟消息给任何游戏对象

为了使这个概念更具体，让我们来看一个有攻击倾向的岗哨机器人的实例。我们也用表 3.0.1 的伪代码来表示状态机。我们的状态机有两个状态：巡逻（Patrol）和进攻（Attack）。然而，状态机出发时并没有这些状态。而是不管当前状态如何，状态机顶部的世界部分总是活动的。当状态机第一次运行的时候，OnEnter 响应被触发。在这个响应内部，第一个状态由一个 SetState 命令设置。在这个例子中，它设置开始状态给 Patrol。

```
BeginStateMachine

  //Global Responses
  OnEnter {
```

```
    SetState( STATE_Patrol )
  }
  OnMsg( MSG_Dead ) {
    //Destroy this game object
  }

  State( STATE_Patrol ) {
    OnEnter {
      //Set initial goal point for patrol
    }
    OnUpdate {
      if( /*see the enemy*/ )
        SetState( STATE_Attack )
      else if( /*goal point reached*/ )
        //set next patrol point as goal
    }
  }

  State( STATE_Attack ) {
    OnEnter {
      //Set goal to be enemy
    }
    OnUpdate {
      if( /*enemy dead*/ )
        SetState( STATE_PATROL )
      else if( /*enemy within weapon range*/ )
        //Shoot enemy
    }
  }

EndStateMachine
```

由于这个状态机是事件驱动的，它惟一的执行方法就是得到一个消息。当一个状态第一次进入时（OnEnter）、当一个状态退出时（OnExit）、当一个游戏信号发生时（OnEnter），或者任何其他定义的消息时（OnMsg( )），它可以得到一个消息。

当查看状态机的时候，想象一个消息被传送给它。消息首先转向当前状态。如果对该消息有一个响应，消息就被消耗了且执行响应。如果在那个状态中没有对消息的响应，消息就被重新发送到状态机顶部的世界响应。这一行为创造了一个非常强大的状态机概念，即这样的思想：不管当前状态如何，个体状态可以有消息响应或者你可以得到对一个消息的世界响应。甚至更强大的是，有的时候对一个消息你可以有一个优先于特定状态的内部响应的世界响应。

因为有这些世界消息响应，一个状态改变可能会自当前状态的外部发生。因为这个原因，OnExit 消息响应是至关重要的。如果一个世界消息响应改变了当前状态，你可以依靠得到当前状态内的 OnExit 消息，在状态实际改变之前清除一切东西。

3.0.5 交待时间（Confession Time）

现在我要做一个重要的交待。你刚才检验过的伪代码确实是用一个普通的 C 编译器编译的（也就是说，它将提供的注释转化为实际代码）。你需要做的所有事情就是使用这些宏定义并在下列函数中放置状态机：

```
#define BeginStateMachine  if( STATE_Global == state ) { if(0) {
#define State(a)           return( true ); } } \
else if( a == state ) { if(0) {
#define OnEnter            return( true ); } \
else if( MSG_RESERVED_Enter == msg->name ) {
#define OnExit             return( true ); } \
else if( MSG_RESERVED_Exit == msg->name ) {
#define OnUpdate           return( true ); } \
else if( MSG_RESERVED_Update == msg->name ) {
#define OnMsg(a)           return( true ); } \
else if( a == msg->name ) {
#define SetState(a)         SetStateInGameObject( go, (int)a );
#define EndStateMachine     return( true ); } } \
else { assert( !"Invalid State" ); \
    return( false );}  return( false );

bool ProcessStateMachine( GameObject* go,
unsigned int state, MsgObject* msg )
{
  // Put state machine inside this function!
}
```

C 风格的宏是一个有意思的东西。通常你可能想避免它们，因为它们可能被滥用并导致 bug。在这种情形下，我们可以开发宏定义构造一个新的状态机语言！我们当然不一定非用这些宏来让状态机工作，它只是让编码误差更少，读起来更简单，写得更快。那是宏的一个好运用！

你可能已经注意到了宏内含有一组返回语句。这些返回语句很方便地报告了一个消息是被处理还是没有被消耗地失败了。这一信息对于知晓一个在一个本地状态没被处理的消息是否需要发送给世界响应至关重要。返回值也帮助在日志中记录一个特定消息是否被状态机处理了。

3.0.6 另一个小交待

我还有一个小的交待。构造宏是用来使你不必使用所有那些花括号的！花括号使状态机更像 C，但是它们根本不需要。不使用花括号，你可以使状态机更优雅而且可能更易读。下面是一个不使用花括号的状态机的例子：

```
BeginStateMachine
```

```
    OnEnter
      //Initialization code here
    OnMsg( MSG_SomeMessage )
      //Response code here

    State( STATE_Roam )
      OnUpdate
        //Update code here
      OnExit
        //Cleanup code here

EndStateMachine
```

3.0.7 状态机构建单元

为了得到最好的结果，要非常仔细地构造这些C宏。它们可以被视为能堆成想要的任何形式的构建单元。为了这种堆能力，你最少需要下面这些语句：

```
BeginStateMachine

EndStateMachine
```

从这最少语句，你可以加入状态并倾听那些状态内部的任何消息。由于根本不要求它有状态，可以仅仅有消息响应！

状态机的名字和消息的名字是简单枚举类型——实际上是无符号整型。由于简单的if-else语句代替了宏，要求的处理是最低限度的。

3.0.8 状态机消息路由选择

当然，要有更多些的支持代码确定消息的路线并使该状态机适当工作。图3.0.1展示了一个关于该结构的概述。

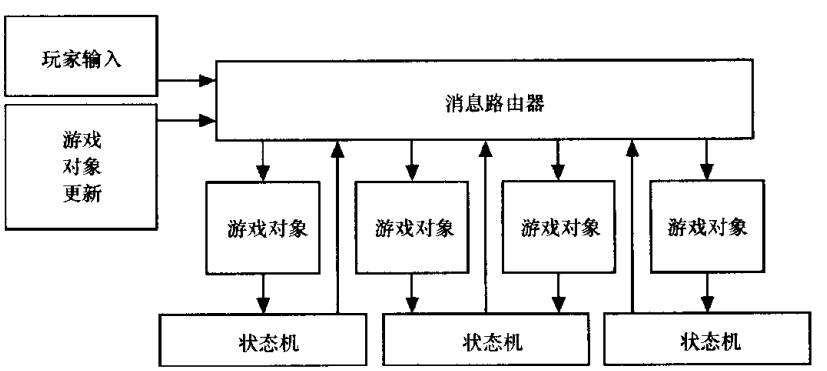


图3.0.1 消息路由选择概述

游戏对象消息的惟一外部来源是玩家的输入以及游戏时钟更新循环。除此以外，消息是状态机自身的产物。当一个消息被发出时，它总是转向消息路由器。然后路由器通过游戏对象发送它到游戏对象拥有的状态机上。万一一个消息应在未来的某个时间发送，路由器将继续保留它直到发送时间已经过去。

注意在图 3.0.1 中，两个不同的游戏对象指向同一个状态机。显然，如果你有两个或更多的对象要表现得一样，它们应该执行同样的精确代码。所以，认识到所有的变量和状态信息都被存储在游戏对象内部而不是状态机内是重要的。多游戏对象使用同一个状态机，你应该始终意识到这个事实。

为了解释消息路由选择和状态改变，我们需要对每一个游戏对象和消息对象内部的一些变量知道得更多些。下面是它们的基本定义：

```
typedef struct
{
  unsigned int unique_id;
  //State machine info
  StateMachineID state_machine_id;
  unsigned int state;              //the current state
  unsigned int next_state;         //the next state
  bool force_state_change;         // has a state change been requested

  //Put other game object info in here
} GameObject;

typedef struct
{
  MsgName name;               //name of message   unsigned int sender_id;
  unsigned int receiver_id;
  float delivery_time;        //deliver message at this time

  //Note that the sender_id and receiver_id are not pointers to
  //game objects.  Since messages can be delayed, the sender or
  //receiver may get removed from the game and a pointer would
  //become dangerously invalid.

  //You can add right here any data you want to be passed
  //along with every message - sometimes it's helpful to let
  //messages convey more info by using extra data.
  //For example, a damaged message could carry with it the
  //amount of damage.

} MsgObject;
```

下面是当一个状态改变被请求时要调用的代码。注意状态改变被请求，而且直到当前消息正被处理时才发生。此外，要认识到这个函数被 SetState 宏调用。

```
void SetStateInGameObject( GameObject* go, unsigned int state )
{
  go->next_state = state;
```

```
    go->force_state_change = true;
}
```

路由器采用一个公式化的消息，它准备好被发出并确认被发往正确的状态。如果有一个请求，它也处理改变状态。你可以在程序清单 3.0.1 中看到路由器的代码。由于路由器需要处理延迟消息（本文稍后将解释的一个概念），它涉及的函数见程序清单 3.0.2。

3.0.9 发送消息

为了从状态机内部发送一个消息，如果有一个简单函数可调用将会有帮助。下面是一个发送消息界面的例子。

```
void SendMsg( MsgName name, unsigned int sender, unsigned int receiver )
{
  MsgObject msg;
  msg.name = name;                    //The name of the message
  msg.sender = sender;                //The sender
  msg.receiver = receiver;            //The receiver
  msg.delivery_time = GetCurTime();   //Send the message NOW

  RouteMessage( &msg );
}
```

注意当一个消息被发出的时候，状态机立即将其送往预期的接收者。对调用来说，这是个很好的特性，因为一个状态机内部的断点能让你看到堆栈，进而看到是谁发送了这个消息。

3.0.10 发送延迟的消息

如前面提及的，我们能给消息一个未来将被发送的时间。路由器通过为未来路由选择存储延迟消息来处理这个命令。在游戏的主循环的某处，需要调用函数 SendDelayedMessages 以使消息能最后在正确的时间被发送出。下面是发送一个延迟消息的界面函数：

```
void SendDelayedMsg( MsgName name, float delay,
unsigned int sender, unsigned int receiver )
{
  MsgObject msg;
  msg.name = name;                    //The name of the message
  msg.sender = sender;                //The sender
  msg.receiver = receiver;            //The receiver
  msg.delivery_time = GetCurTime() + delay;    //Send a future message

  RouteMessage( &msg );
}
```

注意所有的消息包含了发送者和接收者作为惟一 ID，而不是作为指针。因为消息可能被延迟，发送者或接收者可能已被从游戏中删除。由于指针不可能知道这一点，仅用一个指针

标记游戏对象是不安全的，改为用惟一的 ID 查出消息的接收者。这种方法确保了消息只发送给有效的游戏对象。

延迟消息对状态机是一个难以置信的有用工具。考虑下面 heat-seeking 火箭的状态机。火箭被点火，而且如果在 5 秒内没有接触任何东西，它将会自动爆炸。当状态机被初始化时，这个任务通过发送一个延迟消息（MSG_SelfDestruct）完成。过了 5 秒之后，消息被送往状态机并在世界响应中消耗。这时，状态被设置到 Explode，而且火箭将不复存在。

```
BeginStateMachine
  //Global Responses
  OnEnter    //Triggered when state machine first starts up
    SendDelayedMsg( MSG_SelfDestruct, 5.0, go->unique_id,
      go->unique_id );
    SetState( STATE_Armed );

  OnMsg( MSG_SelfDestruct )
    SetState( STATE_Explode );

  State( STATE_Armed )
    OnMsg( MSG_Collision )
      SetState( STATE_Explode );

    OnUpdate
      //Identify closest visible enemy and steer toward

  State( STATE_Explode )
    OnEnter
      //Explode rocket - cause area damage
      //Delete game object

EndStateMachine
```

3.0.11 删除游戏对象

从状态机内删除游戏对象要做一些考虑。由于游戏对象拥有状态机，当我们执行状态机内部代码时不能删除它。解决的办法是设置一个识别应被删除的游戏对象的标志。当执行移到状态外部时，删除它就是合法的。

3.0.12 增强：定义消息的范围

一个悄然出现的问题是，有时消息仅仅在一个特定状态内部是有效的。不幸的是，一个延迟消息有被错误的状态消耗的潜在可能。考虑下面的代码：

```
BeginStateMachine

  //Global Responses
```

```
    OnEnter
      SetState( STATE_Alive );


    State( STATE_Alive )
      OnEnter
        SendDelayedMsg( MSG_TimeOut, 3.0, go->unique_id,
          go->unique_id );

      OnMsg( MSG_TimeOut )
        //Play a sound

      OnMsg( MSG_Dead )
        SetState( STATE_Dead );

    State( STATE_Dead )
      OnEnter
        SendDelayedMsg( MSG_TimeOut, 50.0, go->unique_id,
          go->unique_id );

      OnMsg( MSG_TimeOut )
        SetState( STATE_Alive );

EndStateMachine
```

问题是两个状态都发送和响应 MSG_TimeOut。如果 Alive 状态在收回 MSG_TimeOut 之前得到一个 MSG_Dead，失效的状态就会不正确地得到一个产自 Alive 状态的 MSG_TimeOut，这显然是我们不希望出现的。

解决的办法是仅在一个特定状态内部标志一个消息有效。如果当发送时状态不再活动，该消息就应该被丢弃。实际上，消息现在有了一个范围并且仅在此范围内有效。

这个增强特性能很容易地用每个消息变量内部一个额外的称为“状态”的变量来添加。当消息被发送时，它首先检查消息“状态”是否与游戏对象的当前状态相匹配。只有相匹配消息才被发送。然而，绝大多数时间，你只会想发送消息而不考虑当前状态如何，于是消息应默认为不执行这个检查。

由于只有发送给你自己的延迟消息才应该被标上一个状态，我们可以为发送这种特别种类的消息创建一个更有用的函数。下面就是一个代码示例（如果做了这个增强，其他的发送消息函数需要标志消息状态为无效）。

```
void SendDelayedMsgToCurrentState( MsgName name, float delay,
GameObject* go )
{
  MsgObject msg;
  msg.name = name;              //The name of the message
  msg.state = go->state;        //The state in which the msg is valid
  msg.sender = go->unique_id;       //The sender
  msg.receiver = go->unique_id;     //The receiver
  msg.delivery_time = GetCurTime() + delay;   //Send a future message
```

```
    RouteMessage( &msg );
}
```

3.0.13 增强：记录所有的消息活动和状态变迁

有了当前的结构，探听每一个游戏对象当前的状态并在屏幕上显示它就一下子轻而易举了。你甚至可以观察所有通过消息路由器的消息传输并把它们显示在屏幕上。但是对于实时（hard-core）调试，理想的情形是单独跟踪每一个游戏对象，记录其所有的消息活动和状态变迁，同时加上一个时间印记。令人惊讶的是这个工作其实很容易做。

诀窍是修改状态机宏。当我们插入一些调用宏的简单函数时，监控每一个状态机的工作就变得透明了。下面的宏记录了消息响应和状态变迁：

```
#define OnEnter      return( true ); } \
    else if( MSG_RESERVED_Enter == msg->name ) { \
    LogMessage( go, msg, GetCurTime() );
#define OnExit       return( true ); } \
    else if( MSG_RESERVED_Exit == msg->name ) { \
    LogMessage( go, msg, GetCurTime() );
#define OnUpdate     return( true ); } \
    else if( MSG_RESERVED_Update == msg->name ) { \
    LogMessage( go, msg, GetCurTime() );
#define OnMsg(a)     return( true ); } \
    else if( a == msg->name ) { \
    LogMessage( go, msg, GetCurTime() );
#define SetState(a)  SetStateInGameObject( go, (int)a ); \
    LogStateChange( go, state, (int)a, GetCurTime() );
```

函数 LogMessage 和 LogStateChange 能以任何你喜欢的方式存储信息。建议为每个游戏对象保持一个历史数据的循环缓冲器。然后当一些有趣的事情发生时，你就既能在屏幕上浏览它的历史也能将它转存到一个文件中。由于每一个事件都打上了时间标记，你可以比较不同游戏对象的日志来看一下它们是如何相互作用的。如前所述，如果有三个或更多的游戏对象在一瞬间相互作用，这个特征惊人地有用。现在非常复杂的交互作用也易于调试了。

3.0.14 增强：交换状态机

在复杂的角色内，很难设计一个通用的状态机。解决的办法是构造更易于管理和更专门的字节大小的状态机。有了这种功能，一个角色可以选择运行最适合于情况的状态机。这个函数避免了变得难以管理的过分复杂的状态机。

3.0.15 增强：多状态机

没有规定说一个游戏对象在一个时间只能运行一个状态机。事实上，一个 AI 角色同时

运行几个状态机是很有用的。设想每个角色有一个充当大脑作用的状态机，一个服务于保持跟踪运动目标的状态机（不是运动执行效果）。“大脑”甚至可以通过发送信息形式的命令控制运动状态机。

由于不是所有类型的运动都相同，可以应用交换状态机的思想。“大脑”于是可以运行适于每种特定情况的适当的运动状态机。

3.0.16 增强：一个状态机队列

专用的运动状态机带来了一个有趣的增强。能够将几个状态机排成队列来用。在这样的情况下，只有队列头部的状态机是活动的，其他的将暂时禁止使用。

该思想是：可能一个玩家控制一个 AI 角色相继去了三个分散的地方（想一下 RTS）。为每一个地点将投入一个分离的运动状态机到运动队列。当第一个状态机到达目标时摧毁自己，并且队列中的下一个状态机变为活动的。这个方法实际上解决了在 RTS 游戏中大量的命令问题。

考虑巡逻行为。巡逻是去一个地点的队列并再三重复那个模式。通过使用状态机运动队列，每一个巡逻地点能被放入队列。然而，当一个状态机到达它的目标时，需要把它自己放到队列的末尾以保持循环。

3.0.17 代码外部脚本化行为

本文描述的 AI 引擎显然没被从代码外部以脚本驱动。但不排除一个程序设计师通过利用外部数据智能地对行为施加影响。如果我们创建影响做出决策的变量，就要为特定角色定做一个状态机。事实上，由于属性不同如侵略或恐惧，许多角色应该能够使用相同的状态机却有非常不同的行为。

有趣的是，这个 AI 引擎的原始实现支持代码外部单独脚本驱动的状态机。把逻辑性在代码外部对每一个相关人员来说都是一个调试的恶梦。原始的意图是极佳的，结果却是一个浪费了很多人时间的令人泄气的编程环境。

要学习的教训是逻辑应放在代码内部而数据应放在外部。除非设计目标是让用户写他们自己的 AI，没有令人信服的理由支持任意的对 AI 行为的脚本化（更详细的关于数据和脚本化的讨论请见“1.0 数据驱动设计魔术”）。

3.0.18 结论

即使你从不使用以上的 AI 引擎，在此给出的许多富有启发性的思想也能被应用于任何 AI 引擎或状态机。一些更值得注意的思想如下：

- 统一用消息通信。
- 由于需要使用消息，还需使用统一的时钟。
- 采用数据驱动而不是轮询。
- 跟踪所有的 AI 对象的通信和状态改变。

- 在状态机内总是使用当前的全局响应，不管当前状态如何。
- 允许一个状态机内的全局响应被当前状态重载。
- 允许一个 AI 交换状态机。
- 允许一个 AI 同时运行多个状态机。
- 允许一个 AI 将几个状态机排成队。
- 在代码内部保持复杂的逻辑性而不是从外部 scripting 它。

这里介绍的 AI 引擎是在 AI 对象上实施一个标准结构的有力工具。由于采用了宏状态机伪语言，它的速度惊人并易于添加新的行为。事实上它是如此容易，以致有一个实际的倾向是在状态机内部放置过多的代码。挑战来自决定如何取得均衡。作为一个通用的规则，或许只有高水平的决策应该被保持在状态机内。AI 角色的其他系统，如运动执行和动画，当然应该在别处存放。

程序清单 3.0.1 消息路由器

```
void RouteMessage( MSG_Object* msg )
{
  GameObject* go = GetGOFromID( cur_msg->receiver_id ); //Function not supplied
  if( !go )
  {  //Receiver doesn't exist anymore - discard the message
    return;
  }

  if( msg->delivery_time > GetCurTime() )
  {  //This message needs to be stored until its time to send it
    StoreDelayedMessage( msg );
    return;
  }

  if( RouteMessageHelper( go, go->state, msg ) == false )
  {  //Current state didn't handle msg, try Global state (0)
    RouteMessageHelper( go, 0, msg );
  }

  // Check for a state change
  while( go->force_state_change )
  {  //Note: circular logic (state changes causing state changes)
    //could cause an infinite loop here - protect against this

    //Create a general msg for initializing and cleaning up the state change
    MsgObject tempmsg;
    tempmsg.receiver = go->unique_id;
    tempmsg.sender = go->unique_id;

    go->force_state_change = false;

    //Let the last state clean-up
    tempmsg.name = MSG_RESERVED_Exit;
```

```
      RouteMessageHelper( go, go->state, &tempmsg );

      //Set the new state
      go->state = go->next_state;

      //Let the new state initialize
      tempmsg.name = MSG_RESERVED_Enter;
      RouteMessageHelper( go, go->state, &tempmsg );
   }
}

bool RouteMessageHelper( GameObject* go, unsigned int state, MsgObject* msg )
{
   //Look up correct state machine for this Game Object
   //and send message to that particular one
   //(not implemented here - this always calls the same one)
   return( ProcessStateMachine( go, state, msg ) );
}
```

程序清单 3.0.2 处理延迟消息的函数

```
void StoreDelayedMessage( MsgObject* msg )
{
   //Store this message (in some data structure) for later routing

   //A priority queue would be the ideal data structure (but not required)
   //to store the delayed messages - Check out Mark Nelson's article
   //"Priority Queues and the STL" in the January 1996 Dr. Dobbs' Journal
   //http://www.dogma.net/markn/articles/pq_stl/priority.htm

   //Note: In main game loop call SendDelayedMessages() every game
   //      tick to check if its time to send the stored messages
}

void SendDelayedMessages( void )
{  //This function is called every game tick

   while( /*loop through all delayed messages*/ )
   {
      if( cur_msg->delivery_time <= GetCurTime() )
      {
         RouteMessage( cur_msg );
         RemoveDelayedMessage( cur_msg );
      }
   }
}

void RemoveDelayedMessage( MessageObject* msg )
{
   //Remove this message from the delayed messages data structure
}
```

3.0.19 参考文献

[LaMothe95] LaMothe, Andre, "Building Brains into Your Games," *Game Developer*, www.gamasutra.com/features/programming/061997/build_brains_into_games.htm, August 1995.

[Nelson96] Nelson, Mark, "Priority Queues and the STL," *Dr. Dobb's Journal*, www.dogma.net/markn/articles/pq_stl/priority.htm, January 1996.

[Woodcock99] Woodcock, Steve, "Game AI: The State of the Industry" *Game Developer*, www.gamasutra.com/features/19990820/game_ai_01.htm, August 1999.

3.1 一个有限状态机类

Eric Dybsand

本文定义了一个通用的有限状态机（finite-state machine，FSM）C++类。FSM是计算机科学和数学的抽象概念，多年来已被广泛用于各个方面。本文并不讨论FSM背后的理论，而是简单阐述一个基本的构造单元工具——FSM类，使用它有助于在你的游戏中改进复杂的人工智能决策过程。

关于FSM首先应该了解的事情是，它们是由有限数目的状态组成的简单机器（显然如此，你不这样认为吗？）。一个状态实际上仅仅是一个情形。例如，考虑门，它的状态可以是开的（open）或者关的（closed），以及锁住的（locked）或未锁的（unlocked）。

关于FSM应该知晓的另一个方面是FSM有一个输入，它影响到从一个状态到另一个状态的状态变迁（state transition）。一个FSM可以有简单的（或复杂的）状态变迁函数，它决定了什么状态将成为当前状态（current state）。

新的当前状态称为FSM的状态变迁的输出状态（output state），或者FSM基于输入状态由变迁得到的状态。如果这个概念令你迷惑，再考虑一下作为FSM实例的门。当门处于关且锁住的状态时，也许输入“使用钥匙（use key）”将导致门变迁到未锁的状态（即状态变迁的输出状态和门的新的当前状态）。那么输入“用手（use hand）”将导致门变迁到开的状态。当门处于开的状态时，输入“用手”将使门变迁回关的状态。当门处于关的状态时，输入“使用钥匙”将使门的状态变迁回锁住的状态。当门处于锁住状态时，输入“用手”会使将门的状态变为开的变迁失效，而且门会保持锁住的状态。此外，一旦门处于开的状态，输入“使用钥匙”也会使将门的状态变为锁住的变迁失效。

所以总的来说，FSM是一个有着有限数目状态的机器，其中的一个状态是当前状态。FSM可以接受输入，这将导致一个基于某些状态变迁函数的从当前状态到输出状态的状态变迁，然后输出状态就成为了新的当前状态。

这个问题的答案是：可能性确实无穷！FSM可以构成管理游戏世界的基础，模拟非玩家控制角色（non-player character (NPC)）的情感，维持游戏的状况，分析人类玩家的输入，或者管理一个对象的状态。

例如考虑一个冒险游戏中的NPC怪兽的态度。假设该怪兽可以有下面的状态：狂暴（berserk），愤怒（rage），疯狂（mad），烦恼（annoyed），

以及漠不关心（uncaring）。此外，假设有基于怪兽状态的做不同事情的 AI 游戏代码。我们可以使用 FSM 来管理怪兽的态度，而且从一个状态到另一个状态的变迁方式取决于游戏自身的输入。我们还进而假设输入有看到了玩家（player seen），玩家进攻（player attacks），玩家离开（player gone），怪兽受伤（monster hurt），以及怪兽恢复（monster healed）。于是可以绘出图 3.1.1 所示的状态图。

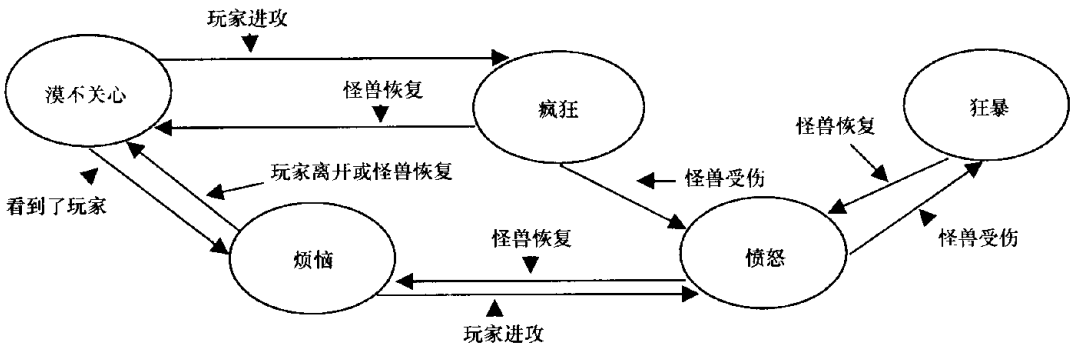


图 3.1.1 一个有限状态机实例

使用这些输入和状态，我们可以建立一个看起来类似表 3.1.1 所示的状态变迁矩阵。

表 3.1.1 怪兽游戏的一个状态变迁矩阵

当前状态	输入	输出状态
uncaring	player seen	annoyed
uncaring	player attacks	mad
mad	monster hurt	rage
mad	monster healed	uncaring
rage	monster hurt	berserk
rage	monster healed	annoyed
berserk	monster hurt	berserk
berserk	monster healed	rage
annoyed	player gone	uncaring
annoyed	player attacks	rage
annoyed	monster healed	uncaring)

所以，依赖于怪兽态度的当前状态和 FSM 的输入，怪兽的态度将会改变。执行基于怪兽态度行为的游戏代码将引起怪兽的不同行为。

显然，我们可以基于更多的状态和输入增加更多的状态变迁。这样做将影响怪兽的态度如何评价和确定，而且使用这样的怪兽态度是我们创建 AI 的方式。

3.1.1 FSMclass 和 FSMstate

现在如何把这些思想付诸实践？就是 FSMclass 和从属于它的 FSMstate 类将在下面的实现中所展示的，如图 3.1.2 所示。

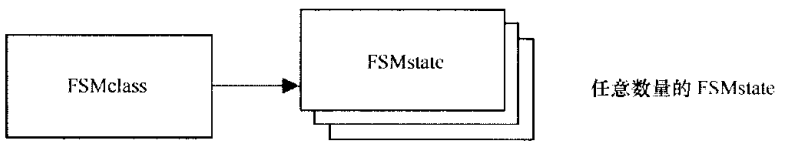


图 3.1.2 FSMclass 可以使用任意数目的 FSMstates

FSMclass 提供了对于任意数目状态的结构，这些状态由 FSMstate 类提供。这两个类相互作用为一个通用的有限状态机提供函数性。通用的 FSM 暗示了这些对象是普遍的，能够支持多种状态类型、多种状态变迁、任意数目的状态变迁，以及 FSM 中的任意数目的状态。以这样的多样性和通用性作为设计目标，我们选择在下述讨论中将看到的这些类和它们的成员。

3.1.2 定义 FSMstate

这是我们用于说明 FSM 的一个状态的类定义：

```
class FSMstate
{
   unsigned m_usNumberOfTransistions;  // maximum number of states supported
   int *m_piInputs;                    // input array for transitions
   int *m_piOutputState;               // output state array
   int m_iStateID;                     // the unique ID of this state

public:
   // constructor accepts an ID for this state and the number of
   //  transitions to support
   FSMstate( int iStateID, unsigned usTransitions );
   // destructor cleans up allocated arrays
   ~FSMstate();

   // access the state ID
   int GetID() { return m_iStateID; }
   // add a state transition to the array
   void AddTransition( int iInput, int iOutputID );
   // remove a state transition from the array
   void DeleteTransition( int iOutputID );
   // get the output state and effect a transition
   int GetOutput( int iInput );
};
```

FSMstate 类的结构器和破坏器实现请看程序清单 3.1.1。

FSMstate 类的成员变量和函数如下：

FSMstate::m_usNumberOfTransistions 控制本状态能够支持的状态变迁的数目。设置该值也决定了输入和输出数组的大小。由于我们正在创建一个“有限”状态机，该值为本状态设置了有限状态变迁限度。

FSMstate::m_piInputs 是一个 m_usNumberOfTransistions 大小的数组，它包含了在状态变迁过程期间使用的输入值。输入数组被状态变迁评价函数用于与接收的输入相比较，并决定相应的输出状态。

FSMstate::m_piOutputState 是一个 m_usNumberOfTransistions 大小的数组，它包含一个相应的输出状态标志符，该标志符指示了在一个状态变迁期间新的变迁状态。

FSMstate::m_iStateID 是惟一用于确定一个 FSM 状态实例的标志符，而且是任何从其他状态到本状态的变迁的输出值。

FSMstate::GetID( )提供了对 FSMstate 类实例的惟一标志符的公共访问。其实现请看类的声明。

FSMstate::AddTransition( )提供了一个对 FSMstate 的实例增加新的输入值和输出状态数组的方法。它的实现请看程序清单 3.1.2。

FSMstate::DeleteTransition( )提供了删除一个存在的输入和它的相应输出状态标志符的方法。它的实现请看程序清单 3.1.3。

FSMstate::GetOutput( )提供了使用输入值决定变迁输出状态标志符并返回该输出的状态变迁函数。实现见程序清单 3.1.4。

3.1.3 定义 FSMclass

现在我们需要实际的 FSMclass 实现了。FSMclass 通过保存一个 FSM 对象的集合对象来工作。

```
class FSMclass
{
   State_Map m_map;        // map containing all states of this FSM
   int m_iCurrentState;    // the m_iStateID of the current state

public:
   FSMclass( int iStateID );        // set initial state of the FSM
   ~FSMclass();                     // clean up memory usage

   // return the current state ID
   int GetCurrentState() { return m_iCurrentState; }
   // set current state
   void SetCurrentState( int iStateID ) { m_iCurrentState = iStateID; }

   // return the FSMstate object pointer
   FSMstate *GetState( int iStateID );
   // add a FSMstate object pointer to the map
```

```
    void AddState( FSMstate *pState );
    // delete a FSMstate object pointer from the map
    void DeleteState( int iStateID );

    // perform state transition based on input & current state
    int StateTransition( int iInput );
};
```

FSMclass 类的构造器和破坏器的实现见程序清单 3.1.5。

FSMclass 类的成员变量和函数如下：

FSMclass::m_map 是 FSMstate 对象的集合对象（在此是指向 FSMstate 对象的指针），并且它由一个 STL <map>实现：

```
typedef map< int, FSMstate*, less<int> > State_Map;
```

关于 STL 和<map>集合对象的特性的一般讨论超出了本文的范围。关于 STL 的信息请看“在游戏编程中使用 STL”。上述函数声明 State_Map 是一个有整型关键字的 STL <map>；该地图包含了指向 FSMstate 对象的指针，而且存取过程中使用的比较函数是整型的 less<>算子。

FSMclass::m_iCurrentState 是 FSMstate 对象的状态标志符，它被认为是 FSM 的当前状态。

FSMclass::GetCurrentState( )提供了对当前 FSMstate 状态的惟一标志符的公共访问。它的实现见类的声明。

FSMclass::SetCurrentState( )为 FSM 提供了设置新当前 FSMstate 对象状态的惟一标志符的公共访问。它的实现见类的声明。

FSMclass::GetState( )提供了一个获得 FSM 内包含的任何 FSMstate 对象指针的方法。其实现见程序清单 3.1.6。

FSMclass::AddState( )提供了一个把 FSMobject 指针增加到包含于 FSM 中的<map>的方法。这是用于记录 FSM 内由 FSMstate 定义的状态关系的一种方法。其实现见程序清单 3.1.7。

FSMclass::DeleteState( )提供了从 FSM 内包含的<map>中删除 FSMobject 指针的一种方法。这是用于从 FSM 内动态删除状态关系的一种方法。注意：当你想从<map>中删除一个当前状态时，要确保在删除旧的当前状态前用 FSMclass::SetCurrentState( )设置一个新的当前状态。其实现见程序清单 3.1.8。

FSMclass::StateTransition( )提供了一种使用接收的输入值初始化状态变迁并返回输出状态标志符的方法。其实现见程序清单 3.1.9。

3.1.4 为 FSM 创建状态

为了在游戏中应用 FSMclass 和 FSMstate 类，我们首先建立 FSMstate 对象：

```
FSMstate *pFSMstate = NULL;
```

```
// create the STATE_ID_UNCARING
try
{
   // FSMstate( int iStateID, unsigned usTransitions )
   pFSMstate = new FSMstate( STATE_ID_UNCARING, 2 );
}
catch( ... )
{
   throw;
}
// now add state transitions to this state
pFSMstate->AddTransition( INPUT_ID_PLAYER_SEEN, STATE_ID_ANNOYED );
pFSMstate->AddTransition( INPUT_ID_PLAYER_ATTACKS, STATE_ID_MAD );
```

And then create an FSMclass object:

```
// create the FSMclass object
try
{
   // FSMclass( int iStateID )
   m_pFSMclass = new FSMclass(STATE_ID_UNCARING);
}
catch( ... )
{
   throw;
}
```

现在把 FSMstate 对象加到 FSMclass 对象：

```
// now add this state to the FSM
m_pFSMclass->AddState( pFSMstate );
```

请为 FSM 中的所有状态重复创建 FSMstate 对象和将它们到增加到 FSMclass 对象的过程。

3.1.5 使用 FSM

为了使用 FSMclass，我们仅需要传递给它一个输入值（它是依赖于游戏的），并且接收一个输出状态（也是依赖于游戏的），然后作用于输出状态。所以你将有一些类似的游戏代码：

```
// something happens in the game that causes an input
iInputID = INPUT_ID_PLAYER_ATTACKS;
   .
   .
// have the FSM do the transition to an output state
m_iOutputState = m_pFSMclass->StateTransition(iInputID);
   .
   .
// some game AI code tests for the output state
```

```
if( m_iOutputState == STATE_ID_MAD )
{
   // some code for the monster to act mad
}
```

它的用法就这么简单！

总之，这个 FSMclass 不是你计算机游戏 AI 需求的终结解决方案。它是一个起点，或是用来创建适合于你自己游戏特性需求的 FSM 的一个构造单元。

这个 FSMclass 能扩展以支持不同输入类型数据或状态标志数据类型。基于接收的输入类型或当前状态，易于增加一个状态或输入特性变迁函数，到输出状态的变迁能够惟一确定。

如果对这个概念有兴趣，你将得到一个开发复杂计算机游戏 AI 的有力工具。

程序清单 3.1.1

```
FSMstate::FSMstate( int iStateID, unsigned usTransitions )
{
   // don't allow 0 transitions
   if( !usTransitions )
    m_usNumberOfTransistions = 1;
   else
    m_usNumberOfTransistions = usTransitions;

   // save off id and number of transitions
   m_iStateID = iStateID;

   // now allocate each array
   try
   {
      m_piInputs = new int[m_usNumberOfTransistions];
      for( int i=0; i<m_usNumberOfTransistions; ++i )
         m_piInputs[i] = 0;
   }
   catch( ... )
   {
      throw;
   }

   try
   {
      m_piOutputState = new int[m_usNumberOfTransistions];
      for( int i=0; i<m_usNumberOfTransistions; ++i )
       m_piOutputState[i] = 0;
   }
   catch( ... )
   {
      delete [] m_piInputs;
    throw;
   }
}
```

```
FSMstate::~FSMstate()
{
   delete [] m_piInputs;
   delete [] m_piOutputState;
}
```

程序清单 3.1.2

```
void FSMstate::AddTransition( int iInput, int iOutputID )
{
   // the m_piInputs[] and m_piOutputState[] are not sorted
   // so find the first non-zero offset in m_piOutputState[]
   // and use that offset to store the input and OutputID
   // within the m_piInputs[] and m_piOutputState[]
   for( int i=0; i<m_usNumberOfTransistions; ++i )
   {
      if( !m_piOutputState[i] )
         break;
   }
   // only a valid offset is used
   if( i < m_usNumberOfTransistions )
   {
      m_piOutputState[i] = iOutputID;
      m_piInputs[i] = iInput;
   }
}
```

程序清单 3.1.3

```
void FSMstate::DeleteTransition( int iOutputID )
{
   // the m_piInputs[] and m_piOutputState[] are not sorted
   // so find the offset of the output state ID to remove
   for( int i=0; i<m_usNumberOfTransistions; ++i )
   {
      if( m_piOutputState[i] == iOutputID )
         break;
   }
   // test to be sure the offset is valid
   if( i >= m_usNumberOfTransistions )
      return;

   // remove this output ID and its input transition value
   m_piInputs[i] = 0;
   m_piOutputState[i] = 0;

   // since the m_piInputs[] and m_piOutputState[] are not
   // sorted, then we need to shift the remaining contents
   for( ; i<(m_usNumberOfTransistions-1); ++i )
   {
      if( !m_piOutputState[i] )
```

```
            break;

        m_piInputs[i] = m_piInputs[i+1];
        m_piOutputState[i] = m_piOutputState[i+1];
    }
    // and clear the last offset in both arrays
    m_piInputs[i] = 0;
    m_piOutputState[i] = 0;
}
```

程序清单 3.1.4

```
int FSMstate::GetOutput( int iInput )
{
    int iOutputID = m_iStateID;     // output state to be returned

    // for each possible transition
    for( int i=0; i<m_usNumberOfTransistions; ++i )
    {
        // zeroed output state IDs indicate the end of the array
        if( !m_piOutputState[i] )
            break;
        // state transition function: look for a match with the input value
        if( iInput == m_piInputs[i] )
        {
            iOutputID = m_piOutputState[i];   // output state id
            break;
        }
    }
    // returning either this m_iStateID to indicate no output
    // state was matched by the input (i.e., no state transition
    // can occur) or the transitioned output state ID
    return( iOutputID );
}
```

程序清单 3.1.5

```
FSMclass::FSMclass( int iStateID )
{
    m_iCurrentState = iStateID;
}

FSMclass::~FSMclass()
{
    FSMstate *pState = NULL;
    State_Map::iterator it;

    // only perform this if there are pointers in the map
    if( !m_map.empty() )
    {
        // first delete any FSMstate objects in the map
```

```
        for( it = m_map.begin(); it != m_map.end(); ++it )
        {
            pState = (FSMstate *)((*it).second);
            if( pState != NULL )
                delete pState;
        }
        // let the map dtor() erase the actual pointer out of the map
    }
}
```

程序清单 3.1.6

```
FSMstate *FSMclass::GetState( int iStateID )
{
    FSMstate *pState = NULL;
    State_Map::iterator it;

    // try to find this FSMstate in the map
    if( !m_map.empty() )
    {
        it = m_map.find( iStateID );
        if( it != m_map.end() )
            pState = (FSMstate *)((*it).second);
    }
    return( pState );
}
```

程序清单 3.1.7

```
void FSMclass::AddState( FSMstate *pNewState )
{
    FSMstate *pState = NULL;
    State_Map::iterator it;

    // try to find this FSMstate in the map
    if( !m_map.empty() )
    {
        it = m_map.find( pNewState->GetID() );
        if( it != m_map.end() )
            pState = (FSMstate *)((*it).second);
    }

    // if the FSMstate object pointer is already in the map, return
    if( pState != NULL )
        return;

    // otherwise put the FSMstate object pointer into the map
    m_map.insert( SM_VT(pNewState->GetID(), pNewState) );
}
```

程序清单 3.1.8

```
void FSMclass::DeleteState( int iStateID )
{
   FSMstate *pState = NULL;
   State_Map::iterator it;

   // try to find this FSMstate in the map
   if( !m_map.empty() )
   {
      // get the iterator object of the FSMstate object pointer
      it = m_map.find( iStateID );
      if( it != m_map.end() )
         pState = (FSMstate *)((*it).second);
   }

   // confirm that the FSMstate is in the map
   if( pState != NULL &&
      pState->GetID() == iStateID )
   {
      m_map.erase( it );      // remove it from the map
      delete pState;      // delete the object itself
   }
}
```

程序清单 3.1.9

```
int FSMclass::StateTransition( int iInput )
{
   // the current state of the FSM must be set to have a transition
   if( !m_iCurrentState )
      return m_iCurrentState;

   // get the pointer to the FSMstate object that is the current state
   FSMstate *pState = GetState( m_iCurrentState );
   if( pState == NULL )
   {
      // signal that there is a problem
      m_iCurrentState = 0;
      return m_iCurrentState;
   }

   // now pass along the input transition value and let the FSMstate
   // do the really tough job of transitioning for the FSM, and save
   // off the output state returned as the new current state of the
   // FSM and return the output state to the calling process
   m_iCurrentState = pState->GetOutput( iInput );
   return m_iCurrentState;
}
```

3.1.6 参考文献

关于 FSM 的更多的信息可以在下面的链接上找到：

http://csr.uvic.ca/~mmania/machines/intro.htm

www.erlang/se/documentation/doc-4.7.3/doc/design_principles/fsm.html

www.microconsultants.com/tips/fsm/fsmartc1.htm

另外一个用 C++代码实现的 FSM 可以在下述网址找到：

http://uw7doc.sco.com/SDK_c++/CTOC-_Using_Simple_Finite_State_Machi.html

一个用 C 的实现可以在下述网址找到：

http://w3.execnet.com/lrs/Writing/Finite%20State%20Machines.html

3.2 博弈树

Jan Svarovsky

对于许多博弈问题，如国际象棋和西洋跳棋，可以定义一棵博弈树（game tree），这棵树上的节点是博弈状态，每个节点的子节点是能够从它通过一次走步到达的状态。在这些游戏中，一个电脑玩家能通过展开博弈树穷举所有的状态进行应对。也可以有一个估价函数，以试图量化一个特定的状态对一个玩家来说价值如何。由于时间约束的原因，搜索在一些点必须停止。在这些点将对剩余状态的值做一些估计。

假设对一个玩家有利的走步一定对另一个玩家不利，这个玩家想最大化盘面的估价函数值而另一个玩家想最小化它。这种“我得到就等于对手失去”类型的博弈称为“零和博弈（zero-sum game）”。例如，井字棋（tic-tac-toe）是一个零和博弈；井字棋博弈树的局部如图 3.2.1 所示。在 1 级深度搜索（one-level-deep）中，一个玩家显然只想得到能产生盘面估价函数定义的最佳盘面的走步。在 2 级深度搜索中，玩家 1 假定无论自己怎样走，玩家 2 都将接着进行前面描述过的 1 级搜索。所以，玩家 1 走任何一步都会使玩家 2 处于最为不利的境地（然后玩家 2 会选择最有利于自己的走步）。这些假设能被扩展到时间允许的任何可能级别的搜索。

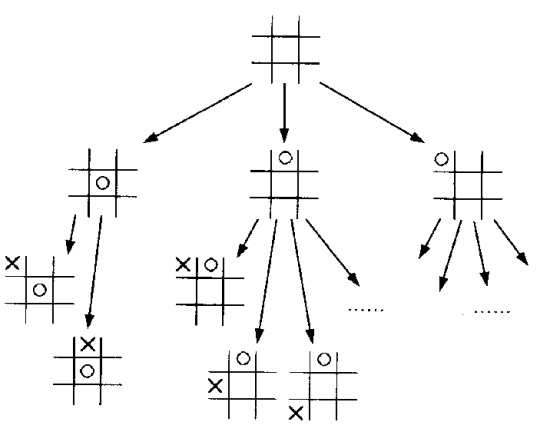


图 3.2.1 井字棋开局部分的博弈树

下面的函数（加上一个非常类似的对应函数 minimize( )）返回最佳期望盘面值，展望一个特定的数量。一般地，对于此刻的每一个有效走步，maximize( )都应该被调用一次，并且返回最佳值的那个走步将被选取。

```
int maximize(int ply)
{
    if (ply == 0 || game_over()) return evaluate_current_board();

    int best = -infinity;

    for (Move *m = first_available_move(); m != NULL;
    m = next_available_move())
    {
        make_move(m);
        int new_value = minimize(ply - 1);
        unmake_move(m);
        if (new_value > best) best = new_value;
    }
    return best;
}

Move *which_move_shall_I_take(int ply)
{
    Move *best_move;
    int best_value = -infinity;

    for (Move *m = first_available_move(); m != NULL;
    m = next_available_move())
    {
        make_move(m);
        int new_value = maximize(ply);
        unmake_move(m);
        if (new_value > best_value)
        {
            best_value = new_value;
            best_move = m;
}
        }
        return best_move;
}
```

3.2.1 极小极大算法的负极大改进算法

与其写两个函数，其中一个以最小化盘面状态为目标而另一个最大化盘面状态，倒不如加入一个否定运算将其简化为一个函数。注意现在估价函数必须为当前的玩家返回盘面的参数，而不总是返回较低值（较低值意味着对这个玩家有利的盘面，而较高的值对另一个玩家有利）。因此盘面状态必须包含或隐含下一步该哪个玩家走。

```
int negamax(int ply)
{
    if (ply == 0 || game_over()) return evaluate_current_board();
    int best = -infinity;
```

```
    for (Move *m = first_available_move(); m != NULL;
m = next_available_move())
    {
        make_move(m);
        int new_value = -negamax(ply - 1);
        unmake_move(m);
        if (new_value < best) best = new_value;
    }
    return best;
}
```

对该函数最有效的系统是将盘面/游戏（board/game）状态提出并被上述 make_move 函数和 unmake_move 函数取反，而且估价函数应被递增地计算而不是每一次被调用时都从零开始计算。

3.2.2 $\alpha-\beta$ 剪枝

Newell、Simon 和 Shaw 于 1958 年发明了 $\alpha-\beta$ 剪枝。这个概念基于这样的观察结果：在一些情形下，对博弈树的一些分枝进行进一步的研究是没有意义的，如图 3.2.2 所示。

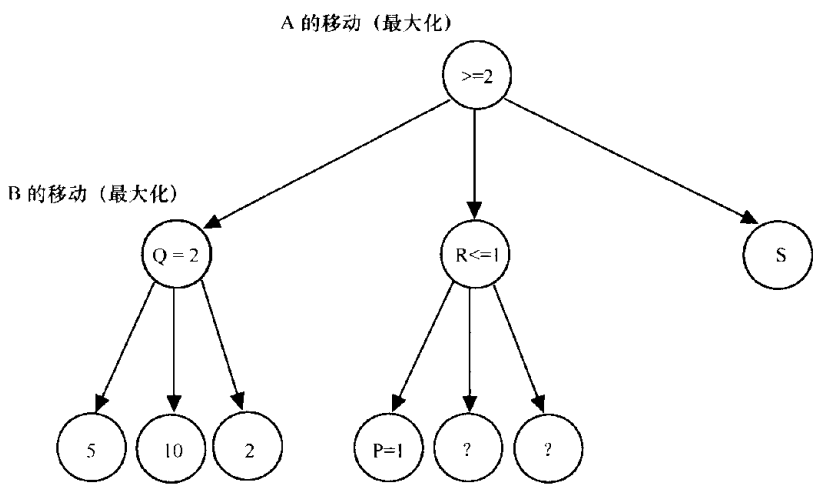


图 3.2.2 有时博弈树的一些分枝可以终止

这里，当我们看见玩家 B 移动 P 将产生一个盘面值 1 时，就知道玩家 A 将永远不会让 B 到达能移动 P 的点。玩家 A 已经知道他能迫使 B 进入情形 Q，这里 B 有望得到的最佳值是 2（记住，B 正试图极小化盘面值）。所以，探查 P 的兄弟节点就没有必要了，因为玩家 A 将永远不会令情形 R 发生。显然对 A 来说 R 要比 Q 更糟糕。

这个概念能被概括为：如果我们知道对方玩家能在另外的地方得到一个更好的结果，那么我们知道当前的盘面状态永远不会被那个玩家变为可达到的。现在搜索“剪掉”了 P 的剩余兄弟节点并直接到达 S。

这实际上意味着我们应该为搜索增加一个额外的参数。它是我们所知道的另一个玩家能基于目前搜索过的部分博弈树得到的最佳值。一旦当前搜索返回比这个“当前最佳”对我们来说更好（对另一个玩家来说更坏）的结果，我们知道就不必再在这儿搜索下去了。当然，这实际上变成了两个参数。一个是对手玩家迄今为止得到的最佳值（称为β），另一个是我们迄今所得到的最佳值（称为α）。α被传递给另一个玩家走步的递归调用，在此α和β被交换。

```
int alphabeta(int ply, int alpha, int beta)
{
    if (ply == 0 || game_over()) return evaluate_current_board();

    for (Move *m = first_available_move(); m != NULL;
    m = next_available_move())
    {
        make_move(m);
        int new_value = -alphabeta(ply - 1, -beta, -alpha);
        unmake_move(m);
        if (new_value > beta) return new_value; // prune
        if (new_value < alpha)
alpha = new_value; // update our "best so far"
    }
    return alpha;
}
```

可以看到这种方法相当理想，你可能想找到尽可能的剪枝步。这意味着你在每一点都首先考虑最佳走步。这大概看起来不可能（因为找到最佳走步是搜索的全部要点），但是实际上已经有一些方法存在，并且在实践中游戏程序差不多总能在正确排序方面取得成功。这种方法给出了搜索代价的理论上的平方根，意味着搜索能进行到深度的两倍。

3.2.3 走步排序方法

一种走步排序方法是 *iterated deepening*（深度迭代法）。在该方法中，不是在所有级别进行直进式搜索，而是搜索的级别逐渐增加，采用上一级的搜索结果来为下一级的走步排序。这种方法也许看起来像做了很多额外的工作，但是，由于搜索具有的指数性质，最后的迭代结果是到目前为止最重要的代价。

上一级的结果能被存储在一个哈希表中（参见[Sedgewick98]），存储计算过的盘面状态的值。这个表将盘面状态的哈希值为盘面值。它还有另一个作用：当不同的走步序列产生同样的博弈状态时，可以避免重复计算盘面值。

可以使用特定每种游戏的启发式方法来进一步优化算法，如在下棋时总是考虑抓住先机。最后，还有一个“杀手”试探：如果一个走步在博弈树的一个兄弟节点证明是最佳的，首先在这个点上试探。

3.2.4 $\alpha-\beta$求精

$\alpha-\beta$实际上是预期盘面值的一个下限和一个上限。α是下限，因为它是你期望能使盘面达到的最低值。你知道另一个玩家将不能使你得到超过β的任何值。如果你相当肯定$\alpha-\beta$搜索将返回什么值，可以不必选择从负无穷到正无穷开始，而是从一个你期望的返回值附近的某个范围开始。如果返回值击中范围的任何一边，我们知道结果实际上是超过这个范围的，所以必须扩大范围并重试。

由于“地平线效应”（horizon effect），固定深度的搜索会很糟糕。如果一个特别糟的走步不久将发生，计算机将会把其他走步在它们在搜索深度后走那很糟的一步时进行排序。这是因为如果糟糕的一步在很远的将来才能发生，它将不能看到。对于选择增加一些分枝的级别的时机有很多存在的方法，但是这些讨论超出了本文的讨论范围。

3.2.5 参考文献

[Eppstein] Eppstein, David, “Strategy and Board Game Programming,” www.ics.uci.edu/~eppstein/180a/970401.html.

[Sedgewick98] Sedgewick, R, *Algorithms in C++*, Addison-Wesley Longman, Inc.,, 1998.

3.3 A*路径规划基础

Bryan Stout

3.3.1 问题

在计算机游戏 AI 中，我们经常需要为一个智能主体在游戏世界规划出一条路径，以使其从来处到达另一处，本文对该问题的一种基本解决方案进行了研究。随书 CD 中含有 PathDemo 程序的一个副本，你可以用它轻松了解 A*（或其他的路径规划技术）是如何工作的。

在路径规划中最普通的问题是障碍物的躲避问题，包括对死胡同（cul-de-sacs）的忽略（有时是发现或探险）。其次可能就是对不同地形的识别，以及在探索公路或空旷的地形、绕开沼泽等众多选择中寻找出效率最高的路径。

3.3.2 方法概述

A*（读作“A 星”）算法在人工智能学术界中是一种历久弥新的算法。自 1968 年以来已用于解决各类问题，15 数码难题（15-puzzle）就是其中很受欢迎的教学实例。幸运的是，它对于路径规划问题也是非常有效的。

A*是一种在状态空间中进行搜索的算法，它主要是通过检查特定状态的相邻或邻接状态来搜索从起始状态到目标状态的最小代价路径。在 15 数码难题中，状态由 4×4 棋盘上的 15 个棋子的棋局组成，邻接状态通过滑动一个棋子到空白区域而达到。在路径规划问题中，状态由主体在游戏世界中占有的特定位置组成，一个邻接状态是通过移动主体到一个邻接位置而达到的。

本质上讲，A*算法反复检查它已经看到但是还没有搜索到的最有希望的位置。当一个位置被探索到而这个位置恰好就是目标时，A*算法结束；否则，它将为进一步的搜索记录这个位置的所有相邻位置。

更具体地说，A*跟踪记录两个状态表，称为 Open 状态表和 Closed 状态表，分别表示未检查状态和已检查状态。开始时 Closed 状态表是空的，Open 状态表仅包括初始状态（主体位于它自己的当前位置）。在每次迭代过程中，A*算法将 Open 表中最有希望的状态取出进行检查。如果这个状态不是目标状态，则它的相邻位置被排序：如果它们是新状态，将被放入 Open 表中；如果它们已经在 Open 表中且这是一个代价更低的路径，关于这些位置的信息将被更新；如果它们已经在 Closed 表中，由于已经被检查

过，它们将被忽略。如果在目标找到以前 Open 表就已变空，就意味着没有从那个起始位置到目标位置的路径。

在 Open 表中“最有希望的”状态必然是具有最低估价值的路径将通过的位置。每一个状态 X 包括有信息来确定这些状态：从开始状态到该状态的最小代价路径的代价（称为 CostFromStart(X)）；到达目标位置剩余距离的代价的一个启发式估价函数；以及整个路径的估价值（定义为 CostFromStart(X)+CostToGoal(X)）。整个路径的估价值是 TotalCost(X)的最小值，它决定的是下一个要检查的状态。此外，每一个状态保持一个指向其父状态（沿最小代价路径到达它的状态）的指针。当目标状态被找到时，由这些链接能回溯到起始状态，以便构造从起始状态到目标状态的路径。请注意，你会在文献中发现，有时也将 CostFromStart(X) 称为 g(X)，将 CostToGoal(X) 称为 h(X)，将 TotalCost(X)称为 f(X)。本文中为了表述得更清晰，将使用我们自己的命名。

程序清单 3.3.1 A*算法

这里是伪码形式的 A*算法：

```
Open: priorityqueue of searchnode
Closed: list of searchnode

AStarSearch( location StartLoc, location GoalLoc,
agenttype Agent ) {
    clear Open and Closed

    // initialize a start node
    StartNode.Loc = StartLoc
    StartNode.CostFromStart = 0
    StartNode.CostToGoal = PathCostEstimate( StartLoc,
        GoalLoc, Agent )
    StartNode.Parent = null
    push StartNode on Open

    // process the list until success or failure
    while Open is not empty {
        pop Node from Open    // Node has lowest TotalCost

        // if at a goal, we're done
        if (Node is a goal node) {
            construct a path backward from Node to StartLoc
            return success
        } else {
            for each successor NewNode of Node {
                NewCost = Node.CostFromStart + TraverseCost( Node,
                NewNode, Agent )
                // ignore this node if exists and no improvement
                if (NewNode is in Open or Closed) and
                (NewNode.CostFromStart <= NewCost) {
                    continue
```

```
                } else {      // store the new or
                              // improved information
                    NewNode.Parent = Node
                    NewNode.CostFromStart = NewCost
                    NewNode.CostToGoal = PathCostEstimate( NewNode.Loc,
                    GoalLoc, Agent )
                    NewNode.TotalCost = NewNode.CostFromStart +
                    NewNode.CostToGoal
                    if (NewNode is in Closed) {
                        remove NewNode from Closed
                    }
                    if (NewNode is in Open) {
                        adjust NewNode's location in Open
                    } else {
                        push NewNode onto Open
                    }
                }
            } // now done with Node
        }
        push Node onto Closed
    }
    return failure     // if no path found and Open is empty
}
```

3.3.3 A*的特性

A*有几个很有用的特性（在此没有进行证明，有兴趣的读者可以在参考文献中查找）。首先，A*能找到一条从起点到终点的路径（如果存在）。其次，只要 CostToGoal(X)估价值是可纳的，它就一定能找到一条最佳路径。这意味着 CostToGoal 总是比实际值估计得低。也就是说，CostToGoal(X)总是小于或等于从 X 到目标位置的实际的最小代价。第三，A*使启发式得到了最有效的应用：如果不计切断具有同样代价的状态间的联系，使用同一启发式函数 CostToGoal(X)去寻找最佳路径时，没有哪一种搜索算法检查的状态会比 A*算法少。

3.3.4 将 A*应用到游戏路径规划

让我们具体来看一下 A*的各个方面是如何应用到计算机游戏中的路径规划上的。关于这个问题的讨论极大地取决于游戏的种类和游戏世界的内部表示。下面的讨论将对其可能的应用提出一些建议。

1. 状态

如上所述，在路径搜索中状态的基本组成部分是位置。然而它并不是惟一的成分。一个主体的朝向或速度同样重要。比如，汽车经常是照直往前走或者稍微转弯，转弯的次数越少汽车运行得就越快，大多数汽车只有在停车才能后退。仅仅依据位置规划一条路径在大多数情况下是可以接受的，但是有时为了避免设计出很难导航的穿过地形或绕过障碍物的路径，

基于速度和朝向来进行规划将会更令人满意。

即使单独考虑位置，究竟选择哪一个位置也颇费思量。在一些游戏中，世界是自然平铺的——实时策略游戏常常使用正方形栅格，许多战争游戏则使用一种可视的六角形栅格。但是许多游戏并不那样划分空间，尤其是那种使用 3D 第一人称视角和世界的斜视图的游戏。在这些情况下，选择一个从中进行搜索的位置集是很重要的。图 3.3.1 展示了一种路径规划的情形及几种划分空间的方式。

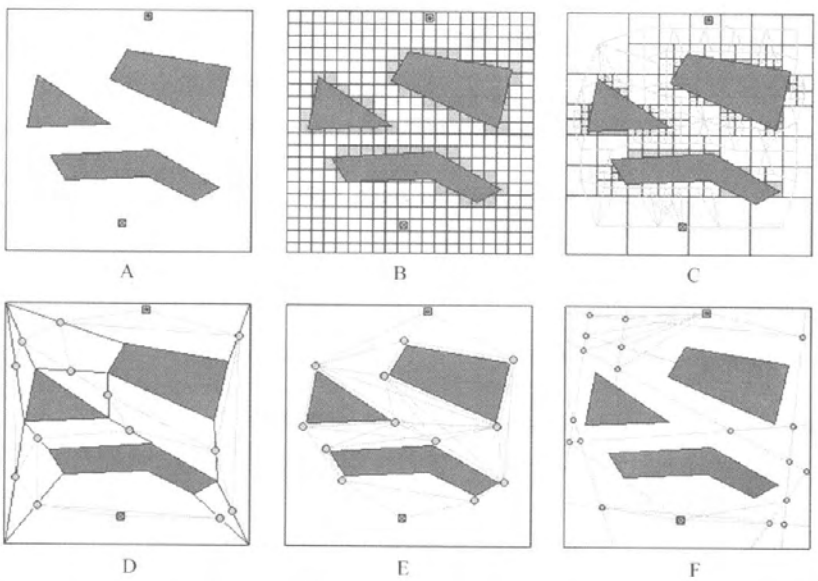


图 3.3.1 划分一个连续空间的几种方式

划分空间的几种方式如下：

- **矩形栅格**。最简单的划分方式是将空间分割为规则的正方形栅格，如图 3.3.1b 所示。位置既可以是正方形的中心点也可以位于它的角落；如果供游戏之用，栅格可以被认为是由它所覆盖的最共同的地形所组成。

- **四叉树**。另一种划分空间的方式是将其划分为不同大小的正方形。用四叉树的方法递归地将一个正方形划分为四个更小的正方形，直到每一个正方形都有一致的（或至少大部分一致）地形，如图 3.3.1c 所示。同样，位置即可以是正方形的中心点也可以位于它的角落。该方法有两个优势：正方形越大（因而就越少），那么对其搜索就更快，其表现形式也更容易存储。

- **凸多边形**。一个更复杂的但可能更健壮的办法是将空间分解为由一致的地形构成的凸多边形。这种方法可能已经用于地图的表示了，因而它能直接使用于路径搜索。如果现有的网格是无用的或低效的，有多个方法能用于将空间分割成多边形。C-cells 是一种分割空间的方法，每个顶点与可见的最近的顶点相连接，而连线分割了空间。另一种是最大区域分解（maximum-area decomposition），在每个凸顶点，与之连接的边被发射出去直到碰撞上障碍物或墙壁，而且在这些线和连接到其他最近顶点的连线中，最短的被选作一条边界。第三种方法就是本卷中其他文章中讨论过的导航网格（3.6 简化的 3D 运动和使用导航网格的寻径）。

类似的技术能用于将可变代价地形划分为具有一致地形的凸多边形。多边形设计好之后，搜索位置能选择在它们的中心和/或它们的边界内的不同的部分。

- **可见点**。并非所有的技术都是将空间分解为区域，有些技术代之以直接指出位置。可见点主要关心的是障碍物的躲避：在每个障碍物的凸顶点的附近放置一个搜索位置，正好能避免与障碍物碰撞就可以了（如图 3.3.1e 所示）。围绕障碍物的最短路径在些顶点近旁经过，就好像一个橡胶圈将起始位置和目标位置连接起来。你或许可以通过将这些点增加到那些从一致凸多边形得到的路径上，把这个方法扩充到考虑地形代价。
- **广义圆柱体**。另外一种主要考虑障碍物躲避的技术是广义圆柱体：相邻障碍物之间的空间可以被看作是一个 2D 圆柱体，它的外形在行进中会改变。在每两个相邻的障碍物之间（包括墙壁或地图的边界），计算一个中心轴线（如图 3.3.1f）。这些线的交点为搜索提供了位置。

对于这些方案中的大多数来说，当一个搜索完成的时候，起始位置和目标位置常常不在搜索位置的集合中，所以它们需要在搜索过程中动态地被添加到原有的位置集合中。

无论哪个方案用于量化连续的空间，我们或许都需要进行实验和改进，以使它能最佳地满足游戏要求：要有足够多的位置才不会忽略合理的路径；但位置又不能太多，否则搜索过程将花费太长的时间。另一个问题是，从任何量化方案找到的路径大都看起来是锯齿状和不太真实的，这意味着路径需要被平滑，不管是在它被指派给主体之前还是在主体沿着这个路径前进的过程中。

2. 相邻状态

状态的相邻状态取决于地图的表示和量化方式。一些量化方式仅仅使用他们的邻接位置作为相邻状态——正方形栅格将认为每个内在点有 8 个相邻状态，如果把对角线上的排除在外就有 4 个——然而在另一些量化方式里，一个位置的相邻状态是它可以看到的所有其他位置。

位置的相邻状态也取决于地形。有的地形也许是不可逾越的，意味着它根本不是其附近位置的相邻状态。主体的类型也是一个决定因素。例如，陆上汽车不能在海上行驶，步兵能横越某些汽车无法穿过的地方。

为了加快搜索速度，我们需要一个有效的方法来计算每个位置的相邻状态。栅格有一个计算相邻位置的自然方法。(x, y) 的相邻位置可以是 $(x+1, y)$、$(x+1, y+1)$，以及 $(x, y+1)$ 等等。由于计算相邻位置的代价常常较高，其他多数模式要求用一些数据结构存储相邻状态的信息以便进行快速查找。

3. 代价

两个位置之间路径的代价函数值（即上述 CostFromStart）代表假定要最小化的任何事物，通常它可以是行进的距离、横越的时间、花费的运动点数，或消耗的燃料。然而其他因素也能加入到代价函数中，比如对于穿越不合需要区域的惩罚和一些审美上的考虑（例如，使斜穿运动的代价比直线运动的代价要高，即使做不到这一点，也要使目标路径看起来更直；请参见关于审美优化的文章“3.4 A*审美优化”中进一步的讨论）。

正像前面考虑过的连接性，在很多游戏中，对所有的主体来说代价可能是不一样的——例

如：公路适宜于有轮机动车极大地加速行进，而适合于步兵的道路可能就不太适合加速前进了。而且，在许多游戏中，旅行代价是不对称的，从 A 到 B 去可能比从 B 到 A 代价要高（比如从 A 到 B 是上坡）。这就是为什么在程序清单 3.3.1 中代码的代价函数依赖于旅行的主体及旅行的两个端点。

此外，在搜索中这些地形代价需要被迅速查找出来，实际上它可能最好与连接性信息一起储存。这样一次查找就可以确定两个位置是否相邻，如果是，就为特定主体找到代价了。

4. 估价

对到达目标的路径代价的估计，是从起始位置开始的已知距离的补充。如果你想要确保发现一条最佳路径的话，这段距离不应被高估。一个常用的做法是用从给定位置到目标的实际的地图距离乘以每单位距离的最小地形代价。因为路径不可能比最直接的“乌鸦飞行线”还短，这个数字将是一个低估值（除非你的游戏有类似于即时定位系统的东西）。在很多游戏中，这个最小距离是 2D 或者 3D 空间中两个点之间的欧氏距离，但是当游戏中涉及严格的正方形或者六角形块的时候，最短的块路径通常比在块的中心点之间的欧氏距离稍长些。因此，在一个正方形的栅格中，一个远处的块（3，5）有一个最小距离 2 + 3 * sqrt(2)，而并非欧氏距离的 sqrt(34)。这个实际上的最短距离然后被乘以一个典型的地形代价。这个代价应该包括以前讨论的所有涉及相邻节点之间代价的因素。

但是，保证一条最佳路径并不是惟一的考虑因素，还要考虑搜索的速度，并且 CostToGoal 值的质量对搜索效率有极大的影响。请看图 3.3.2 和 3.3.3。在图 3.3.2a 中，CostToGoal 被设置为零，实际上被低估了，并且我们看到在到达目标之前搜索是进行环形扩展的，因为没有启发式的信息指导它在正确的方向上进行搜索。在图 3.3.2 b 中，我们看见每正方形的一个精确的权值 1 使搜索沿着一条直线接近目标。在图 3.3.3 中，起点和终点在代价较高的地形中（每正方形代价为 8）。因为估价值 1 是对它的过低的估计，搜索的边界接近于图 3.3.2 a 中非启发式搜索的圆环。在图 3.3.3 b 中，我们看见甚至一个每正方形 5 的估价也对搜索具有相当的指导性。因此估价值很重要，当估价相当准确时，或许是决定性的。实际上，在某些情形下，为了得到快速搜索，冒着得到一条次优路径的危险（文章“A*速度优化”讨论了这个概念），我们可以对代价进行高估（overestimate）。因此不是使用每单位的一种最小地形代价，而是使用一个典型的代价，这个代价或者是固定的，或者在一定的起点与终点之间通过取样来动态决定。

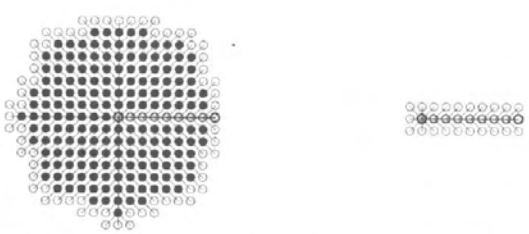


图 3.3.2 在空旷地形中进行 A*搜索 a:有一个启发式权值 0 b：有一个启发式权值 1

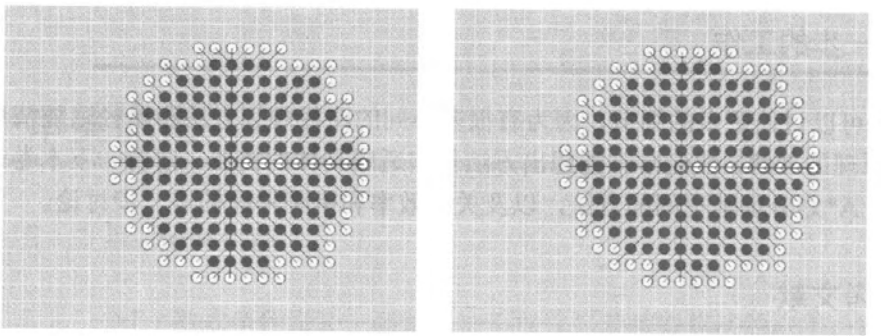


图 3.3.3 A* 在代价较高的地形中搜索 a:有一个启发式权值 1 b：有一个启发式权值 5

5. 目标

典型的目标是一个单独的位置，但也不必一定如此。例如，如果一个油量很低的汽车正试图规划一个到最近的加油站的路径，每一个搜索中的新节点 *N* 都对到每个加油站的剩余距离做了一个估价，而且其中的最小值被用作 CostToGoal(N)值。这个方法保证了搜索过程同时解决了最近目标与最佳路径两个问题。

3.3.5 A*的弱点

虽然 A*是一种较好的搜索算法，但是必须明智地使用它。否则，将会造成资源浪费。在一张大地图上，Open 表和 Closed 表中可能有数百甚至数千个节点，占据的存储空间可能比存储较小的系统（例如控制台系统）提供的空间更多。在任何系统上，A*都可能由于花费太多 CPU 时间而不可接受。

A*效率最差的情形就是用它来判断开始位置和目标位置之间是否无路径。那样的话，它将在确定目标不在这些位置集合中之前，检查从位置出发的每个可达位置（如图 3.3.4 中所示）。避免这个问题的最好方法是用手工或者用算法做一个地图的预分析（pre-analysis），以使程序能查找两个位置是否互相可达，比方说在同一个岛屿上。如果不能，搜索工作根本就不会开始。

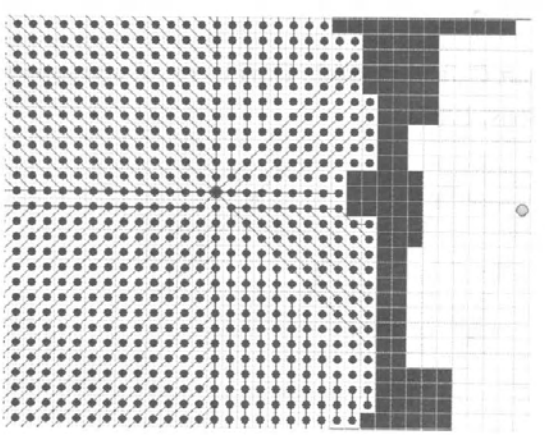


图 3.3.4 在不存在通向目标的路径的情形下进行搜索

3.3.6 进一步的工作

我们还可以讨论更多的细节，因为有很多不同的路径规划方法和路径追随情形。有了对A*的工作原理的理解，我们常常能想出方法使它适应具体的需要。请看一下本卷中的其他文章中关于用A*划分场地空间的方法，以及关于效率和审美考虑的进一步讨论。

3.3.7 参考文献

1. 网站

[Woodcock] Woodcock, Steven, "The Game AI Page: Building Artificial Intelligence into Games", www.gameai.com.

2. 综合性的AI教科书

下面是两本近来非常好的综合性的AI教科书，它们正好都采用了以主体为中心的模式来讨论AI：

[Russell95] Russell, Stuart, and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.

[Nilsson98] Nilsson, Nils J., *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

3. 研究性的教科书

下面这些书探讨了搜索的一般问题，可以追溯到AI研究的早期：

[Barr81] Barr, Avron, and Feigenbaum, Edward A., eds., *The Handbook of Artificial Intelligence*, volume 1, Addison-Wesley, 1981.

[Kanal88] Kanal, L., and Kumar, V., eds., *Search in Artificial Intelligence*, Springer-Verlag, 1988.

[Pearl84] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[Shapiro] Shapiro, Stuart C., and Eckroth, David, eds., *Encyclopedia of Artificial Intelligence*, 2 volumes, John Wiley & Sons, 1987.

3.4 A*审美优化

Steve Rabin

为游戏角色设计一条路径决不仅仅只是搜索算法的应用。它也包括了构造一个令人赏心悦目的路径和执行结果。为角色设计的路径可以用以下三个主要方法加以改进：使路径更直、使路径更平滑和使路径更直接。简单地通过最大化响应率就可以改善路径的执行效果。所有这些优化为玩家带来了审美上更加愉悦的体验。由于提供一种令人满意的体验是游戏的终极目标，这些优化相当重要并直接影响了A*的内部与外部编码。

3.4.1 直路径

由A*计算出的路径看起来常常像是被醉汉构造出来的一样。它们摇摇摆摆地到达目的地，虽然效率较高但看上去确实不太自然。这是一个破坏了所有游戏AI可信度的严重问题。解决这个问题有两种方法：其一是促进A*算法中直路径的构造；其二是在计算完路径后再对其进行清理。

构造更直的路径包含仔细估算A*算法内部的额外开销。请观察一下图3.4.1和图3.4.2所示的由A*计算出的两条路径。

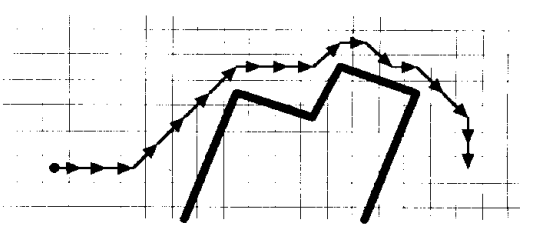


图3.4.1 典型的A*路径

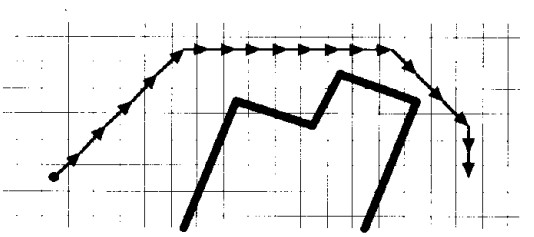


图3.4.2 取直的A*路径

令人惊讶的结果是这两条路径竟然经过了相同的距离！由于两条路径

有相同的代价，A*不能区分它们，因而简单地选择了它偶然发现的第一条路径。让 A*选择直路径的诀窍是代价函数。如果正被考虑的新移步与上一步不是以直线相连，就加入一个额外代价（处罚）因子。请注意我们关注的并不是整条路径的直线性，而仅仅是对与上一步不在一条直线上的新移步进行处罚。

一个合理的处罚值是取给定方向正常代价的一半。事实上，在一个形状规则的栅格上，对非直选择的任何程度的处罚（即使是 0.0000001）都会促使 A*选择直路径。然而，并不是在任意一个网络上都是如此。

在此一句忠告：对非直路径进行处罚将导致导航器做更多的工作，从而延缓计算进程。显然，为了找到最直的路径，这个算法必须考虑更多的变更。事实上搜索需要占用相当长的时间，但是如果采用分级方法，额外时间就不成问题了。请务必理解这其中的折衷。

3.4.2 多边形搜索空间中的直路径

对于多边形搜索空间，上面的技巧就不太灵了。因为三角形并不像矩形栅格那样形状一致，相似路径代价一样的情况非常罕见。因此没有必要寻找更直的路径了。相反，存在另一个问题：既然三角形在大小和面积上变化多端，那么路径将比以往更弯曲。我们可以在路径计算出之后对它的直线性进行优化。Greg Snook 的寻径文章“3.6 简化的 3D 运动和使用导航网格进行寻径”讨论了一种极好的解决这个问题的方法 。

3.4.3 平滑路径

遗憾的是，通过 A*计算的路径通常因有过多的急转弯而显得漏洞百出。即使你想方设法将路径变得更直，急转弯依然会使你的角色看上去就像机器人。对急转弯应用转动减幅（rotational dampening）技术，可以稍稍掩饰一下，但是在每个锐角转角处将摆动得很厉害。有一个更好的方法可以解决这个问题。

计算机图形学中有一个现成的算法可以使你的路径（只不过是空间中的点序列）变得平滑。一个简单的 Catmull-Rom 样条就能够完成这项任务，因为它产生了一条能够经过原始路径中所有控制点的曲线（不像 Bezier 曲线，虽然比较平滑但是不能经过所有的控制点）。显然，直接经过所有点比较好，因为 A*认为他们是通畅的并且避免了障碍物的妨碍。

可是如何从实际输入的点序列获得一个更平滑的点序列呢？Catmull-Rom 公式要求 4 个输入点，然后给出一条位于第 2 点和第 3 点之间的光滑曲线。图 3.4.3 更好地解释了这个概念。

为了得到第 1 个输入点和第 2 个输入点之间的点，你可以为这个函数输入第 1 个点两次，接着输入第 2 点和第 3 点。为了获得第 3 点和第 4 点之间的点，你可以先为函数输入第 2 点和第 3 点，然后输入第 4 点两次。

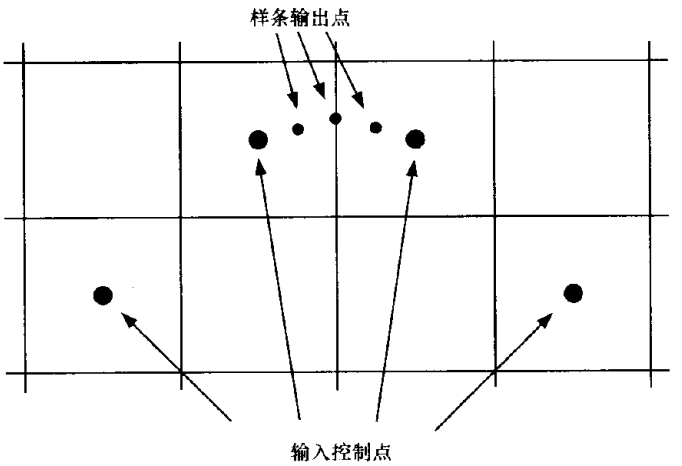


图 3.4.3 从控制点得到样条点

每次使用 Catmull-Rom 公式时，它将在第 2 个点和第 3 个输入点间近似 **u**%的地方给出一个点，这里 **u** 是你自己传递的一个数。公式如下（其中的点可以是 2D 的也可以是 3D 的）：

```
output_point = point_1 * (-0.5f*u*u*u + u*u - 0.5f*u) +
               point_2 * (1.5f*u*u*u + -2.5f*u*u + 1.0f) +
               point_3 * (-1.5f*u*u*u + 2.0f*u*u + 0.5f*u) +
               point_4 * (0.5f*u*u*u - 0.5f*u*u);
```

注意：如果 **u** 是零，结果是 point_2；当 **u** 等于 1 时，结果为 point_3。就像你看到的，样条确实直接经过了输入点。

3.4.4 预先计算的 Catmull-Rom 公式

既然速度是个问题，你可能想规定 **u** 仅仅取某几个间隔，比如 0.0、0.25、0.5 和 0.75。通过固定 **u** 的所有可能取值，你可以对每个 **u** 预先计算公式。注意：这些公式既适用于 2D 点也适用于 3D 点。以下是在不同间隔下的公式：

```
// u = 0.0
output_point =   point_2;

// u = 0.25
output_point =   point_1 * -0.0703125f + point_2 * 0.8671875f +
                 point_3 * 0.2265625f + point_4 * -0.0234375f;

// u = 0.5
output_point =   point_1 * -0.0625f + point_2 * 0.5625f +
                 point_3 * 0.5625f + point_4 * -0.0625f;

// u = 0.75
output_point =   point_1 * -0.0234375f + point_2 * 0.226563f +
```

```
                point_3 * 0.8671875f + point_4 * -0.0703125f;

// u = 1.0
output_point =    point_3;
```

有了 Catmull-Rom 公式，你所需要做的就是沿着 A*找到的路径走一遍并构造一条新路径。记住，在开始时要把第 1 个点输入两次，快结束时输入最后一个点两次。如果 A*路径只有两个点，就不要运用样条了。

新路径的点数是原来的 4 倍，你或许想消除其中的冗余点。让新路径通过一个剪除路径共线点的函数将显著地缩减点序列。

图 3.4.4 展示了运用 Catmull-Rom 样条前后的一条典型路径。注意它依然是一个点序列（piece-wise linear），但是路径现在变得更平滑了。对于大规模的实例，这个路径是极为平滑的。

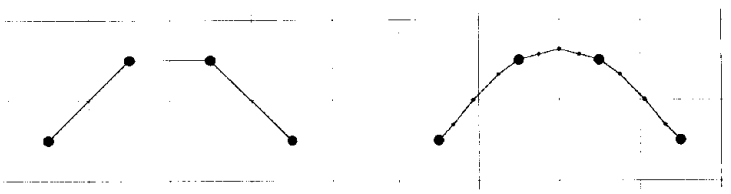


图 3.4.4 应用样条前后的路径点

3.4.5 改进分级路径的直接性

一种非常重要的 A*技术是分级路径（hierarchical pathing）技术。问题是，由此产生的路径可能不那么理想。应用分级路径时，在两个不同的阶段进行寻径：首先找到大范围的路径，然后再在局部级别进行寻径。例如，一个城堡可以被分成一个个的房间，你可能想从地牢到达到王宫。解决的办法是首先寻找房间之间的大范围的路径（通过在房间的连接性图上运行 A*），一旦发现了那条路径，你就可以在遇到的每个连接的房间之间进行寻径了。这样做节省了大量的时间。但令人遗憾的是，整条路径看起来却相当糟糕，因为目标点将总是隔壁房间的门，致使角色总要穿过每扇门的中心。当门任意大时，看起来就更糟糕了。图 3.4.5 说明了这个问题。

有一种简单、雅致的方法能得到理想的路径，但是它要花费 2 倍的计算时间。可以采用这样的技巧：总是寻找到隔壁的门之外的门的路径。然后，每当角色穿过第一个门口时，抛弃剩余路径并且重复这一过程。当道路的下半部分总被废弃时，它确实构造了最直接和最美观的路线。图 3.4.6 展示了最后的路径。

因为考虑到了将来的路径，该技术总能发现穿过门口的最佳通路。下面的例子是一个按步骤的寻径执行指南，说明了这个方法是如何工作的：

（1）在房间的连接图上用 A*寻找房间到房间的最好路径。

（2）结果是依次经过下面的房间序列：1，2，3，4。

（3）从起始点到子目标 1 进行寻径（如图 3.4.7a）。

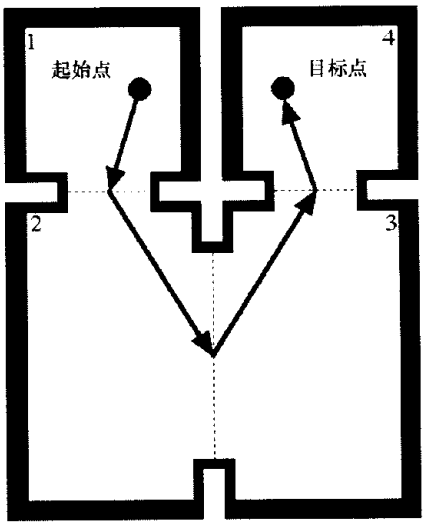


图 3.4.5 穿过几个房间的路径

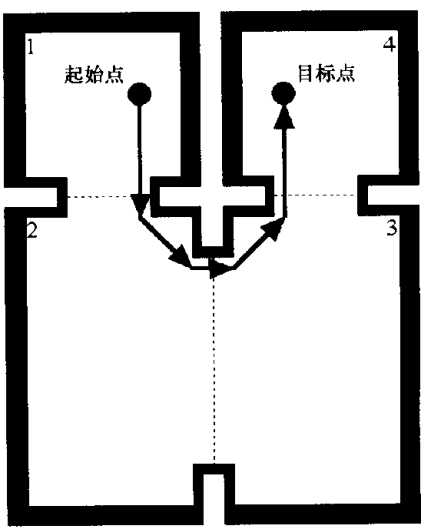
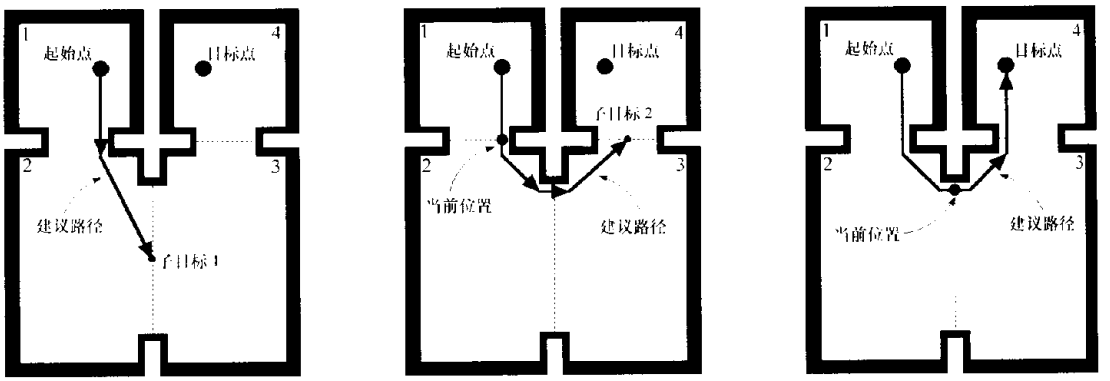


图 3.4.6 优化过的穿过几个房间的路径

（4）让角色一直走，直到他进入房间 2。
（5）抛弃剩余的路径并寻找到子目标 2 的路径。
（6）让角色一直走，直到他进入房间 3。
（7）抛弃剩余的路径并对到目标的路径进行寻找。
（8）令角色走向最终目标。

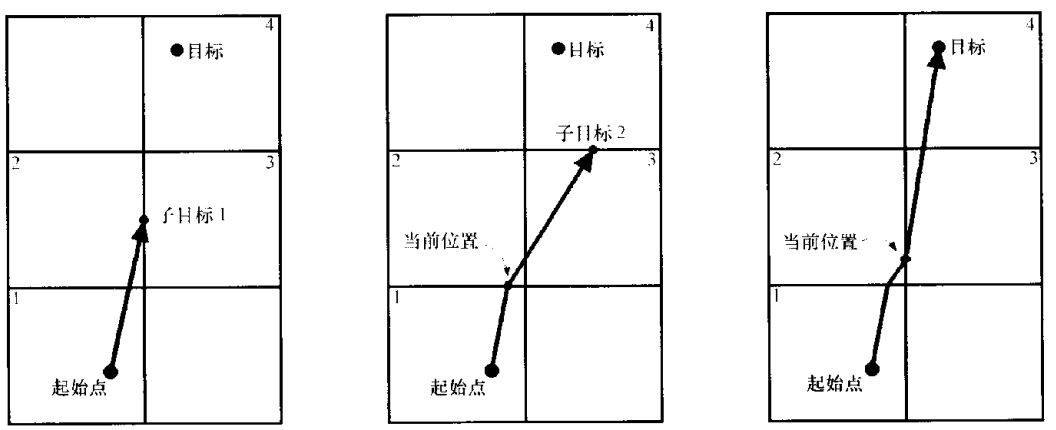


a. 到步骤（3） b. 到步骤（5） c. 到步骤（7）

图 3.4.7 在不同步骤中的计算过的路径

3.4.6 空旷区域上的分级寻径

一组连接的房间和一组连接的区域之间并没有真正的差别。可以在空旷区域上应用同样的原理，得到的路径虽不十分直，但非常接近。你也需要认识到这些区域对于玩家来说是相当大的。图 3.4.8 展示了用于空旷区域的分级寻径步骤。



a. 到步骤（3） b. 到步骤（5） c. 到步骤（7）

图 3.4.8 在不同步骤过程中计算过的路径

3.4.7 在分级搜寻过程中减少停顿

注意每当角色进入一个新房间时，必须计算一条新的局部路径，这显然要花时间。当角色进入每个门口时就会表现出停顿。搜寻本身不能被加速了，但新路径请求可以被预料并在之前进行大略计算。这个简单的技巧使得角色在贯穿路径时保持了流畅的运动。

3.4.8 最大化响应率

控制器响应率对玩游戏来说是决定性的。当玩家在RTS游戏中发出一个行动命令时，期望个体能立即响应。但是在寻径和搜索算法世界中，有时要花费一段时间寻找那条路径。我们需要在此使用一些技巧以便为玩家提供即时响应的感觉。

第一个技巧是通过播放一种确定个体接收到了命令的声音以进行迟延。这个技巧给出了即时反馈，使个体意识到命令并将立刻执行它。同时，导航器以全速尝试发现路径。

另一技巧是通过播放一个“准备好移动（get ready to move）”的动画实现延迟。角色的运动表示他将要移动，即使他实际上可以一步也不走。你甚至可以使角色转向“最可能（best-guess）”的方向，以便当最后的路径找到时，他已经准备好移动了，这样迟延时间甚至可以更长。

你可以通过在最终的路径准备好之前就使你的角色移向“最可能猜中”的方向，使这个思想再深化一步。不幸的是，你可能会完全猜错，并且角色必须原路返回。除非导航器极慢，否则最好避免使用这种方法。

移动成群的人花费的时间更长。如果玩家抓住20个个体并且要他们穿越地图，你可能要等待很长的时间才能得到处理过的20条路径。对付这种情形有两个技巧。第一个技巧是将路径请求排队并且让每个个体在它的请求被服务时才运动。这种方法看来似乎有即时反馈，因为至少有一些个体开始立即运动。当每个个体开始运动的时候有一点像爆炒玉米花，但总的来说，对玩家是相当满意的。第二个技巧是在20个个体的小组中选择一个头目，并且只为他寻找一条道路。然后告诉其他19个个体跟随这个头目前进。但是，这种方法可能变得非常复杂，因为可能有大规模的团伙，头目可能在中途死掉，并且每个个体应该最终停止在惟一的目的地。

3.4.9 结论

所有这些技术都被设计用来使寻径对玩家更透明。目标永远是即时找到好的直接路径。由于这是一个很困难的问题，希望你能把这些技术精华用于目前的导航器，以便获得看起来更好且最终让玩家感觉更满意的路径。

3.4.10 参考文献

[Patel99] Patel, Amit J., “Amit’s Thoughts on Pathfinding,” http://theory.stanford.edu/~amitp/GameProgramming/, November 27, 1999.

[Stout96] Stout, W. Bryan, “Smart Moves: Intelligent Path-Finding,” *Game Developer*, October/November 1996, pp. 28–35, www.gamasutra.com/features/19990212/sm_01.htm.

3.5 A*速度优化

Steve Rabin

传统的A*是一个很慢的算法，永远达不到你希望的那么快。由于已有许多对其进行优化的算法，因此理解它为什么慢是很有必要的，从而使你能充分利用优化。

关于A*首先需要注意的是它受搜索空间的支配。通常，待搜索连接的绝对数目是一个A*能工作多快的指示数字。在一个1 000×1 000正方形的矩形栅格内，有1 000 000个可能搜索的正方形。不管如何优化你的代码，在这类世界中寻找一条任意的路径都要做大量工作。解决的办法是对搜索空间进行优化。

一旦搜索空间得到优化，就该更深入地观察实际的A*实现了。由于A*可能消耗大量的存储器，优化存储地址分配和每个数据存取就是至关重要的。A*也需要大量的排序，但是使用某些专门的数据结构可以迅速有效地进行处理。

最后，加速A*的最好方法就是根本不把它用于简单的情形。可以构造某种测试以确定你是否绝对需要触发导航器。很多时候并不需要使用完整的A*实现就可以确定简单的路径。例如，试着运行一个盲目通向目标的直线路径，测试一下看看它是否与墙或者其他物体相撞。无庸置疑，有时候这个简单的解决办法将会得到极好的效果。

3.5.1 搜索空间优化

1. 简化搜索空间

最大的胜利往往来自于在较少数据中进行搜索。如果你用一个简化的连接图来表示世界，A*将大大加速工作。实际上可以有几种选择。由于速度并不是惟一考虑的因素，其他一些正反两方意见也在这里进行了讨论。图3.5.1提供了每种技术的简单图解。

2. 矩形或六角形栅格

描述

一种一致的矩形或者六角形的栅格被覆盖到世界上。每个栅格空间的大小与最小的角色的尺寸成比例。因此，在A*搜索过程中一个栅格空间内的角色锁定了那个空间。参见图3.5.1A。

正方

- 考虑到躲避，障碍物和角色能很容易地在栅格内被标明。这样产生了一个一步到位的解决办法来找到一条经过静态和动态物体的路径。
- 特别适合于基于 2D 平铺的世界。

反方

- 通常会产生最大的搜索空间。
- 矩形的栅格与 3D 世界匹配得不是很好。
- 路径看上去像是在一个棋盘上移动。

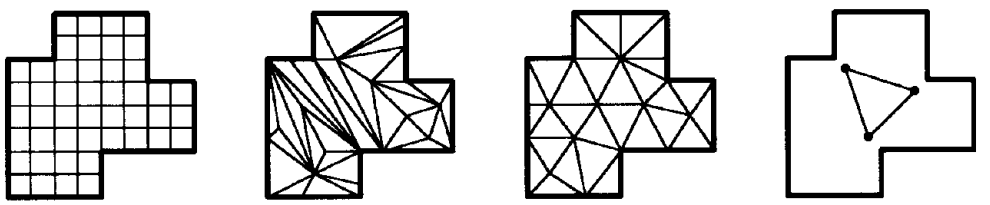


图 3.5.1 表示搜索空间的 4 种选择

3. 真实的多边形地板

描述

在 3D 游戏世界中，地板多边形被明确标明并且作为搜索空间被直接使用。这种多边形地板与渲染几何图形是相同的，因此繁易程度可以随心所欲。参见图 3.5.1B。

正方

- 数据结构已经在 3D 世界中存在。
- 能够通过 BSP 树较为容易地实现。

反方

- 三维世界中的地板上可能有任意数目的多边形。
- 不能表现诸如桌子或椅子等的障碍物（因为地板存在于这些物体之下）。
- 需要有在一个多边形内选择路径点的算法解决方案。

4. 多边形地板表示

描述

由一位艺术人员或者级别设计师创建一个专门用于寻径的多边形的地板表示。多边形可以在不允许角色走动的地方被消除，例如在桌子或椅子下。参见如图 3.5.1C。

正方

- 小搜索空间的表示。
- 能够通过 BSP 树快速实现。
- 障碍物可以一并表示。

反方

- 要求由艺术人员或级别设计师构造。

- 不能表示空间内的角色。
- 对于在多边形内选择路径点需要有算法解决方案。

5. 可见点

描述

点被放置在世界中的凸角上（从每个拐角处稍微伸出去）。然后每个点与它能“看见”的所有其他点相连接。这样建立了一张连接图，这张图描述了要求的绕开墙体的最小路径。如图 3.5.1D 所示。

正方

- 能创建最小的搜索空间表示。
- 障碍物可以一并表示。
- 产生的路径相当直。

反方

- 需要算法或设计师辅助建立连接图。
- 如果障碍物被毁坏，不能从图中删除。
- 不能表示空间内的角色。
- 不大适合宽度大的实体，比如一个大块头的角色。
- 有弯曲墙体的世界能使连接图变得不必要地复杂。

如你所见，没有显而易见的选择。每种表示法都有它自己的利弊。如果你的世界是有极少动态障碍物的 3D，那么使用可见点是一种合理的选择。如果你的世界是基于平铺的 2D 或者有成群的运动角色（如在一个大的 RTS 游戏内），那么矩形栅格可能是最好的选择。请记住，你对于搜索空间表示法的选择对速度和适应性有巨大地影响。

6. 可见点解释

既然可见点是一种极其可行的选择，值得对它做一些更多地解释。这种技术要求你逐步建立一张能周游世界的图。点被放置在凸出的拐角处并且与他们能够看见的所有其他点相连接。这就像为绕开墙体的惟一目的建造了一个超速干道系统。现在的问题是你怎样能上下这个超速干道。

为了登上超速干道，需要测试在高速公路上起始点和每一个点之间的可见度。因为你可能会比较数千个点，所以使用其他空间划分技术（像分级搜索那样）是很重要的。一旦你有了一个潜在的上斜坡（on-ramp）点的序列，就把它们放入 A*的 Open 表中并开始运行该算法。对于探索的每个点，必须用目标点测试它的可见度。如果你发现一个点能看见目标点，则得到一个潜在的下斜坡（off-ramp）点。图 3.5.2 显示了一个简单的例子。

7. 分级寻径

分级寻径是一种加快寻径进程的强大技术。不管使用的是哪一种搜索空间表示法，这个技术实际上都能简化搜索空间。因此如果你的世界表示比较大，也仍然有希望。关键是要对世界进行分级分解。

考虑一座城堡。可以认为它是一个单独的建筑物或者由门相连接的房间的集合（一张大

范围的连接图)。导航器在两个不同的阶段工作。知道了起始和终止房间，它首先寻找房间到房间的路径。一旦得到了房间到房间的路径，导航器就继续在当前房间到序列中下一个房间的微型路径问题上工作。这样，在进行第一步之前，导航器并不需要计算整条路径。那些微型路径在“需要知道”的基础上，在每个新房间被进入时才被算出。这种方法显著地减少了搜索空间和最终花费的计算路径时间。

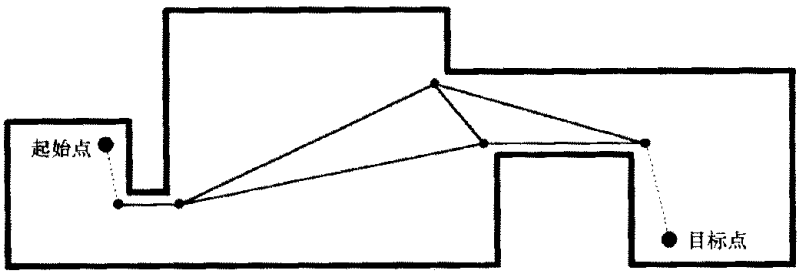


图 3.5.2 可见点的例子

如果你的世界已经被分级构造，这种技术真的是好极了。3D 世界也能采用一种简单的堆积木模式（building-block paradigm）来构造。下面考虑一个螺旋形的楼梯。通常，一个螺旋形的楼梯会使大多数的导航器面临困境。那些结构是真正的 3D 的，大部分是环形的，并且可以盘旋很长一段距离。一个螺旋形楼梯可以用楼梯的直角转弯块（quarter-turn piece）组合而成。然后这样的块能被无数次复制，从而产生一个非常长的螺旋形的楼梯。

如果一个楼梯是以这样的方式构造的，那么分级导航器将能很明显地计算出一条上楼梯的路径。导航器将首先计算通过每一个直角转弯块的大范围的路径，然后将迅速寻找从开始点到每个直角转弯块的局部路径。在更为复杂的几何结构上计算一条复杂的路径突然间变得简单了。图 3.5.3 显示了一个例子。

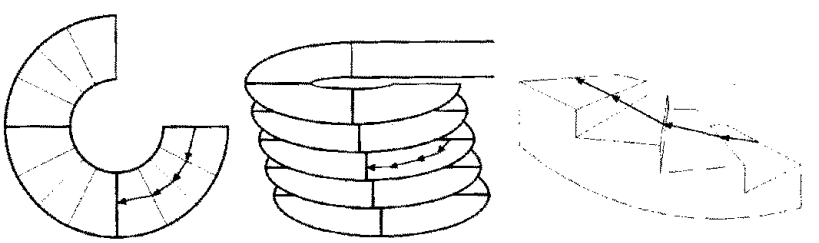


图 3.5.3 在一个螺旋形楼梯上的分级寻径

分级寻径并非局限于有门的房间范围内。你能很容易地将其思想延伸到巨大的由不同区域缝合在一起的地形上。虽然不再有可识别的场所“门”，但从一个区域到另一个区域的全部接缝都变成了门。

设想有一个一望无垠的世界由这些缝合在一起的土地块构造而成。现在想象让一个角色从一个土地块的末端走向另一个。没问题！导航器首先寻找穿过土地块的网络的路线，然后寻找从当前的土地块到下一个的局部路径。稍做工作，你甚至可以设想规划一条从城堡的王宫到一座完全不同的城堡下面的第 9 层地牢的路线，即使城堡可能相隔 10 英里！

8. 计算局部路径时避免停顿

因为每当角色进入一个新房间时必须计算一条路径，所以当新路径被构造时，角色在每个门口不表现出停顿是很重要的。为了避免停顿，路径请求必须被预料到并在角色实际需要之前进行计算。这个简单的技巧使得角色在贯穿路径时保持了流畅的运动。

3.5.2 算法优化

1. 使用启发式代价

有时为启发式代价设计一个算法与其说是科学倒不如说更像巫术。启发式代价背后的思想是估计从一个特殊的节点到目标节点的实际代价。这里有一个有趣的事实：如果始终知道到目标的实际代价，A*将循着一条路径直达目标而不会浪费任何搜索时间去走错误的路径。但是如果启发式碰巧过高估计了实际代价，该启发式就变为“不可采纳的（inadmissible）”，而且算法不可能发现最佳路径（还可能找到一条很糟糕的路径）。

保证永不过高估计代价的方法是通过计算节点和目标之间的几何距离。第一次进行 A*编码时，在进行优化之前最好这样做。因为代价永远不会高于这个距离，最佳路径总能被发现。

2. 高估启发式代价

第二个有趣的事实是：使用一个略微高估的启发式代价常能带来对合理的路径的更快搜索。可是代价应该被高估多少呢？为了回答这个问题，你需要了解当剩余路径的代价被人为地增加时会发生什么。

如果总代价的启发式部分（总费用＝节点费用＋启发式的费用）比实际大，关于 Open 表中的哪个节点该被选取的推理过程就失真了。由于 A*总是选择总代价最小的节点，这种失真促使更接近于目标的节点被选取。

当你观察一个试图在一堵墙周围找到路径的 A*搜索时，能够看见由探索过的节点（Closed 表上的节点）逐步显示出来的轮廓。这个轮廓是了解应用启发式估计效果最简单的方式。

当启发式等于零时，搜索绕着起始点以环形向外扩展。当启发式使用到目标点的欧氏距离时，搜索的轮廓看起来就像一个椭圆形，并以开始点和目标点为焦点。当启发式被高估时，轮廓改变成菱形或者六边形，并且起始点和目标点为其最长轴的端点。图 3.5.4、3.5.5 和 3.5.6 显示了试图克服一个大障碍时使用不同启发式代价进行搜索的轮廓扩展情况。

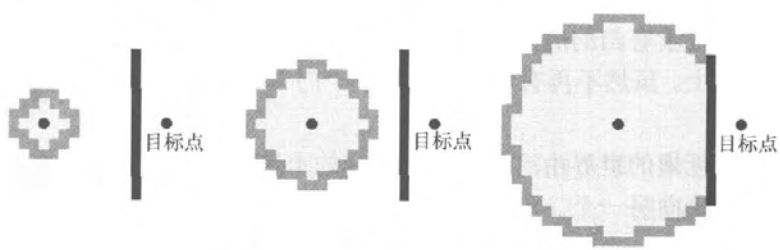


图 3.5.4 采用的启发式代价为零

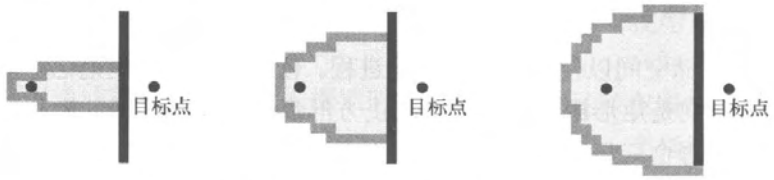


图 3.5.5 启发式使用的是到目标点的欧氏距离

图 3.5.6 采用高估的启发式代价

这一切意味着什么呢？意味着通过高估启发式代价，能使搜索向离目标最近的节点努力推进。这给克服开始点和目标点之间的大障碍的搜索带来了压力。如果实际的解决办法要求在到达目标之前返回，一个高估的启发式代价则会使搜索速度慢下来。但是，如果多数时间都有一条绕过大的堵塞障碍物的路径，那么使用高估的启发式代价将更快。图 3.5.7 说明了这一点：非高估启发式搜索探索的节点要比高估启发式代价的多 3 倍。

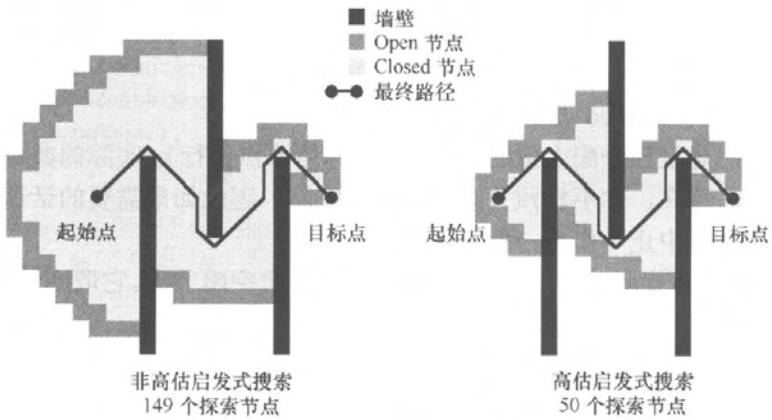


图 3.5.7 一个非高估启发式搜索与一个高估启发式搜索的比较

最后，得到适量的高估值还需要实验。令人遗憾的是，如果搜索空间不是在一个栅格上，精确观察搜索的轮廓或许不太可能，需要测量一些诸如 Closed 表的大小和 Open 表的最大长度的指标。

Closed 表的最终大小表明有多少节点被探索；Open 表的最大长度是表示探索每个节点要用多长时间的一个好指标（当 Open 表变大时，其操作需要相对长的时间）。当调整启发式代价时，你可以试一下典型的搜索并观察 Open 表和 Closed 表的大小以便确定好的启发式值。通过在实际游戏上进行测试，你将能把启发式代价调整到一个合理的水平。

3. 从搜索空间分离寻径数据

A*需要大量的存储空间以便存储每个搜索进程。依照惯例，这些记忆存储在每个可搜索节点内。如果搜索空间是矩形栅格，每个栅格正方形都包含寻径节点的数据。如果搜索空间为多边形网格，那么每个三角形包含寻径节点数据。由于单个的搜索几乎从未覆盖搜索空间内的每个节点，没理由将这些惊人的存储空间都专门用于寻径。例如，一个1 000×1 000的平铺世界有大约100万寻径节点只是呆在那里，大部分时间未被使用。

解决办法是从搜索空间中分离出寻径节点数据。这个办法减少了巨大的存储总开销并且也能使搜索加速。有趣的是，通过从搜索空间分离节点数据，可以考虑同时搜索，因为现在多个节点数据可以指向同一个真正的节点。然而通常情况下同时搜索并不是一个好主意——虽然在某些情形下它可能是有效的。

4. 预分配最小存储量

从搜索空间分离节点数据要求每个搜索使用一些额外的存储块。我们完全能快速地分配节点数据，但A*可能每次搜索数百个节点，因此这不是一个合理的解决办法。改进的方式是为每个搜索预分配足够大的能被循环使用的存储块。

A*储存如此多的是什么？全部是跟踪搜索进程的数据。对于探索过的每个节点，算法需要保存以下的信息：

（1）一个指向父节点的指针；

（2）到达该节点的代价；

（3）总代价（代价＋启发式估计）；

（4）该节点是否在Open表中；

（5）该节点是否在Closed表中。

该方法的主要思想是预分配大量的这种节点（称为节点银行）。实际的数目依据最大搜索的大小会有所不同。现在，你不想预分配太多存储空间，因此如果需要的话这个序列应该能增长，或者迫使该搜索中止（作为一种选择）。

当A*第一次探索新节点时，需要从节点银行中请一个空闲节点。它取回一个空闲节点时，需要填入上述信息，以便个性化这个新节点。

5. 在主节点表内存储探索过的节点

一旦某个节点从节点银行被个性化，就需要为快速检索将其存储在某处。适宜于这种行为的最佳数据结构是哈希表。哈希表允许以常量的时间存储和查找数据。所以我们将所有的已探索节点存储在这个主节点表中。这张哈希表允许我们即时查明一个特殊的节点是否在Closed表或Open表中。记住，由于节点数据存储空间已经被分配，哈希表仅仅包含指向这些节点的指针。

在此你可能会问“Closed表在哪里？”答案是，它存在于主节点表内部。所有的已探索节点都被储存在主节点表内，并且Closed表碰巧也在相同的地方。这并不是问题，因为每个节点都被清楚标明了是否在Open表或Closed表中。那么Open表在哪里呢？Open表被另外保留，但是主节点表中也包含指向所有与Open表上节点相同的节点的指针。为什么要重复

呢？因为有时使用主节点表能更快发现你所需要的节点，而有时使用 Open 表更快。完全是因为速度。

在 A*算法中当任何特定的节点被探索时，可能该节点已经在同一搜索过程中被探索过了。为简单起见，你需要一个函数，返回指向特定节点数据的一个指针，无论它是否在之前被搜索过。

该函数首先检查主节点表看看节点是否已经被探索过了。如果是，函数就简单地返回指向现有节点的指针。如果节点不在主节点表中，就从节点银行中取出一个空闲节点，为该节点赋以表示想要的节点的初值，并且返回其指针。实际上，函数完全隐藏了从节点银行分配新节点和取得已经存在节点的细节。

```
Node* GetNode( MasterNodeList nodelist, NodeLocation node_location )
{
  //GetNodeFromMasterNodeList accesses the hash table of nodes
  Node* node = GetNodeFromMasterNodeList( nodelist, node_location );
  if( node ) {
    return( node );
  }
  else
  {   //Not in the Master Node List - get new one from the Node Bank
    Node* newNode = GetFreeNodeFromNodeBank();
    newNode->location = node_location;
    newNode->onOpen = false;
    newNode->onClosed = false;

    //StoreNodeInMasterNodeList places the node into the hash table
    StoreNodeInMasterNodeList( nodelist, newNode );
    return( newNode );
  }
}
```

6. 优化 Open 表

A*的优点来自其指导搜索朝着最有希望的方向进行的能力。达到此目标的方法是把所有下一步要搜索的节点放到 Open 表中，然后按对搜索来说最有希望的节点到最没有希望的节点进行排序。问题是 Open 表倾向于变大，并且每当它经历 A*循环时，最有希望的节点必须从表中提取出来。提取的节点是具有最小总代价（到达该节点的代价＋到目标剩余代价的启发式估计）的节点。事实表明，存储 Open 表的最好方式是把它作为优先队列进行排序。

优先队列可以作为一个二分堆（binary heap）实现。一个二分堆是一棵排序树，它的特性是双亲节点总是比它的孩子节点具有更低的值。然而在兄弟间是无序的，因此堆不是完全有序的树。因为这个有意思的性质，插入和提取元素（删除最小元素）的时间复杂度仅为 $O(\log n)$。幸运的是，那几乎就是 A*需要就 Open 表做的全部操作。

7. 实现一个优先队列

从头开始实现一个优先队列超出了本文的讨论范围，但是有一种使用 STL 来实现的简单

方法。无论你是否正在使用 STL，一定要去查阅一篇关于优先队列的重要论文[Nelson96]。该文极为清晰地描述了优先队列，使你能够构造一个不用 STL 帮助的优先队列。另外，请参考任何标准的算法和数据结构书以获得更多的信息。

为了正确使用优先队列，请使用以下 4 个 A*需要在 Open 表上执行的操作：

（1）提取有最小代价的节点（并对表重新排序）：$O(log\ n)$。

（2）在 Open 表中插入一个新节点：$O(log\ n)$。

（3）更新一个已经存在于 Open 表中的节点的总代价（并对表重新排序）：$O(n+log\ n)$。

（4）确定 Open 表是否为空表：O(1)。

事实上 STL 用所谓的容器适配器（container adapter）来实现优先队列。但是，能在其上执行的操作非常有限。实际上，它没有执行操作#3（更新一个节点的总代价并对表重新排序）的接口。但是，我们能在一个 STL 矢量容器（vector container）上使用 STL 堆操作来构造自己的优先队列！

程序清单 3.5.1、3.5.2、3.5.3 和 3.5.4 包含了连同节点对象、堆对象以及 STL 比较对象一起的 4 个 Open 表操作（全部使用 STL 以 C++实现）。此外，在本书所附 CD 中，你将发现 Greg Snook 的寻径文章“3.6 简化的 3D 运动和使用导航网格进行寻径”，使用几乎相同的实现 STL 优先队列的代码。

8. 使用优化的主节点表和 Open 表的 A*

关于使用上述思想并没有太多的技巧，但是以防万一，程序清单 3.5.5 给出了使用主节点表和 Open 表实现 A*算法的基本思想，其中也包括一些其他小技巧，例如不搜索刚刚搜索过的节点。

3.5.3 结论

从根本上说寻径是一个艰难的计算问题，因此最好的策略始终是设法将问题简化。在做出任何优化算法的努力之前，要确保世界以最简单合理的方式表示。一旦表示方式被决定了，一并包含某种分级计划也是非常重要的。该计划通常包含了一些预处理，它要求关卡设计师和艺术人员致力于搜索空间的表示。虽然它增加了研发资金的开销，但是没有更好的方法来加速寻径。

一旦搜索空间最终定了下来，在做任何优化尝试之前使导航器在该空间正确工作是非常重要的。A*不是一种很简单的算法，并且如果包括了许多优化，调试它是极其困难的。当你准备进行优化时，由从搜索空间中分离出节点数据开始；下一步是为 Open 表实现一个优先队列，并为主节点表/Closed 表实现一个哈希表；最后，当一切效验如神，并且游戏稳定时，你就可以运行启发式代价了。为了得到最好的结果，在研发期间你要对启发式进行若干次调整，以使游戏世界定义得更好。

在所有这些技术已经被实现之后，下一步是“欺骗”（cheat）。一些给出即时寻径感觉的技术能在本书文章“3.4 A*审美最优”中找到。其中包含了使玩家认为一条道路已经被发现而实际上你正在延迟的技巧。如果玩家感到游戏的响应非常灵敏，寻径器看起来透明而不显眼，就达到了加速的核心目的。

程序清单 3.5.1 节点对象

```
class Node
{
public:
   NodeLocation location;  // location of node (some location representation)
   Node* parent;           // parent node (zero pointer represents starting node)
   float cost;             // cost to get to this node
   float total;            // total cost (cost + heuristic estimate)
   bool onOpen;            // on Open list
   bool onClosed;          // on Closed list
};
```

程序清单 3.5.2 优先队列对象

```
class PriorityQueue
{
public:
   //Heap implementation using an STL vector
   //Note: the vector is an STL container, but the
   //operations done on the container cause it to
   //be a priority queue organized as a heap
   std::vector<Node*> heap;
};
```

程序清单 3.5.3 STL 比较函数

```
class NodeTotalGreater
{
public:
   //This is required for STL to sort the priority queue
   //(its entered as an argument in the STL heap functions)
   bool operator()( Node * first, Node * second ) const {
      return( first->total > second->total );
   }
};
```

程序清单 3.5.4 4 种 Open 表操作

```
Node* PopPriorityQueue( PriorityQueue& pqueue )
{  //Total time = O(log n)

   //Get the node at the front - it has the lowest total cost
   Node * node = pqueue.heap.front();

   //pop_heap will move the node at the front to the position N
   //and then sort the heap to make positions 1 through N-1 correct
   //(STL makes no assumptions about your data and doesn't want
   //to change the size of the container.

   std::pop_heap( pqueue.heap.begin(), pqueue.heap.end(),
      NodeTotalGreater() );
```

```
  //pop_back() will actually remove the last element from the heap
  //(now the heap is sorted for positions 1 through N)
  pqueue.heap.pop_back();

  return( node );
}

void PushPriorityQueue( PriorityQueue& pqueue, Node* node )
{ //Total time = O(log n)

  //Pushes the node onto the back of the vector (the heap is
  //now unsorted)
  pqueue.heap.push_back( node );

  //Sorts the new element into the heap
  std::push_heap( pqueue.heap.begin(), pqueue.heap.end(),
    NodeTotalGreater() );
}

void UpdateNodeOnPriorityQueue( PriorityQueue& pqueue, Node* node )
{ //Total time = O(n+log n)

  //Loop through the heap and find the node to be updated
  std::vector<Node*>::iterator i;
  for( i=pqueue.heap.begin(); i!=pqueue.heap.end(); i++ )
  {
    if( (*i)->location == node->location )
    { //Found node - resort from this position in the heap
      //(since its total value was changed before this function
      //was called)
      std::push_heap( pqueue.heap.begin(), i+1,
        NodeTotalGreater() );
      return;
    }
  }
}

bool IsPriorityQueueEmpty( PriorityQueue& pqueue )
{
  //empty() is an STL function that determines if
  //the STL vector has no elements
  return( pqueue.heap.empty() );
}
```

程序清单 3.5.5 用一个主节点表和一个优先队列 Open 表实现的 A*

```
MasterNodeList g_nodelist;

bool FindPath( GameObject* gameobject, WorldLocation goal )
```

```
{
  //Get a path in progress if it exists for this game object with
  //this goal
  //A path may have been started and not finished from last game tick
  //If no path in progress, it returns an empty path structure
  Path* path = GetPathInProgress( gameobject, goal );

  if( !path->initialized )
  {  //The InitializePath function fills out the path structure for
     //this path request
     //It initializes a clean MasterNodeList and a clean Open list
     InitializePath( path, gameobject, goal );

     //Create the very first node and put it on the Open list
     Node* startnode = GetNode( g_nodelist, GetNodeLocation(
       gameobject->pos ) );
     startnode->onOpen = true;      //This node goes on Open list
     startnode->onClosed = false;   //This node not on Closed list
     startnode->parent = 0;         //This node has no parent
     startnode->cost = 0;           //This node has no cost to get to
     startnode->total = GetNodeHeuristic( startnode->location,
       path.goal );
     PushPriorityQueue( path.open, startnode );
  }

  while( !IsPriorityQueueEmpty( path->open ) )
  {
     //Get the best candidate node to search next
     Node* bestnode = PopPriorityQueue( path.open );

     if( AtGoal( bestnode, goal ) )
     {  //Found the goal node - construct a path and exit
        //The complete path will be stored inside the game object
        ConstructPathToGoal( gameobject, path );
        return( true );   //return with success
     }

     while( /*loop through all connecting nodes of bestnode*/ )
     {
        Node newnode;
        newnode.location = /*whatever the new location is*/;

        //This avoids searching the node we just came from
        if( bestnode->parent == 0 ||
            bestnode->parent->location != newnode.location )
        {
           newnode.parent = bestnode;
           newnode.cost = bestnode->cost + CostFromNodeToNode(
             &newnode, bestnode );
           newnode.total = newnode.cost;
```

```
            //Get the preallocated node for this location
            //Both newnode and actualnode represent the same node
            //location, but the search at this point may not want
            //to clobber over the data from a more promising route -
            //thus the duplicate nodes for now

            Node* actualnode = GetNode( g_nodelist,
              newnode.location );

            //Note: the following test takes O(1) time (no searching
            //through lists)
            if( !( actualnode->onOpen && newnode.total >
              actualnode->total ) &&
               !( actualnode->onClosed && newnode.total >
              actualnode->total ) )
            {  //This node is very promising
              //Take it off the Open and Closed lists (in theory)
              //and push on Open
              actualnode->onClosed = false;   //effectively removing it from Closed
              actualnode->parent = newnode.parent;
              actualnode->cost = newnode.cost;
              actualnode->total = newnode.total;

              if( actualnode->onOpen )
              {  //Since this node is already on the Open list,
                //update it's position
                UpdateNodeOnPriorityQueue( path.open, actualnode );
              }
              else
              {  //Put the node on the Open list
                PushPriorityQueue( path.open, actualnode );
                actualnode->onOpen = true;
              }
            }
          }
        }

        //Now that we've explored bestnode, put it on the Closed list
        bestnode->onClosed = true;

        //Use some method to determine if we've taken too much time
        //this tick and should abort the search until next tick
        if( ShouldAbortSearch() ) {
          return( false );
        }
      }

      //If we got here, all nodes have been searched without finding
      //the goal
```

```
    return( false );
}
```

3.5.4 参考文献

1. A*算法

[Heyes-Jones98] Heyes-Jones, Justin, "A* Algorithm Tutorial," www.gamedev.net/reference/programming/ai/article690.asp, 1998.

[Patel99] Patel, Amit J., "Amit's Thoughts on Pathfinding," http://theory.stanford.edu/~amitp/GameProgramming/, November 27, 1999.

[Stout96] Stout, W. Bryan, "Smart Moves: Intelligent Path-Finding," *Game Developer*, -October/November 1996, pp. 28–35, www.gamasutra.com/features/19990212/sm_01.htm.

2. 数据结构

[Lewis91] Lewis, Harry R., *Data Structures and Their Algorithms*, HarperCollins Publishers Inc., 1991.

[Nelson96] Nelson, Mark, "Priority Queues and the STL," *Dr. Dobb's Journal*, www.dogma.net/markn/articles/pq_stl/priority.htm, January 1996.

3.6 简化的3D运动和使用导航网格进行寻径

Greg Snook

对于游戏程序设计师来说，使一个对象从点A智能地运动到点B一直是一个挑战。对一个3D空间中的对象这样做是一个更大的挑战。在当今复杂的3D环境世界中，这成了一项必须完成的首要任务。本文提出了一个有助于克服这些障碍并以最小的工作量使所有对象安全地到达点B的相当简单的方法：欺骗（Cheat）。

是的，欺骗。实时游戏几乎没有时间来计算真正的3D对象到场景的交互以及寻径，而且对于大多数的应用来说，做这些工作的代码复杂性常常是不必要的。我们在此提出了一种更容易的方法。寻找一种简单的、可扩展的方法碰碰运气并且以一种对玩家来说可信的方式移动鼠标。我们承认如下的事实：最容易的方法几乎总是包含了欺骗。

3.6.1 简述

我们所需要的是一种将3D空间简化为更惯用的2D空间的方法。2D空间中的对象以一种高度可预测的方式运动，并且能被玩家直观地控制。此外，我们掌握了许多为对象的运动创建智能路径的2D搜索算法。我们要创建的是一种允许对象在一个伪3D环境中运行而为玩家提供的是一个全3D外观的方法。为了达到这个目的，我们使用一个三角形的网格来将3D空间表现为一个不规则的2D游戏场地。

该思想源于这样的事实：对于大多数的游戏环境而言，可以相当简单地预知在哪些地方对象能运动以及在哪些地方不能运动。从这一点说，可以创建一个几何图形的简单集合来定义一个“可以走动的”表面区域。可以把这个区域想象成一个典型3D环境中的房间。由于角色是有人类特点的，而且游戏所属的行星是有地心引力的，你可以假定游戏对象将在这个房间的地板上度过他们的大部分时间。你也可以假定他们不能穿行柱子、桌子、汽水机，以及其他占用地板空间的对象。我们可以用覆盖了空旷、可以走动的表面区域的简单、粗略的几何图形定义地板上剩余的部分。将这个几何图形看作一种多边形地毯，我们称之为导航网格（navigation mesh）。它代表了环境内部对象能围绕其运动的区域。图3.6.1显示了随书CD中提供的navimesh程序中用到的3D环境。线框多边形展示了导航网格，它定义了对象可以运动的区域。

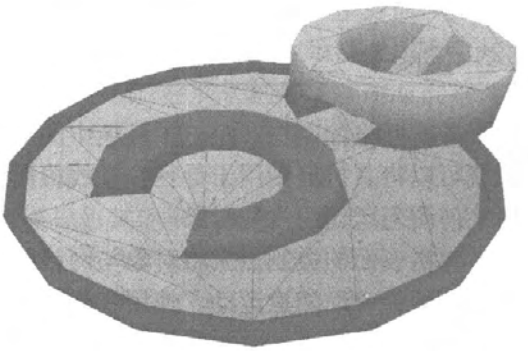


图 3.6.1 一个具有用线框画出的相关导航网格的 3D 环境

从某种意义上说，这个导航网格对象能被像基于平铺的 2D 游戏使用的栅格系统一样处理。每个网格多边形被认为是一个栅格单元，只是它与邻近的单元以三边而不是四边相连。稍做一些改进，我们甚至可以将这样的网格用于传统的基于栅格的算法，如可视线（line-of-sight）探测和寻径。更多的好处是这种 2D 栅格的替代物可以有不规则形状和大小的单元，像盘旋楼梯或是盘山公路，甚至像桥或梯形通道之类的东西相重叠——当准备访问对于上述我们都熟知并喜爱的经过时间检验的 2D 算法时。

利用一个导航网格也可以减少一个对象和它的静态环境之间必需的碰撞检测的数量。由于导航网格已经代表了环境中空旷表面区域的一个近似，我们的对象要求仅与网格的边而不是与真正的场景几何图形相碰撞。通过将一个控制点从对象投影到网格上，我们可以很容易地跟踪对象的运动及与 2D 直线相交（而不是与全 3D 多边形相交）的碰撞检测。在要求更多碰撞细节的情形，网格单元仍而可与真实的场景几何图形集相联系以适应于改进的测试。这样与一个单元的边相碰撞的对象将被传递到解决与房间几何图形相关的碰撞问题的程序。以这样的方式将过程数据与这样的单元相联系，起到了为运动的物体进行快速而有可能不太好的邻近度检测的作用，每当一个对象进入一个特定的单元或与某个特殊边界相碰撞时，这个思想能扩展到触发陷阱、门，以及开关等。

3.6.2 构造

导航网格几何图形需要遵从一些简单的规则以便正确工作。首先，它需要完全由三角形组成以保证每一个单元包含在同一个平面内。其次，全部网格必须是邻接的，所有邻接的三角形共享两个顶点和一条边。最后，在同一平面内，没有两个三角形交叠。也就是说，一个三角形的单元内任何给定点对于该单元来说是惟一的。这将极大地支持我们的算法并为玩家提供可信的运动。

导航网格对玩家是不可见的。我们仅仅将其用在场景后面以限制角色的运动并决定路径。因此，它需要仅由最小数量的必要多边形来构成，以代表对象能够在其中运动的区域。细节丰富的导航网格可以产生非常精确的结果，但是对大多数实时游戏来说它们的总开销将成为一个限制因素。网格应该仅包含必需的使玩家更易进行可信运动的单元，而用不着表现出场地上的每一个细枝末节。

3.6.3 滚动骰子并且移动鼠标

让我们首先研究一下使用导航网格控制在一个 3D 环境中的对象运动。一旦能使对象与网格进行恰当的相互作用，就可以将其扩展到用于寻径和视线检测中。但是重中之重的事情是：我们需要得到围绕网格几何图形运动的一些对象。

我们使用一个固定在导航网格表面的控制点来将对象与网格连接起来。这个控制点永远也不会离开网格，但是它能够在网格表面随意运动。使用我们的多边形地毯例子，想象一个人站在一个房间中。控制点能被设想成位于这个人正下方的一个弹子，在他或她的两脚间静止不动。无论这个人在房间里运动到什么地方，弹子都跟着一起滚动，总是保持在地毯上位于这个人正下方的位置。

所有预期的对象运动都被转换为相对于这个控制点的运动，依次在导航网格表面得到解决。然后对象相对于新的控制点位置运动。在我们的例子中，把那个弹子踢向墙面并让它弹回，然后令人运动到新的弹子位置。

基本的过程如下所示，假定每一个对象保持了一个网格上的控制点，而且我们知道哪一个网格单元当前包含了控制点。

（1）将对象期望的运动矢量映射到它的当前单元所在的平面上。这样就沿着单元的平面把运动转化为了一个 2D 矢量。我们称这个新的矢量为一个运动路径，并把它表示为一条线段。线段的端点是控制点的起始位置和期望的最终位置，两者都转换到相应单元平面的 2D 空间中。

（2）测试相对于单元的 2D 三角形边的运动路径。由三角形的特性我们知道，离开单元的路径必定与三角形的确定的一条边相交。于是，我们测试运动路径的 2D 线段与代表单元三角形的三条线段的任何可能的交点。三个可能结果中只有一个是该测试的结果：

a. 我们的路径与一个非共享的边（即一个不与邻近单元相连的边）相交。这意味着我们遇到了某个固定的物体。解决运动路径矢量与单元的边之间的碰撞，调整运动路径以适应任何方向上的改变，并且重复第 2 步。

b. 我们的路径与一个共享的边相交。移动到邻接的单元并重复从第一步开始的所有过程，将当前矢量投影到新的单元平面并测试它的边。

c. 剩下的惟一可能是我们的运动路径并没有离开当前单元。测试过程结束并找到了对象的新静止位置所在的单元。我们将结果即 2D 运动路径终点转换回 3D 空间，以寻找控制点的真正的 3D 位置并使对象相应运动。

显然，对于复杂的导航网格，这将成为一个非常繁琐的过程。对于遇到的每一个单元，我们都需要将一个任意的 3D 矢量投影到一个 3D 平面上。这里可以将结果矢量连同单元的边一同转换到 2D 空间，在此可以执行我们的直线相交测试。一旦完成了关于这个网格的运动，就需要取消转换和投影以产生 3D 空间中的新的控制点。

实时进行此项工作的工作量非常大，尤其是如果你有很多对象要测试，或者你的对象计划在给定框架内越过许多单元的时候。然而对于简单的环境而言，它是合理的并由此在导航网格内产生很好的适应性。对于复杂的环境，我们还可以用更仔细的计划（请看“欺骗”）和附加的导航网格几何图形规则显著地加速这个过程。该规则就是：为便于快速投影，所有的

单元的法线必须沿着一个预先确定的基轴朝向同一个方向。

再想象一下地板上有导航网格地毯的房间。地板上所有单元的法线都向上，于是它们满足我们的网格的新需求。也就是说，在我们的环境中所有的单元法线有一个正的 y 值。注意我们不要求新的导航网格是平坦的，只是不再允许单元的法线是 90 度或远离我们所选的轴。

有了这个新规则，投影就变得极为简单了。我们仅需沿着所选定的轴发出尺寸。在铺了地毯的房间的例子中，将点投影到地板上现在与将它们的 y 值投到窗体一样简单了。另外，当我们已经沿着导航网格单元完成了运动路径的处理时，得到了一个新的 2D（x，z）位置以及包含它的单元。利用该单元的平面方程，可以用（x，z）解出 y 并很容易地转换回 3D 空间。新的运动跟踪过程可简化如下：

（1）创建一个由控制点和期望位置组成的运动路径，通过去除他们的公共轴线方向的值简化为 2D 点。

（2）如前所述，相对于单元三角形的边测试 2D 运动路径，直到找到一个包含了该运动路径的目标终点的单元。

（3）利用新的控制点位置（x，z）和它所属单元的平面方程，解出 y 并将控制点再转换回 3D 空间。

Navimesh 例程包括了一些阐明这一过程的简单的类。在源代码中，一个称为 NavigationCell 的类被用于表示一个单独的三角形单元，并且 NavigationMesh 表示那些单元的一个集合对象。让我们首先来查看一下 NavigationCell，因为它做了大部分工作。

NavigationCell 用下列成员定义了网格的一个单独单元：

```
Plane    m_CellPlane;      // A plane containing the cell triangle
vector3 m_Vertex[3];       // the three vertices of this cell
Line2D m_Side[3]; // a 2D line representing each cell wall
NavigationCell* m_Link[3];     // pointers to cells that attach to
                           // this cell on each of its three
                           // sides.  A NULL link denotes a solid edge.
```

Vector3，Plane 和 Line2D 是非常简单的辅助类（它们的源代码也被给出）。区别之处在于 Line2D 事实上被看作是一条经过两个点的光线。它有一个暗含的方向，即从端点 A 指向端点 B。对于 2D 线段也留下了一个垂直的“法线”。这个法线用于将点划分为在直线左侧或是右侧。这里“左”和“右”的概念是这样规定的：假设你正站在线段的端点 A 向着端点 B 的方向看。正如你在源代码中看到的，将点关于直线进行划分的能力在我们的运动处理中会起重要的作用。

NavigationCell 的主要用途是在我们的过程中执行这样的一步：决定一条路径如何与单元的边交互作用。NavigationCell 包含一个成员函数，将一个 2D 线段分为三条单元边并且返回一个结果。ClassifyPathToCell( )这个函数是应用导航网格的基本构造块。该函数的返回值可以是下述枚举类型中的一个：

```
enum PATH_RESULT
{
NO_RELATIONSHIP = 0,  // the path does not cross this cell
ENDING_CELL,          // the path ends in this cell
EXITING_CELL          // the path exits this cell
```

```
};
```

在最后结果是 EXITING_CELL 的情形，为调用函数提供了穿越的单元的边以及与边的2D交点。这使我们可以将任何2D路径与单元进行比较并确定将出现什么类型的交点。当与一条固体的边发生交点，我们可以用交点来计算新的方向并再测试。程序清单3.6.1详细说明了ClassifyPathToCell( )函数。

NavigationMesh必须用这个函数按上述定义的步骤处理我们的运动。NavigationMesh的成员函数ResolveMotionOnMesh( )管理着整个过程，用ClassifyPathToCell( )测试每一个遇到的单元。它包括一个3D控制点，一个指向其当前占有单元的指针，以及运动发生后的预期的控制点的位置。它将控制点的真正最终位置和新的控制点所在的单元返回给调用函数。程序清单3.6.2详细阐明了ResolveMotionOnMesh( )函数的使用。

3.6.4 到此仅完成一半

现在我们已经了解了如何应用导航网格来控制对象的运动，下面再看看这个网格的其他一些应用。最常见的应用就是寻径。记住我们的网格是由相连的单元组成的，它们共享公共的边，正如一个栅格或六角形格图。那么任何适用于栅格或六角形格图的传统的寻径算法都能相应地转换到我们的网格上来。

事实上，对于搜索算法使用多边形网格算不上什么新技术。由于寻径算法是被设计用来在相连的节点数据的数据库上工作的，它们在穿越相连顶点集合时很有作用。作为游戏程序设计师，我们对这些适用于栅格和六角形格图的方法已经变得熟视无睹了，它仅仅是在其中能使用这些方法的环境的一个小的子集。

使用我们的导航网格，确实可以为多边形上的寻径方法增加一个小小的创新：不使用网格顶点，而是使用每一个单元边的中点来代替。为什么呢？有两个原因，这两个原因都可以依赖于游戏环境得到论证。第一个原因是可以使用导航网格来限制对象在环境中的运动。如果产生了一条沿着单元顶点的路径，对象就始终在单元的边上运动。在一个导航网格中这是代价最高的运动，因为沿着一个单元的边运动恰恰意味着不断地与单元的边界发生碰撞，由此引起了大量额外的、不必要的交点测试。

第二个原因纯粹是美感。如果我们假设网格被设计为使用最小数目的多边形，它就会支持这样的理由：在我们的空地中并没有许多顶点或多边形的边。可以看一下图3.6.2a和3.6.2b，它们显示了一个走廊的俯视图，以及一个合理数目的多边形来定义其中的空地。如果在单元边界上产生一条路径（图3.6.2a），我们将花费大部分时间倚着走廊的墙拖动自己。使用单元边的中点（图3.6.2b），则可以沿着走廊产生一个更悦目的路径。

前面提到，两个理由都是可以论证的。你可以简单地增大导航网格的复杂性并增加代码以避免沿着固体的网格边界拖动路径，但是我发现经过边的中点会更容易些（而且概念上更直观）。在应用中，也证明了对于关卡设计者来说，在为寻径而不必关心多余的顶点位置和单元边界的情况下，创建网格更容易。从本质上讲，我们所做的一切就是避开网格顶点来创建一条更可信的路径。

那么如何来构造一条路径呢？与任何寻径情形一样，需要选择对你的游戏环境来说最佳的路径。最佳优先（Best-first）搜索，Dijkstra算法，以及由来已久的A*算法都能被应用于

导航网格的单元。在本书的其他地方对各种搜索方法有一些极好的解释，所以我在这儿就不准备深入阐述了。请查阅本书中 Steve Rabin 和 Bryan Stout 所写的详细讨论 A*算法的文章（3.3，3.4，3.5）。你也可以查阅文本最后推荐的关于寻径的参考文献。

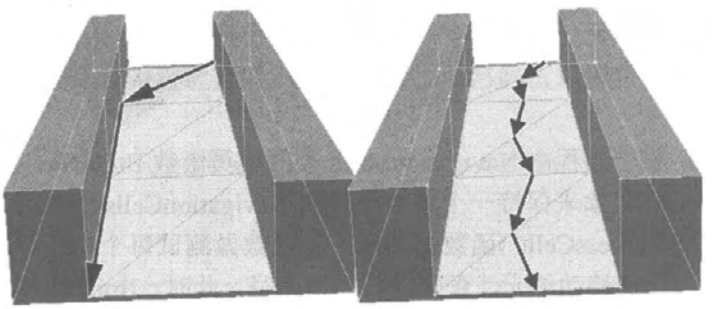


图 3.6.2 一个简单的走廊和导航网格的两个俯视图展示了路径 a:沿着单元的边产生 b:经过单元边的中点

为了示范，navimesh 示例代码表明了如何在导航网格上应用 A*算法。虽然它可能使用的是最复杂的算法，但是 A*算法能得到高精确度的结果，并且常常比其他方法在内存使用和搜索时间方面效率更高。然而，它的效率取决于一个好的启发式的应用。启发式有助于使搜索算法朝着目标行进，防止展开所有的网格（除非必要）。

使用最好的启发式纯属一个特定于游戏的问题。只是要知道对象在游戏地形中运动得多么好，而且需要相应地调整启发式。你甚至可能需要为每一个对象类型分别定制启发式，考虑它爬陡坡的能力，高速转弯的能力等等。然而在多数情形，这个启发式只不过是从一个给定单元到目标的大约距离。为了达到示范的目的，我们所采用的也是这样一个启发式。

为了运行 A*算法，我们保持了一个需要被处理的单元的列表。在 navimesh 示例代码中，这些“空旷（Open）”的单元排列在一个有序表中，称为一个 NavigationHeap。单元按照到达目标需要花费的代价从最佳到最差依次排列。因此，每次从堆里拉出一个单元时，我们知道正在处理单元的当前“最佳推测”，它将提供到目标的最佳路径。

为了开始寻径，需要将第一个单元（我们的目标）放在堆上，然后弹出并处理堆上的每一个单元，直到到达起始位置或者堆为空为止。如果在到达目标之前堆就空了，我们就知道在两个位置之间没有可用的路径。

为了处理一个单元，要检查它的每一个相邻单元。通过将与当前单元相关的代价加到穿越单元到每一个相邻单元所要求的距离上，决定移动到每一个单元的距离。这种对于每一个相邻单元的“到达代价（Arrival Cast）”接着被增加到相邻单元自己的启发式值上，来为每一个相邻单元取得一个优先权分值。

我们现在来考查每一个相邻单元的分值或代价来做下面两者中的一个。如果相邻单元当前不在开放的堆内，我们必须根据它的分值来为其排序。实际上将推迟处理它。如果相邻单元已经在堆内，我们需要看一下新的分值是否比其当前排序所依据的分值更好。如果新的分值是一个改进，为了较早处理需要将单元在堆内向上移动到其新的优先位置。如果新的分值不是一个改进，就丢弃它，因为一个更加理想的路径已经穿越了这个单元。无论在哪种情形，

每当一个单元被增加到堆上或在堆内改变位置时，都要记录下设置了当前“到达代价”的单元的特性。

这样做使单元能留意下一个沿着生成了的路径到目标最近的单元。你将注意到 navimesh 例子在反向搜索方向时运行了 A*算法，从目标单元开始搜索一条向后的到当前位置的路径。当搜索完成时，每一个单元包含了一个沿着生成的路径到下一个最近单元的链接。我们可以通过这些链接从当前的位置以正确次序跳跃到目标，并且为游戏对象建立一个最终的途经点（waypoint）列表。

在示例代码中，整个过程由 NavigationMesh 类的成员函数 BuildNavigationPath( )进行。它使用 NavigationHeap 对象来保持一个将被处理的 NavigationCells 的列表。当每一个单元被从堆内拉出时，它的 ProcessCell( )函数被调用，该函数做测试每个相邻单元的工作，如前所述。必要时在堆内增加或移动单元，直到找到一条路径。此时，BuildNavigationPath( )对路径上的单元进行遍历，将每个单元的中点增加到最终的 NavigationPath 途经点列表上。整个过程的源代码见程序清单 3.6.3。

3.6.5 它是有效的，但不是那么完美

正如你可以通过 navimesh 示例程序中画的蓝线看到的，通过多边形对象建立一个路径产生了一个锯齿状的结果。你很少会发现导航几何图形是为产生一个直线路径而建立的。网格的特性使得我们的路径从一个单元到另一个单元蜿蜒而行，迂回曲折（见图 3.6.3 a）。任何一个机械使用该路径的对象在玩家看来将非常奇怪。幸运的是，我们将讨论一个能够使路径变得相当平滑的最终应用：视线确定。

回到我们正在解决的如何围绕网格移动对象的问题中来，我们定义了一个函数 ClassifyPathToCell( )，来与到一个单元的一条 2D 直线的运动相比较。函数的结果告诉我们路径是终止于单元内部，与一个固体边相交，或是穿过单元到其相邻单元。我们现在能够再使用该函数来进行一个视线测试，通过向前跳到视线所及的最远的途经点使路径变得平滑。

每当我们到达路径内的一个途经点时，就往前看列表中的下几个途经点。通过创建一个从当前位置到每个这样的途经点的运动直线，我们可以快速地检测该途经点是否“看得见”。为了这样做，要检测相对于每个单元的从当前位置到途经点的采用 ClassifyPathToCell( )函数的路径。如果函数返回与一个固体边的交点，我们知道从当前位置是看不见那个途经点的。反之，如果我们到达那个途经点时没有遇到这样的一个交点，我们知道该点对我们来说是可以看见的。通过搜索链上最远的能看见的途经点，我们可以跳过一些弯弯曲曲的途经点并且使路径变得平滑。图 3.6.3a 展示了通过跳过一些多余的可以看得见的途经点得到新的更平滑的路径。

这个方法能用来进行所有类型的可见性测试。使用 ClassifyPathToCell( )函数，可以测试网格上的任意两点是否能相互看到。它在敌方 AI（enemy AI）中有一些非常有用的应用，因为你可以快速地测试敌方对象是否能够在任何特定瞬间看到玩家的位置。NavigationMesh 类的 LineOfSightTest( )成员函数详细给出了决定点的可见性的过程。

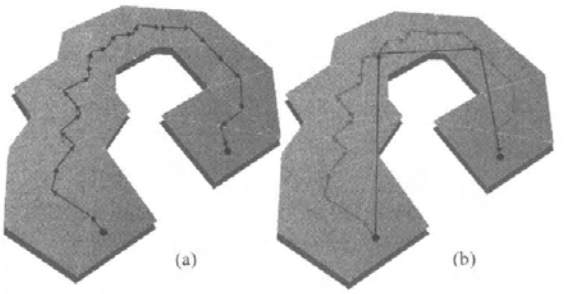


图 3.6.3 a：一个使用 A*算法得到的路径例子　b：采用视线测试平滑后得到的同一路径

3.6.6 结论

我希望本文说明了 3D 空间并不需要进行那么复杂的计算。我的目的是想表明使用类似的 2D 方法（有一点儿欺骗性），可以极大地简化游戏环境而不影响玩家的 3D 体验。最后的结果是它是一个对于在 3D 空间中移动对象非常灵活有用且没有太多 3D 运算的工具。

本方法允许你简化对象/场景碰撞运算，创建复杂的路径，并且测试环境空间中任何两点之间的可见性。然而，我们仍然可以对导航网格进行进一步的改进。他们能被构造成高层次的单元而不是刚体三角形，对于近似于照片效果的有更多细节的路径可以采用像棋盘格似的多分辨率（multiresolution）网格，甚至以动画来表现流动的表面。现在暂且给你留下这些研究思路。请试试你能在此基础上对网格的新应用达到多高的层次。

程序清单 3.6.1　一个 2D 直线与一个导航网格单元相交

```
NavigationCell::PATH_RESULT NavigationCell::ClassifyPathToCell(const
Line2D& MotionPath, NavigationCell** pNextCell, CELL_SIDE& Side,
vector2* pPointOfIntersection)const
{
    int InteriorCount = 0;

    // Check our MotionPath against each of the three cell
    //  walls
    for (int i=0; i<3; ++i)
    {
        // Classify the MotionPath endpoints as being either
        // ON_LINE, or to its LEFT_SIDE or RIGHT_SIDE.
        // Since our triangle vertices are in clockwise order,
        // we know that points  to the right of each line are
        // inside the cell.  Points to the left are outside.
        // We do this test using the ClassifyPoint function of
        // Line2D

        // If the destination endpoint of the MotionPath
        // is Not on the right side of this wall...
        if (m_Side[i].ClassifyPoint(MotionPath.EndPointB()) !=
        Line2D::RIGHT_SIDE)
        {
```

```
            // ..and the starting endpoint of the MotionPath
            // is Not on the left side of this wall...
            if
            (m_Side[i].ClassifyPoint(MotionPath.EndPointA()) !=
            Line2D::LEFT_SIDE)
            {
                // Check to see if we intersect the wall
                // using the Intersection function of
                // Line2D
                Line2D::LINE_CLASSIFICATION IntersectResult =
                MotionPath.Intersection(m_Side[i],
                pPointOfIntersection);

                if (IntersectResult ==
                Line2D::SEGMENTS_INTERSECT ||
                IntersectResult ==
                Line2D::A_BISECTS_B)
                {
                    // record the link to the next
                    // adjacent cell (or NULL if no
                    // attachment exists) and the
                    // enumerated ID of the side we hit.

                    *pNextCell = m_Link[i];
                    Side = (CELL_SIDE)i;
                    return (EXITING_CELL);
                }
            }
        }
        else
        {
            // The destination endpoint of the MotionPath
            // is on the right side.  Increment our
            // InteriorCount so we'll know how many walls we
            // were to the right of.

            InteriorCount++;
        }
    }

    // An InteriorCount of 3 means the destination endpoint of
    // the MotionPath was on the right side of all walls in the
    // cell.  That means it is located within this triangle,
    // and this is our ending cell.

    if (InteriorCount == 3)
    {
        return (ENDING_CELL);
    }
```

```
    // We reach here only if the MotionPath does not
    // intersect the cell at all.
    return (NO_RELATIONSHIP);
}
```

程序清单 3.6.2 在一个导航网格上解析运动

```
void NavigationMesh::ResolveMotionOnMesh(const vector3& StartPos,
NavigationCell* StartCell, vector3& EndPos,
NavigationCell** EndCell)
{
    // create a 2D motion path from our Start and End
    // positions, tossing out their Y values to project them
    // down to the XZ plane.
    Line2D MotionPath(vector2(StartPos.x,StartPos.z),
    vector2(EndPos.x,EndPos.z));

    // these three will hold the results of our tests against
    // the cell walls
    NavigationCell::PATH_RESULT Result =
    NavigationCell::NO_RELATIONSHIP;
    NavigationCell::CELL_SIDE WallNumber;
    vector2 PointOfIntersection;
    NavigationCell* NextCell;

    // TestCell is the cell we are currently examining.
    NavigationCell* TestCell = StartCell;

    //
    // Keep testing until we find our ending cell or stop
    // moving due to friction
    //
    while ((Result != NavigationCell::ENDING_CELL)
                && (MotionPath.EndPointA() !=
                MotionPath.EndPointB()))
    {
        // use NavigationCell to determine how our path and
        // cell interact
        Result = TestCell->ClassifyPathToCell(MotionPath,
            &NextCell, WallNumber, &PointOfIntersection);

        // if exiting the cell...
        if (Result == NavigationCell::EXITING_CELL)
        {
            // Set if we are moving to an adjacent cell or
            // we have hit a solid (unlinked) edge
            if(NextCell)
            {
                // moving on. Set our motion origin to the
                // point of intersection with this cell
                // and continue, using the new cell as our
```

```
                // test cell.
                MotionPath.SetEndPointA(PointOfIntersection);
                TestCell = NextCell;
            }
            else
            {
                // we have hit a solid wall.
                // Resolve the collision and correct our
                // path.
                MotionPath.SetEndPointA(PointOfIntersection);
                TestCell->ProjectPathOnCellWall(WallNumber,
                MotionPath);

                // add some friction to the new MotionPath
                // since we are scraping against a wall.
                // we do this by reducing the magnitude of
                // our motion 10%
                vector2 Direction =
                    MotionPath.EndPointB() -
                    MotionPath.EndPointA();
                Direction *= 0.9f;
                MotionPath.SetEndPointB(MotionPath.EndPointA() +
                Direction);
            }
        }
        else if (Result == NavigationCell::NO_RELATIONSHIP)
        {
            // Although theoretically we should never
            // encounter this case, we do sometimes find
            // ourselves directly on a vertex of the cell.

            // This can be viewed by some routines as being
            // outside the cell.  To accommodate this rare
            // case, we force our starting point into the
            // current cell by nudging it back so we may
            // continue.

            vector2 NewOrigin = MotionPath.EndPointA();
            TestCell->ForcePointToCellCollumn(NewOrigin);
            MotionPath.SetEndPointA(NewOrigin);
        }
    }

    // we now have our new host cell
    *EndCell = TestCell;

    // Update the new control point position,
    // solving for Y using the Plane member of the
    // NavigationCell
    EndPos.x = MotionPath.EndPointB().x;
```

```
    EndPos.z = MotionPath.EndPointB().y;
    TestCell->MapVectorHeightToCell(EndPos);
}
```

程序清单3.6.3 用A*算法在网格上建立一个导航路径

```
bool NavigationMesh::BuildNavigationPath(NavigationPath& NavPath,
NavigationCell* StartCell, const vector3& StartPos,
NavigationCell* EndCell, const vector3& EndPos)
{
    bool FoundPath = false;

    // Increment our path finding session ID
    // This identifies each path finding session
    // so we do not need to clear out old data
    // in the cells from previous sessions.
    ++m_PathSession;

    // load our data into the NavigationHeap object
    // to prepare it for use.
    m_NavHeap.Setup(m_PathSession, StartPos);

    // We are doing a reverse search, from EndCell to
    // StartCell.  Push our EndCell onto the Heap as the first
    // cell to be processed.

    EndCell->QueryForPath(&m_NavHeap, 0, 0);

    // process the heap until empty, or a path is found
    while(m_NavHeap.NotEmpty() && !FoundPath)
    {
        NavigationNode ThisNode;

        // pop the top cell (the open cell with the lowest
        // cost) off the Heap
        m_NavHeap.GetTop(ThisNode);

        // if this cell is our StartCell, we are done
        if(ThisNode.cell == StartCell)
        {
            FoundPath = true;
        }
        else
        {
            // Process the Cell, Adding its neighbors to the
            //  Open Heap as needed
            ThisNode.cell->ProcessCell(&m_NavHeap);
        }
    }

    // If we found a path, build a waypoint list
```

```
    // out of the cells on the path
    if (FoundPath)
    {
        NavigationCell* TestCell = StartCell;
        vector3 NewWayPoint;

        // Setup the Path object, clearing out any old data
        NavPath.Setup(this, StartPos, StartCell, EndPos,
        EndCell);

        // Step through each cell linked by our A* algorithm
        // from StartCell to EndCell
        while (TestCell && TestCell != EndCell)
        {
            // add the link point of the cell as a way point
            // (the exit wall's center)
            int LinkWall = TestCell->ArrivalWall();

            NewWayPoint = TestCell->WallMidpoint(LinkWall);
            NewWayPoint = SnapPointToCell(TestCell,
            NewWayPoint);
            // just to be sure

            NavPath.AddWayPoint(NewWayPoint, TestCell);

            // and on to the next cell
            TestCell = TestCell->Link(LinkWall);
        }

        // cap the end of the path.
        NavPath.EndPath();
        return(true);
    }

    // no path exists between the two points provided.
    // i.e. "you can't get there from here"
    // This will never happen on a contiguous mesh.
    return(false);
}


bool NavigationCell::ProcessCell(NavigationHeap* pHeap)
{
    if (m_SessionID==pHeap->SessionID())
    {
        // once we have been processed, we are closed
        m_Open  = false;

        // query all our neighbors to see if they need to be
        // added to Open heap
```

```
        for (int i=0;i<3;++i)
        {
            if (m_Link[i])
            {
                // The Distances between the wall midpoints
                // of this cell are held in the order
                // ABtoBC, BCtoCA and CAtoAB.

                // abs(i-m_ArrivalWall) is a formula to
                // determine which distance measurement
                // to use.  We add this distance to known
                // m_ArrivalCost to compute the total cost
                // to reach the next adjacent cell.

                m_Link[i]->QueryForPath(pHeap, this,
                m_ArrivalCost+m_WallDistance[abs(
                i-m_ArrivalWall)]);
            }
        }
        return(true);
    }
    return(false);
}

bool NavigationCell::QueryForPath(NavigationHeap* pHeap,
NavigationCell* Caller, float arrivalcost)
{
    if (m_SessionID!=pHeap->SessionID())
    {
        // this is a new session, reset our internal data
        m_SessionID = pHeap->SessionID();

        if (Caller)
        {
            m_Open  = true;
            ComputeHeuristic(pHeap->Goal());
            m_ArrivalCost = arrivalcost;

            // Remember the triangle wall this caller is
            // entering from
            if (Caller == m_Link[0])
            {
                m_ArrivalWall = 0;
            }
            else if (Caller == m_Link[1])
            {
                m_ArrivalWall = 1;
            }
            else if (Caller == m_Link[2])
            {
```

```
                m_ArrivalWall = 2;
            }
        }
        else
        {
            // We are the cell that contains the starting
            // location of the A* search.

            m_Open  = false;
            m_ArrivalCost = 0;
            m_Heuristic = 0;
            m_ArrivalWall = 0;
        }
        // add this cell to the Open heap
        pHeap->AddCell(this);
        return(true);
    }
    else if (m_Open)
    {
        // A true m_Open means we are already in the Open
        // Heap.  If this new caller provides a better path,
        // adjust our data.  Then tell the Heap to resort our
        // position in the list.

        if ((arrivalcost + m_Heuristic) < (m_ArrivalCost + m_Heuristic))
        {
            m_ArrivalCost = arrivalcost;

            // Remember the triangle wall this caller is
            // entering from
            if (Caller == m_Link[0])
            {
                m_ArrivalWall = 0;
            }
            else if (Caller == m_Link[1])
            {
                m_ArrivalWall = 1;
            }
            else if (Caller == m_Link[2])
            {
                m_ArrivalWall = 2;
            }
            // ask the heap to resort our position in the
            // priority heap
            pHeap->AdjustCell(this);
            return(true);
        }
    }
    // this cell is closed
    return(false);
```

```
}

void NavigationCell::ComputeHeuristic(const vector3& Goal)
{
    // our heuristic is the estimated distance (using the
    // longest axis delta) between our cell center point
    // and the goal location


    float XDelta = fabs(Goal.x - m_CenterPoint.x);
    float YDelta = fabs(Goal.y - m_CenterPoint.y);
    float ZDelta = fabs(Goal.z - m_CenterPoint.z);

    m_Heuristic = __max(__max(XDelta,YDelta), ZDelta);
}
```

3.6.7 参考文献

[Patel99] Patel, Amit J., "Amit's Thoughts on Pathfinding," http://theory.stanford.edu/~amitp/GameProgramming/, November 27, 1999.

[Heyes-Jones99] Heyes-Jones, Justin, "A* Algorithm Tutorial," www.gamedev.net/reference/programming/ai/article690.asp, November 27, 1999.

[Stout96] Stout, W. Bryan, "Smart Moves: Intelligent Path-Finding," *Game Developer*, also www.gamasutra.com/features/19990212/sm_01.htm, October 1996.

3.7 Flocking：一种模拟群体行为的简单技术

Steven Woodcock

Flocking（有时也称为 swarming 或 herding）是由 Craig Reynolds 于 1987 年在一篇为 SIGGRAPH 所写的论文“Flocks, Herds, and Schools: A Distributed Behavioral Model”中首次提出的一种技术。在该文中，Reynolds 提出了 3 个简单的规则，当把它们组合在一起时，为自治主体（也称为 *boids*）群给出了一种类似于鸟群、鱼群或蜂群的群体行为的逼真形式（请看图 3.7.1 中一个这种行为活动的例子）。这些 Reynolds 称之为定向行为（steering behaviors）的规则是：

- **分离**（Separation）。定向时要避免与本地 flock 同伴拥挤。
- **列队**（Alignment）。驶向本地 flock 同伴的平均航向。
- **聚合**（Cohesion）。定向时朝着本地 flock 同伴的平均位置移动。

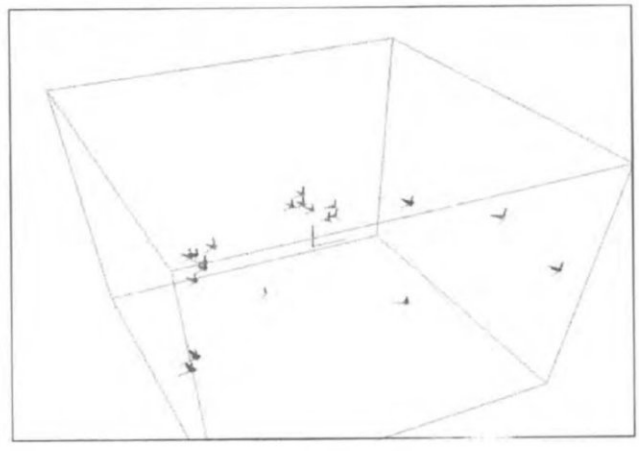


图 3.7.1 一个活动的 flocking 演示的屏幕

分离规则给了一个主体试图与其他邻近的主体保持一定的距离的能力。当确保以一个“看似自然”的接近度模拟真实世界中的群体时，它可以避免主体拥挤在一起。本文介绍的代码通过下述途径达到了这个目的：为一个 flock 的每个成员测试与其相邻同伴的接近程度，然后调整其航向以得到一个期望的距离。

列队规则为一个主体提供了与其他邻近主体列队的能力（即与其他邻近主体航向和/或速度相同）。与分离类似，本文将列队说明为：通过每一

个 flock 成员观察邻近同伴，然后调整它的航向和速度以与其邻近同伴的平均航向和速度相匹配。

聚合规则给了一个主体与其他邻近主体“聚合（group）”的能力，从而模拟自然界中的类似行为。本文中再次通过让每一个主体检测其邻近主体，平均它们的位置然后调整其与之匹配的航向，从而达到上述目的。

1. 第四规则

Reynolds 在稍后的实现和论文中又增加了有时被称做 flocking 的“第四规则”的规则：

- **躲避**（Avoidance）。使避免撞上局部区域内的障碍或敌人。

躲避规则为主体提供了使它绕过障碍和避免碰撞的能力。这种控制行为是这样完成的：通过赋予每个主体“向前看”一段距离的能力，决定与一些对象的碰撞是否可能，然后调整它的航向以避免碰撞。类似地，它也适用于躲避某些其他类型的主体（如兔子躲避狐狸，或者鸽子躲避老鹰）。

2. 无存储

注意定向行为与状态信息、保存 flock 知识的主体、它所前往的环境，以及喜好无关。flocking 是一个无状态算法，其中在两次更新之间没有保存任何信息：每个 boid 在每一轮更新中对它的环境重新评估。这样做不仅与其他能提供类似行为的方法相比减少了存储需求，而且它允许 flock 对于变化了的环境条件进行实时反应。因而，flock 显示出了涌现行为（emergent behavior）的要素：flock 中的任何一个成员都不知道 flock 将到何处，但是该 flock 像一个整体一样运动、躲避障碍和敌人，并以一种流动的、动态的方式彼此保持步调一致。

3. 此概念对有计算机游戏有什么用处？

Flocking 为个体运动和构造一个更为真实的玩家可以探索的环境提供了一个强有力的工具；它已被非常成功地用于多种商业游戏中。例如，Unreal（Epic）和 Half-Life （Sierra）都为许多怪兽以及威胁性小的生物（如鸟和鱼）使用了 flocking 算法。Enemy Nations (Windward Studios)使用了一个改进的 flocking 算法来控制部队队形和越过 3D 环境的运动。与使用简单脚本的结果相比，被创造的动物群体能在实时策略游戏或角色扮演游戏中在地形中更真实地游走，射手群或剑客群能真实地穿过桥梁或者绕过巨石和其他障碍。一个第一人称视角射击类游戏中的怪兽能以一种更可信的方式在城堡的大厅里徘徊，躲避玩家可能袭击的位置，但是当 flock 变得足够大时也许会发动反攻。可能性几乎是无穷的。

3.7.1 实现

1. 矢量和运动

迅速回顾一下 flock 成员的运动机制将有助于理解代码的工作机制。

图 3.7.2 举例说明了局部空间（local space，boid 本身相关的空间）的概念。“前”向着 Z 轴的正向，“左”向着 X 轴的正向，“上”则是与 boid 的顶部垂直。本文中的 boid 具有典型

的 Reynolds 常使用的三角翼形状，当然也可以是其他任何想要的形状。

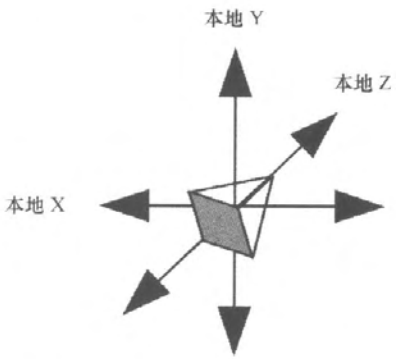


图 3.7.2 为每一个 boid 定义本地空间

图 3.7.3 展示了另一个重要的因素：朝向。朝向指的是 roll，pitch 和 yaw，仅仅是一个特定的对象如何在局部空间被定向的一种指示，如图 3.7.2 所示。Roll 就是绕着局部空间的 Z 轴旋转（朝着前后）。Pitch 是绕局部的 X 轴的旋转（左右偏转）。Yaw 就是绕 Y 轴旋转（贯穿 boid 直接上下运动）。理解这些朝向非常重要，因为当我们决定运动中 boid 的朝向时会用到它们。本文提出的思想通过在每一轮更新期间构造一个速度矢量（velocity vector）来起作用，它将调整 boid 的局部 X，Y 和 Z 的朝向以满足 4 种定向行为规则的要求。

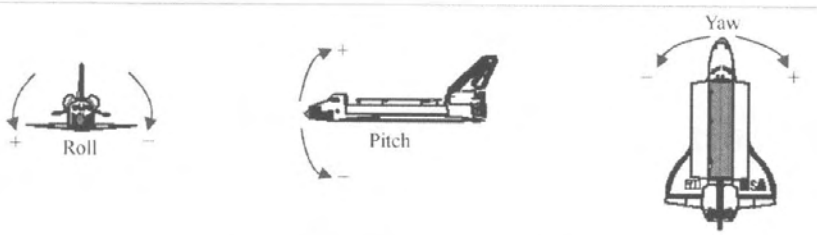


图 3.7.3 三个轴的旋转定义为 roll，pitch 和 yaw

关于这个思想如何起作用的另一个重要方面是一个 boid 的局部上要求的解决冲突的方式。观察一会儿 4 个定向行为规则，就会看到对如何区分这些行为的优先关系并没有什么控制——所有的行为都被认为同等重要。这与真实世界的多数正常行为大致吻合：一个鸽子可能想不但赶上它的同伴，而且在追赶的过程中还要避免穿过一个鹰群。虽然如此，但这个概念没有考虑到生与死的情况。

你可以用许多方法解决这个问题——例如，自动赋给躲避以较高的优先级，不过本文采用的是一个矢量积聚（vector accumulation）的途径。一个特定的 boid 用来满足所有 4 个定向行为规则的朝向变化，在应用于 boid 的运动之前被加到了一个变化矢量（change vector）上。依照惯例，这个变化矢量被转化为一个单位矢量以使积聚的变化保持适当的比例。该办法使得每一个定向行为都能对 boid 最终的运动状态的改变产生相应的影响，同时也使 boid 可以满足（至少部分满足）两个或更多个相互冲突的指令。随着时间的流逝，我发现这个方法一般情况下要比其他的方法更令人满意。

2. 约束

boid 上的一些约束限制了他们如何能够运动和作出反应。也许这一实现中最有影响力的是每一个 boid 的感知域(perception range)，它限制了一个 flock 同伴能够在周围的环境中“看”多远来探测其他的同伴、潜在的障碍或者敌人。这个域越大，flock 组织得越好，粘着性越强，越能更好地避开敌人和障碍。减小这个域值将导致较不稳定的 flock，boid 群遭遇障碍或敌人等时常会分裂。

关于我们的主体怎样才能运动的另一个约束是他们的速率和最大速率变化（maximum velocity change)。在真实世界中，flock 中的动物在跟上同伴方面受到其运动速度与转弯速度等方面的约束。本文通过忽略加速度稍微简化了 3D 环境中的运动问题，并且把精力完全集中到了速率上；速率的改变被限定在全局最大速率的某个比例。这有助于防止我们的演示中的主体原地急转或当试图赶上它们的同伴时以不合理的速度迸发。它也提供了一个关于他们能多快地减速或为了避免障碍而改变路线的控制限制。如果允许“无限响应（infinite response)”，他们可能会在以无限灵活和速度转弯并绕开障碍物之前，径直飞向障碍物的表面——这是一种不太真实的行为。

为了达到演示的目的，最后一个约束是我们的 boid flock 的世界。这就达到了本文的目的，我已经随手创建了一个 Box 类，它定义了我们的 boid 能运动的世界。任何偏离了这个 Box 边界的 boid 都会神奇地迁移到对面，并保持它以前具有的相同的运动特性。结果是离 Box 边界太近的 folck 可能会丢失一些成员到“对面的世界”，在那儿他们可能会失去与主 flock 的联系并形成他们自己的 flock。

事实上所有这些参数都是可以调整的，于是我们可以看到使用它们的潜在的效果。

3.7.2 代码

三个类组成了本文的核心内容：CBox，CFlock 和 CBoid。图 3.7.4 描述了这些类的组织结构。

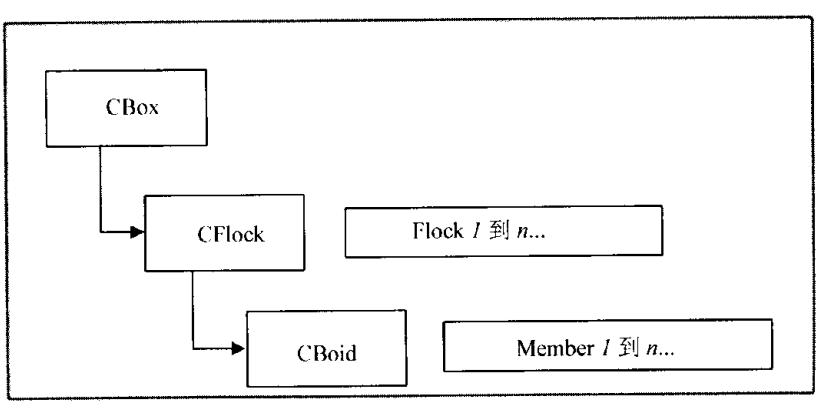


图 3.7.4 Cbox 类定义了我们的 flock 可以飞行的基本世界

每一个 flock 由一个 CFlock 对象实例来表示。可以有多个 CFlock 对象归属于 CBox 内部，

并且它们可以在任何一个点被创建或销毁（虽然本文只是在初始化时创建它们）。CFlock 对象用来组织和简化对 flock 的成员的访问。

类似地，一个 CBoid 对象代表一个 flock 的每个成员。当然，可以有多个 CBoid 与每个 CFlock 相联系（如果不是这样，它可能是一个相当小的 flock），而且像 CFlock 对象一样，它们可以在任何一个点被创建或销毁。虽然本文只是在初始化时创建 flock 的成员，为个体的成员加上一些使用期限也不是什么难事（请看“局限性与可能的改进”部分），于是个体就可以变老和死亡。

1. CBox 类

如我们所料的，CBox 是一个相当简单的类：

```
class CBox
{

public:

   ////////////////////////////////
   // constructors and destructors
   ////////////////////////////////

   // Constructor #1.
   // Creates a Box with default values of 50 meters
   // on any side not specified.
   CBox (float lv=50.0, float wv=50.0, float hv=50.0);

   // Destructor
   virtual ~CBox();

   ////////////////////////////
   // miscellaneous functions
   ////////////////////////////

   // GetBoxLength.
   // Returns the length of the Box, in meters.
   float GetBoxLength (void);

   // GetBoxWidth..
   // Returns the width of the Box, in meters.
   float GetBoxWidth (void);

   // GetBoxHeight.
   // Returns the height of the Box, in meters.
   float GetBoxHeight (void);
```

该类提供了一种简单的方法来确定参数并重新得到世界的边界。

2. CFlock 类

CFlock 类代表了一个基本的 boid 的 flock，并且主要起了一个组织工具的作用而完全不是每个 flock 严格的表示。它的各种函数都相当简单，而且它主要处理的是一些簿记工作，这些簿记可能在你处理 flock 的时候用到。CFlock 类的定义见程序清单 3.7.1。

在程序中，flock 首先被创建并且接着有一个或多个 boid 增加到其中。创建类时要考虑到 flock 的实时产生和删除（虽然演示程序没有这样做）。一个 CFlock 对象的列表被保存到简单静态数组 CFlock::ListOfFlocks[ ]中（为达到本文的目的也简化了）。在每一轮更新中，调用 flock 的 CFlock::Update( )方法以更新该 flock 的所有成员。新的 boid 可以在任何时间用 CFlock::AddTo( )增加，如果需要，成员可以经由 CFlock::RemoveFrom( )被删除。CFlock::GetCount( ) 和 CFlock::GetFirstMember( )提供了获取关于一个特定 flock 的特殊状态信息的方法，而调试方法 CFlock::PrintData( )提供了更详尽的信息。

Flock 能在任意时间被创建，虽然演示程序只是在初始化时创建它们。每一个 flock 可以有任意数目的 CBoid 对象作为成员。注意一个 flock 的成员对于他们属于什么样的 flock 一无所知，但是 flock 自己却知道谁是它的成员。为了达到本文的目的，一个 CBoid 对象内有一个类型为 CFlock 的成员，指明它在开始被分配于其上的那个 CFlock 对象，但是如果需要，增加允许 boid 改变它们所“效忠”的 flock 的代码也丝毫不难（参见“局限性与可能的改进”）。

3. CBoid 类

CBoid 类实现了真正的 flocking 算法的核心内容，因此，它相当简洁。它包含在清单 3.7.2 中。这个类处理了一个特殊的主体运动和存在的所有方面：如何运动，如何感知它的环境，如何区分它的动作的优先次序。

每一个 CBoid 对象表示一个单独的个别主体。与 CFlock 非常象，CBoid 对象能被随意创建或销毁。一旦被创建，它们就如先前所描述的，由 CFlock::AddTo( )，Flock::RemoveFrom( ) 和 CFlock::Update( )间接管理。

每一个 CBoid 对象通过它的 CFlock::FlockIt( )方法来更新，它通过创建一个特定的 boid 能看见的 flock 同伴的列表开始（基于它的感知域值）。如果选项被激活，一个敌人（其他 flock 的成员）的列表也被创建。

该方法然后开始实现前述的定向行为规则，将速度矢量的序列累加起来以适应主体的要求。调用方法 CBoid::KeepDistance( )（分离行为），CBoid::MatchHeading( )（列队行为）和 CBoid::SteerToCenter（聚合行为）来确定 boid 将想做什么事情。如果选项被激活，其他 flock 成员的躲避行为通过对 CBoid::FleeEnemies( )的调用来模仿。

这里实现的另外的一个方法是 CBoid::Cruising( )。该方法试图模仿一个 boid 的“预期的巡航速度（cruising speed）”，如果任何事情都由它决定并且它不受任何其他影响。该方法的主要目的是给任何自由徘徊的 boid（flock 同伴看不到它们）一些运动的“意图”。

在更新过程的每一个阶段，我们将按比例积累所有的预期速率矢量变化将其加和到一个累积矢量中。CBoid::Flockit( )方法最后的一个检查确保了一个个体永远不会超过它的最大允许速度或者最大速率变化。

最后两个方法起到了“清除”的作用以保证每个行为看起来都很自然。作为最终速度矢

量变化结果，CBoid::ComputeRPY( )通过必要的计算确定 boid 的正确朝向。CBoid::WorldBound( )做了一些合理的测试以确定是否有主体偏离于它所处的 CBox 世界对象的边界之外，如果有，就把其放回到盒中。

有各种各样的私有方法处理可见性和链表管理；对于他们来说没有什么特殊之处，它们与 flocking 完全不相干。调试方法 CBoid::PrintData( )提供了在每次更新（update-by-update）的基础上关于一个特定 boid 的详尽的信息。

3.7.3 局限性与可能的改进

严峻的现实使任何 flocking 行为的（原始）演示都有一些局限性。本文根本没有完全实现躲避障碍物，虽然它的确考虑到了以“敌人 flock”形式进行躲避。Boid 仍然分配给它们初始时所属的 flock，但是你可以很容易地设想出给一个 boid 重新分配其所属 flock 的代码，它将会忽略旧的那个 flock。类似地，在此实现中的 boid 几乎是全能的：它具有一个惊人的 360° 全球形的视野；没有对于只能看见正前方的对象或 flock 同伴的限制。任何一个（真实）游戏实现都会希望使用一个更为真实的视野。

可以相当容易地得到其他可能实现的列表。许多已经用了 flocking 行为规则和它的变种的人已经在单独的主体上实现了“生命时钟”，使得那些偏离同伴太远的 boid 死去，但是如果他们有足够时间保持与自己兄弟的关系，就可以“繁殖”新的 flock 成员。你也可以研究捕食者和捕食行为，修改基本代码以允许一种类型的 flock“喂养”另一种。

程序清单 3.7.1 CFlock 类定义

```
class CFlock
{

  public:

    ////////////////////
    // static variables
    ////////////////////

    // number of flocks
    static int FlockCount;

    // list of flocks
    static CFlock * ListOfFlocks[MAX_FLOCKS];

    ////////////////////////////////
    // constructors and destructors
    ////////////////////////////////

    // Constructor.
    // Creates a new flock.
    CFlock (void);
```

```
    // Destructor.
    ~CFlock (void);

    ////////////////////
    // flocking functions
    ////////////////////

    // Update.
    // Updates all members of a flock.
    void Update (void);

    ////////////////////////
    // miscellaneous functions
    ////////////////////////

    // AddTo.
    // Adds the indicated boid to the flock.
    void AddTo (CBoid * boid);

    // GetCount.
    // Returns the # of boids in a given flock.
    int GetCount (void);

    // GetFirstMember.
    // Returns a pointer to the first boid in a
    // given flock (if any).
    CBoid * GetFirstMember (void);

    // PrintData.
    // Dumps all data describing a given flock.
    void PrintData (void);

    // RemoveFrom.
    // Removes the indicated boid from the flock.
    void RemoveFrom (CBoid * boid);

  private:

    int    m_id;              // id of this flock
    int    m_num_members;     // number of boids in this flock
    CBoid  *m_first_member;   // pointer to first member

};
```

程序清单 3.7.2 CBoid 类定义

```
class CBoid {

  public:

    ////////////////////
```

```
// static variables
//////////////////

// visible friends list (work space reused by each boid)
static CBoid * VisibleFriendsList[MAX_FRIENDS_VISIBLE];

//////////////////////////////
// constructors and destructors
//////////////////////////////

// Constructor #1.
// Creates an individual boid with randomized position,
// velocity, and orientation.
CBoid (short id_v);

// Constructor #2.
// Creates an individual boid with specific position,
// velocity, and orientation.
CBoid (short id_v,
       vector * pos_v, vector * vel_v, vector * ang_v);

// Destructor
~CBoid (void);

//////////////////////////
// public flocking methods
//////////////////////////

// FlockIt.
// Used for frame-by-frame updates; no time
// deltas on positions.
void FlockIt (int flock_id, CBoid *first_boid);

//////////////////////////
// miscellaneous functions
//////////////////////////

// AddToVisibleList.
// This visibility list is regenerated for each member each
// update cycle, and acts much like a push-down queue; the
// latest member added to the list becomes the first one
// when the list is sequentially accessed.  Mostly I did
// this for speed reasons, as this allows for fast inserts
// (and we don't delete from this list, we just rebuild it
// each update cycle).
void AddToVisibleList (CBoid *ptr);

// ClearVisibleList.
// Clears the visibility list.
void ClearVisibleList (void);
```

```
    // GetNext.
    // Returns the "next" pointer of the invoking member.
    CBoid * GetNext();

    // LinkOut.
    // Removes a member from a list.
    void LinkOut ();

    // PrintData.
    // Dumps all data describing a given member.
    void PrintData (void);

    // SetNext.
    // Set the "next" pointer of an individual member.
    void SetNext (CBoid *ptr);

    // SetPrev.
    // Set the "prev" pointer of an individual member.
    void SetPrev (CBoid *ptr);

private:

    ///////////////
    // data members
    ///////////////

    // supplied with constructor(s)
    short   m_id;                          // member individual ID
    float   m_perception_range;            // how far member can see
    vector  m_pos;                         // position of member
                                           //(in meters)
    vector  m_vel;                         // velocity of member
                                           // (meters/sec)
    vector  m_ang;                         // orientation of member


    // computed
    float   m_speed;                       // overall speed of member
    u_short m_num_flockmates_seen;         // # of flockmates this
                                           // member sees
    u_short m_num_enemies_seen;            // # of enemies this
                                           // member sees
    CBoid   *m_nearest_flockmate;          // pointer to nearest
                                           // flockmate (if any)
    CBoid   *m_nearest_enemy;              // pointer to nearest
                                           // enemy (if any)
    float   m_dist_to_nearest_flockmate;   // distance to
                                           // nearest flockmate
                                           // (if any), in
```

```
                                        // meters
float   m_dist_to_nearest_enemy;        // distance to
                                        // nearest enemy
                                        // (if any), in
                                        // meters
vector  m_oldpos;                       // last position
vector  m_oldvel;                       // last velocity
CBoid   *m_next;                        // pointer to next
                                        // flockmate
CBoid   *m_prev;                        // pointer to
                                        // previous
                                        // flockmate

////////////////////
// flocking methods
////////////////////

// Cruising.
// Generates a vector indicating how a flock boid would
// like to move, if it were all up to him and he was under
// no other influences of any kind.

vector CBoid::Cruising (void);

// FleeEnemies.
// Generates a vector for a flock boid to avoid the
// nearest enemy (boid of a different flock) it sees.

vector CBoid::FleeEnemies (void);

// KeepDistance.
// Generates a vector for a flock boid to maintain his
// desired separation distance from the nearest flockmate
// he sees.

vector CBoid::KeepDistance (void);

// MatchHeading.
// Generates a vector for a flock boid to try
// to match the heading of its nearest flockmate.

vector CBoid::MatchHeading (void);

// SeeEnemies.
// Determines which enemy flock boids a given flock boid
// can see.

int CBoid::SeeEnemies (int flock_id);

// SeeFriends.
```

```
    // Determines which flockmates a given flock boid can see.
    int CBoid::SeeFriends (CBoid *first_boid);

    // SteerToCenter.
    // Generates a vector to guide a flock boid towards
    // the "center of mass" of the flockmates he can see.

    vector CBoid::SteerToCenter (void);

    // WorldBound.
    // Implements a world boundary so that flocks don't fly
    // infinitely far away from the camera, instead remaining
    // in a nice viewable area.  It does this by wrapping flock
    // boids around to the other side of the world, so (for
    // example) they move out the right and return on the left.

    void CBoid::WorldBound (void);

    ////////////////////////
    // miscellaneous functions
    ////////////////////////

    // AccumulateChanges.
    // Adds vector values in changes into the accumumlator
    // vector. Returns magnitude of accumulator vector after
    // adding changes.

    float CBoid::AccumulateChanges (vector &accumulator,
      vector changes);

    // CanISee.
    // Determine whether a given invoking boid can see the boid
    // in question. Returns the distance to the boid.

    float CBoid::CanISee (CBoid *ptr);

    // ComputeRPY.
    // Computes the roll/pitch/yaw of the flock boid based on
    // its latest velocity vector changes.  Roll/pitch/yaw are
    // stored in the "ang" data member as follows:
    //   pitch is about the x axis
    //   yaw is about the y axis
    //   roll is about the z axis
    // All calculations assume a right-handed coordinate
    // system:
    //   +x = through the left side of the object
    //   +y = up
    //   +z = through the nose of the model
    void CBoid::ComputeRPY (void);
};
```

3.7.4 资源与致谢

事实上研究这一独特领域需要快速发现 Web 上几乎每个涉及、源出或被其他 flocking/swarming/herding 应用激发灵感的 flocking/swarming/herding 应用，不然这几乎是不可能的。本文描述的实现也不例外。非常感谢 Christopher Kline（Mitre 公司）独创的最早发表在他的 C++ Boids 实现中（在其网站上可以找到）的计算 roll/pitch/yaw 的方法（在此进行了很大改动）。也要感谢 Mike Louie (Boeing)在数学变换方面的帮助（我讨厌死矩阵了）。

除 Christopher 的网页（www.media.mit.edu/~ckline/boids/，它包括了很多极好的演示和示例代码）之外，也可以在互联网或是在书店中找到许多其他极好的关于本主题的参考文献。也许最好的寻找更多信息的出发点就是“flocking 之父” Craig Reynolds。Craig 的网站：www.red.com/cwr。还请看 Reynolds, C. W., “Flocks, Herds, and Schools: A Distributed Behavioral Model,” in *Computer Graphics*, 21(4),SIGGRAPH '87 Conference Proceedings, pages 25–34, 1987。

Microsoft DirectX SDK 在资源光盘中也提供了两个相当简单易学的实现。在 *DirectX 7a* CD 上，可以在下述目录找到：*\DXF\samples\multimedia\d3dim\src\boids* 和*\DXF\samples\multimedia\dmusic\src\dmboids*。两个版本都以用一个看起来相当自然的“力场（force-field）”方法的障碍躲避为特色。

最后，还有一本极好的书，它除了讨论一般的人工生命的主题外，还讨论了 flocking 和 boid，那就是由 Steven Levy 所写的《人工生命》（Artificial Life）。

3.8 用于视频游戏的模糊逻辑

Mason McCuskey

本文是对于一种称为模糊逻辑的人工智能技术的介绍。

说明模糊逻辑的最好方法是解释它与传统逻辑有哪些不同之处。传统逻辑致力于“真”与“假”的思想——也就是开或关，零或1，是或否，正或负。

模糊逻辑允许我们使用非“一刀切”的概念，也就是说，一些东西需要用一个形容词来表示“到什么程度”或者“多少”。例如，模糊逻辑允许我们量化地模拟大小概念，如“非常大”，“十分小”，“中等”，“巨大”等等。

模糊逻辑在游戏 AI 中有无数的应用。例如，我们可以使用模糊逻辑来模拟计算机控制的角色的情感：“恼怒的”对“极度愤怒”，“有一点儿紧张”对“恐惧的”，“快乐”对“狂喜”等等。这反过来允许我们创造一个更像人类的或比使用传统逻辑（“黑”或“白”）所能创建的都聪明的 AI。

3.8.1 模糊逻辑如何工作

为了详细说明模糊逻辑如何工作，让我们首先来回顾一下传统逻辑是如何工作的。传统逻辑使用的是“一刀切集合”。一个一刀切集合是这样一种集合：一个给定元素要么属于它要么不属于它。例如，让我们定义一个一刀切集合 M，它由所有在 5 与 10 之间的实数组成：

$M = [5, 10]$

M 的特征函数看起来如图 3.8.1 所示（为了这个例子，让我们假定我们的论域是所有 0 与 20 之间的实数，如图 3.8.1 所示）。

这是一个一刀切集合，因为我们的论域中的任何数字或在集合 M 中或不在集合 M 中，即，如果该数字是在 5 与 10 之间，就返回函数值 1，否则就返回 0。

一个一刀切集合对非黑即白情节作用得非常好，但是在某些情形下它就会崩溃。我们可以断定 7 英尺是高的，于是我们可以声明所有高人的集合为“所有最少 7 英尺高的人”。我们建立一个函数，如果给定的高度比 7 英尺大就返回 1，否则就返回 0。问题是如果有一些人高 6 英尺 11.5 英寸，它不在我们的“高人”集合中，即使几乎没有人怀疑他们是高的这一事实。所以，为了试图解决这个问题，我们可以将要求的最低高度降低到 6 英尺，

但是这就像让一个5英尺11英寸的人和一个6英尺的人并排站在一起，却说他们一个“高”，另一个“不高”一样愚蠢。一刀切的集合的严格性不利于我们的工作。

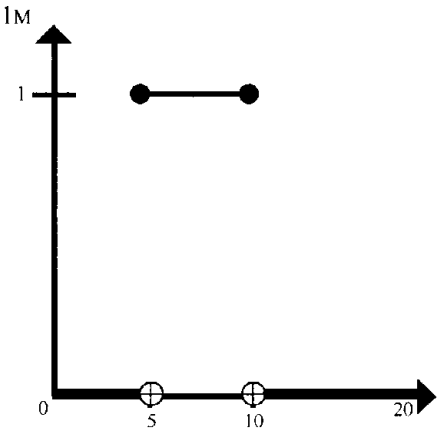


图3.8.1 集合M的表示

换句话说，一刀切的集合没有给我们的就是区分某事物在集合中是“多少”（或者什么程度）的能力。然而模糊集合，给了我们这种能力。使用模糊集合，我们可以“模糊（flex）”“在集合中”与“不在集合中”之间的界限，包括一些如“仅仅有一点儿在集合中”或“差不多全在集合中”。

我们通过改变特征函数的返回值来做到这一点，即让特征函数不但返回0或1，而且也返回0与1之间的一个值，它表明给定数字在什么程度上属于集合。回到我们前面的例子，如果0表示“不高”而1表示“高”，那么0.5就表示“有些高”（或者，“在高的人中属于中等”），而且0.01可以表示“有一点儿高”（刚好勉强能在高的集合中）。

图3.8.2表示了所有高大的人的模糊集合。

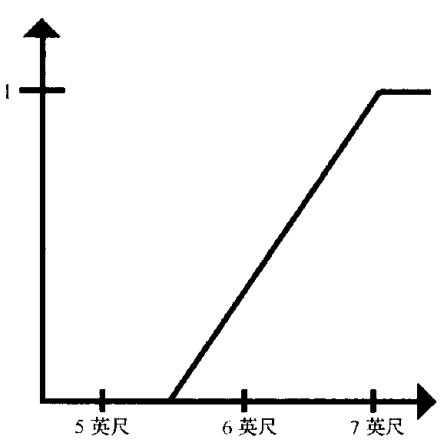


图3.8.2 集合“高大的人”的表示

请比较一下图3.8.2中的图形与图3.8.1中一刀切集合的图形。图3.8.2中的模糊集合有一

个斜坡——5 英尺的人是不高的，但是从 5 英尺 5 英寸开始，高度就逐渐开始属于高的集合了，直到最后达到 7 英尺，他们就完全属于高的集合了。

那是一个模糊集合。

3.8.2 模糊逻辑运算

现在我们明白了一个模糊集合是什么，下面让我们在其上进行一些运算。

图 3.8.3 定义了另一个模糊集合：“大约 6 英尺高”的人的集合。

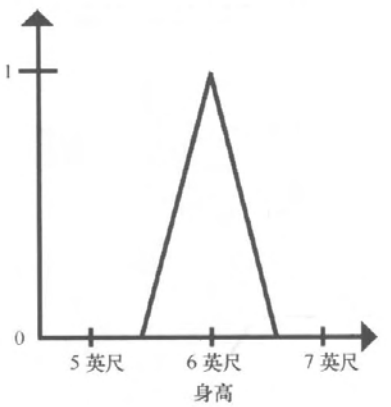


图 3.8.3 集合“大约 6 英尺高”的表示

这儿有一个关于模糊集合的 AND 运算的例子。图 3.8.4 展示了“高且（AND）大约 6 英尺的人”的模糊集的图形。

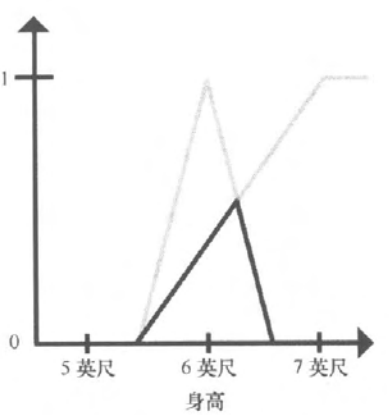


图 3.8.4 “高且大约 6 英尺的人”的集合的表示

基于同样的思想，图 3.8.5 展示了“高或（OR）6 英尺的人”的模糊集合的图形。

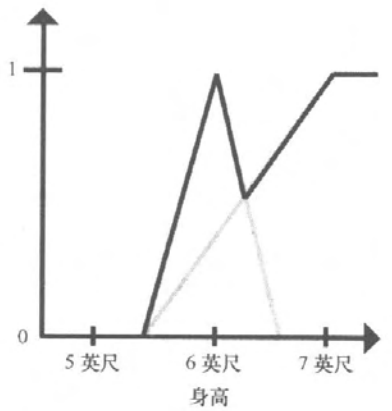


图 3.8.5 “高或（OR）6 英尺的人”集合的表示

最后，图 3.8.6 展示了一个非运算的例子：“不高的人”的集合。

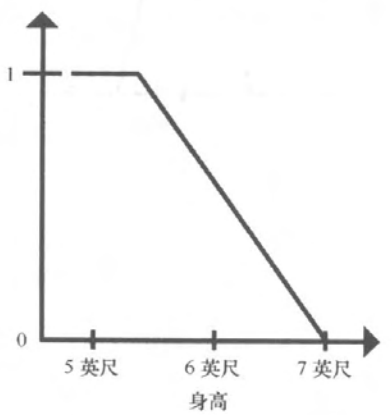


图 3.8.6 “不高的人”集合的表示

3.8.3 为模糊控制而刹车

现在我们知道了如何创建模糊集合并在其上进行运算了，下一步就是应用模糊集来完成一些功能，这称为模糊控制（fuzzy control）。

为了举例说明，让我们模拟一下交通。也许我们正在开发一个城市模拟游戏，而且我们想让城市中的小汽车表现得更为真实。我们有一队汽车，且想让队中的每一辆汽车加速或减速，就像它被一个真人驾驶一样。这意味着没有汽车会撞上另一辆汽车的尾部，并且每辆汽车内有一个谨慎的驾驶员，他试图在他（或她）的汽车与其他前面的汽车之间保持适当的距离（两倍车长）。

这一情形可以很容易地用模糊逻辑来模拟，因为对于每辆车而言，我们仅需考虑两个变量（在模糊逻辑中，它们被称为语言变量（linguistic variables）：

（1）这辆车与它前面的车之间的距离（我们称这个语言变量为距离）。

（2）这辆车与它前面的车之间的距离增量。如果两辆车之间的空间是增加的，我们就有一个正的距离增量；如果空间是缩短的，就有一个负的距离增量。如果该空间既不增加也不缩短，距离增量就为零。

经过数小时仔细研究在真实的高速公路上的真实的汽车，我们得到了一些规则。对每辆汽车：

- 如果距离增量是零并且距离大约是两倍车长，则保持当前速度。
- 如果距离增量是负的并且距离比两倍车长小，则减速。
- 如果距离增量是正的并且距离比两倍车长大，则加速。

还有许多规则，它们都依照同样的模式并且被汇总在表 3.8.1 中。

表 3.8.1　　汽车 AI 的规则

		距离增量				
		快速缩短（负）	缩短（负）	稳定（零）	增加（正）	快速增加（正）
距离	很小	急刹车	急刹车	减速	减速	保持速度
	小	急刹车	减速	减速	保持速度	加速
	正好（两倍车长）	减速	减速	保持速度	加速	加速
	大	减速	保持速度	加速	加速	全速
	很大	保持速度	加速	加速	全速	全速

现在我们已经得到了规则，下面我们需要明确模糊逻辑的已经用于描述距离和距离增量的所有条件。这意味着我们需要定义 15 个模糊集合：距离、距离增量和汽车动作（我们简称为动作）各 5 个。表 3.8.2、3.8.3 和 3.8.4 及图 3.8.7、3.8.8 和 3.8.9 中的图形概括了这些集合。

表 3.8.2　　距离的模糊集合定义

距离标记	模糊集
很小	小于一倍车长
小	大约一倍车长
正好	大约两倍车长
大	大约三倍车长
很大	小于三倍车长

表 3.8.3　　距离增量的模糊集合定义

距离增量标记	模糊集
快速缩短	约为负（当前车速的一半）
缩短	小于零
稳定	约为零（两车以大体相同的速度移动）
增加	大于零
快速增加	约为当前车速的一半

表 3.8.4 动作的模糊集合定义

动作标记	动作
急刹车	速度减半 (speed /= 2)
减速	减速到当前速度的一半 (speed -= speed / 2)
保持速度	不采取动作
加速	增加当前速度的一半 (speed += speed / 2)
全速	速度加倍 (speed *= 2)

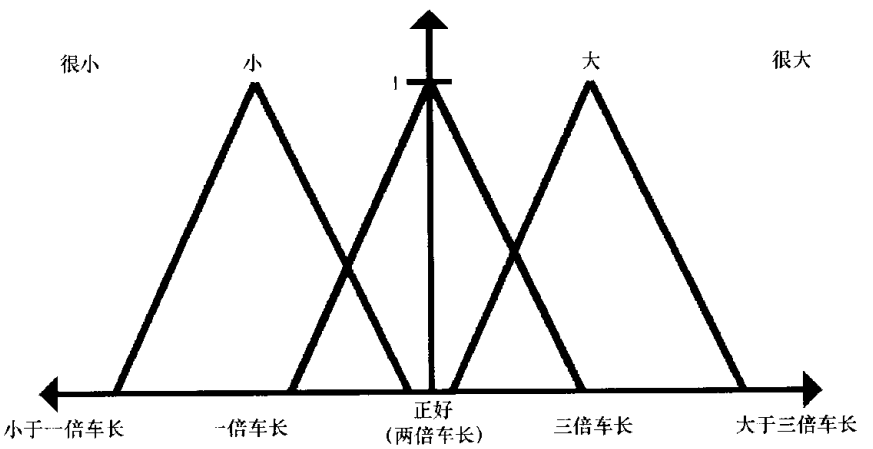


图 3.8.7 距离定义的表示

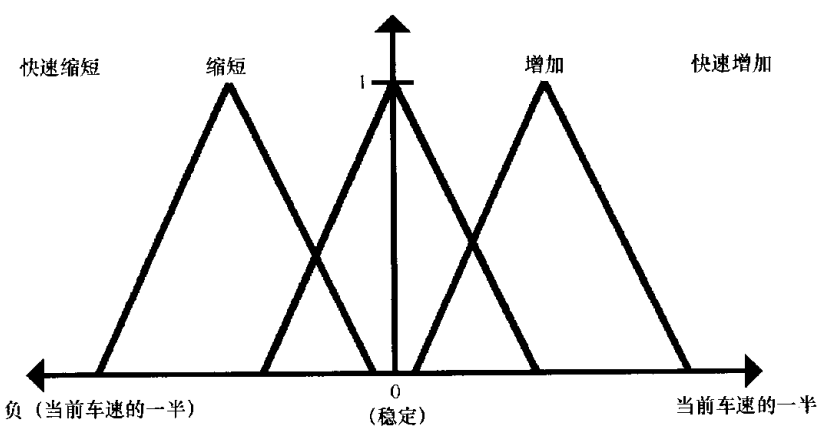


图 3.8.8 距离增量定义的表示

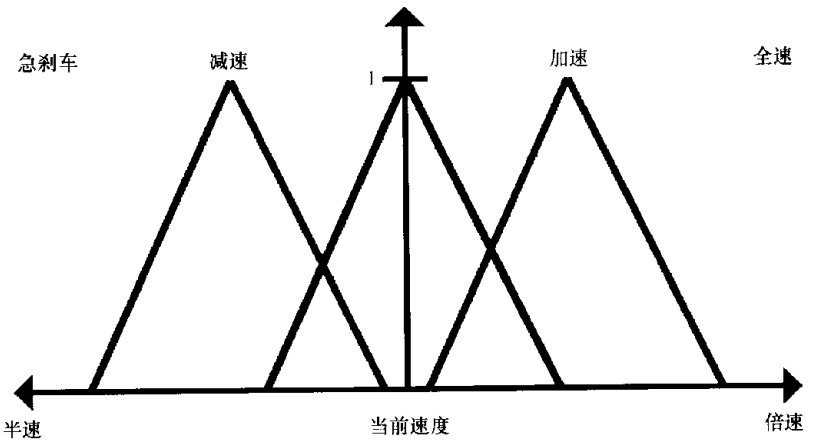


图 3.8.9 动作定义的表示

现在让我们举一个实际的例子来看看数字是如何通过我们刚才创建的模糊控制系统流动的。图 3.8.10 展示了我们为距离选定的一个实际的值；图 3.8.11 展示了距离增量的一个实际值。

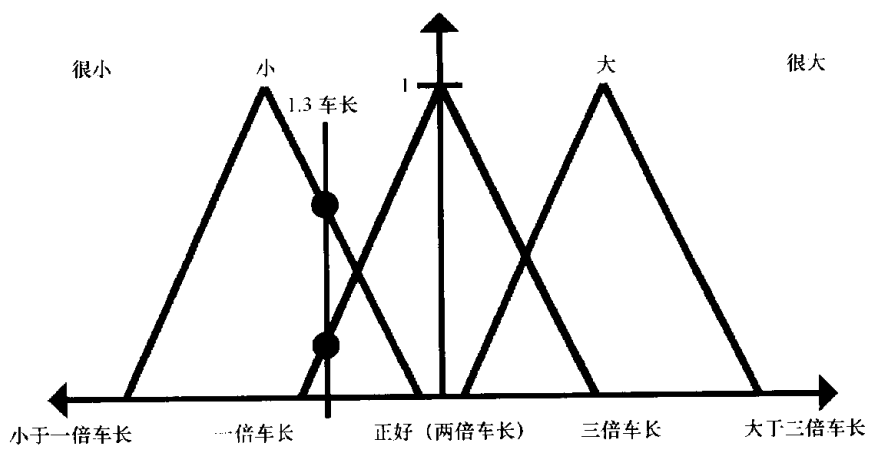


图 3.8.10 我们为这个例子选取的距离值

正如你看到的，我们选了 1.3 作为距离变量，0.25 作为距离增量。模糊集告诉我们，一个 0.25 的距离增量是“slightly growing”（它以大约 0.3 的程度隶属于“growing”集合），而且一个 1.3 的距离是“mostly small”（它以大约 0.75 的程度隶属于“small”集合）。注意我们也可以说距离是“barely perfect”（它以 0.1 隶属于“perfect”集合），且距离增量是“mostly stable”（它以大约 0.6 的程度隶属于“stable”集合）。

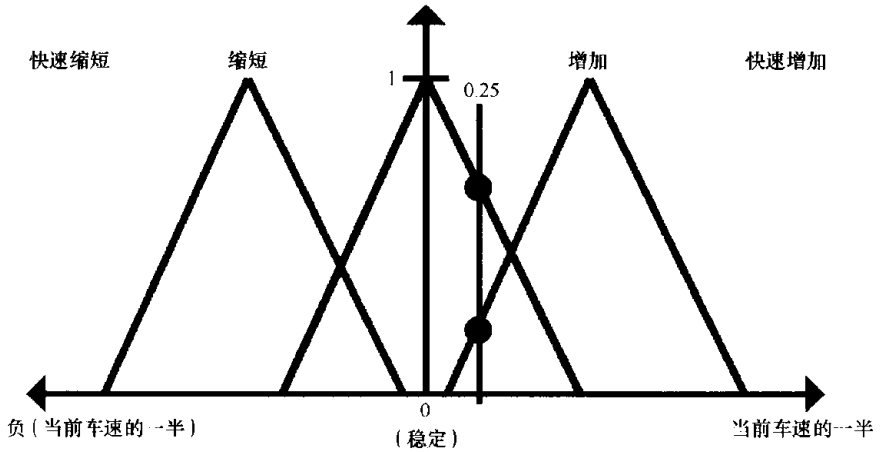


图 3.8.11 我们为这个例子选取的距离增量值

这意味着我们正在处理 4 个集合：其中距离属于“小”和“正好”这两个集合，距离增量属于“增加”和“稳定”这两个集合。给出这些集合的组合，我们知道无论做出什么决定都将基于以下 4 条规则：

- 如果距离是“小”且距离增量是“增加”，保持当前速度。
- 如果距离是“小”且距离增量是“稳定”，减速。
- 如果距离是“正好”且距离增量是“增加”，加速。
- 如果距离是“正好”且距离增量是“稳定”，保持速度。

下一步是估计每一条规则为“真”的程度。

让我们看一下第一条规则。我们需要保持当前速度的程度依赖于陈述“距离小且距离增量在增加”的“真实”度。我们知道距离对“小”的隶属度是 0.75，并且距离增量对“增加”的隶属度是 0.3。于是，我们知道了模糊陈述“距离小且距离增量在增加”的结果是 0.3。这是因为 0.3 是两个陈述都保持为真的最大程度。

我们也能同样估计其他 3 条规则的“真实”度，得到下面的结果：

- 距离小且距离增量增加：“真实”度为 0.3。
- 距离小且距离增量稳定：“真实”度为 0.6。
- 距离正好且距离增量增加：“真实”度为 0.1。
- 距离正好且距离增量稳定：“真实”度为 0.1。

这意味着我们可能的动作“保持速度”得到分值 0.3 和 0.1，“减速”得到一个分值 0.6，而“加速”的分值为 0.1。

我们这里使用的得到最终的值的方法称为一种 defuzzification 方法。有许多可用的 defuzzification 方法，需要从中选择一个适合于你的应用的方法。然而多数时间在行动图形的“真正的”区域上执行一个简单的质心（center-of-mass）计算就足够好了（参见图 3.8.12）。

这个计算给了我们最终的行动过程，即“减速”到一个大约 0.25 的程度。现在就是一个简单地应用 25%的减速规则到汽车的当前速度的问题了。由于“减速”到一个程度 1.0 是汽车速度的 0.75，25%的减速规则意思是我们应该用大约 0.81 去乘汽车速度。

所以汽车渐渐地减速下来，它对于输入标准（距离小且距离增量增加）是合理的。

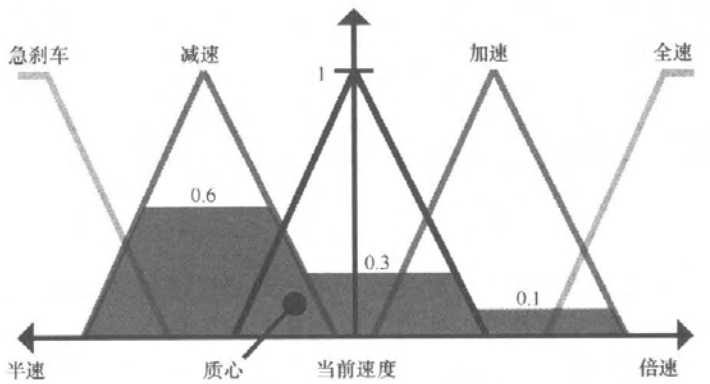


图 3.8.12 使用 defuzzification 方法

当然，计算机要每秒执行几百次我们刚才经过的整个过程，模拟一个谨慎的驾驶员的老练控制。

3.8.4 模糊逻辑的其他应用

模糊逻辑和模糊控制能被用于各种游戏情形。通常的想法是模糊逻辑可以被用在我们试图模拟一个人类专家的任何地方。应用模糊逻辑的其他好地方还包括敌人 AI（如战斗中的魔鬼对玩家的游侠的恐惧到了什么程度？），非玩家角色（如店主多么信任玩家？），flocking 算法（我离部队里的其他人相距多远？），以及其他无数地方。

模糊逻辑也可以被用于表现非生物事件，比如给出风速和方向，描述云是如何运动的。

3.8.5 结论

模糊逻辑是一个有着许多应用的有力工具。非常幸运，本文解释了模糊逻辑和模糊逻辑如何工作的过程，也给出了一切在你的游戏中何处运用模糊逻辑的建议。

如果你有疑问或是意见，请与我联系或访问我的网站。请在作者部分查看我的联系方式。

3.8.6 资源

[Bauer00] Bauer, Peter, Nouak, Stephan, and Winkler, Roman, www.fl11.uni-linz.ac.at/pdw.fuzzy/index.html, March 21, 2000.

[Nguyen99] Nguyen, Hung T., and Walker, Elbert A., *A First Course In Fuzzy Logic,* CRC Press, 1999.

[Rao95] Rao, Valluru B., and Rao, Hayagriva Y., *C++ Neural Networks and Fuzzy Logic,* IGD Books Worldwide, 1995.

[Woodcock00] Woodcock, Steven M., "Game AI," www.gameai.com, March. 21, 2000.

3.9 神经网络初探

André LaMothe

数字计算机在计算上的限制已经通过许多途径被认识到了。我们的确能不断使它们变得速度更快，体积更小，价格更廉，但数字计算机处理的将永远是数字信息，因为它是基于确定性的二进制计算模型的。另一方面，神经网络（Neural nets），却基于不同的计算模型。它基于高度并行、分布式、概率性的模型，不需要模拟一个计算机程序解决问题的方法，而是模拟一个由单元构成的网络，该网络能够发现、确定或通过对问题分而治之再把结果放在一起，以一种更生物学的方法与问题的可能解决方案联系起来。本文是一个关于神经网络及其工作原理的简要介绍。

3.9.1 生物学仿真

神经网络是从我们自己的大脑得到灵感的。某些人的大脑说："我想知道我是如何工作的？"然后开始创造一个它自身的模型。是不是有些神秘？标准的神经元（neurode）模型是一种基于人类神经元的简化模型，50 多年前就被发明了。如图 3.9.1 中所示，一个生物神经元由三个主要部分组成：

- **树突**。负责收集输入信号。
- **细胞体**。负责主要的处理过程及信号求和。
- **轴突**。负责向其他树突传递信号。

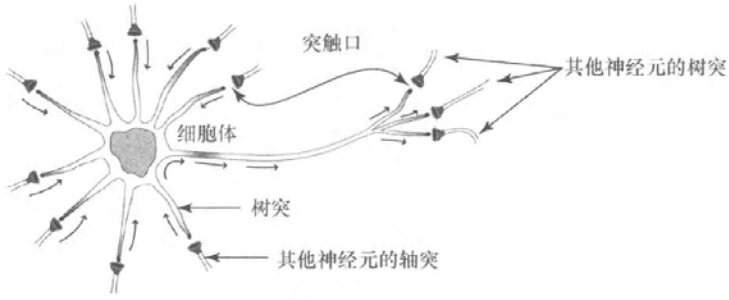


图 3.9.1 一个基本的生物神经元

人类大脑平均有 100 000 000 000 个或者说 10^{11} 个神经元，并且每个神经元通过树突与多达 10 000 个神经元相连。信号通过基于钠、钾和氯离子的电化学过程被传递。信号通过积累由这些离子引起的电位差来传递。化学过程并不重要，而可以将信号看作简单的从轴突传递到树突的电脉冲。

从一个树突到一个轴突的连接部位称为突触（synapses），它们是基本的信号传递点。

那么一个神经元是如何工作的呢？并没有一个简单的答案，但是为达到我们的目的，下面的解释就足够了。树突收集从其他神经元接收的信号，然后细胞体进行所谓的求和，并且基于该结果激活轴突并传输信号。激活视许多因素而定，但是我们可以用一个传递函数来模拟它，该函数处理总计过的输入并且如果传递函数被满足的话，就产生一个输出。此外，在实际的神经元中输出是非线性的，也就是说，信号不是数字的而是模拟的。事实上，神经元不断地接收和发送信号，并且它们的真实模型依赖于频率且必须被在 S-域（S-domain，频域）中进行分析。事实上，一个简单生物神经元的真实传递函数已经被导出了，而推导过程要写满几黑板。

现在我们已经对“神经元是什么”和“我们正试图模拟什么”有了一定的了解，下面让我们谈谈能用神经网络为视频游戏做些什么。

3.9.2 对游戏的应用

神经网络似乎是我们正期待的答案。如果我们能给游戏中的角色哪怕一点儿智力，可以想象一个游戏将会变得多么棒！在某种意义上这是可能的。神经网络只是以一种粗糙的方法模拟了神经元的结构，而不是高水平的推理和演绎功能——至少不是其经典含义。寻找将神经网络技术应用于游戏 AI 的方法要颇费思量，但是一旦你掌握了它，就能与确定性算法、模糊逻辑以及遗传算法结合起来，为你的游戏产生非常健壮的思维模型。毫无疑问，它将比任何你用上百个 if-then 语句或是脚本逻辑达到的效果都要好。神经网络能被用于如下方面：

- **环境扫描与分类。**一个神经网络能被馈入解释为视觉或听觉信息的信息，这些信息随后被用于选择一个输出响应或训练网络。这些响应能被实时学习和修正，以便对其进行优化。
- **记忆。**一个神经网络能被游戏生物用作一种存储形式。神经网络能通过经历一个响应集合进行学习，然后当一个新的体验发生时，网络就能够以关于应该做什么的最可能的推测来进行响应。
- **动作控制。**一个神经网络的输出能用于控制一个游戏生物的动作。输入可以是游戏引擎中的各类变量，然后网络就可以控制游戏生物的动作了。
- **响应映射。**神经网络的确善于“联想”，就是从一个空间到另一个空间的映射。有两种不同的联想：自体联想（autoassociation），它是一个输入与其自身的映射；异体联想（heteroassociation），它是一个输入与其他东西的联想。响应映射在后端或输出端使用了一个神经网络，在其控制下产生另一个间接层或一个对象的行为。本质上，我们可以有许多控制变量，但是我们只对几个特定的组合有一刀切式的响应，可以用它们训练网络。然而，在输出端使用一个神经网络，我们可以得到其他的响应，它们与定义好的那些有同样的活动领域。

前述例子可能看起来有一点儿模糊，的确如此。重要的是神经网络是一种能以我们想要的任何方式使用的工具。关键是要巧妙地使我们的 AI 编程更容易，使我们的游戏生物响应得更智能。

3.9.3 神经网络 101

本节覆盖了神经网络讨论中使用的基本术语和概念。这并不容易做到，因为神经网络实在是一个是众多学科的交叉领域，并且每个学科都为它创造了自己的词汇表。我们这里描述的词汇表是那些众所周知的词汇表的一个较好的交集。此外，神经网络理论充满了冗余研究，这意味着许多人可能做了重复的研究工作，结果是创建了大量的具有不同名称的神经网络。我试图在本文中尽量采用最通用的名称以免陷入名称混淆。在本文稍后部分我们将介绍一些截然不同的网络，完全可以用他们适当的名称来代表。如果你对有的概念还不熟悉，不要过于惊慌，暂时只管读下去，大部分概念将在本文其余详细的上下文中再次讨论。让我们开始吧。

现在我们已经看到了一个神经元的“通俗化”版本了，下面让我们看一下基本的人工神经元（artificial neuron），再在其上进行讨论。图 3.9.2 是一个标准神经元（standard neurode）或人工神经元的图示。你可以看到有许多标为 X_1 – X_n 和 B 的输入。每一个输入都有一个连接权值 w_1 – w_n，还有一个附属于它们的 b。此外，有一个求和点 Y 和一个单独的输出 y。神经元的输出基于一个传递函数或“激励（activation）”函数，它是一个神经元的总输入的函数。输入来自于 X_i 和一个修正点 B。B 可以被认为是一段过去的历史、记忆或倾向。

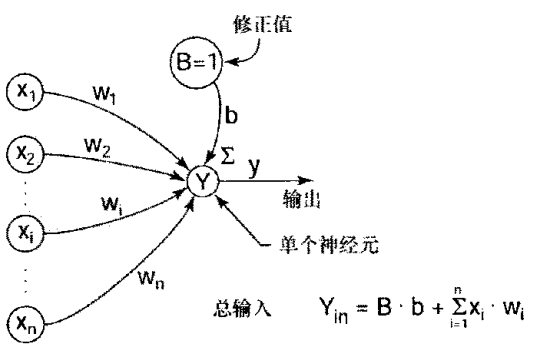


图 3.9.2 一个单独的有 n 个输入的神经元

神经网络的基本运算如下：每个输入 X_i 与其的连接权值相乘然后求和，和的输出被称为输入激励（input activation）Y_a，这个激励值然后被馈给激励函数 $f_a(x)$，最后的输出是 y。运算公式如下：

$$Y_a = B * b + \sum_{i=1}^{n} X_i * w_i \tag{3.9.1}$$

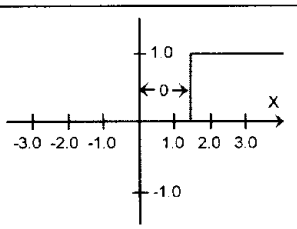
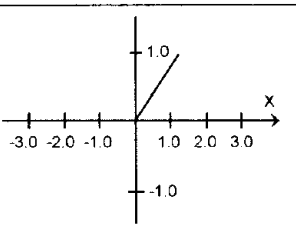
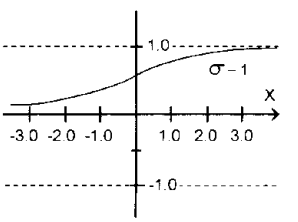
以及 $y = f_a(Y_a)$。$f_a(x)$ 的不同形式随后讨论。

在继续下面的讨论之前，我们需要先来谈谈输入 $\mathbf{X_i}$，权值 $\mathbf{w_i}$，以及它们各自的域。在多数情形，输入由集合（$-\infty, +\infty$）中的正整数和负整数组成。然而，许多神经网络使用更简单的二价（bivalent）的值（意味着它们只有两个值）。采用这样一种简单的输入形式是因为最

终的所有输入都是二进制的（binary）或是两极的（bipolar），复杂的输入会被转化为纯二进制或两级表示。此外，多数时间我们是试图解决计算机问题，如图像或声音识别，它们适宜于二价表示。不过，这个规则并不是一成不变的。在二价系统中使用的值主要有二进制系统中的 0 和 1，或者两级系统中的－1 和 1。除了两级表示在数学上要优于二进制表示以外，这两个系统很相似。每个输入上的权值 $\mathbf{w_i}$ 也是在典型的（$-\infty, +\infty$）范围内，正值和负值分别被称做兴奋（excitatory）和抑制（inhibitory）。额外的输入 B（修正值）总是 1.0，且与 b 成比例或被 b 相乘，即在某种意义上可以说 b 是它的权值。公式 3.9.1 中的第一项说明了这个概念。

下面继续进行我们的分析。一个神经元的激励 $\mathbf{Y_a}$ 一旦被找到了，就会应用于激励函数，然后就可以计算出输出 y 了。有许多激励函数，各有不同的用途。基本的激励函数 $f_a(x)$ 如表 3.9.1 所示。

表 3.9.1　　激励函数 fa(x)

阶跃函数	线性函数	指数函数
		
公式 3.9.2 $F_s(x) = 1, \text{if } x \geqslant \theta$ $0, \text{if } x < \theta$	公式 3.9.3 $F_l(x) = x$, 对所有的 x	公式 3.9.4 $F_e(x) = 1/(1+e^{-\sigma x})$

每个函数的公式都相当简单，但每个函数都是为模拟或适合不同的特性而导出的。

在许多神经网络和模型中，当输入信号达到临界状态时，阶跃（step）函数被用于激活神经元。这就是因子 $\boldsymbol{\theta}$ 的用途，它模拟了神经元将被激活的临界输入水平或阈值。当我们想使神经元的输出更接近于输入激励时，使用线性激励（linear activation）函数。这种激励函数被用于模拟线性系统，如基本的均速运动。最后，指数激励（exponential activation）函数用于创建一个非线性响应（non-linear response），这是产生一个有非线性响应的神经网络和模拟非线性过程的最佳可能途经。指数激励函数是高级神经网络的关键，因为线性和阶跃函数的合成通常也是线性的或阶跃的，永远无法用它们创建一个有非线性响应的网络。因此，我们需要用指数激励函数来处理欲用神经网络进行解决的非线性问题，但并非锁定了指数函数。依赖于期望的网络特性，也可以使用双曲（Hyperbolic）函数、对数（logarithmic）函数和超越函数（transcendental functions）。最后，如果需要，我们可以对所有这些函数进行按比例缩放或位移。

正如你可以想象到的，一个单独的神经元并不能为我们做太多的工作，所以我们需要取一组神经元创建一个神经元层，如图 3.9.3 所示，这个图说明了一个单层神经网络。图 3.9.3 中的神经网络有许多输入节点和输出节点。依照惯例，它称为一个单层神经网络，因为输入层并不计入到层数当中，除非它是惟一的层（在这种情形，输入层也是输出层，也就只有一层了）。图 3.9.4 展示了一个两层神经网络，输入层仍然没有计数，而且内部的层被称做是“隐含（hidden）”层，输出层也称为响应层（response layer）。理论上，对一个神经网络可以拥有

的层数并无限制；然而，从多个不同的层中导出关系以并到易处理的训练方法可能很难。创建一个多层神经网络的最好方法是使每个网络只有一或两层，然后将它们作为构成或功能模块连接起来。

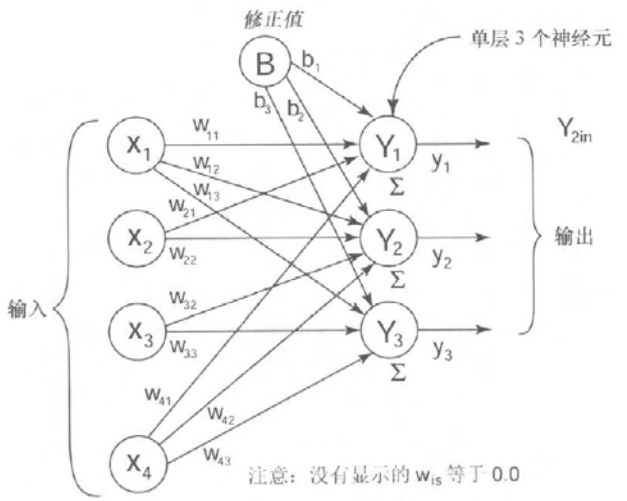


图 3.9.3 一个四输入，三神经元的单层神经网络

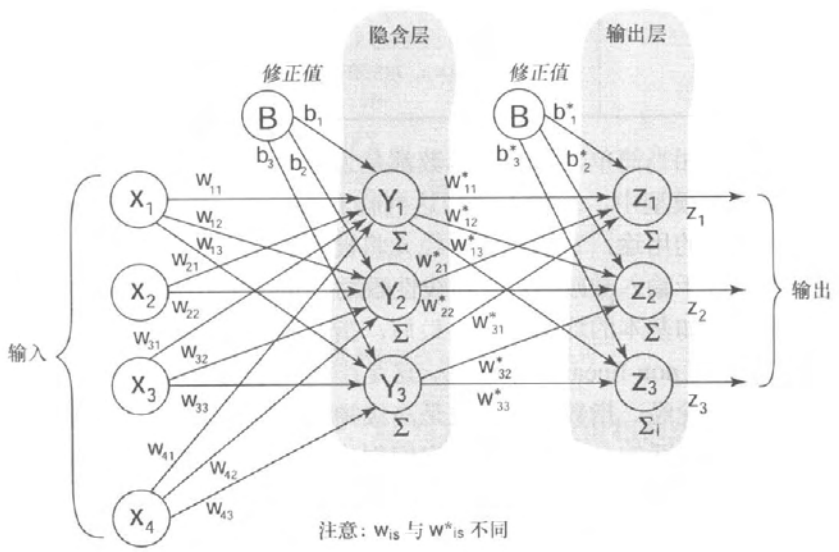


图 3.9.4 两层神经网络

好了，现在让我们讨论一下有关时态（temporal）或时间相关的话题。众所周知，我们的大脑与数字计算机相比是十分慢的。事实上，人类的大脑有毫秒级的时间周期，而数字计算机的时间周期是十亿分之一秒级，并且很快还会变成低于十亿分之一秒级。它代表了信号从神经元传递到神经元所花的时间。人工神经元也在这种意义上模拟了事实：一层层地执行计算并且把结果顺序传输下去。这个模型有助于更好地模拟生物系统（如人类）中信号传输中的时间延迟。

我们差不多快介绍完预备知识了，让我们再讨论几个更高级的概念和术语作为结束。你可能会被问到“神经网络是做什么的？”，这是一个好问题，而且是一个很难确切回答的问题。问题其实应当是“你想让神经网络做什么？”神经网络主要是一种映射工具，帮助我们将一个空间映射到另一个空间。本质上说，他们是一种存储器类型。像任何存储器一样，我们可以用一些类似的术语来描述它们。短期记忆（short-term memory，STM）和长期记忆（long-term memory，LTM）这两种记忆神经网络都有。STM 是一个神经网络记住刚学习过的东西的能力，而 LTM 是一个神经网络在它新的学习过程中回忆起一段时间以前学过的东西的能力。

这使我们遇到了“适应性（plasticity）”的概念，或者换句话说，一个神经网络是如何处理新信息或训练的。一个神经网络能不能学习更多的信息并且仍然正确地回忆起先前存储的信息？如果可以，神经网络会因为保存了这么多信息以致数据开始重迭或有了公共的交集而变得不稳定吗？这一领域被称为稳定性（stability）。我们想使一个神经网络起码要有一个好的 LTM、有一个好的 STM、是可塑的（在多数情形），以及显示出稳定性。当然，有一些神经网络并没有模拟存储器。它们更多是为了功能映射，所以这些概念并不适用。

现在我们知道有关存储器的概念了，下面让我们讨论几个帮助我们度量和理解这些特性的数学要素吧。

神经网络的主要应用之一是作为存储器，它能够通过处理不完全或“噪声”输入产生响应。响应可以是输入本身（自体联想，autoassociation）或另一种与输入完全不同的输出（异体联想，heteroassociation）。此外，映射可以从一个 n 维空间到一个 m 维空间且非线性导入。底线是我们想以某种方式在神经网络中存储信息以使输入（精确的输入及噪声）能被并行地处理。这意味一个神经网络是一种超维（hyperdimensional）存储器单元，因为它能将一个输入 n 元组和一个输出 m 元组联系起来，这里 m 可以等于 n，但是不一定非要如此。

神经网络所做的工作实质上是将一个 n 维空间划分为区域，它惟一地将输入映射到输出，或将输入分为不同的类，就像一个漏斗。现在，当输入数据集（我们称之为 S）中输入值（矢量）的数目增加时，逻辑上说神经网络分离信息将更为困难。当一个神经网络中充满了信息时，由于输入空间不能再将所有被划分的事物限定在一个有限的维数，输入值将被重叠恢复。这种重叠导致了“串音（crosstalk）”，即一些输入实际上并没有产生应有的区别。串音可能是想得到的，或者可能不想得到。虽然不是所有情形都关心这个问题，但它是联想存储器神经网络所涉及的，所以为了说明这个概念，让我们假定现在试图将 n 元输入矢量与一些输出集合联系起来。输出集合对于网络固有机能的影响不如输入集合那么大。

如果一个输入集合 S 是二进制的，我们可以考察形如 1101010...10110 的序列。让我们假定每个输入比特矢量只有 3 个二进制位，因此整个输入空间就是由下面的 8 个矢量构成的：

$\mathbf{v}_0 = (0,0,0)$，$\mathbf{v}_1 = (0,0,1)$，$\mathbf{v}_2 = (0,1,0)$，$\mathbf{v}_3 = (0,1,1)$，$\mathbf{v}_4 = (1,0,0)$，$\mathbf{v}_5 = (1,0,1)$，$\mathbf{v}_6 = (1,1,0)$，$\mathbf{v}_7 = (1,1,1)$

为了更精确，这个矢量集合的基可以表示为：

$$\mathbf{v} = (1,0,0) * \mathbf{b}_2 + (0,1,0) * \mathbf{b}_1 + (0,0,1) * \mathbf{b}_0$$

这里 $\mathbf{b}_i$ 可以取值为 0 或 1。

例如，如果我们令 $\mathbf{b}_2$=1，$\mathbf{b}_1$=0，以及 $\mathbf{b}_0$=1，则可以得到

$\mathbf{v} = (1,0,0) * 1 + (0,1,0) * 0 + (0,0,1) * 1 = (1,0,0) + (0,0,0) + (0,0,1) = (1,0,1)$

这就是可能输入集中的 $\mathbf{v}_5$。

一个基是一个特殊的矢量总和，它描述了一个空间中的矢量集合。所以 v 描述了空间中的所有的矢量。简而言之，输入集中的正交（orthogonal）矢量越多，它们在一个神经网络中的分布越好，他们被恢复得就越好。正交是指独立的矢量，或者换句话，如果两个矢量是正交的，它们的点积就是 0，它们彼此到另一个矢量的投影是 0，而且不能相互表示出来。

集合 v 中有许多正交矢量，但是它们可以分成不同的子集——例如，$\mathbf{v}_0$ 与所有的矢量都正交，所以我们总是包含它。但是如果我们在集合 S 中包括了 $\mathbf{v}_1$，其他适合的并保持正交的矢量就只能是 $\mathbf{v}_2$ 和 $\mathbf{v}_4$，或表示为集合：

$\mathbf{v}_0 = (0,0,0)$，$\mathbf{v}_1 = (0,0,1)$，$\mathbf{v}_2 = (0,1,0)$，$\mathbf{v}_4 = (1,0,0)$

为什么呢？因为对于 **i**，**j** 的所有从 0 到 3 的取值，$\mathbf{v}_i \bullet \mathbf{v}_j$ 都等于 0。换句话说，所有矢量对的点积都为 0，所以它们都是正交的。所以，这个集合作为一个神经网络的输入矢量时作用得非常好。然而，集合：

$\mathbf{v}_6 = (1,1,0)$，$\mathbf{v}_7 = (1,1,1)$

可能作为输入作用得不好，因为 $\mathbf{v}_6 \bullet \mathbf{v}_7$ 是非零的，或者在一个二进制系统中，它是 1。下一个问题是“我们能够度量这种正交性吗？”回答是肯定的。在二进制矢量系统中，有一个量度称为汉明距离（hamming distance），它用来度量二进制位矢量之间的 n 维距离。汉明距离就是简单的两个矢量间不同的位的数目。例如，矢量：

$\mathbf{v}_0 = (0,0,0)$，$\mathbf{v}_1 = (0,0,1)$

有汉明距离 1，而矢量：

$\mathbf{v}_2 = (0,1,0)$, $\mathbf{v}_4 = (1,0,0)$

的汉明距离为 2。

我们可以将汉明距离在二进制位矢量系统中用于度量正交性，它有助于我们确定输入矢量是否将有许多重叠。确定一般的输入矢量的正交性要难一些，但原理是一样的。

到这儿对于概念和术语的解释就要结束了，让我们再深入看一些你能用在自己的游戏 AI 中的神经网络的实例。我们将探讨用于执行逻辑功能、对输入进行分类以及将输入与输出联系起来的神经网络。

3.9.4 纯逻辑，Mr. Spock

最早的人工神经网络是由 McCulloch 和 Pitts 在 1943 年提出的。这些神经网络由许多神经元组成并典型地用于计算逻辑函数如 AND、OR、XOR，以及它们的联合。图 3.9.5 是一个基本的有两个输入的 McCulloch-Pitts 神经元的表示。如果你是一个电子工程师，你将立即看到 McCulloch-Pitts 神经元与晶体管或 MOSFET 非常相似。在任何情形下，McCulloch-Pitts 神经元都没有修正值并且有简单的激励函数 $f_{mp}(X)$，如公式 3.9.5 所示。

$$f_{mp}(x) = 1, \quad \text{if } x \geq \theta$$
$$0, \quad \text{if } x < \theta \qquad (3.9.5)$$

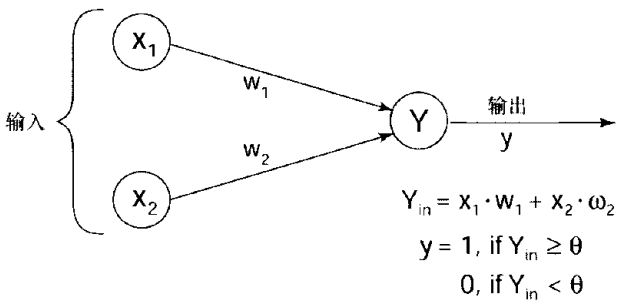


图 3.9.5 McCulloch-Pitts 神经元

MP（McCulloch-Pitts）神经元通过对输入 $\mathbf{X}_i$ 和权值 $\mathbf{w}_i$ 的乘积求和并将结果 $\mathbf{Y}_a$ 应用于激励函数 $f_{mp}(X)$来起作用。McCulloch-Pitts 的早期研究主要关注于用神经元模型构造复杂的逻辑线路。此外，神经网络的一个规则是从一个神经元传递一个信号要用一个时间步。这有助于更真实地模拟生物上的自然神经元。

让我们来看一些完成了基本的逻辑操作和 MP 神经网络的例子。逻辑 AND 操作有以下的真值表：

$\mathbf{X}_1$	$\mathbf{X}_2$	输出
0	0	0
0	1	0
1	0	0
1	1	1

我们可以用一个二输入 MP 神经网络来模拟它，权值设为 $\mathbf{w}_1=1$，$\mathbf{w}_2=1$，且 $\theta=2$。这个神经网络如图 3.9.6a 所示。正如你看到的，所有的输入连接都能正确工作。例如，如果我们输入 $\mathbf{X}_1=0$，$\mathbf{Y}_1=1$，激励值将为：

$$\mathbf{X}_1*\mathbf{w}_1 + \mathbf{X}_2*\mathbf{w}_2 = (1)*(1) + (0)*(1) = 1$$

如果我们给激励函数 $f_{mp}(X)$输入 1，结果是 0，这是正确的。作为另一个例子，我们试一下输入 $\mathbf{X}_1=1$，$\mathbf{X}_2=1$，激励值将是：

$$\mathbf{X}_1*\mathbf{w}_1 + \mathbf{X}_2*\mathbf{w}_2 = (1)*(1) + (1)*(1) = 2$$

如果我们将 2 输入给激励函数 $f_{mp}(X)$，那么结果是 1，这是正确的。其他的情形也正确。OR 操作与之类似，但是阈值变为 1 而不是 AND 中的 2。你可以试着运行一下真值表来检验一下结果。

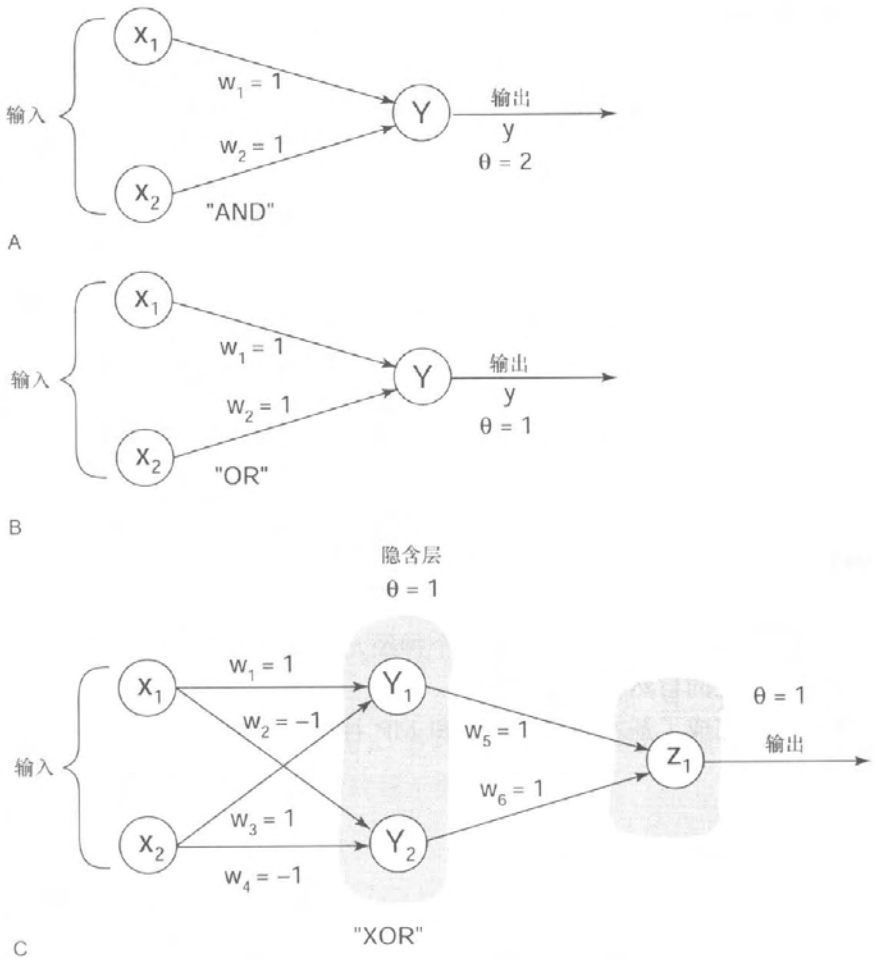


图 3.9.6 用 McCulloch-Pitts 网络实现的基本逻辑操作

XOR 操作就有所不同了，因为它事实上有两层，即预处理的结果在输出层中被进一步处理。这是需要用多于一层的神经网络来解决某一问题的理由的一个好实例。在神经网络中 XOR 是一个普通的问题，常用于测试一个神经网络的执行。在任何情形之下，在一个单独层 XOR 都不是线性可分离的，它必须被分解为更小的问题，然后再将结果合在一起。让我们以 XOR 作为最后一个 MP 神经网络的例子。XOR 的真值表如下：

X1	X2	输出
0	0	0
0	1	1
1	0	1
1	1	0

只有当输入不同的时候 XOR 才为真。这是一个问题，因为两个输入映射到了同一个输出。XOR 不是线性可分离的，如图 3.9.7 所示。正如你看到的，没有办法能用一条直线分离

开正确的响应。但重要的是能用两条直线来分离正确的响应，这正是那两层的功能。第 1 层预处理或解决问题的一部分，剩下的一层来最终完成处理工作。参考图 3.9.6c，我们看到权值是 $w_1 = 1$，$w_2 = -1$，$w_3 = 1$，$w_4 = -1$，$w_5 = 1$，$w_6 = 1$。网络是这样工作的：每一层并行计算 X_1 和 X_2 是否是相反的，（0，1）或（1，0）被馈给第 2 层，该层把它们合起来并当为真时激活。本质上说，我们创建了逻辑函数：

$z = ((X_1 \text{ AND NOT } X_2) \text{ OR } (\text{NOT } X_1 \text{ AND } X_2))$

如果你想对基本的 McCulloch-Pitts 的神经元做一下实验，CD 上的程序清单 3.9.1 是一个完整的二输入、单神经元模拟器，你能用它进行实验。

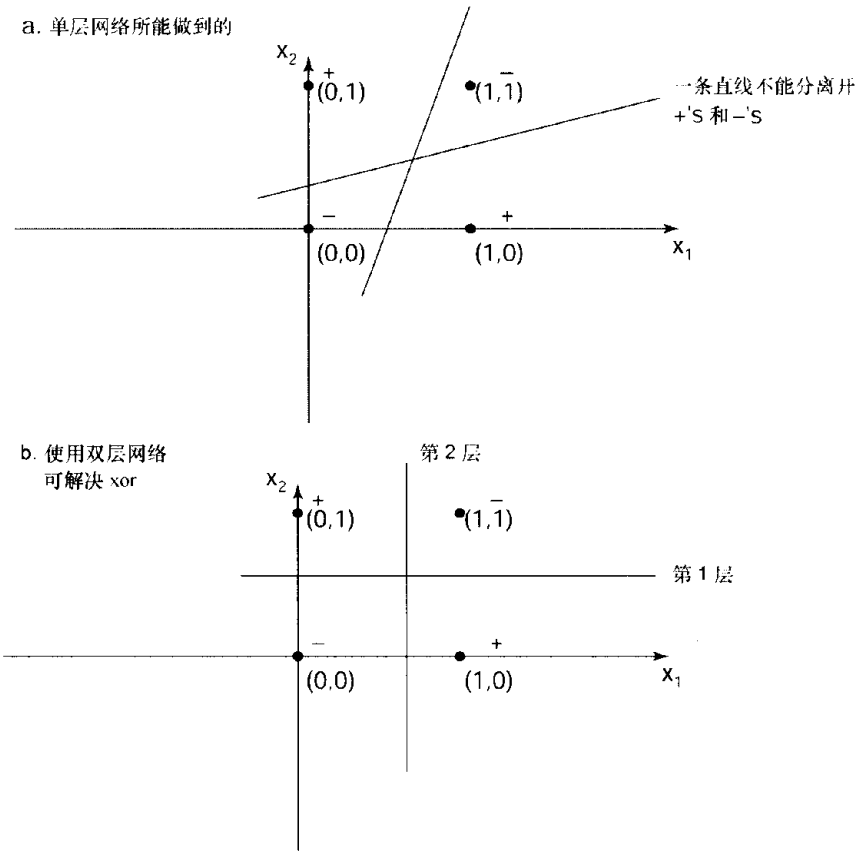


图 3.9.7 使用 XOR 函数来说明线性分离

关于 McCulloch 和 Pitts 所提出的基本构造部件就讨论到这里。现在让我们转到更现代的例如那些用于将输入矢量分类的神经网络。

3.9.5 分类与“图像”识别

在此我们将准备开始看一下真正的神经网络（对它们有一些束缚）。为了继续下面关于 Hebbian 和 Hopfield 神经网络的讨论，让我们先分析一个一般的神经网络的结构，它说明了许多概念，如线性可分离性，两级表示，以及神经网络的有存储器的仿真。

让我们从查看图 3.9.8 开始，它表明了我们使用的基本的神经网络模型。如你所见，这是一个有 3 个输入的单节点网络，包括修正值和一个单输出。我们将看一下是否能用这个网络来解决前面用 McCulloch-Pitts 神经元已经解决过的逻辑 AND 函数问题。

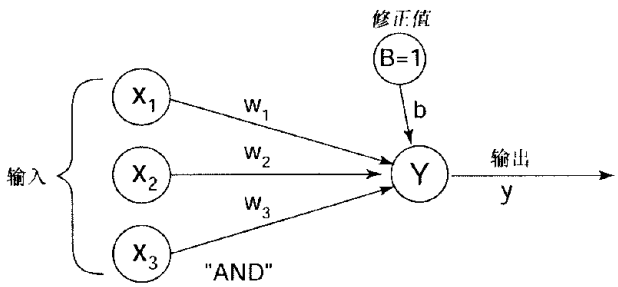


图 3.9.8 用于讨论的基本神经网络模型

让我们首先采用两极表示。所有的 0 都用−1 来代替，所有的 1 不变。采用两极表示的输入和输出的逻辑 AND 函数的真值表如下：

X_1	X_2	输出
−1	−1	−1
−1	1	−1
1	−1	−1
1	1	1

公式 3.9.6 给出了我们将要使用的激励函数 $f_c(x)$。

$$f_c(x)=\begin{cases}1, & \text{if } x \geqslant \theta \\ -1, & \text{if } x < \theta\end{cases} \tag{3.9.6}$$

注意这个函数是具有两极形式输出的阶跃函数。在继续讨论之前，让我先告诉你一个事实：修正值和阈值最终所起的作用相同，这给了我们神经元中的另一种自由度，使神经元以非它们不可的方式来响应。我们将对这个概念进行简要说明。

图 3.9.8 中的单神经元网络能为我们实现一个分类。它将告诉我们输入是否属于同一个类。例如，“这是一棵树还是不是一棵树？”或者在我们的例子中，这个输入（它正好是一个 AND 逻辑）是在+1 类还是−1 类？这是多数神经网络的基础和我要对线性可分性反复说明的理由。我们需要提出一种映射了我们的输入和输出的空间的线性分割，以使一个空间的固定描述将其分离。这样我们需要提出能够达到这个目的的正确的权值和修正值。但是如何达到这个目标呢？仅仅反复进行试验？还是有一个方法论？答案是训练一个神经网络有许多方法。这些训练方法在不同的数学前提下工作且能被证明，不过现在，我们仅仅直接使用一些

可以得到期望结束的方法。这些训练将我们带到了如下的学习算法和更为复杂的网络。

当各种各样的输入及特定的激励函数 $f_c(x)$馈入给神经网络时，我们来试着找一下能给出正确结果的权值 w_i和修正值 **b**。让我们写下神经元的激励值并看一下是否能推断出一些权值与输入之间的对我们有帮助的关系。给定分别具有权值 w_1 和 w_2 的输入 $\mathbf{X}_1$ 和 $\mathbf{X}_2$，以及 **B**=1 和修正值 **b**，我们有下面的方程式：

$$X_1 * w_1 + X_2 * w_2 + B * b = \theta \qquad (3.9.7)$$

由于 B 总是等于 1.0，该方程简化为：

$$X_1 * w_1 + X_2 * w_2 + b = \theta$$

$$\cdot$$

$$\cdot$$

$$X_2 = -X_1 * w_1 / w_2 + (\theta - b)/w_2 \quad （解出 X_2）$$

它是什么？是一条直线！而且如果左边 $X_1 * w_1 + X_2 * w_2 + b$ 大于或者等于 θ，神经元将会输出 1，否则神经元将会输出－1。所以，该直线是一个判定边界，图 3.9.9 a 说明了这个概念。在这个图中，你可以看到该直线的斜率是 $-w_1/w_2$，并且 X_2 上的截距为 $(\theta - b)/w_2$。现在你能明白为什么我们可以去掉 θ 了吗？它是一个常数小数，而且总是能通过缩放 b 来处理任何误差，所以我们假设 $\theta = 0$。最后得到的公式为：

$$X_2 = -X_1 * w_1 / w_2 - b/w_2$$

我们想得到的是权值 w_1 和 w_2，以及修正值 b，以使它分离我们的输出或将它们划分到单一的没有重叠的部分，这是线性可分离性的关键。图 3.9.9b 显示了许多满足要求的判定边界，我们可以从中选择任何一个。让我们选择最简单的值吧，它们是：

$$w_1 = w_2 = 1$$

$$b = -1$$

由这些值，我们的判定边界变为：

$$X_2 = -X_1 * w_1 / w_2 - b/w_2 \rightarrow X_2 = -1 * X_1 + 1$$

它的斜率是－1 而且 $\mathbf{X}_2$ 上的截距是 1。如果我们将逻辑 AND 的输入矢量代入到这个方程，并且采用激励函数 $f_c(x)$，就可以得到正确的输出。例如，如果 $X_2 + X_1 - 1 > 0$，那么激活这个神经元；否则输出是－1。让我们用 AND 输入试一下看能得到什么：

输入	$\mathbf{X}_1$	$\mathbf{X}_2$	**输出($\mathbf{X}_2 + \mathbf{X}_1 - 1$)**
－1	－1	(－1)+(－1) －1 = 3 < 0	不激活，输出 －1
－1	1	(－1) + (1) －1 = －1< 0	不激活，输出－1
1	－1	(1) + (－1) －1 = －2 < 0	不激活，输出－1
1	1	(1) + (1)－1 = 1 > 0	激活，输出 1

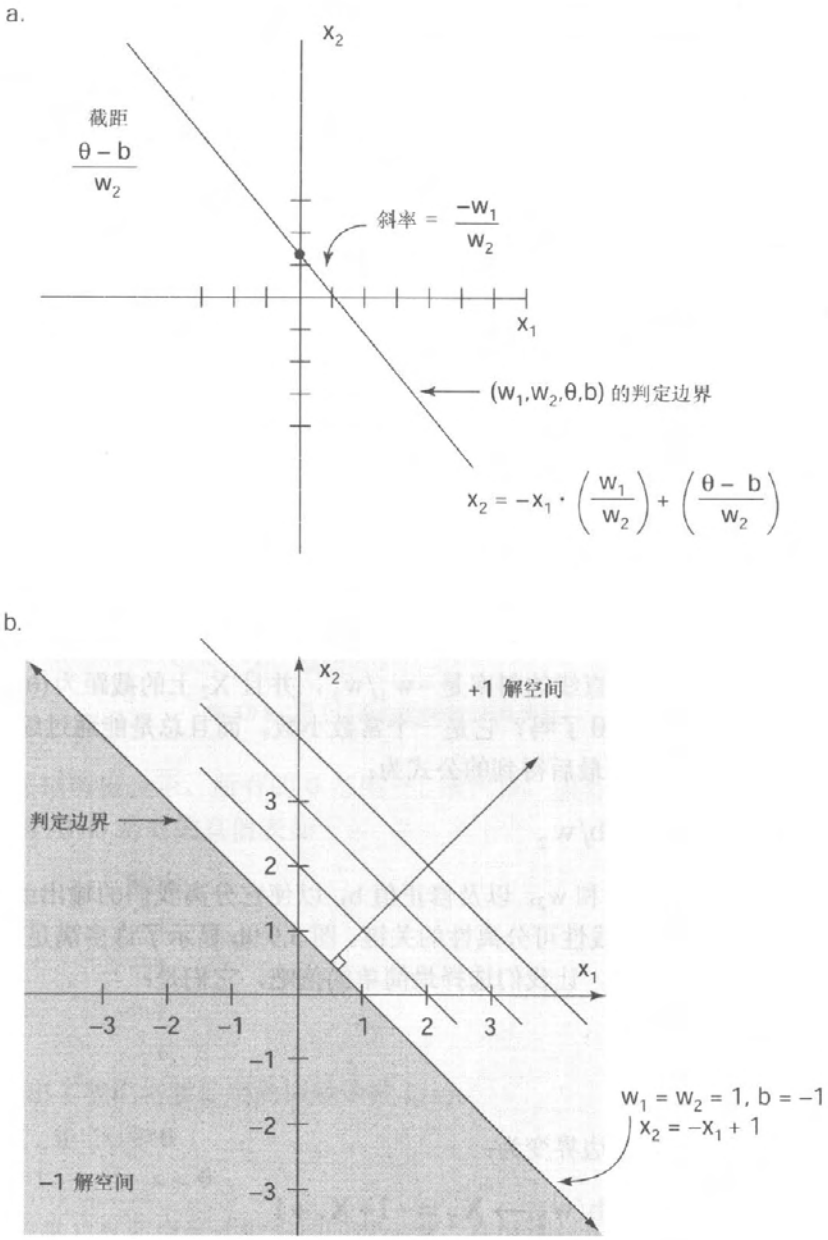


图 3.9.9 由权值、修正值和θ产生的数学上的判定边界

如你所见，具有正确的权值和修正值的神经网络极好地解决了问题。此外，还有一整个族的权值也能做到这一点（在与它垂直的方向上调整判定边界）。然而，在此有一点极为重要。若没有修正值或是阈值，仅有通过原点的直线也是可能的，因为 $\mathbf{X}_2$ 上的截距必须是 0。这是非常重要的，也是使用一个修正值或阈值的基础，所以这个例子是一个重要的证明，因为它彰显了这个事实。

所以，我们是不是更接近于发现如何用算法找到权值了？是的，我们现在有了一个几何学的模拟，它是找到一个算法的开端。

3.9.6 Hebbian 的 Ebb

现在我们准备讨论第一个学习算法和它在神经网络上的应用了。由 Donald Hebb 发明的算法是最简单的学习算法之一，它是以一种基于使用输入矢量修改权值的方式，使权值能够产生输入和输出的最好可能的线性分离。可惜算法只能说是差强人意。实际上，它对于正交的输入作用得非常好；但是对非正交的输入，算法就崩溃了。虽然算法没有为所有的输入产生正确的权值，它仍是多数学习算法的基础，所以我们从这儿开始研究。

在我们看到这个算法之前，要记住它是为一个单神经元、单层神经网络设计的。当然你可以在层内放置许多神经元，但是它们都是并行工作的，也能并行学习。你开始看到神经网络所展示出的大规模的并行化了吗？一个多神经元网络不再使用一个单权值矢量，而是使用一个权值矩阵。算法很简单，它的运行步骤如下：

假设：

- 输入矢量是两极形式 **I** = (−1，1，…−1，1)的且包含 **k** 个元素。
- 有 **n** 个输入矢量，而且我们将集合称作 **I**，且第 **j** 个元素为 $\mathbf{I}_j$。
- 输出被称为 $\mathbf{y}_j$ 而且有 **k** 个，每一个 $\mathbf{I}_j$ 都有一个输出。
- 权值 $\mathbf{w}_1$−$\mathbf{w}_k$ 包含在一个一维矢量 $\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, \ldots \mathbf{w}_k)$中。

1．将所有的权值初始化为 0，且令它们包含在一个有 n 个分量的矢量 w 中。将修正值 b 也初始化为 0。

2．For j = 1 to n，do:

$b = b + y_j$　　（此处 y 是期望的输出）

$w = w + Ij * y_j$　　（记住，这是一个矢量运算）

end do

这个算法勉强称得上是一个“累加器”，基于输入和输出的变化移动判定边界。惟一的问题是它有时（或根本）不能把边界移动得足够快，于是“学习”就不会发生。

那么我们如何来使用 Hebbian 进行学习呢？答案是与前述方法相同，只是现在我们有了一个算法的方法能对网络进行训练，于是我们称这个网络为 Hebb 或者 Hebbian 网络(Hebbian net)。

让我们取可信赖的逻辑 AND 函数作为一个例子，并看看算法是否能够找到正确的权值和修正值来解决这个问题。下面的求和等价于运行该算法：

$\mathbf{w} = [\mathbf{I}_1*\mathbf{y}_1] + [\mathbf{I}_2*\mathbf{y}_2] + [\mathbf{I}_3*\mathbf{y}_3] + [\mathbf{I}_4*\mathbf{y}_4] = [(-1,-1)*(-1)] + [(-1,1)*(-1)] + [(1,-1)*(-1)] + [(1,1)*(1)] = (2,2)$

$\mathbf{b} = \mathbf{y}_1 + \mathbf{y}_2 + \mathbf{y}_3 + \mathbf{y}_4 = (-1) + (-1) + (-1) + (1) = -2$

因此，$\mathbf{w}_1$=2，$\mathbf{w}_2$=2，且 **b**=−2。在前一节中我们从几何上导出了简单形式的与之成比例的值 $\mathbf{w}_1$=1，$\mathbf{w}_2$=1，**b**=−1。太棒了！有了这个简单的学习算法，我们可以训练一个神经网络（由单个的神经元组成）来对输入集合进行响应，或将输入划分为真或假，1 或−1。现在如果我们将神经元排列在一起构造一个神经元的网络，而不是简单地将输入划分为 On 或 Off，就可以和输入的模式联系起来。这是下一个神经网络（Hopfield 网络）结构的基础。还有一

件事：用于 Hebb 的激励函数是一个阶跃函数，有阈值 0 及两极输出 1 和－1。

为了对 Hebbian 学习和如何实现一个真正的 Hebb 网络有一个感性认识，CD 上的程序清单 3.9.2 包含了一个完整的 Hebbian 神经网络模拟器（Hebbian Neural Net Simulator）。这个程序是自解释的，但是有两个有趣的特性：你可以从三个激励函数中选择一个，而且可以输入任何类型的数据。通常，我们将选定阶跃激励函数，且输入/输出将是二进制的或两极的。然而，从发现的角度出发，也许你利用这些增加的自由度会发现一些有意思的东西。尽管我建议你从阶跃函数和两极形式的输入和输出开始。

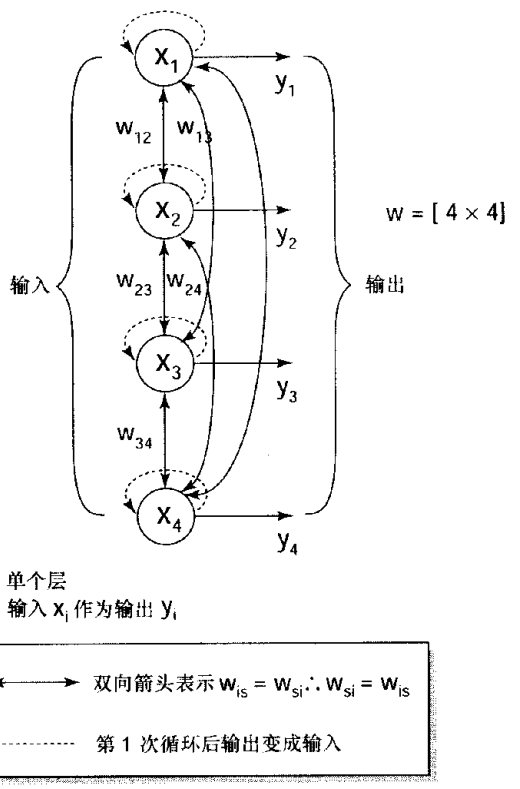


图 3.9.10 一个 4 节点自体联想 Hopfield 神经网络

3.9.7 运行 Hopfield

John Hopfield 是一个物理学家，他喜欢摆弄神经网络（这对我们有好处）。他提出了一种简单（至少在结构上）但有效的称为 Hopfield 网络的神经网络，被用于自体联想。你可以输入一个矢量 x 并取回这个 x（希望如此！）。图 3.9.10 展示了一个 Hopfield 网络。它是一个单层网络，神经元的数目等于输入的 $\mathbf{X}_i$ 的数目。这个网络是全接连的，也就是每一个神经元与其他每一个神经元相连接，输入节点也是输出节点。这个结构可能会让你觉得不可思议，因为有反馈（feedback）。反馈是 Hopfield 网络的关键特征之一，也是收敛到正确结果的基础。

Hopfield 网络是一种迭代的自体联想存储器。这意味着它能花一个或多个循环返回正确的结果。让我解释一下：Hopfield 获取一个输入然后将它反馈回来，最后得到的输出可能是

或可能不是期望的输入。在输入矢量返回之前这个反馈循环可能发生许多次。因此，一个 Hopfield 网络的操作序列如下：首先，我们基于想要自体联想的输入矢量确定权值，然后输入一个矢量看能从激励函数得到什么。如果结果与我们最初输入的相同，我们就完成任务了，否则，我们取出结果矢量并将其再反馈回网络。

现在让我们看看用于 Hopfield 网络的权值矩阵和学习算法。

Hopfield 网络的学习算法基于 Hebbian 规则，简单地对乘积进行求和。然而，由于 Hopfield 网络有许多输入神经元，权值不再是一个单一的数组或矢量，而是极其紧凑地存储在一个单一矩阵中的矢量集。这样一个 Hopfield 网络的权值矩阵 W 是基于下面的公式创建的：

假设：

- 输入矢量是两极形式 **I** = （−1,1,…−1, 1）的，且包含有 k 个元素。
- 有 m 个输入矢量，并且我们称其集合为 I，第 j 个元素是 I_j。
- 输出称为 $\mathbf{y}_j$ 且有 k 个，每一个 $\mathbf{I}_j$ 都有一个输出。
- 权值矩阵 W 是方阵并且是 k×k 维的，因为有 k 个输入。

$$\mathbf{W}_{(k\times k)} = \sum_{i=1}^{k} \mathbf{I}_i^t \times \mathbf{I}_i \tag{3.9.8}$$

注意：每个外积都是 k×k 维的，因为我们是将一个列矢量与一个行矢量相乘。

对于所有的 i，$\mathbf{W}_{ii} = 0$

注意这里没有修正值项而且 W 的主对角线元素一定全为零。权值矩阵不过是对所有的 i 从 1 到 n，由转置矩阵 I_i^t 与 I 相乘产生的矩阵的和。除了用输入与其自身相乘代替输入与输出相乘，对于一个单神经元这差不多与 Hebbian 算法一样，它与自体联想情形的输出是相等价的。最后，激励函数 $f_n(x)$如下：

$$f_h(x) = \begin{cases} 1, & \text{if } x \geqslant 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{3.9.9}$$

$f_h(x)$是一个有着二进制输出的阶跃函数。这意味着输入也必须是二进制的，但是我们是否说过输入是两极的？噢，它们是，也不是。当权值矩阵产生以后，我们可以将所有输入转换到两极表示，但是对普通的操作使用输入的二进制形式，而且 Hopfield 网络也将是二进制的。这个转换不是必要的，但它使关于网络的讨论变得更为容易。

总之，让我们来看一个例子。假设我们想创建一个 4 节点的 Hopfield 网络并且想让它恢复下面这些矢量：

$\mathbf{I}_1 = (0,0,1,0)$，　$\mathbf{I}_2 = (1,0,0,0)$，$\mathbf{I}_3 = (0,1,0,1)$

注意：它们是正交的

转换到两极表示的(*)，我们有：

$\mathbf{I}_1^* = (-1,-1,1,-1)$，　$\mathbf{I}_2^* = (1,-1,-1,-1)$，　$\mathbf{I}_3^* = (-1,1,-1,1)$

现在我们需要计算 $\mathbf{W}_1$，$\mathbf{W}_2$，$\mathbf{W}_3$，这里 $\mathbf{W}_i$ 是每个输入的转置矩阵与其自身的乘积：

$$\mathbf{W}_1 = [\mathbf{I}_1^{*t} \times \mathbf{I}_1^*] = (-1,-1,1,-1)^t \times (-1,-1,1,-1) =$$

$$\begin{matrix} 1 & 1 & -1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & 1 & -1 \\ 1 & 1 & -1 & 1 \end{matrix}$$

$$W_2 = [I_2^{*t} \times I_2^{*}] = (1,-1,-1,-1)^t \times (1,-1,-1,-1) =$$

$$\begin{matrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{matrix}$$

$$W_3 = [I_3^{*t} \times I_3^{*}] = (-1,1,-1,1)^t \times (-1,1,-1,1) =$$

$$\begin{matrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{matrix}$$

然后我们相加 $\mathbf{W}_1 + \mathbf{W}_2 + \mathbf{W}_3$，得到：

$$W_{(1+2+3)} =$$

$$\begin{matrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & 3 \\ -1 & -1 & 3 & -1 \\ -1 & 3 & -1 & 3 \end{matrix}$$

令主对角线元素为零，可以得到最终的权值矩阵：

$$W =$$

$$\begin{matrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & 3 \\ -1 & -1 & 0 & -1 \\ -1 & 3 & -1 & 0 \end{matrix}$$

好，现在我们可以开始看到最酷的事了。让我们输入原始矢量看一下结果。为了做到这一点，可以简单地用该矩阵去乘输入，然后用激励函数来处理每一个输出值。这里是结果：

$I_1 \times W = (-1,-1,0,-1)$ 且 $f_h((-1,-1,0,-1)) = (0,0,1,0)$

$I_2 \times W = (0,-1,-1,-1)$ 且 $f_h((0,-1,-1,-1)) = (1,0,0,0)$

$I_3 \times W = (-2,3,-2,3)$ 且 $f_h((-2,3,-2,3)) = (0,1,0,1)$

输入被完美地恢复了，而且它们理应如此，因为他们都是正交的。作为最后一个例子，让我们假设输入（视觉的，听觉的，等等）是一个很小的噪声，仅有一个错误在里面。让我们取 $\mathbf{I}_3 = (0,1,0,1)$并加上一些噪声给 $\mathbf{I}_3$，得到 $\mathbf{I}_3^{noise} = (0,1,1,1)$。现在让我们看看如果把这个噪

声矢量输入到 Hopfield 网络会发生什么：

$$I_3^{noise} \times W = (-3,2,-2,2) 且 f_h((-3,2,-2,2)) = (0,1,0,1)$$

太神奇了，原始矢量被恢复了！这太棒了！所以我们可以有一个山看起来像树（橡树，垂柳，云杉，红杉）的比特模式填充的存储器。如果我们输入另外一棵相似的树，又如说垂柳（还没有存入网内），我们的网将（很有希望地）输出一个垂柳，表明那是它“认为”看起来像的模式。

这是联想记忆的长处：我们并不需要教给网络每一个可能的输入。我们只是必须充分地训练它来给它一个好办法。然后“接近的”输入通常将收敛于一个实际被训练过的输入。这是图像或语言识别系统的基础。

为了完成我们关于神经网络的研究，我已经为你提供了一个 Hopfield 自联想模拟器，它允许你创建最多有 16 个神经元的网络。Hebb 网络也类似，但是你必须使用一个阶跃激励函数，而且当训练时你的输入样本必须是两极的而联想（运行）时必须是二进制的。CD 上的程序清单 3.9.3 包含了模拟器的代码。

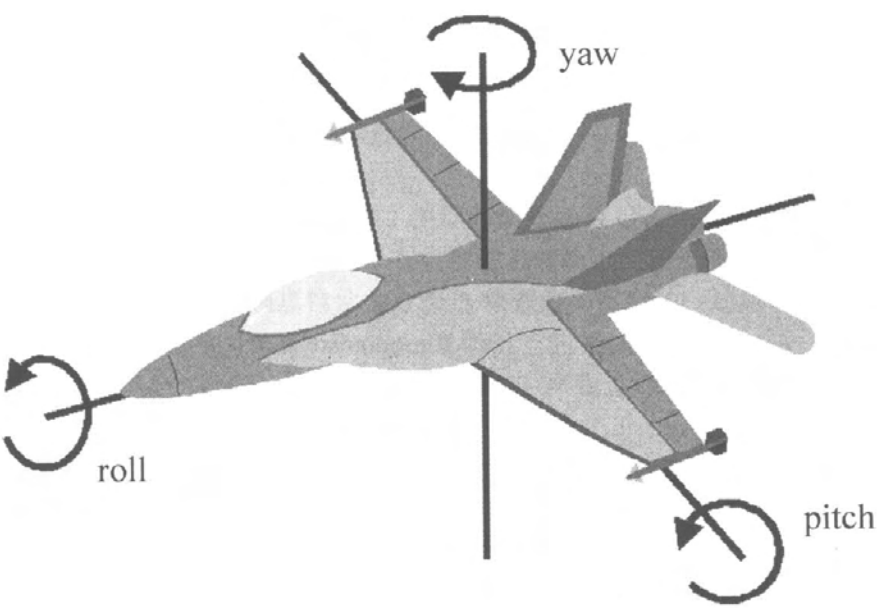
3.9.8 结论

我希望本文已经给了你一个关于神经网络是什么以及如何创建一些有效的计算机程序来模拟它们的思想。我们讨论了基本的术语和概念，一些数学基础，并且以一些更普遍的神经网络结构作为结束。

然而，仍然有太多的关于神经网络的东西要学习：感知机（Perceptrons）、模糊联想记忆（fuzzy associative memories，FAMs）、双向联想记忆（bidirectional associative memories，BAMs）、Kohonen maps、学习机（Adalines）、Madalines、反向传播网络（back-propagation networks）、自适应共振理论网络（adaptive resonance theory networks）和盒中的大脑状态（brain state in a box）等等。好了，就到这儿吧，我的神经网络想玩 PlayStation 2 了！

第 4 章

多边形技术



4.0 为 OpenGL 优化顶点提交

Herbert Marselas

在OpenGL 中有大量用于提交和渲染顶点的函数，范围从简单的即时模式函数到更为复杂的多顶点函数，以及厂家指定的扩展。然而性能可能随所用的函数不同而有很大差异。

4.0.1 即时模式

即时模式函数（例如 glVertex*，glColor*，glNormal*）时常用于构造并快速渲染。这些函数很容易使用，因为每个函数都对应于顶点的一个不同的组分的提交：位置、颜色、法线、纹理坐标，等等。然而，即时模式函数如此易于使用（一个组分一个组分地提交）也是导致它们成为性能最低的函数的原因。

这是由于两个因素。首先，渲染一个单独的顶点需要调用几个函数。其次，每个函数必须被进入，随后完成很少量的工作，然后退出。进入和退出一个函数所必需的时间叫做函数开销（function overhead）。不管函数所做的工作量大小，这项开销都要产生，而且它代表了使用该函数所需要的固定时间。如果函数做许多工作，那么这项开销与完成的工作相比是比较低的。如果函数做的工作较少，或函数被调用很多次，这项开销就会快速增加。

图 4.0.1 显示了用即时模式函数 glTexture3f、glColor4f 和 glVertex3f 提交 300 个着色的、有纹理的且转换过的顶点所需 CPU 周期的时间总量。这 300 个顶点包含 100 个 3 像素的、离散的、均质的直角三角形。这些计时是在 450MHzPentium II 微机上的 Microsoft Windows 98 下使用一个流行的 OpenGL 用户图形卡得到的。产生这些数据的源代码作为一个 Microsoft Visual C++ 6 项目见所附的 CD。

可以使用较小的转换过的三角形去掉在转换（它们已经被转换）、光照（它们已预采光）和光栅化（它们非常小）操作中所花费的时间。这保证我们能有效地测量进入每个函数、储存数据和返回所必需的时间。提交和渲染所有的 300 个顶点总共需要大约 163 154 个 CPU 周期。

提交单个顶点的位置、颜色和纹理坐标平均需要花费大约 544 CPU 周期。然而，在执行过程中会有一些峰值。这在 glVertex 函数中能看到，在一帧中第一次调用该函数要花费大约 38238 个 CPU 周期（可能用于分配数据），尽管证实这一点需要进行更详细的驱动程序分析。以后调用平均值为

每次 308 个 CPU 周期，峰值为每次 1500 个 CPU 周期。完整的分析包含在所附 CD 上一个 Microsoft Excel 97 电子表格中。

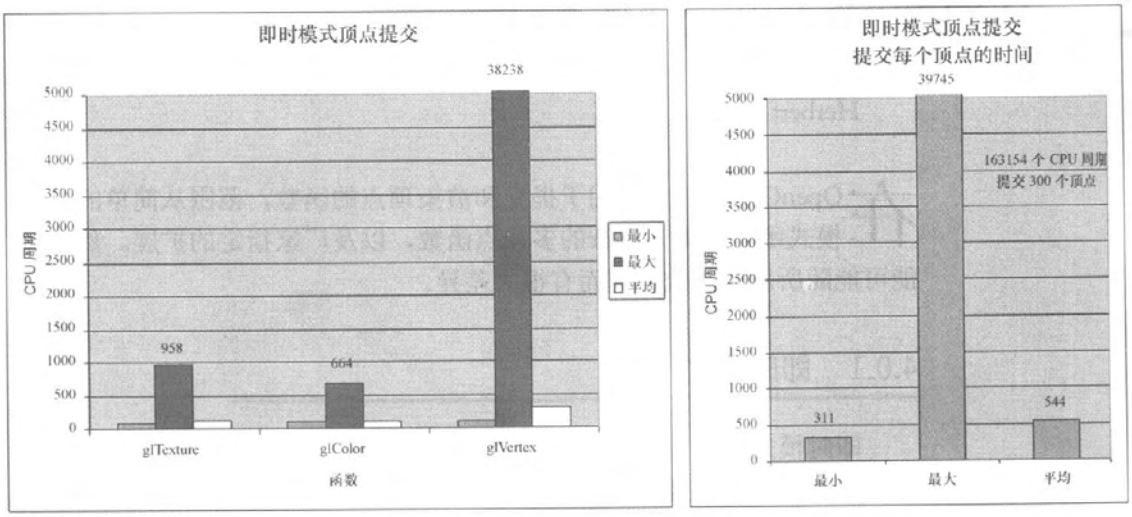


图 4.0.1 使用即时模式提交 300 个顶点（100 个离散的三角形）的 CPU 周期

要想改善性能，应该减少函数开销，而最简单的方法就是通过减少为提交和渲染这 300 个顶点而调用的函数的数量。调用一或两个函数提交和渲染所有 300 顶点将比调用 900 个函数（我们刚刚就是这样做的）性能上要好得多。

4.0.2 交叉存取数据

如果顶点数据已经包含在一个单独结构中，glInterleavedArrays 可在一个单独函数调用中提交顶点的所有成分。glInterleavedArrays 能够提交大量的标准交叉存取的顶点结构，范围从轻量级仅含位置的顶点到重量级的具有位置、法线、漫反射颜色和纹理坐标的顶点。

glInterleavedArrays 只能提交一个指向被渲染的顶点的指针。实际渲染数据必须调用另一个函数，例如 glDrawArrays、glDrawElements 或 glArrayElement。

将 glInterleavedArrays 用于前文即时模式的例子，一次函数调用可以用来为单个顶点提交所有的数据。然而，顾名思义，glInterleavedArrays 也可以接受一个为渲染而提交的顶点数组。这允许我们用一次函数调用提交所有的 300 个顶点，而不像在即时模式中那样每个顶点进行 3 次调用（共 900 次）。

在试验数据的情形下，一个有 300 个顶点的数组用 glInterleavedArrays GL_T2F_C3F_V3F 顶点结构格式来生成。这有效地复制了被即时模式函数所提交和渲染的数据。

图 4.0.2 中给出了用 glInterleavedArrays 和 glDrawArrays 提交和渲染顶点所需的时间总量。提交 300 个顶点（原文误为 triangle，译者注）工作量的平均时间是大约 72 821 个 CPU 周期。这比用即时模式函数提交和渲染相同工作量所需要时间的一半还少（约为 44%）。

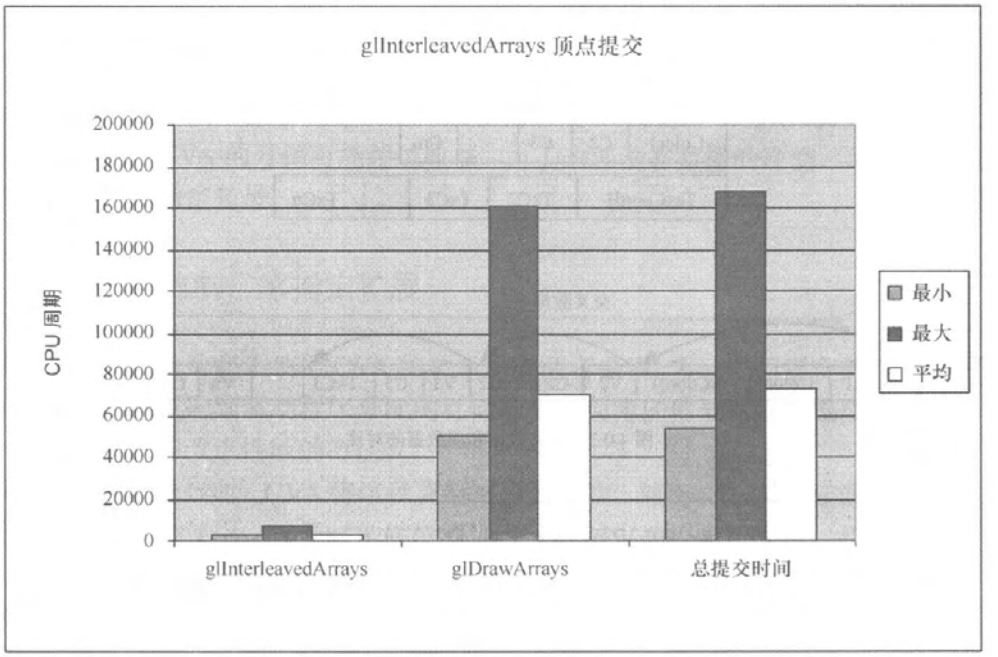


图 4.0.2 用 glInterleavedArrays 提交顶点

4.0.3 步数据和流数据

另一个可替换的顶点提交界面是 gl*Pointer 函数。类似于 glInterleavedArrays，指向顶点数据的指针用 gl*Pointer 函数来提交（例如 glVertexPointer，glColorPointer）。被提交的顶点数据随后用 glDrawArrays、glArrayElement 或 glDrawElements 函数进行渲染。

类似于 glInterleavedArrays，gl*Pointer 也有一个统一的步参数。步指定为从一个顶点组分的开始到下一个顶点的字节数。当步比零大时，gl*Pointer 函数操作本质上与对 glInterleavedArrays 进行一个单一调用是相同的。当步是零（数据被紧紧地压缩在一起）时，数据被称为流数据（参见图 4.0.3）。

当使用 SIMD（单指令多数据）指令集，如 Intel 的 SSE（Streaming SIMD Instructions）或 AMD 的 3DNow!的指令进行转换、光照和（或）剪切顶点的时候，流数据是非常重要的。如果数据采用的是交叉顶点格式，数据必须逐点移入和移出 CPU 的 SIMD 寄存器。若数据采用的是流格式，则 CPU 能快速且容易地移动大块数据进入 SIMD 寄存器进行处理。

对几何体和光照即使不利用 CPU 的 SIMD 指令，仅仅通过使用 gl*Pointer 函数也能使性能有一个明显的改善。

用有 glDrawArrays 的 gl*Pointer 函数（见图 4.0.4）提交和渲染 300 个顶点平均要花费大约 51 212 个 CPU 周期的工作量，作为对比，用 glInterleavedArrays 和 glDrawArrays（图 4.0.3）则花费大约 72 821 个 CPU 周期。时间上大约减少了 30%。如果我们再依靠 SIMD CPU 指令来实现几何体和光照操作的话，性能的改进会更加显著。

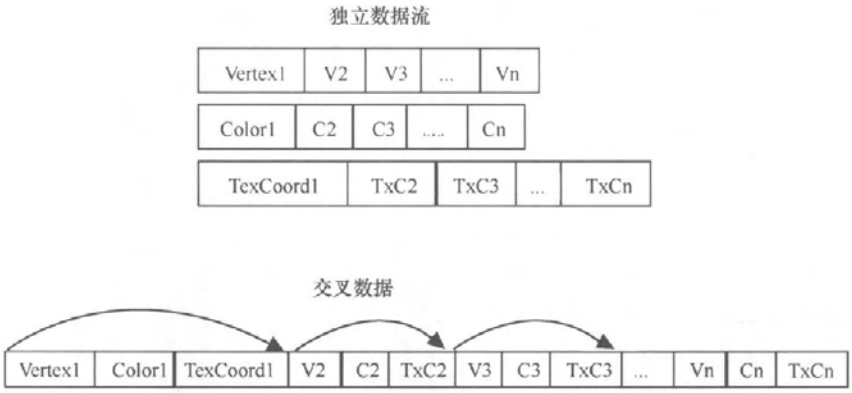


图 4.0.3 交叉数据和流数据的对比

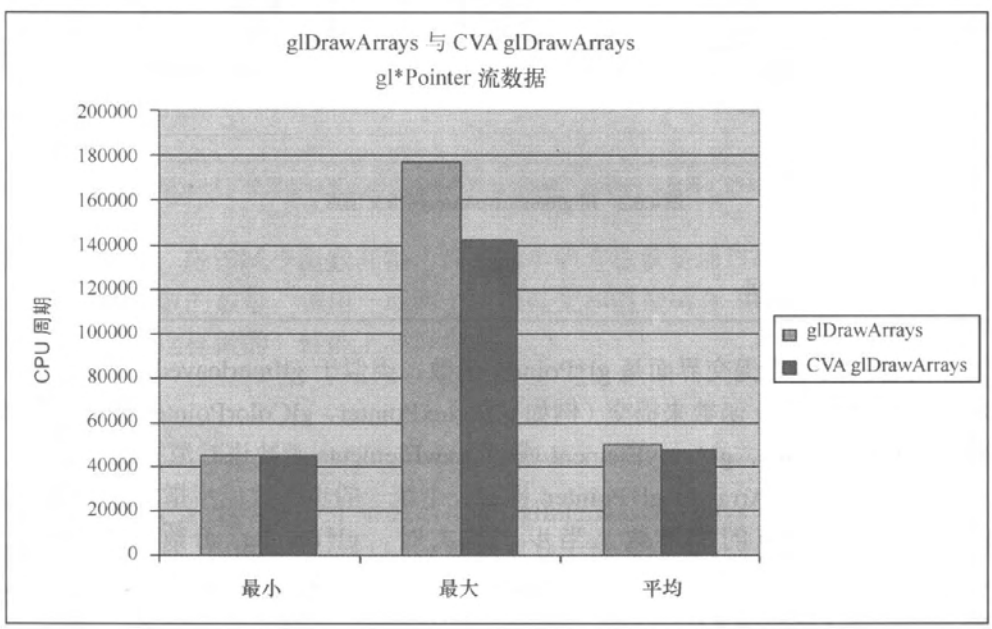


图 4.0.4 有 CVA 与没有 CVA 的 glDrawArrays 的顶点提交时间比较

4.0.4 编译过的顶点数组

编译过的顶点数组扩展（EXT_compiled_vertex_array）是建立在函数 glInterleavedArrays 和 gl*Pointer 的功能基础上的。编译过的顶点数组（CVA）函数允许在由 glInterleavedArrays 或 gl*Pointer 所提供的数组中指定一个数据范围的应用，该范围不会被应用改变。它们允许驱动程序对数据范围进行一次优化，并一直重用优化过的版本直到应用解除了数据锁定。

为了达到最佳存取，编译过的顶点函数允许 CPU 的转换和光照进行数据重排，导致了有效速度的增加。它也允许渲染硬件为了获得更快的性能而修正数据，或者甚至允许在图形适配器上为更快速存取进行一个本地复制。

在我们的测试工作量（参见图 4.0.4）下，使用 CVA 和不使用 CVA 之间性能差别不是很

大，但这只是因为正在试图做的是对函数开销进行量化。如果顶点正被转换、光照和剪裁，或有更多的操作，这种性能上的差别将会相当大。

对静态数据或在被修正之前多次使用的数据来说 CVA 是非常有用的。如果数据只是被使用一次，那么使用 CVA 的开销可能超过收益。为了改善动态数据的性能，目前惟一的通用替代方案是使用厂家指定扩展。

4.0.5 取消数据复制厂家指定扩展

对于即时模式函数和不使用 CVA 的数组函数两者来说，为渲染而提交的数据必须从应用程序分配的存储器复制到驱动程序分配的存储器中。因为任何数据复制都要花时间，取消复制是一种容易实现的改进性能的方法。

当顶点数组被锁定时，CVA 将这种复制减少为一次。然而，CVA 假定数据是静态的或将在修正之前被重复地使用。对于时常被更新或改变的动态数据数组来说，仍然存在从应用存储器复制数据到驱动程序存储器的问题。

消除复制的惟一方法是为应用程序直接在驱动程序分配的存储器中储存顶点，一些厂家支持这种扩展。nVidia 扩展 wglAllocateMemoryNV 就是一种这样的厂商指定扩展。它在应用程序能储存顶点数据的地方把直接可存取的存储器分派给图形卡。这就消除了为任何驱动程序进行数据复制的需求，当顶点数组被提交并随即被渲染时性能也得到改善，因为数据已经准备好了。

请检验你的厂商为它们自己的 OpenGL 所做的扩展。

4.0.6 数据格式

需要考虑的第二个方面是为渲染三角形而被提交的顶点采用的格式。顶点表（像用于测试工作量的）是最常用的格式。在一个顶点表中，每个三角形由 3 个顶点定义（如图 4.0.5A）。然而，当三角形共享顶点时，顶点表会不合理地重复相同的顶点。一个替代方案是使用三角形条带（triangle strip）或扇形（fan），如图 4.0.5B 所示。

离散的、条带的或扇形三角形可由 glDrawArrays 和 glDrawElements 的模式参数识别。这些模式参数分别是 GL_TRIANGLES、GL_TRIANGLE_STRIP 和 GL_TRIANGLE_FAN。

减少绘制大量三角形所需顶点数目的另一个方法是利用一个顶点索引的分离数组建立来自顶点数组的面（如图 4.0.5C）。glDrawElements 与 glDrawArrays 类似，但它增加了一个接受面顶点索引表的新参数。

对于一些复杂的三角形网格来说，使用三角形条带和面顶点索引将顶点与三角形的比降低到接近 1:1 是可能的。然而，在许多数据类型上，与面顶点索引结合的三角形列表是极快速的，并且创建时需要更少的预处理。

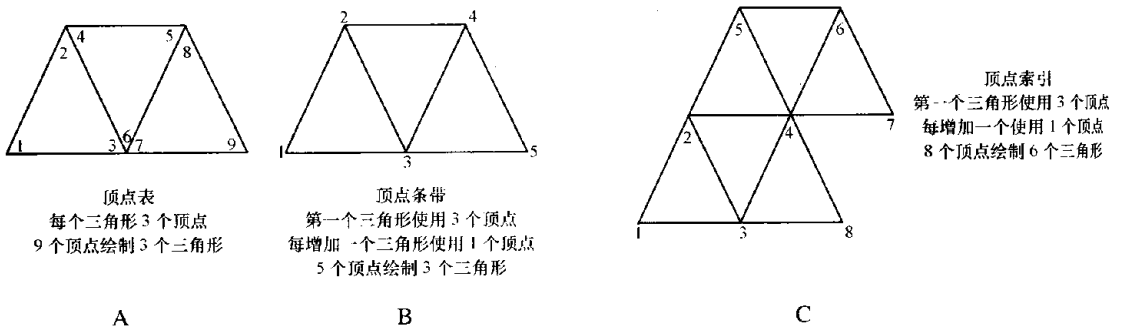


图 4.0.5 A，B，C 分别显示的是 A:3 个离散的三角形，B:3 个条带三角形，C:6 个索引过的三角形的顶点

4.0.7 一般建议

这里有许多提高顶点提交和渲染性能的一般建议。

（1）当使用索引过的数据时，应注意对单个三角形所有的顶点尽可能互相接近地协同定位。如果一个三角形所需的顶点在数组中分离得太远，当它跳来跳去时，可能使图形适配器不断地重复处理数组的子部分。

（2）用材质、阴影和纹理设置对顶点数据进行预排序，可以有助于增加在一次函数调用中能被提交和（或）渲染的顶点数量。

（3）对每一顶点提交的信息量应尽可能小。不要包含只是偶尔使用的附加数据。这一点必须用不断变更顶点格式的方法来取得平衡。例如，如果颜色信息很少使用，提交的顶点就不附加颜色信息。

（4）在提交太少数据和太多数据之间有一个平衡问题。大部分数组函数至少需要提交 10~50 个顶点来克服函数开销。而提交 32k~64k 的顶点数据则是上限。这些极限数量随图形适配器不同会有所变化。

（5）花费过多时间将大量顶点数据导入单个缓冲器（非驱动程序分配缓冲器）可能会引起许多问题，以至 CPU 难以解决。这些问题包括 cache 问题、使图形卡迟滞，和拥有太多数据以至函数无法运行。

4.0.8 结论

（1）即时模式函数在刚开始时可能易于使用，但是对于为渲染而提交顶点而言，它们是性能最低的函数（参见图 4.0.6）。

（2）在一次函数调用中，提交和（或）渲染合理数目的顶点。

（3）对静态数据或不常变化的数据，使用编译过的顶点数组（CVA）。

（4）为了获得最好的 CPU 转换和光照性能，使用带 CVA 的流数据格式。

（5）一些厂家将会提供专门的顶点提交扩展以使它具有更高的性能。

（6）在渲染离散或条带三角形时，使用索引过的顶点数据，将渲染三角形所使用的顶点数目减到最少。

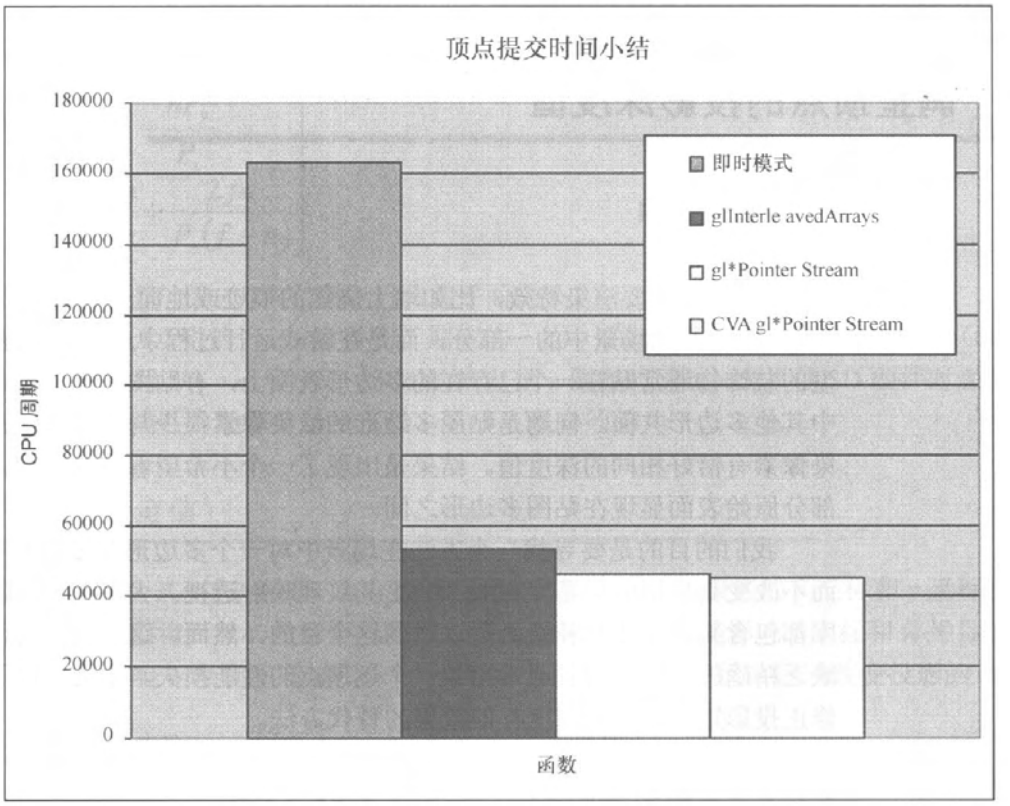


图 4.0.6 不同函数进行顶点提交和渲染的时间的简要比较

4.0.9 参考文献

[ARB] OpenGL Extensions. OpenGL ARB. www.opengl.org

[Kempf97] Kempf, R., and Frazier, C., *OpenGL Reference Manual* 2nd Edition, Addison-Wesley Developers Press, 1997.

[Spitzer00] Spitzer, John F., Maximizing OpenGL Performance for GPUs. 08 March 2000. www.nvidia.com.

4.1 调整顶点的投影深度值

Eric Lengyel

许多游戏需要渲染特效，比如墙上烧焦的痕迹或地面上的脚印，它们不是原始场景中的一部分，而是在游戏运行过程中产生的。这些类型的装饰物通常贴在一个已存在的多边形表面上，有贴图的多边形与场景中其他多边形共面。问题是贴图多边形的渲染像素很少与共面多边形的渲染像素有恰好相同的深度值。结果是出现了一个不希望看到的图案，即有部分原始表面显现在贴图多边形之间。

我们的目的是要寻找一个方法在场景中对一个多边形深度进行补偿，而不改变其投影的屏幕坐标或不变更其纹理映射透视。大多数的 3D 图形库都包含某种多边形补偿函数以达到这个目的。然而，这些解决方法通常缺乏精确的控制，而且通常招致一个逐顶点的性能损失。本文提供了一个修正投影矩阵以达到深度补偿效果的替代方法。

4.1.1 考察投影矩阵

让我们先来考察一下标准的 OpenGL 透视投影矩阵在一个视角空间中一点 $\mathbf{P} = (P_x, P_y, P_z, 1)$ 上的效果。为了简化矩阵，我们假设可视棱台的中心轴（view frustum）在视角空间的 Z 轴上（即，被 4 个侧面切出的近裁剪平面（clipping plane）上的矩形有 *left* = –*right* 和 *bottom* = – *top* 的特性）。设到近裁剪平面的距离为 n，到远裁剪平面的距离为 f，我们有：

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & -\dfrac{f+n}{f-n} & -\dfrac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} nP_x \\ nP_y \\ -\dfrac{f+n}{f-n}P_z - \dfrac{2fn}{f-n} \\ -P_z \end{bmatrix} \tag{4.1.1}$$

为了要完成投影，我们需要用它的 w 坐标（其值为 $-P_z$）来除这个结果。由这个除法得到下面的投影 3D 点，我们把它记为 $\mathbf{P}'$。

$$\mathbf{P}'=\begin{bmatrix} -\dfrac{nP_x}{P_z} \\ -\dfrac{nP_y}{P_z} \\ \dfrac{f+n}{f-n}+\dfrac{2fn}{P_z(f-n)} \end{bmatrix} \tag{4.1.2}$$

因为摄像机指向负 z 方向，所以近裁剪平面位于 $z=-n$，远裁剪平面位于 $z=-f$。将 $-n$ 和 $-f$ 代入公式 4.1.2 中的 P_z，从而得到了期望的 -1 到 1 的 z 值，与规一化绑定裁剪体（clipping volume）。记住这个从[$-n$.. $-f$]到[-1..1]的映射是 z 倒数的函数。为了使 3D 硬件对深度缓冲器中的值所做的线性内插保持透视正确，这一点是必须的。

4.1.2 矫正深度值

很明显，从公式 4.1.2 来看，为 w 坐标保留 $-P_z$ 值将保证被投影的 x 坐标和 y 坐标也保留下来。此后，我们将只关心投影矩阵的右下 2×2 部分，因为这是影响 z 坐标和 w 坐标的惟一部分。可以不影响 w 坐标而通过引入一个 $1+\varepsilon$（ε 是某个较小值）因子改变投影的 z 坐标：

$$\begin{bmatrix} -(1+\varepsilon)\dfrac{f+n}{f-n} & -\dfrac{2fn}{f-n} \\ -1 & 0 \end{bmatrix}\cdot\begin{bmatrix} P_z \\ 1 \end{bmatrix}=\begin{bmatrix} -(1+\varepsilon)\dfrac{f+n}{f-n}P_z-\dfrac{2fn}{f-n} \\ -P_z \end{bmatrix} \tag{4.1.3}$$

在除以 w 之后，投影 z 坐标得到下值。

$$\begin{aligned} P_z' &= (1+\varepsilon)\frac{f+n}{f-n}+\frac{2fn}{P_z(f-n)} \\ &= \frac{f+n}{f-n}+\frac{2fn}{P_z(f-n)}+\varepsilon\frac{f+n}{f-n} \end{aligned} \tag{4.1.4}$$

将这个式子与等式 4.1.2 中的 z 坐标做一下比较，可以看到我们已经发现了一个用常量 $\varepsilon\dfrac{f+n}{f-n}$ 来补偿投影深度值的方法。

4.1.3 选择一个适当的 ε

由于 Z-buffer 的非线性特性，公式 4.1.4 中所给出的常偏移量在摄像机离得远处比近处对应于一个很大的视觉空间差。虽然对于一些应用来说常偏移量可能作用得很好，但还没有单一的解决方案能够作用于每个应用的所有深度。我们最好是选择一个适当的 ε，它给出视觉空间偏移量 δ 和深度值 P_z，它们共同表示了我们正补偿的对象。为了给 ε 确定一个公式，让我们把从公式 4.1.1 得到的标准投影矩阵应用于一点，它的 z 坐标已被某个小的 δ 补偿过。

$$\begin{bmatrix} -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_z+\delta \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{f+n}{f-n}(P_z+\delta) - \frac{2fn}{f-n} \\ -(P_z+\delta) \end{bmatrix} \tag{4.1.5}$$

除以 w 后，对于投影的 z 坐标我们有下面的值。

$$\begin{aligned} P_z' &= \frac{f+n}{f-n} + \frac{2fn}{(P_z+\delta)(f-n)} \\ &= \frac{f+n}{f-n} + \frac{2fn}{P_z(f-n)} + \frac{2fn}{f-n}\left(\frac{1}{P_z+\delta} - \frac{1}{P_z}\right) \end{aligned} \tag{4.1.6}$$

使这个结果与公式 4.1.4 相等并化简，最后得到：

$$\varepsilon = -\frac{2fn}{f+n}\left(\frac{\delta}{P_z(P_z+\delta)}\right) \tag{4.1.7}$$

做一些实验能够找到对于某个特殊应用的较好的 δ 值。应该记住的是 δ 是一个视觉空间偏移量，这样，当 P_z 变得较大的时候就会不太有效了。对于一个 m bit 整数深度缓冲器来说，我们想确保：

$$|\varepsilon| \geq \frac{1}{2^m-1}\left(\frac{f-n}{f+n}\right) \tag{4.1.8}$$

因为更小的 ε 值将不能产生足以改变整数深度值的偏移量。用等式 4.1.7 的右边替换 ε，解出 δ 得：

$$\delta \geq \frac{kP_z^2}{1-kP_z} \tag{4.1.9}$$

或者：

$$\delta \leq \frac{-kP_z^2}{1+kP_z} \tag{4.1.10}$$

这里常量 k 由下式给出：

$$k = \frac{f-n}{2fn(2^m-1)} \tag{4.1.11}$$

当补偿一个对着摄像机的多边形的时候（平常的情形），等式 4.1.9 给出了 δ 的最小有效值，而当补偿一个远离摄像机的多边形时，等式 4.1.10 给出了 δ 的最大有效值。

4.1.4 实现

下面的示例代码说明了等式 4.1.3 中的投影矩阵如何在 OpenGL 下得到实现。函数 LoadOffsetMatrix 取与传递给 OpenGL 函数 glFrustum 同样的 6 个值。同时也取δ和P_z值来计算ε。

4.1.5 源代码

```
#include <gl.h>

void LoadOffsetMatrix(GLdouble l, GLdouble r,
GLdouble b, GLdouble t,
GLdouble n, GLdouble f,
GLfloat delta, GLfloat pz)
{
    GLfloat  matrix[16];

    // Set up standard perspective projection
    glMatrixMode(GL_PROJECTION);
    glFrustum(l, r, b, t, n, f);

    // Retrieve the projection matrix
    glGetFloatv(GL_PROJECTION_MATRIX, matrix);

    // Calculate epsilon with equation (7)
    GLfloat epsilon = -2.0F * f * n * delta /
((f + n) * pz * (pz + delta));

    // Modify entry (3,3) of the projection matrix
    matrix[10] *= 1.0F + epsilon;

    // Send the projection matrix back to OpenGL
    glLoadMatrixf(matrix);
}
```

4.2 矢量摄像机

David Paull

矢量摄像机是基于矩阵的摄像机的一种一般化形式，广泛应用于传统的图形引擎。由于矩阵有几个典型的连接操作，常常难以阅读。矢量摄像机只用简单的矢量来描述它的朝向、位置、视野，以及宽高比（aspect ratio）。这种格式允许对全局图像管道进行一些有趣的优化。矢量摄像机与基于矩阵的摄像机所使用的信息是相同的。世界到摄像机（world-to-camera）矩阵（观察矩阵，view matrix）可以分解为 4 个矢量。如你在图 4.2.1 所看到的，一个观察矩阵的确由 4 个矢量构成。

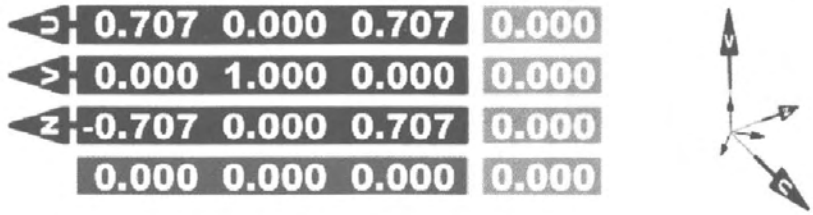


图 4.2.1 观察矩阵由 4 个矢量构成

3 个矢量代表 3 个轴，它们定义了摄像机的朝向，另一个矢量代表摄像机在世界坐标空间中位置。总的来说，这提供了 6 个自由度。在一些图形引擎中，你也许需要对观察矩阵进行转换以与矢量摄像机相一致。图 4.2.2 显示了一个有着矢量摄像机和一个盒子模型的场景。该图也显示了定义摄像机视域界限的观察金字塔。

矢量摄像机的主要优点是它对本地坐标空间和世界坐标空间都能进行操作。摄像机的朝向和位置矢量都存储在世界空间中；然而，它们能用模型的本地到世界矩阵的逆矩阵反向变换回本地空间。摄像机和物体不必互相相对运动，摄像机的新朝向和位置都是相对于本地空间物体的。这些是渲染物体所需要的惟一的变换。现在矢量摄像机在本地空间了，它能够投影本地空间坐标，而且不再需要进一步的变换了。几乎不费吹灰之力，矢量摄像机现在能够通过模型中的每个本地空间顶点而循环，并且将它们映射到 2D 屏幕坐标。如果模型是静态的，比如一座山脉，那么模型数据可以被存储在世界空间中。这样就可以考虑一个更快的编码方式了。有了世界空间中的物体和摄像机，不再需要计算逆矩阵，也根本不需要进行任何变换了。

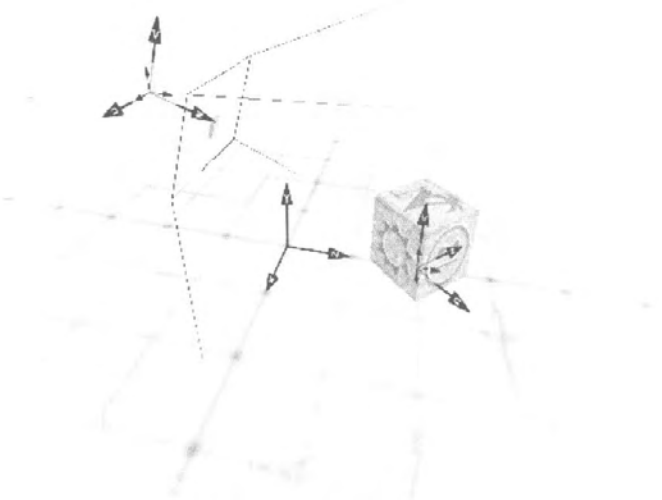


图 4.2.2 矢量摄像机的说明

这些图中使用了左手坐标系统，其 *Y* 轴竖直向上。矢量摄像机的位置是（0, 2, −2），并且有一个小的关于 *U* 轴的旋转使摄像机向下倾斜。盒子的位置是（0, 0, 2），而且有一个关于 *Y* 轴的 45° 旋转。更小的箭头表明世界（*x*, *y*, *z*）轴矢量以帮助说明旋转。

4.2.1 矢量摄像机初步

矢量摄像机使用 3 个矢量代表它的朝向。如果摄像机没有进行任何旋转并且位于(0,0,0)，则 *U*、*V* 和 *N* 矢量分别平行于 *X*，*Y* 和 *Z* 的初始矢量。*U* 矢量指向右方，*V* 指向上方，而 *N* 指向摄像机面对的方向。图 4.2.3 显示了矢量摄像机和它的 3D 屏幕。这个 3D 屏幕是用摄像机的 *U*，*V* 和 *N* 矢量和两个视野参数（field of view parameter）创建的。视野参数是使用用户定义的视野计算的，这个视野然后被宽高比进行缩放。有许多计算视野参数的方法。对于这些例子，我采用了下面的代码：

```
float AspectRatio = ScreenHeight/ScreenWidth;
float FOV = pi/2;
float hFrac = tan(FOV*0.5);
float vFrac = tan(FOV*0.5*AspectRatio);
```

如果摄像机没有进行任何旋转或平移，矢量将如下定义：

```
O vector = -U vector * hFrac + V vector * vFrac
S vector =  U vector * hFrac * 2
T vector = -V vector * vFrac * 2
```

3D 屏幕是由添加这些矢量产生的。对于本例，与近平面的距离是 1.0，这样从摄像机的位置开始，添加 1.0 * *N* 矢量。然后添加 *O* 矢量，它定义的 3D 点将被称为屏幕原点。*S* 和 *T*

矢量都从这个点发出去。*S* 和 *T* 分别定义了屏幕的 *X* 和 *Y* 常量直线。这就好比添加了一个屏幕大小的纹理到了由 *S* 和 *T* 矢量定义的四边形中。使用摄像机的世界空间位置和世界空间矢量 *S* 和 *T*，任何世界空间中的点能都可以用下面的方法被映射到屏幕坐标系。产生一个矢量，它的起点是摄像机的世界空间位置，它终止于被投影顶点的世界空间位置。如果顶点是在摄像机的视野中，我们可以计算光线与矢量摄像机的 3D 屏幕的交点。这个交点在图 4.2.3 和图 4.2.4 中标记为 *P* 矢量。然后利用矢量的点积计算沿着 *S* 和 *T* 矢量的距离，这样其实就将值转换到了 2D 空间。这些 2D 值根据当前的显示分辨率进行重新缩放，最后 2D 屏幕坐标就可以显示了。

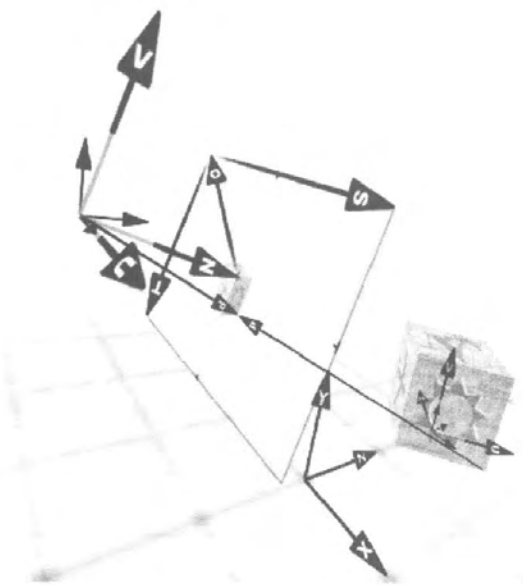


图 4.2.3 使用矢量 *P*

4.2.2 本地空间优化

摄像机矢量被存储在世界空间；然而模型典型地存储在本地空间（local space，有时称为模型空间）。模型以（0,0,0）为其中心并且伴随一个本地到世界矩阵。这个本地到世界矩阵规定了物体将如何进行旋转和平移，以使它终止于其最终的世界空间中的朝向和位置。由于模型数据存储于本地空间，如果能够在本地空间工作将会非常有利。为了达到这个目的，摄像机必须从世界空间移动到本地空间。摄像机必须围绕物体进行旋转和平移，这样摄像机对物体的新朝向和位置就保持了同样的空间关系，好象物体从局部空间变换到世界空间，并且被摄像机在世界空间观察一样。解决的方案就是采用模型的本地到世界矩阵的逆矩阵。一个旋转矩阵的逆由两部分计算。第一部分实现了上部的 3×3 旋转矩阵的转置，第二部分用 3 个点积来计算新的位置。这个逆矩阵能够执行世界到本地的反向操作。

请注意这里需要保持两个真正的本地到世界的变换。摄像机的朝向和位置假设已经应用了本地到世界的变换；这样我们仅需考虑物体的本地到世界矩阵就可以了。新产生的世界到本地矩阵定义了摄像机的朝向和位置将如何旋转和平移以保持物体与摄像机之间的空间关

系。正如你在图 4.2.3 中所看到的，盒子经过了本地到世界变换，这是一个较慢的逐个顶点处理过程。也请注意将该图中摄像机的世界位置和朝向与图 4.2.4 中的进行比较。你可以看到在图 4.2.4 中，立方体在它的本地坐标空间中，没有进行任何的旋转或平移。然而摄像机已经被旋转和平移了，于是摄像机产生了与图 4.2.3 中同样的图像。将摄像机移动到本地空间中要比将本地数据移动到世界空间中计算起来更快。为将摄像机移动到本地空间只需进行 4 个变换，而将一个具有 N 个顶点的形状从本地移动到世界空间则需要 N 个变换。一旦模型数据和摄像机数据都在同一坐标空间，模型数据的投影所需的计算量就极少了。

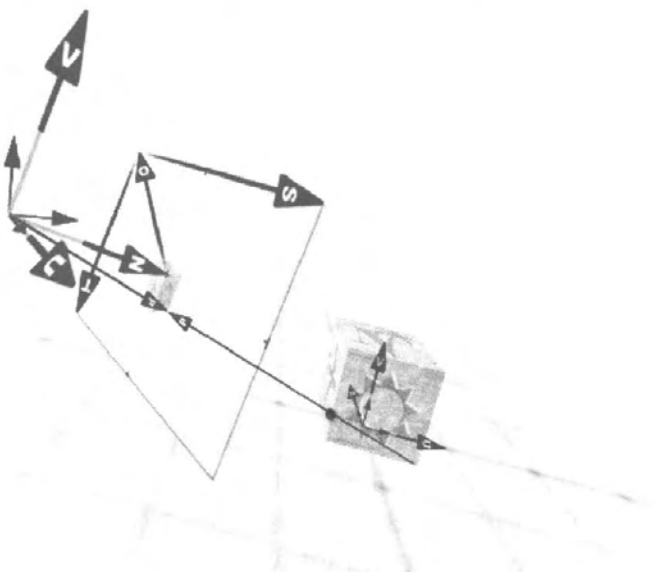


图 4.2.4 立方体位于一个本地坐标空间内

现在摄像机位于本地空间，可以进行一些另外的优化了。如果你为模型中的每一个三角形存储平面法线，使用一个点积就可以执行反向操作了。如果三角形面对摄像机，它的所有顶点都标记为可见的。当整个物体已经被背面剔除时，则只有可见点需要被映射。

4.2.3 结论

矢量摄像机是用于软件或硬件 3D 渲染引擎中表现摄像机运算的一个便利的方法。它的简单性提供了直观地放置和操作摄像机的自由度。摄像机能够在本地坐标空间工作允许进行几种众所周知的优化，以获得比用传统的基于矩阵的摄像机更大的渲染速度。矢量摄像机增长了 25%的帧速率。通过减少需要变换的数目、计算较小的用于最终显示的数据包，并使用一个较低的总内存量来达到这个目的。映射运算十分灵活，它可以应用于任何基于聚焦的棱台（ocus-based frustum），如阴影体（shadow volume）。有时，最好的优化就是重新设计方法，而不是对现有的方法进行可怜的压榨！

本书中包括有矢量摄像机的 OpenGL 源代码，最新版本的源代码请访问 Tanzanite Software 网站 www.tanzanite.to。本文中使用的所有图示都是用 TechNature 引擎渲染的。

4.3 摄像机控制技术

Dante Treglia II

游戏吸引玩家并令其迷醉的主要因素是它的交互特性。在游戏中玩家能够扮演众多不同的角色：从性感、衣着暴露的武士，到有着迷人魅力足以吸引公主的意大利管道工人，这便是诱使许多人购买游戏的动力。但是为了让玩家通过不同的眼睛观察世界，游戏需要有固定的摄像机模型。这就是引入摄像机控制技术的由来。本文将概述几个基本技术，你可以用它们为游戏开发适当的摄像机模型。

4.3.1 一种基本的第一人称摄像机

1. 观察

基本的第一人称视角（first-person）摄像机模型依赖于“观察”应用程序，如 OpenGL 的 gluLookAt()。给定一个摄像机位置、观察方向，以及向上的矢量，这个函数将返回一个朝向观察矩阵。这个观察矩阵然后被置于 OpenGL MODELVIEW 矩阵堆栈，并且当场景中的每个物体被渲染的时候，与它们的朝向矩阵相连接。这种摄像机模型非常易于实现，而且十分有用。在 C++矩阵库中能找到该函数的一种实现。

2. 欧拉角

三维空间中的朝向能够用 3 个欧拉角表示：偏转角（yaw），倾斜角（pitch）和转角（roll）（也称为方位角（azimuth），仰角（elevation）和转角（roll））。yaw，pitch，和 roll 分别说明了在 Y 轴，X 轴和 Z 轴方向的旋转。

一种计算朝向矩阵的方法是连接 3 个轴的旋转矩阵。然而，为了达到控制的目的，有必要保持当前位置以及关于摄像机的 X 轴（侧边）、Y 轴（向上）和 Z 轴（向前）的信息。向前、侧边和向上矢量分别被用于计算摄像机的前进、平移和跳跃。下面的代码给出了一个计算这些矢量的函数，它们能够被用作 gluLookAt( )函数的参数以产生摄像机的观察矩阵。

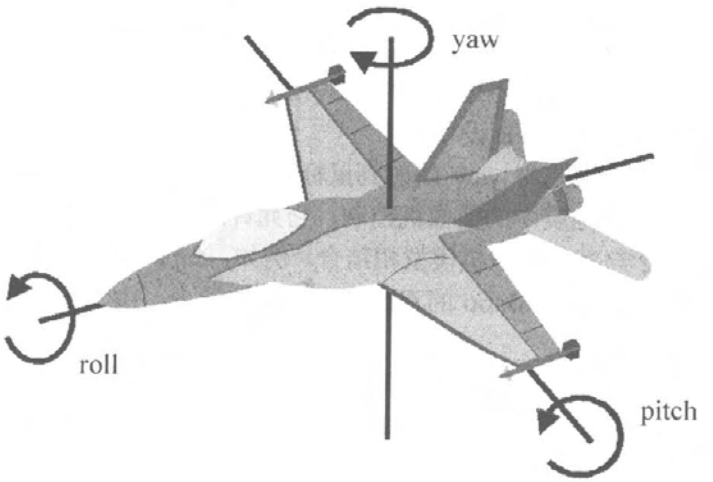


图 4.3.1 一个用 yaw，pitch 和 roll 角描述的飞机图像

```
void FlyCam::ComputeInfo() {
    float cosY, cosP, cosR;
    float sinY, sinP, sinR;

    // Only Want to Calc these once
    cosY = cosf(Y);
    cosP = cosf(P);
    cosR = cosf(R);
    sinY = sinf(Y);
    sinP = sinf(P);
    sinR = sinf(R);

    // Fwd Vector
    fwd.x = sinY * cosP;
    fwd.y = sinP;
    fwd.z = cosP * -cosY;

    // Look At Point
    at = fwd + eye;

    // Up Vector
    up.x = -cosY * sinR - sinY * sinP * cosR;
    up.y = cosP * cosR;
    up.z = -sinY * sinR - sinP * cosR * -cosY;

    // Side Vector (right)
    side = CrossProduct(fwd, up);
}
```

3. 控制

将摄像机的朝向和位置贯穿场景进行移动是与游戏有很大联系的。例如，第一人称视角

射手的摄像机位置很可能会沿着环境轮廓移动。精确的飞行模拟器将依赖于其他的环境因素，诸如引擎推力、高度、空气状况、扰动等等。在此我仅打算抛砖引玉地讨论其必要的成分。我们假定用户的输入是键盘和鼠标，不过要记住这些技术易于应用到几乎任何输入设备，包括操纵杆、控制台控制器，甚至于 VR 设备。

使用欧拉角最直观的控制就是将 yaw 映射到鼠标的（相对于屏幕的）*X* 位置，将 pitch 映射到 *Y* 位置。该映射将模拟地震中的摄像机控制，它允许用户以两个自由度迅速地改变他们的观察。这项技术能用于与其他摄像机模型相结合。例如，*Super Mario64* 主要是一个第一人称视角游戏，但是也包括了通过 Mario 的双眼考查世界的能力。由于 roll 不是一种普通的人类体验，所以通常被忽略了。

摄像机与环境的交互作用是一个游戏特有的话题。正如我在前面提到的，向前、侧边和向上的矢量被用于控制摄像机的位置，而且应该直接与你的游戏引擎捆绑在一起。无论你如何选择合并这些矢量，我强烈建议你运用以时间为基础的物理。这将确保你的控制是与帧速率无关的，并且使游戏感觉更真实。在一无所知的情况下，你应该将用户的输入解释为在希望的方向上的速率。由于你有摄像机的正交朝向基矢量（向前，侧边，向上），其实现就是简单的一维物理问题了：

```
position += deltaTime * inputSpeed * forward
```

加速度也能被用于获得衰减效果。在示例代码中，我已经将这些控制技术合并到一个飞行摄像机了，它允许你在任何方向移动视线。

4.3.2 脚本摄像机

脚本摄像机（scripted camera）是许多游戏的至关重要的部分，从角色扮演游戏中的电影场景到飞越一个高尔夫球场的直升机。大多数采用这种摄像机技术的游戏使用动画包来为摄像机编写脚本，然后将动画输入到它们的游戏引擎中。对静态路径来说这是一个非常绝妙的解决方案，但是对于动态路径会怎样？比如假设你想使玩家有一个魂飞魄散的体验，并且当瞄准目标敌人、同伙、天堂或地狱之门时飞越场景。

1. B 样条曲线

给定一个控制点集，B 样条曲线是一种可适性好、容易且高效的生成平滑曲线的方法。有其他几种曲线生成算法和改进算法能产生更好的适应性和效率，我鼓励大家去进行探索，但是 B 样条是一个极好的起点。我将讨论的 3 次 B 样条的实现是基于一个矩阵形式的基本函数的，如公式 4.3.1 所示。给出 4 个控制点的一个集合和一个参数 u（它由一系列的细分均匀地从 0 变化到 1），该矩阵将产生一条平滑曲线段。对于控制点的每一个元素(x, y, z)，应用 Build( )函数。

公式 4.3.1 3 次 B 样条基本函数

$$B-Spline=\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 0 & 4\\ -3 & 3 & 3 & 1\\ 1 & 0 & 0 & 0\end{bmatrix}\frac{1}{6}$$

```
void Spline::Build() {
  float u, u_2, u_3;
  int i, j, k;
  int index;

  index = 0;
  // For each control Point (Minus the last three)
  for (i = 0; i < controlCnt - 3; i++) {

    // For each subdivision
    for(j = 0; j < curveSubD; j++) {
      u = (float)j / curveSubD;
      u_2 = u * u;
      u_3 = u_2 * u;

      for(k = 0; k < 3; k++) {
        // Position
        curveData[index].pos[k] =
          (
            (-1*u_3 + 3*u_2 - 3*u + 1) *
controlData[i  ].pos[k] +
            ( 3*u_3 - 6*u_2 + 0*u + 4) *
controlData[i+0].pos[k] +
            (-3*u_3 + 3*u_2 + 3*u + 1) *
controlData[i+1].pos[k] +
            ( 1*u_3 + 0*u_2 + 0*u + 0) *
controlData[i+2].pos[k]
          ) / 6.0F;
      }

      index ++;
    }
  }
}
```

注意参数 u 是自乘的且是三次方的，这样我们可以得到一个 3 次样条。也请注意最后 3 个控制点没有被使用，因为这个算法每次利用 4 个连续的控制点。我将这个附加的部分放在了算法中，这样可以保持曲线的连续性。例如，下面的控制点将产生一个非常近似于环的结果。

```
C:  48.000000 2.000000  48.000000 // 没有绘制
```

```
C:  48.000000 2.000000 -48.000000
C: -48.000000 2.000000 -48.000000
C: -48.000000 2.000000  48.000000
C:  48.000000 2.000000  48.000000
C:  48.000000 2.000000 -48.000000 // 没有绘制
C: -48.000000 2.000000 -48.000000 // 没有绘制
```

将B样条用于我们的目的还需要做些工作。曲线给出了摄像机的位置，但是我们还需要一个目标和一个向上的矢量。对于多重曲线，每一个控制点都应该与一个目标位置相关联。这样，当摄像机沿着曲线运动时，将继续聚焦于目标。一旦它遇到一个新的目标位置，摄像机控制逻辑可以简单地在点之间进行插值以达到期望的效果。一个更复杂但是适应性更好的方法是使用两条B样条曲线，一条用于摄像机位置，另一条用于目标位置。

2. 技巧

给定一个控制点集，游戏引擎不但能在一帧中计算整条曲线，而且可以只计算曲线上需要的部分。这样可以减少每帧的计算量，并且也减少了存储曲线数据所需的存储量。对任何曲线部分B样条仅需4个控制点。于是，通过连续地在一个新控制点进行循环，无论它是个随机点还是一个仔细计算过的点，都能生成一条长度有限的平滑曲线。本文的演示软件说明了该解决方案。

虽然曲线保证是连续的，部分之间的距离却不是连续的。这样，使摄像机递增地穿过曲线就不是一个胜任的解法了。摄像机将显著地在控制点之间改变速率，而且帧速率也将改变。令曲线穿过屏幕的最好的方法莫过于使用距离和速度计算了。为了精确计算距离，你需要计算曲线平面上每个子部分之间的距离。这个方法需要进行大量的计算，但是很值得。使用该函数可以为一个给定距离计算一个适当的索引。

```
int Spline::GetIndexAtDistance(float distance) {
  int index = 0;

  if (distance < 0.0) return -1;
  // Forward Push
  while (index < curveCnt &&
distance > curveData[index].distance)
  {
    index++;
  }
  if (index >= curveCnt) return -1;
  return index;
}
```

使用B样条的另一个有用的技巧是利用曲线的切线来迫使摄像机的朝向变到曲线上。如果你想实现一个过山车的话，这将尤其有用。每一个曲线点的前一点减去该曲线点可以计算出近似值。不过一个更精确的方法是计算B样条基本函数的导数。公式2给出了这个导数。示例代码中执行了这样的计算。

方程 4.3.2 B 样条基本函数的一阶导数

$$B-Spline' = \begin{bmatrix} 0 & -1 & 2 & -1 \\ 0 & 3 & -4 & 0 \\ 0 & -3 & 2 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \frac{1}{2}$$

下面我将谈谈法线计算的一个缺陷。寻找一条曲线的法线是个简单的问题。一种方法就是取当前控制点的切线，并且与下一个控制点的切线相交。另一方面，寻找合适的法线也是一个困难的问题。一条曲线在任意点上都可以有法线，这样就有无限多条法线，问题是要找到能生成你所希望的结果的那一条法线。我发现通过与切线相交而得的自然法线，通常能得到想要的结果。然而，由于我使用了叉积，当曲线的方向改变时，右手规则有时会使法线受到影响。为了弥补这一点，我们可能通过检查当前法线与前一条法线的点积来看是否它们的方向相差大约 180°。如果是这样，我将设置一个标志使法线反转。有些应用下，我简单地给出控制点的法线，当细分曲线的时候内插出下一条自然（或特殊）的法线。

3. Catmull-Rom

由于我是以 B 样条为主题进行讨论，我将谈谈曲线生成函数的一种变异，称为 Catmull-Rom 曲线。这两种曲线最大的差别就是 Catmull-Rom 曲线通过所有的控制点，而 B 样条曲线则不。不过我必须提醒你这种方法得到的“曲线性”不如 B 样条得到的看起来舒服。尽管如此，你可能会发现得到通过控制点的曲线是有用的或必要的。其基本函数在公式 4.3.3 中列出。

公式 4.3.3 Catmull-Rom 基本函数

$$Catmull-Rom = \begin{bmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \frac{1}{2}$$

$$Catmull-Rom' = \begin{bmatrix} -3 & -1 & 4 & -1 \\ 9 & 3 & -10 & 0 \\ -9 & -3 & 8 & 1 \\ 3 & 1 & -2 & 0 \end{bmatrix} \frac{1}{2}$$

4.3.3 摄像机技巧

1. 放大

假设你想在游戏中有一个具有高倍望远镜的高性能狙击步枪，而且希望玩家能够在整个狙击范围内看得见。在 OpenGL 中完成它的一种快速而简单的方法是使用 gluPerspective( )函

数的 FOV 参数。下面的代码片断将使得摄像机用余弦函数进行放大和缩小。

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(cos(tempAng += 0.03F)*10 + 33,
        640.0F/480.0F, 1.0, 3000.0);
glMatrixMode(GL_MODELVIEW);
```

2. 衰减

衰减（damping）是使得摄像机控制看起来更好且感觉更好的关键。下面的函数返回一个逼近目标矢量的矢量，当它到达目标时就会慢下来。然而，这个方法有一个主要的缺点：它是与帧速率相关的。当帧速率增加时，衰减的效果就会下降。

```
vector3 dampType1(vector3 currX, vector3 targetX) {
    return currX + ((targetX - currX) / 16.0F);
}
```

一种解决衰减问题的可能方案是应用物理学知识。将加速度和摩擦力应用于摄像机位置将产生期望的结果，而且物理方程是帧速率无关的。但是，物理学方法更适用于一个交互式的解法。它们并非只是简单地给出一个当前和目标位置界面，当摄像机是脚本化的或被固定的动画所影响时将更有用。弹力是理想的解决方案。

公式 4.3.4 弹性方程

$$F = ma = -k_s x - k_d v$$

让我们将弹性方程分解为便于使用的部分并且得到一个函数。首先，让我们假设质量是 1。x 代表从弹簧的静止（目标）状态到当前状态的位移。两个常数 k_s 和 k_d 分别代表虎克弹性常数（spring constant）和阻尼常数（damping constant），v 是目标位置的速率。实现该衰减的函数如下。这是一个适用 C++的绝好时机！

```
vector3 SpringDamp(
        vector3 currPos,        // 当前位置
        vector3 trgPos,         // 目的位置
        vector3 prevTrgPos,     // 前一个目的位置
        float deltaTime,        // 及时变换
        float springConst,      // 虎克常数
        float dampConst,        // 衰减常数
        float springLen) {
    vector3 disp;               // 替换
    vector3 velocity;           // 速度
    float forceMag;             // 力系数

    // 计算弹力
    disp = currPos - trgPos;
    velocity = (prevTrgPos - trgPos) * deltaTime;
    forceMag = springConst * (springLen - disp.length()) +
        dampConst * (DotProduct(disp, velocity) /
```

```
            disp.length());

        // 应用弹力
        disp.normalize();
        disp *= forceMag * deltaTime;
        return currPos += disp;
    }
```

3. 第三人称视角（Third-Person）摄像机

在有关摄像机的话题中，我也需要谈到颇有价值的第三人称视角摄像机。作为一个实例，我已经在示例代码中增加了一个基于弹簧的第三人称摄像机模型。场景中的角色由一个样条进行调节。这不过是模拟一个角色可能会随机采取的朝向。在一个交互式的游戏中，实际角色的朝向能够由游戏逻辑，录制好动画，甚至于前面讨论的第一人称视角摄像机所控制。不管怎样，摄像机追踪在角色后面给定距离的一个位置。我使用弹力衰减方程来为摄像机给出一种真实感觉。

4. 四元数

最后需要注意的是四元数，它已经成为游戏编程的一个主要部分了，并在摄像机朝向技术中扮演着重要的角色。利用四元数表示朝向有许多优点。欧拉角的 3 参数表示法需要三角学和 9 个参数的正交矩阵。而四元数，却仅需 4 个参数且计算代价更小。

观察内插法时，欧拉角实现法有其固有的许多 BUG。假设你希望将物体在 Y 轴上旋转 90°（yaw = π/2）。因为每一个旋转都是独立计算的，这个操作使 X 轴旋转到了负的 Z 轴。这样，在 X 轴方向旋转一个角度 θ 与在 Z 轴方向旋转一个角度 $-\theta$ 的结果就是一样的。换句话说，当你在 yaw 上施加一个变化时，摄像机将会 roll。这种参数异常称为万向节锁（gimbal lock）。就是因为这个锁，通过这些奇异点的内插产生了奇怪且很可能不希望出现的结果。而四元数却没有这个问题，而且能被很容易地进行插值。通过以四元数来表示摄像机朝向，我们可以实现在两个视点之间的平滑插值。

更成熟的朝向控制的实现技术（如那些用于商业飞行模拟器中的技术）四元数表达用角速度。不过为达到我们的目的，用欧拉角来增加围绕某个角度的旋转就足够了，它比直接重新计算一个四元数更直观。因此，有这样一个由给定的 3 个欧拉角生成四元数的函数是非常有用的：

```
quaternion &quaternion::SetEuler(float yaw, float pitch,
float roll) {
    float cosY = cosf(yaw / 2.0F);
    float sinY = sinf(yaw / 2.0F);
    float cosP = cosf(pitch / 2.0F);
    float sinP = sinf(pitch / 2.0F);
    float cosR = cosf(roll / 2.0F);
    float sinR = sinf(roll / 2.0F);
    SetValues(
        cosR * sinP * cosY + sinR * cosP * sinY,
        cosR * cosP * sinY - sinR * sinP * cosY,
```

```
            sinR * cosP * cosY - cosR * sinP * sinY,
            cosR * cosP * cosY + sinR * sinP * sinY
        );
        return *this;
    }
```

4.4 一种快速的圆柱棱台相交测试算法

Eric Lengyel

许多游戏在试图渲染一个复杂的物体之前，首先要确定一个几何学上简单的对象包围体是否可见。由于它们的计算效率，球体和盒通常被用作包围体，但是有时对象的自然形态是适于被一个圆柱体来包围的。尽管我们达不到一个包围球或包围盒所能达到的测试速度，但本文提出了一种确定一个任意圆柱体是否潜在地与视域棱台（view frustum）相交（因而该对象是否可见）的快速算法。

本算法的效率取决于这样的事实：我们可以将问题简化为确定是否有一条线段与一个适当修正过的可视棱台相交。假设一个圆柱体由一个半径和空间中代表两个端面中心的两个点描述，我们根据该圆柱体相对于可视棱台的 6 个平面的有效半径分别将它们向外移动。有效半径依赖于圆柱体的朝向，范围从零（当圆柱体与平面垂直）到实际半径（当圆柱体与平面平行）。

当对象的包围圆柱有较大的高度与半径的比时，圆柱体测试要比球面和包围盒测试要优越。例如，当为无限远光源渲染阴影体时，由于一个阴影体通常要比产生它的物体尺寸要长，一个阴影体完全包容在光线照到物体包围球后的发散部分内。这就使得采用圆柱包围体成了一个自然的选择。因为这样的体积使用一个包围球将会包含大量的无效区，当体积实际上并不可见时，球体可见性测试却返回了正值。虽然一个包围盒通常包含一个较小数量的可接受的无效区，包围盒可见性测试对于一维尺寸比另两维尺寸大得多的盒子并不适用。这是因为包围盒测试只有测定出包围盒完全处于视域棱台 6 个面中任何一面的外边时，才判定包围盒不可见。一个长矩形的包围盒可以容易地跨过视域四棱锥远裁剪平面，但包围盒测试却会返回正值。

4.4.1 视域棱台

我们的视域棱台将完全由 4 个量描述。第一个量是焦距 l，它确定了视野。当水平视角为 θ，焦距由下面的公式给出：

$$l = \frac{1}{\tan(\theta/2)} \tag{4.4.1}$$

第二个量是宽高比 a，它是视域棱台端面的高度除以它的宽度。在与摄像机相距 l 处，一个与摄像机的视线方向垂直的平面将视域四棱锥切开，

形成一个矩形，矩形的左和右边位于 $x = \pm 1$，而它的上下边位于 $y = \pm a$。

剩下的两个描述视域棱台的量是最小和最大的深度，它们定义了近平面的距离 n 及远平面距离 f。在与摄像机相距 n 处，视锥端面矩形以 $x = \pm n/l$ 和 $y = \pm na/l$ 为界。这些值被传递给 OpenGL 函数 glFrustum。

使用 l 和 a 的值，在视点坐标系中，6 个截面的向内的单位长度的法线可以由下面的公式给出：

表 4.4.1 视域棱台面法线

平面	法线
近端面	(0, 0, − 1)
远端面	(0, 0, 1)
左端面	$\left(\frac{l}{\sqrt{l^2+1}}, 0, -\frac{1}{\sqrt{l^2+1}}\right)$
右端面	$\left(-\frac{l}{\sqrt{l^2+1}}, 0, -\frac{1}{\sqrt{l^2+1}}\right)$
上端面	$\left(0, -\frac{l}{\sqrt{l^2+a^2}}, -\frac{a}{\sqrt{l^2+a^2}}\right)$
下端面	$\left(0, \frac{l}{\sqrt{l^2+a^2}}, -\frac{a}{\sqrt{l^2+a^2}}\right)$

圆柱体相交测试将在视线空间发生，使我们可以利用这些法线的对称性和表中众多的零。

4.4.2 计算有效半径

让我们称圆柱中心轴的两个端点为 $\mathbf{P}_1$ 和 $\mathbf{P}_2$，圆柱的半径为 r（参见图 4.4.1）。现在我们选择视域棱台 6 个截平面中的一个，将它的法线标记为 $\mathbf{N}$，并且称位于该法线和圆柱体的轴之间的夹角为 α。圆柱体相对于这个平面的有效半径由下面的简单公式给出：

$$r' = r \sin \alpha \tag{4.4.2}$$

最直接的确定 $\sin \alpha$ 值的方法可能就是计算一个叉积的量值了，但我们可以通过一些三角变换用少得多的运算来计算同一个值。回忆恒等式：

$$\sin^2 \alpha + \cos^2 \alpha = 1 \tag{4.4.3}$$

它给了我们公式 2 的替换形式：

$$r' = r\sqrt{1 - \cos^2 \alpha} \tag{4.4.4}$$

$\cos \alpha$ 值由下式给出：

$$\cos\alpha = \frac{(\mathbf{P}_2 - \mathbf{P}_1)\cdot \mathbf{N}}{\|\mathbf{P}_2 - \mathbf{P}_1\|} \tag{4.4.5}$$

表示圆柱体轴的法线矢量只需要计算一次。对每一个视域棱台，可以通过简单地对一个法线求点积来计算式 4.4.5，该法线至多有 2 个非 0 组分。

计算有效半径并不是绝对必要的，因为有可能采用实际的半径，从而节省 5 个可能的开平方运算。在执行许多圆柱体可见性测试的情形下，采用实际的半径可能会比较理想，而且开平方运算在目标机器上较慢。采用实际半径的缺点是它增加了可见圆柱体的数目，如果实际的半径较大，这个增加可能是显著的。如果速度也是考虑的因素，那么应该通过实验来决定使用有效半径或实际半径。

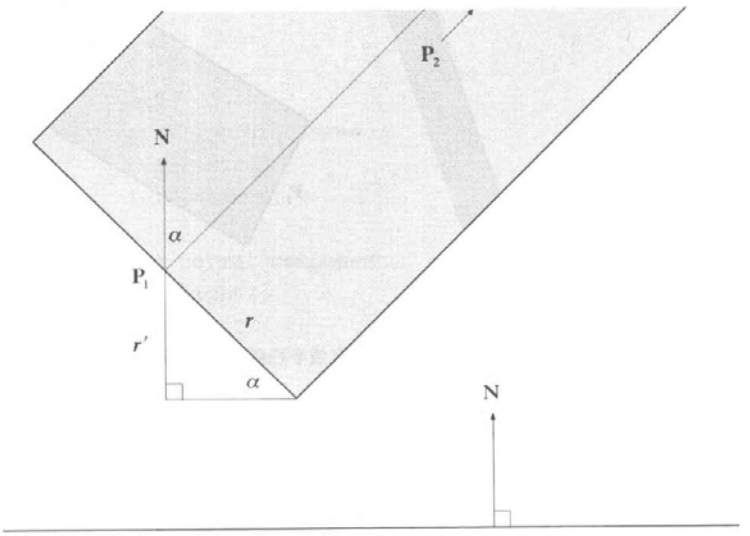


图 4.4.1 计算一个有效半径

4.4.3 算法

相交测试通过分别访问 6 个视域棱台中的每一个进行。我们首先考虑近平面和远平面，因为它们是平行的，故由此产生了同样的有效半径。一旦我们已经发现至少有一部分圆柱体位于这两个平面之间，我们就可以继续考虑 4 个侧平面了。对于每一个平面，我们首先计算圆柱体的有效半径r'并且根据此距离将平面向外移动，如图 4.4.2 所示。这样的效果是将圆柱体简化为一条线段，但是它稍微会增加一些代价，即在我们的可见集合中多包括了几个实际上并不与视域棱台相交的圆柱体。

调整一个平面之后，我们接着测试两个端点$\mathbf{P}_1$和$\mathbf{P}_2$以确定它们位于平面的哪一边。这可以通过将每个端点的坐标代入到下面的平面方程中得到：

$$\mathbf{P}\cdot\mathbf{N} - d = 0 \tag{4.4.6}$$

这里，对于近平面$d = n - r'$，对于远平面$d = -f - r'$，而对其他任何 4 个侧平面

$d = -r'$。公式 6 左边的符号表明了点 P 位于平面的哪一侧。由于平面的法线指向视域棱台的内部，任何位于平面反侧的点就会位于视域棱台的外部。这样，$\mathbf{P}_1$ 和 $\mathbf{P}_2$ 如果都位于平面的反侧，那么我们立即就会知道该圆柱体是不可见的，算法结束。任何视域棱台内部的点必定位于所有 6 个平面的正侧，所以只要 $\mathbf{P}_1$ 和 $\mathbf{P}_2$ 都位于单个平面的正侧，我们就得不出任何结论，而只好继续测试下一个平面了。

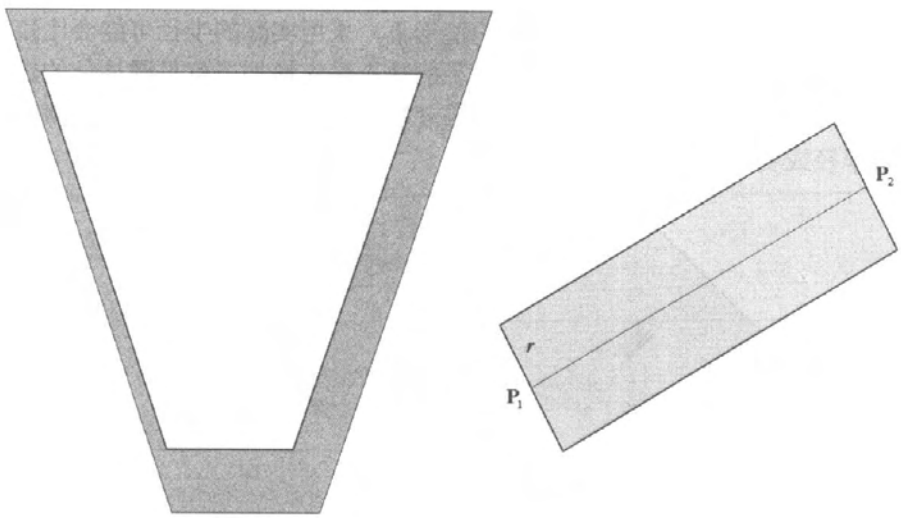


图 4.4.2 阴影区域代表每个截面膨胀有效半径后，视域棱台增加的体积

剩下的情形是一个端点位于正侧而另一个位于反侧，这时我们计算线段与平面的交点，并且用它来代替外部的那个端点。这会有效地去除我们所知道的处于视域棱台外部的圆柱部分。为了找到相交点，我们使用直线的参数方程：

$$\mathbf{P} = \mathbf{P}_1 + t(\mathbf{P}_2 - \mathbf{P}_1) \tag{4.4.7}$$

这里 $0 \le t \le 1$。用这个公式的右边来代换公式 6 中的 $\mathbf{P}$，我们可以解出交点的 t 值：

$$t = \frac{d - \mathbf{P}_1 \cdot \mathbf{N}}{(\mathbf{P}_2 - \mathbf{P}_1) \cdot \mathbf{N}} \tag{4.4.8}$$

把它代回到方程 7 中就得到了新的端点。在用它替换了外部的端点之后，我们继续对下一个平面进行测试。

如果我们已经访问完了视域棱台的 6 个平面，并且未遇到两个端点都位于一个平面的反侧的情形，那么圆柱体至少是部分可见的。当然，这意味着我们不必为访问的最后一个平面替换任何端点。我们只要知道至少有一个端点位于最后的一个平面的正侧，就知道圆柱体部分与视域棱台相交了。

4.4.4 实现

程序清单 4.4.1 中的示例代码实现了圆柱体可见性测试。Frustum 类封装了视域棱台并且通过指定局部长度、宽高比、近平面距离及远平面距离来构造。表 4.4.1 中所列法线的成分在构造器内部预先计算。成员函数 CylinderVisible 确定了由两个点和一个半径决定的圆柱体是否与视域棱台相交并返回 true 或 false。

程序清单 4.4.1

```
#include "mtxlib.h"

class Frustum
{
    private:

        // Near and far plane distances
        float        nearDistance;
        float        farDistance;

        // Precalculated normal components
        float        leftRightX;
        float        leftRightZ;
        float        topBottomY;
        float        topBottomZ;

    public:

        // Constructor defines the frustum
        Frustum(float l, float a, float n, float f);

        // Intersection test returns true or false
        bool CylinderVisible(vector3 p1, vector3 p2,
float radius) const;
};

Frustum::Frustum(float l, float a, float n, float f)
{
    // Save off near plane and far plane distances
    nearDistance = n;
    farDistance = f;

    // Precalculate side plane normal components
    float d = 1.0F / sqrt(l * l + 1.0F);
    leftRightX = l * d;
    leftRightZ = d;

    d = 1.0F / sqrt(l * l + a * a);
    topBottomY = l * d;
```

```
    topBottomZ = a * d;
}

bool Frustum::CylinderVisible(vector3 p1, vector3 p2, float radius) const
{
    // Calculate unit vector representing cylinder's axis
    vector3 dp = p2 - p1;
    dp.normalize();

    // Visit near plane first, N = (0,0,-1)
    float dot1 = -p1.z;
    float dot2 = -p2.z;

    // Calculate effective radius for near and far planes
    float effectiveRadius = radius * sqrt(1.0F - dp.z * dp.z);

    // Test endpoints against adjusted near plane
    float d = nearDistance - effectiveRadius;
    bool interior1 = (dot1 > d);
    bool interior2 = (dot2 > d);

    if (!interior1)
    {
        // If neither endpoint is interior,
        // cylinder is not visible
        if (!interior2) return (false);

        // p1 was outside, so move it to the near plane
        float t = (d + p1.z) / dp.z;
        p1.x -= t * dp.x;
        p1.y -= t * dp.y;
        p1.z = -d;
    }
    else if (!interior2)
    {
        // p2 was outside, so move it to the near plane
        float t = (d + p1.z) / dp.z;
        p2.x = p1.x - t * dp.x;
        p2.y = p1.y - t * dp.y;
        p2.z = -d;
    }

    // Test endpoints against adjusted far plane
    d = farDistance + effectiveRadius;
    interior1 = (dot1 < d);
    interior2 = (dot2 < d);

    if (!interior1)
    {
        // If neither endpoint is interior,
```

```
        // cylinder is not visible
        if (!interior2) return (false);

        // p1 was outside, so move it to the far plane
        float t = (d + p1.z) / (p2.z - p1.z);
        p1.x -= t * (p2.x - p1.x);
        p1.y -= t * (p2.y - p1.y);
        p1.z = -d;
    }
    else if (!interior2)
    {
        // p2 was outside, so move it to the far plane
        float t = (d + p1.z) / (p2.z - p1.z);
        p2.x = p1.x - t * (p2.x - p1.x);
        p2.y = p1.y - t * (p2.y - p1.y);
        p2.z = -d;
    }

    // Visit left side plane next
    // The normal components have been precalculated
    float nx = leftRightX;
    float nz = leftRightZ;

    // Compute p1 * N and p2 * N
    dot1 = nx * p1.x - nz * p1.z;
    dot2 = nx * p2.x - nz * p2.z;

    // Calculate effective radius for this plane
    float s = nx * dp.x - nz * dp.z;
    effectiveRadius = -radius * sqrt(1.0F - s * s);

    // Test endpoints against adjusted plane
    interior1 = (dot1 > effectiveRadius);
    interior2 = (dot2 > effectiveRadius);

    if (!interior1)
    {
        // If neither endpoint is interior,
        // cylinder is not visible
        if (!interior2) return (false);

        // p1 was outside, so move it to the plane
        float t = (effectiveRadius - dot1) / (dot2 - dot1);
        p1.x += t * (p2.x - p1.x);
        p1.y += t * (p2.y - p1.y);
        p1.z += t * (p2.z - p1.z);
    }
    else if (!interior2)
    {
        // p2 was outside, so move it to the plane
```

```
        float t = (effectiveRadius - dot1) / (dot2 - dot1);
        p2.x = p1.x + t * (p2.x - p1.x);
        p2.y = p1.y + t * (p2.y - p1.y);
        p2.z = p1.z + t * (p2.z - p1.z);
    }

    // Visit right side plane next
    dot1 = -nx * p1.x - nz * p1.z;
    dot2 = -nx * p2.x - nz * p2.z;

    s = -nx * dp.x - nz * dp.z;
    effectiveRadius = -radius * sqrt(1.0F - s * s);

    interior1 = (dot1 > effectiveRadius);
    interior2 = (dot2 > effectiveRadius);

    if (!interior1)
    {
        if (!interior2) return (false);

        float t = (effectiveRadius - dot1) / (dot2 - dot1);
        p1.x += t * (p2.x - p1.x);
        p1.y += t * (p2.y - p1.y);
        p1.z += t * (p2.z - p1.z);
    }
    else if (!interior2)
    {
        float t = (effectiveRadius - dot1) / (dot2 - dot1);
        p2.x = p1.x + t * (p2.x - p1.x);
        p2.y = p1.y + t * (p2.y - p1.y);
        p2.z = p1.z + t * (p2.z - p1.z);
    }

    // Visit top side plane next
    // The normal components have been precalculated
    float ny = topBottomY;
    nz = topBottomZ;

        dot1 = -ny * p1.y - nz * p1.z;
    dot2 = -ny * p2.y - nz * p2.z;

        s = -ny * dp.y - nz * dp.z;
    effectiveRadius = -radius * sqrt(1.0F - s * s);

        interior1 = (dot1 > effectiveRadius);
    interior2 = (dot2 > effectiveRadius);

        if (!interior1)
    {
        if (!interior2) return (false);

        float t = (effectiveRadius - dot1) / (dot2 - dot1);
```

```
            p1.x += t * (p2.x - p1.x);
            p1.y += t * (p2.y - p1.y);
            p1.z += t * (p2.z - p1.z);
        }
        else if (!interior2)
        {
            float t = (effectiveRadius - dot1) / (dot2 - dot1);
            p2.x = p1.x + t * (p2.x - p1.x);
            p2.y = p1.y + t * (p2.y - p1.y);
            p2.z = p1.z + t * (p2.z - p1.z);
        }

            // Finally, visit bottom side plane
        dot1 = ny * p1.y - nz * p1.z;
        dot2 = ny * p2.y - nz * p2.z;

            s = ny * dp.y - nz * dp.z;
        effectiveRadius = -radius * sqrt(1.0F - s * s);

            interior1 = (dot1 > effectiveRadius);
        interior2 = (dot2 > effectiveRadius);

            // At least one endpoint must be interior
        // or cylinder is not visible
        return (interior1 | interior2);
    }
```

4.5 3D 碰撞检测

Kevin Kaiser

实时物理引擎是创建一个使玩家能很容易地打消疑虑的 3D 游戏环境的决定性因素。物理引擎不是仅提供真实的图像效果，而是要提供在图像中物体之间的真实互动。这种交互作用为玩家提供了真实性的基础；换句话说，当事物表现得和它们在真实世界中一样时，玩家能更好地理解和游览世界。首先且可以证明的是建立一个实时物理模拟最重要的步骤是要有精确的碰撞检测；一旦碰撞被检测出来，模拟就可以相应地做出反应了。本文将有助于为建立一个精确的物理模拟打下基础，从一个实时物理引擎的至关紧要的部分开始：3D 碰撞检测。

4.5.1 算法概述

本文讨论的两个基本的碰撞算法是：

- 包围球（bounding sphere）碰撞检测——为了使编码简洁和说明易于理解，我们将例用包围球。该代码的主要部分是针对于另一个包围球的半径来检验一个包围球的半径，以确定可能的碰撞。
- 三角形对三角形的碰撞检测——在试图弄明白这个算法之前，也许你需要好好复习一下微积分学；它使用参数方程来确定一个三角形与另一个三角形的平面的碰撞点，然后确定这些碰撞点是否位于对方三角形内。

4.5.2 包围球碰撞检测

碰撞检测最好以分级步骤执行：对象包围球对对象包围球、多边形包围球对多边形包围球，然后是三角形对三角形。我们将从生成包围球开始讨论。计算包围球非常简单；所要做的就是找到对象的中心，然后计算从中心到对象上的一个顶点的最大距离。在存储了每一个包围球半径的情况下，你可以通过对两个对象的半径加和，然后取两个中心点之间的距离，来进行包围球碰撞检测。如果距离比两个半径的和还大，这两个球必定不碰撞。

让我们一步步执行一下。首先，需要确定网格的中心。一种做法是创建一个包围盒（bounding box）并寻找两个对角线的中点（见图 4.5.1）。为了计算包围盒，你需要找到整个对象的最小和最大的 x，y 和 z 值。这可以由对顶点进行迭代并保存一个“当前”最小值和最大值来完成。检验完

所有的顶点之后，将得到包围盒的最大长度。最小和最大值将用来创建包围盒。

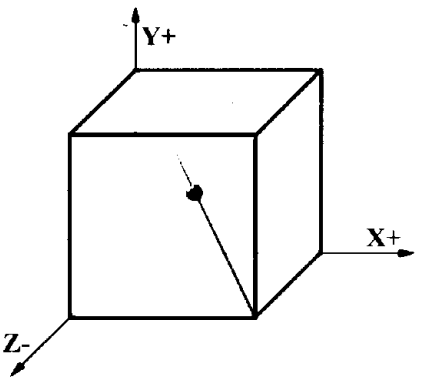


图 4.5.1 寻找中心点

假设一个包围盒有 8 个最远顶点（ABCDEFGH，参见图 4.5.2），让我们给它们分配坐标：

```
A = (minx, miny, minz)
B = (minx, maxy, minz)
C = (maxx, maxy, minz)
D = (maxx, miny, minz)
E = (minx, miny, maxz)
F = (minx, maxy, maxz)
G = (maxx, maxy, maxz)
H = (maxx, miny, maxz)
```

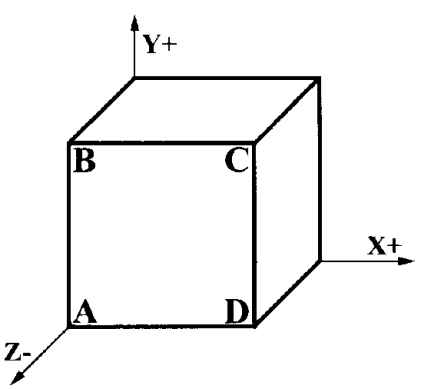


图 4.5.2 创建一个包围盒

现在寻找中心点，它可以通过计算包围盒上的最大点和最小点（由 A 和 G 表示）的平均值得到。

```
// Midpoint formula: Given A(x1,y1,z1) and B(x2,y2,z2),
//   the midpoint of the line the passes through A and B is
//   [(x1+x2)/2, (y1+y2)/2, (z1+z2)/2]
```

```
//
center.x = (A.x + G.x)/2;
center.y = (A.y + G.y)/2;
center.z = (A.z + G.z)/2;
```

包围球的半径可以很容易地通过对物体的顶点进行循环计算出来，每次循环都计算中心点与当前顶点之间的距离。如果该距离比当前最大距离值还大，就以新的距离值替换最大距离。循环结束后，最大距离就是包围球的半径了。(当然，一种简单的优化是只对特殊端点进行开平方运算。)

```
// Distance formula:
//  dist = sqrt[ ((x2-x1)^2)+((y2-y1)^2)+((z2-z1)^2) ]
//  distsq = ((x2-x1)^2)+((y2-y1)^2)+((z2-z1)^2)
//
foreach vertex v in object {
  current_distance_sq = distsq(object.center, v);
  if (current_distance_sq > max_distance_sq)
    max_distance_sq = current_distance_sq;
}
object.bs_radius = sqrt(max_distance_sq);
```

紧接着会在多边形级别重复这样的过程。包围球检验非常快速而简单，这也说明当你不得不对大量的多边形相对于彼此进行检验时，为什么从这个检测开始是比较有利的。当已经生成了必要的包围盒、包围球，以及每一个对象和多边形的中心点之后，你就可以准备深入本文的实质内容了：三角形对三角形的相交检测！记得把你的微积分书拿出来——你也许会需要它。

4.5.3 三角形对三角形的碰撞检测

这种三角形对三角形碰撞检测的方法直接依赖于一些易于理解但需要技巧的数学知识。想像一下在3D空间给定两个三角形(参见图4.5.3)。我们需要从这两个三角形收集许多信息。如果你记得不错，平面方程是 $Ax+By+Cz+D=0$。我们要通过取顶点的叉积来确定 A、B、C 和 D。

```
// given a triangle tri1 with vertices a, b and c.
// vector3 a, b, c;
vector3 v1, v2, cross_v1xv2;

// create vectors v1, v2 (tri1.b - tri1.a,
//                        tri1.c - tri1.a)
v1 = tri1.b - tri1.a;
v2 = tri1.c - tri1.a;

// NOTE: You may be able to skip this step and substitute your
// own surface normals if you already have them stored somewhere.
// Take cross product of v1 and v2 (this is the normal
// vector of the cross product of v1 and v2)
```

```
cross_v1xv2 = CrossProduct(v1, v2);

// Then we plug these values back into Ax+By+Cz+D=0
tri1.pA = cross_v1xv2.x;
tri1.pB = cross_v1xv2.y;
tri1.pC = cross_v1xv2.z;

// Following this rule: Ax+By+Cz+D=0
// if point P(x0,y0,z0) is a point on the polygon
// A = cross_v1xv2.x
// B = cross_v1xv2.y
// C = cross_v1xv2.z
// D = (-A*x0-B*y0-C*z0)
tri1.pD = -DotProduct(cross_v1xv2, P);
```

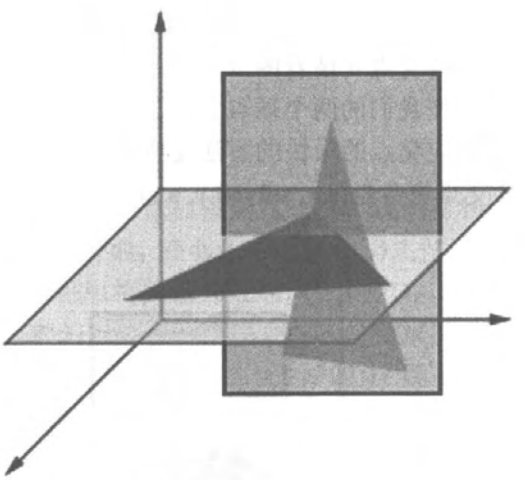


图 4.5.3 两个相交三角形

1. 直线与平面的交点

现在我们有了三角形 1 的平面方程，可以接着进行第二步了：看看三角形 2 是否与三角形 1 的平面碰撞。这需要多个步骤才能完成。主要思想是给定三角形 2 的两个顶点，取这两个点定义的直线并确定该直线与三角形 1 的平面的碰撞点。如果碰撞点是在两个顶点之间，则三角形 2 与三角形 1 的平面碰撞；如果碰撞点不在两个顶点之间，我们就通过三角形 2 的其他两条直线进行迭代，看看在那些点之间是否有碰撞点。

我们使用微积分参数方程来解决这种直线与平面的交点问题。给定两个顶点，$\mathbf{a}(x0,y0,z0)$ 和 $\mathbf{b}(x1,y1,z1)$，令 $\mathbf{a}(x0,y0,z0)*t = \mathbf{b}(x1,y1,z1)*(1-t)$。$t$ 是一个范围从 0 到 1 的插值因子。当 $t=0$ 时，位于 ***b*** 点，而当 $t=1$ 时位于点 a。如果我们将参数方程代入平面方程的每一个分量，可以解出 t：

$$A*(x0*t + x1*(1-t)) + B*(y0*t + y1*(1-t)) + C*(z0*t + z1*(1-t)) + D = 0$$

它可以化简为：

$$t = -(A*x1 + B*y1 + C*z1 + D) / (A*(x0-x1) + B*(y0-y1) + C*(z0-z1))$$

下面是求解 t 的代码：

```
// i0 = (A*x0) + (B*y0) + (C*z0)
i0 = (tri1->pA*a->x) + (tri1->pB*a->y) + (tri1->pC*a->z);
// i1 = (A*x1) + (B*y1) + (C*z1)
i1 = (tri1->pA*b->x) + (tri1->pB*b->y) + (tri1->pC*b->z);
// Be wary of possible divide-by-zeros here (i.e. if i0 == i1)
final_t = -(i1 + tri1->pD) / (i0-i1);

// Then plug final_t back into the functions x(), y() and z()
// to get the point of intersection from line to plane
final_x = (((a->x)*(final_t))+((b->x)*(1-final_t)));
final_y = (((a->y)*(final_t))+((b->y)*(1-final_t)));
final_z = (((a->z)*(final_t))+((b->z)*(1-final_t)));
```

这将给出直线与平面的最终交点（请看图 4.5.4）。当然，我们已经计算出的 t 值必须是在 0 和 1 之间，否则交点就不在我们的两个顶点之间了！在这个步骤中你还需要考虑的一个特殊情况是垂线段的存在。确定交点的最快的方法是将点 a 或点 b 的 x 值和 y 值代入三角形的平面方程并解出 y。那么交点将会是(**a**.x, **solved**.y, **a**.z)。

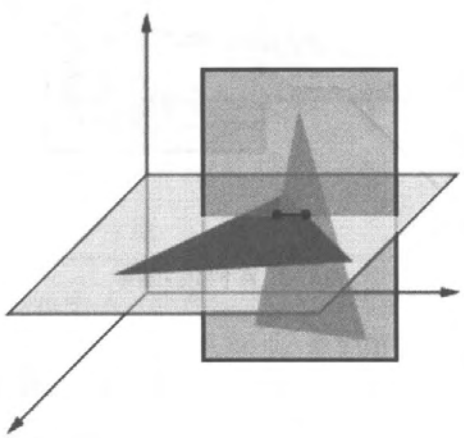


图 4.5.4 确定碰撞点

2. 三角形“平化（Flattening）”

现在假设使用的是右手坐标系。想像将三角形相对于一个坐标面依三角形的朝向进行平化。它可能会丢失掉 y 坐标并保持 x 和 z 坐标。这里引入这个概念并不是专为丢掉 y 坐标，只是需要丢掉适当的坐标以使它变平。决定去掉哪个坐标的一个好办法是观察平面的法线；如果你确定哪个分量的绝对值最大，那么就能找到一个平面，以它来平化一个三角形，使三角形不会成为一条“直线”（比如一个直角三角形丢掉了 y 坐标）。例如，如果 x 分量最大，你将投影到 yz 平面。不论朝向如何，这样产生的三角形将是平伸的（就好像它是平躺在一张桌子上，参见图 4.5.5）。这是非常有利的，因为现在我们可以利用基本代数学来检验，当以

一种类似的方法进行平化时，最终的交点是否位于平化过的三角形内部。程序清单 4.5.1 中的代码将多边形相对于一个坐标平面进行了有效的平化。

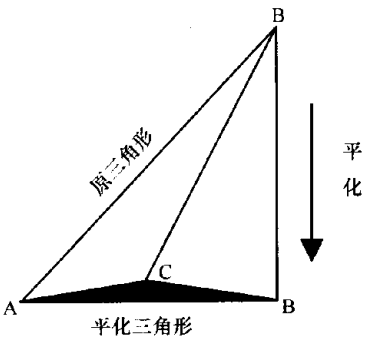


图 4.5.5 顶点投影

3. 点在三角形中的测试

现在已经将坐标平化了，我们需要做一些代数方面的工作来决定平化的交点是不是位于平化的三角形内部。有几种流行的方法能做到这一点；我们将利用平化三角形的每条直线的方程。注意不管你投影到哪一个平面，在我们的讨论中仍然仅涉及平化过的点的 x 和 y 坐标。这是因为通过投影顶点，我们可以将问题有效地简化到 2D；即 x 和 y 坐标。首先，我们需要寻找一个一定位于三角形内部的点。最容易找到的满足要求的点就是三角形的中心，它可以作为顶点的平均值被计算：$((x0+x1+x2)/2, (y0+y1+y2)/2)$。现在我们知道了三角形内部是什么方向了，需要看看点是否在由每一对顶点所决定的直线的“内”侧（参见图 4.5.6）。

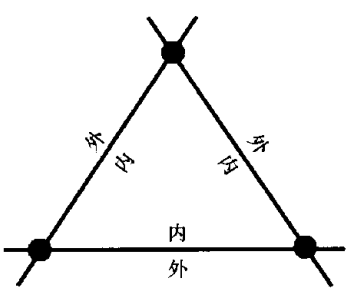


图 4.5.6

给定两个顶点 **v0** 和 **v1**，首先寻找通过它们的形如 $y=mx+b$ 的直线方程。记住斜率（m）的公式是 $(y1-y0)/(x1-x0)$，并且你可以通过计算过的斜率值和直线上的一个已知点来得到 b 的值。现在我们有了斜截式（slope-intercept form）的直线方程了，可以确定平化后的交点是否位于直线的相当于三角形的内侧的那一侧。可以由比较 y 值做到这一点。如果你将平化过的交点的 x 的坐标代入到直线 $y=mx+b$ 中，将得到直线在 x 处的 y 值。接着，通过检验 y 值确定我们先前计算过的中心点是在直线“上方”还是“下方”。我们知道它是在三角形内部的，于是我们的交点必须有一个与中心点相同的关于直线的“方向”的 y 值。如果有，则交点是

相对于直线 ab 在三角形的“内部”的。为直线 bc 和 ca 重复上面的步骤。如果每次测试后交点都在内侧，则该点必定位于三角形的内部。还必须考虑一个特殊情形：一个投影过的垂线段。不能用 $y=mx+b$ 表示垂线。如果遇到这种情况，你可以检验 x 坐标而不是 y 坐标，即首先要确定垂线的哪一侧是内侧。然后，检验投影过的交点的 x 值。如果它是在垂线的“内”侧，那么它就在相对于直线的内部。请看程序清单 4.5.2 中的实现过程。

4. 检验在两个三角形内的所有直线

当然，如果三角形的一条直线没有相交，仍然需要检验其他直线。一个直线/三角形碰撞完全能够表明两个三角形是碰撞的。请看程序清单中这个示例代码的剩余部分。

做完所有这些之后，如果没有碰撞发生，就需要颠倒一下，以三角形 2 为源重新开始进行这个过程。这样能确保一个完备的碰撞检测。

程序清单 4.5.1

```
if (x==FALSE) { // dropping x coordinate
  a1 = tri->a.y;
  b1 = tri->a.z;
  a2 = tri->b.y;
  b2 = tri->b.z;
  a3 = tri->c.y;
  b3 = tri->c.z;
  a4 = vert->y;
  b4 = vert->z;
  inside = 0;
}
else if (y==FALSE) { // dropping y coordinate
  a1 = tri->a.x;
  b1 = tri->a.z;
  a2 = tri->b.x;
  b2 = tri->b.z;
  a3 = tri->c.x;
  b3 = tri->c.z;
  a4 = vert->x;
  b4 = vert->z;
  inside = 0;
}
else if (z==FALSE) { // dropping z coordinate
  a1 = tri->a.x;
  b1 = tri->a.y;
  a2 = tri->b.x;
  b2 = tri->b.y;
  a3 = tri->c.x;
  b3 = tri->c.y;
  a4 = vert->x;
  b4 = vert->y;
  inside = 0;
}
```

程序清单 4.5.2

```
// These are used to check for vertical line segments in the
// flattened triangle; you cannot graph vertical lines in 2D
// using y=mx+b, so we have to instead check if the flattened
// intersection point lies between the x coordinates of any
// vertical line and the center point of the triangle to see if
// the flattened intersection point lies on the inside of the
// triangle with respect to the vertical line segment.
AB_vert = BC_vert = CA_vert = FALSE;

// y=mx+b for outer 3 lines
if ((a2-a1)!=0) {
  m1 = (b2-b1)/(a2-a1);  // a->b
  bb1 = (b1)-(m1*a1);    // y/(mx) using vertex a
} else if ((a2-a1)==0) {
 AB_vert = TRUE;
}

if ((a3-a2)!=0) {
  m2 = (b3-b2)/(a3-a2);  // b->c
  bb2 = (b2)-(m2*a2);    // y/(mx) using vertex b
} else if ((a3-a2)==0) {
 BC_vert = TRUE;
}

if ((a1-a3)!=0) {
  m3 = (b1-b3)/(a1-a3);  // c->a
  bb3 = (b3)-(m3*a3);    // y/(mx) using vertex c
} else if ((a1-a3)==0) {
 CA_vert = TRUE;
}

// find average point of triangle (point is guaranteed
center_x = (a1+a2+a3)/3;       // to lie inside the triangle)
center_y = (b1+b2+b3)/3;

// See whether (center_x,center_y) is above or below the line,
// then set direction to UP if the point is above or DOWN if the
// point is below the line

// a->b
if (((m1*center_x)+bb1) >= center_y)
  DIRECTION(direction,UP);
else
  DIRECTION(direction,DOWN);
if (AB_vert==TRUE) {
  if ((a1<a4)&&(a1<center_x)) // vert projected line
    inside++;
  else if ((a1>a4)&&(a1>center_x)) // vert projected line
```

```
    inside++;
  } else {
    if (direction==UP) {
      if (b4 <= ((m1*a4)+bb1)) // b4 less than y to be inside
        inside++;              // (line is above point)
    } else if (direction==DOWN) {
      if (b4 >= ((m1*a4)+bb1)) // b4 greater than y to be inside
        inside++;              // (line is below point)
    }
  }

  // b->c
  if (((m2*center_x)+bb2) >= center_y)
    DIRECTION(direction,UP);
  else
    DIRECTION(direction,DOWN);
  if (BC_vert==TRUE) {
    if ( (a2 < a4)&&(a2 < center_x) ) // vert projected line
      inside++;
    else if ( (a2 > a4)&&(a2 > center_x) ) // vert projected line
      inside++;
  } else {
    if (direction==UP) {
      if (b4 <= ((m2*a4)+bb2)) // b4 less than y to be inside
        inside++;              // (line is above point)
    } else if (direction==DOWN) {
      if (b4 >= ((m2*a4)+bb2)) // b4 greater than y to be inside
        inside++;              // (line is below point)
    }
  }

  // c->a
  if (((m3*center_x)+bb3) >= center_y)
    DIRECTION(direction,UP);
  else
    DIRECTION(direction,DOWN);
  if (CA_vert==TRUE) {
    if ( (a3 < a4)&&(a3 < center_x) ) // vert projected line
      inside++;
    else if ( (a3 > a4)&&(a3 > center_x) ) // vert projected line
      inside++;
  } else {
    if (direction==UP) {
      if (b4 <= ((m3*a4)+bb3)) // b4 less than y to be inside
        inside++;              // (line is above point)
    } else if (direction==DOWN) {
      if (b4 >= ((m3*a4)+bb3)) // b4 greater than y to be inside
        inside++;              // (line is below point)
    }
  }
```

```
if (inside==3) {
  return TRUE;
} else {
  return FALSE;
}
```

程序清单 4.5.3

```
// Scroll thru 3 line segments of the other triangle
// First iteration  (a,b)
p=line_plane_collision((vertex_ptr)&tri2.a,(vertex_ptr)&tri2.b,
                                          (triangle_ptr)&tri1);

// Determine which axis to project to
// X is greatest
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          TRUE,TRUE,FALSE);
// Y is greatest
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          TRUE,FALSE,TRUE);
// Z is greatest
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          FALSE,TRUE,TRUE);

if (temp==TRUE) {
  // Point needs to be checked to see if it lies between the two
  //  vertices.
  // First check for the special case of vertical line segments
  if ((tri2.a.x == tri2.b.x)&&(tri2.a.z == tri2.b.z)) {
    if (((tri2.a.y <= p.y)&&(p.y <= tri2.b.y))||
        ((tri2.b.y <= p.y)&&(p.y <= tri2.a.y)))
      return TRUE;
  }
  // End vertical line segment check

  // Now check for point on line segment
  if (point_inbetween_vertices((vertex_ptr)&tri2.a,
(vertex_ptr)&tri2.b,(triangle_ptr)&tri1)==TRUE)
    return TRUE;
  else
    return FALSE;
}

// Second iteration (b,c)
p=line_plane_collision((vertex_ptr)&tri2.b,(vertex_ptr)&tri2.c,
                                      (triangle_ptr)&tri1);

// Determine which axis to project to
```

```
// X is greatest
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          TRUE,TRUE,FALSE);
// Y is greatest
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          TRUE,FALSE,TRUE);
// Z is greatest
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          FALSE,TRUE,TRUE);

if (temp==TRUE) {
  // Point needs to be checked to see if it lies between the two vertices
  // First check for the special case of vertical line segments
  if ((tri2.b.x == tri2.c.x)&&(tri2.b.z == tri2.c.z)) {
    if (((tri2.b.y <= p.y)&&(p.y <= tri2.c.y))||
        ((tri2.c.y <= p.y)&&(p.y <= tri2.b.y)))
      return TRUE;
  }

  // Now check for point on line segment
  if (point_inbetween_vertices((vertex_ptr)&tri2.b,
(vertex_ptr)&tri2.c, (triangle_ptr)&tri1)==TRUE)
    return TRUE;
  else
    return FALSE;
}

// Third iteration (c,a)
p=line_plane_collision((vertex_ptr)&tri2.c,(vertex_ptr)&tri2.a,
                                    (triangle_ptr)&tri1);

// Determine which axis to project to
// X is greatest
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          TRUE,TRUE,FALSE);
// Y is greatest
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          TRUE,FALSE,TRUE);
// Z is greatest
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                          FALSE,TRUE,TRUE);

if (temp==TRUE) {
  // Point needs to be checked to see if it lies between the two vertices
```

```
  // First check for the special case of vertical line segments
  if ((tri2.c.x == tri2.a.x)&&(tri2.c.z == tri2.a.z)) {
    if (((tri2.c.y <= p.y)&&(p.y <= tri2.a.y))||
        ((tri2.a.y <= p.y)&&(p.y <= tri2.c.y)))
      return TRUE;
  }

  // Now check for point on line segment
  if (point_inbetween_vertices((vertex_ptr)&tri2.c,
(vertex_ptr)&tri2.a, (triangle_ptr)&tri1)==TRUE)
    return TRUE; // Intersection point is inside the triangle and on
  else          // the line segment
    return FALSE;
}
return FALSE; // Default value/no collision
```

4.6 用于交互检测的多分辨率地图

Jan Svarovsky

本文描述了一种减少接近测试（proximity test）数目的方法，如果游戏具有大量大小不同的对象，则必须执行接近测试。简单地为每一个对象相对于其他对象进行测试，其代价将以对象数目的平方阶增加，会变得非常大！若邻近测试的代价昂贵会尤其糟糕。

4.6.1 使用栅格

简单的解决方案是用一个基于栅格的地图对世界进行切分。每个栅格正方形都有一个对象的连接表，这些对象的中心位于该栅格上。因为对象大小都是非零的，它们可能会在邻接地图栅格内重叠。到了要搜索所有对象间可能的碰撞的时候，每个对象仅须对连接表上位于其后的以及其东、东南和正南相关联的对象进行测试。任何对于北方和西方以及与连接表中较早的对象的碰撞都在其他对象进行检验时被检测过了。这使你可以避免对同一碰撞检验两次。

4.6.2 对象大小变化的问题

当对象的大小变化幅度很大时，这个方法有一些问题。这种方法只是保证了如果你的游戏对象比地图栅格要小的话，能够发现所有的碰撞。如果有较大的游戏对象，可以将地图栅格做得更大。然而，这将意味着那些更小的实际上分隔很远的对象却要相对于彼此进行测试，我们可以设计一个更好的栅格系统以避免这种情况的发生，见图 4.6.1。

如果你将地图栅格做得比一些游戏对象要小，就有这样的危险：由于对象在地图栅格上离得很远以至于不会互相检验，其碰撞就没有被检测出来，而事实上它们确实碰撞了。

可以通过给每一个对象“脚”来解决这个问题。这里，对象不是直接位于地图中（除非它们足够小）；而是更小的辅助对象位于该对象接触的地图栅格中。这些脚的管理很简单，不过有些笨拙（如图 4.6.2）。

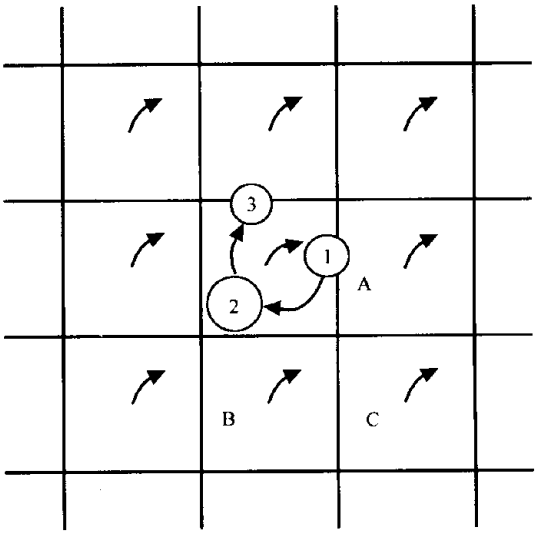


图 4.6.1 对象 2 相对于对象 3 及地图栅格 A，B 和 C 进行检验

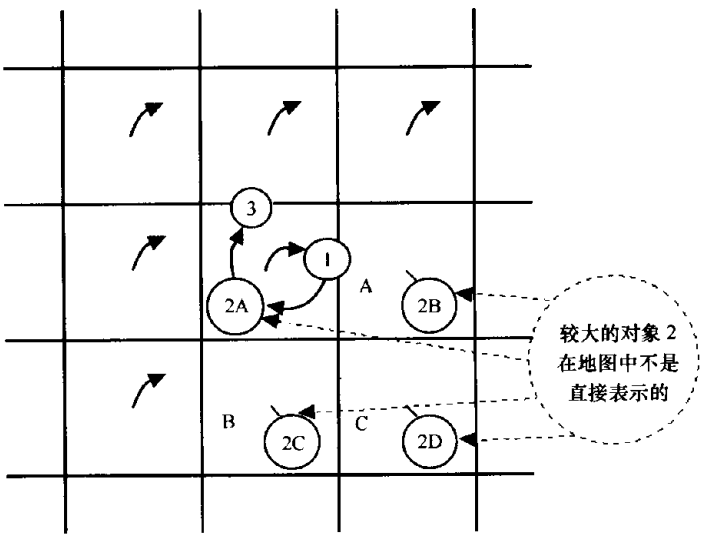


图 4.6.2 较大的对象 2 现在有脚了

4.6.3 多分辨率地图

这里我建议使用一种有多个地图分辨率的替换方法。地图栅格的尺寸以 2 次幂增长以使两个不同坐标系统之间的转换变得更为容易。每个对象都位于地图中，栅格取可能的最小值，虽然事实上可能会比对象要大。当进行碰撞检测时，你不仅要在自己的地图上，而且要在接触到的分辨率更低的地图栅格上（更大的地图栅格）进行检验。与只检验连接表上位于后面的对象很类似，你不必在更高分辨率的地图上进行检验。更小的对象当进行它们自己的检验的时候将会找到你，所以你不必去搜索它们（如图 4.6.3）。

最简单的是得到所有地图栅格的分辨率，一直低到用一个地图栅格覆盖整个世界。每个

更低分辨率的地图的数据会减少4倍，所以需要的存储量将越来越小。以我的经验，如果你在某个水平上中止分辨率的下降，可能常常会在游戏开发中发现一个游戏对象（比如一个爆炸的特效球）只不过大了一点儿，却使你的游戏完全崩溃了。持续使地图分辨率下降的惟一额外代价是需要搜索完所有的地图，直到最低分辨率的一个。

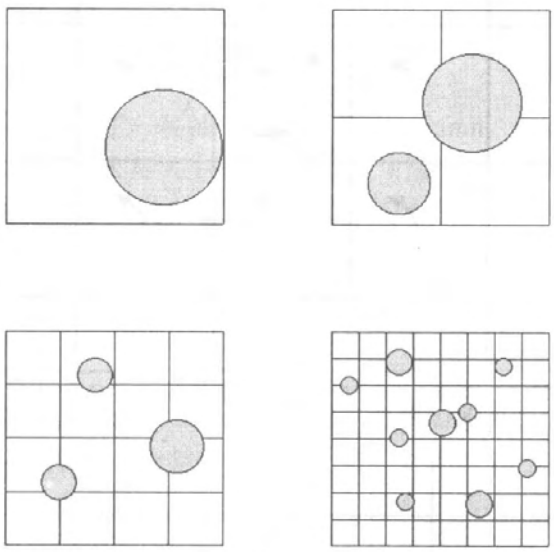


图4.6.3 多个简单地图：每个对象位于可能的最适合的分辨率地图上

你可以简单地为分辨率增加一个较低的限度，即做一个判定——在物体不能与其他对象正确碰撞时，你不允许更大的对象吗？或者是将它们放入到最低分辨率地图栅格？我建议允许它们位于最低分辨率地图中，假设这些对象只是在游戏开发过程中太大了，到了发行的时候你将需要调整地图栅格的大小来适应游戏中用到的最大对象。

4.6.4 源代码

```
#include <stdio.h>
#include <assert.h>
#include "mtxlib.h"

// This is your game object base class
class GameObject;

/////////////////////////////////////////////////////////////////////////
//
// External definitions

// If the map decides two objects are close enough together, it will
//   call this function, which you have to provide
extern void process_collision(GameObject *a, GameObject *b);

/////////////////////////////////////////////////////////////////////////
```

```
//
// A game object. Derive your own objects off this
class GameObject
{
public:
    GameObject()
    {
        NextInMap = NULL;
        MapSquare = NULL;

        MapRes = 0;
    }

    // The object is in a singly linked list hanging off one of
    //  the map squares
    GameObject *NextInMap;

    // And this is the map square that this object is hanging off,
    //  ie the start of that list
    GameObject **MapSquare;

    // The resolution of the map the object is sitting in
    int MapRes;

    // Take the object out of the map's linked list
    void RemoveFromMap();

    // calls "process_collision" on all the relevant other objects in
    //  the map, as per the article
    void ProcessCollisions(class Map *my_map);

private:

    // Do one resolution of map, used by ProcessCollisions()
    void ProcessOneLevel(Map *my_map, GameObject **map_who,
    GameObject *walker, int current_res);
};

////////////////////////////////////////////////////////////////////////
//
// The map.

// for efficiency's sake, the map dimensions etc are constants, you
//  could simply turn them into variables if you so wished

// (1 << this) is number of map squares at highest res
#define MAP_HI_RES_SHIFT        (8)

// and the number of map squares at the lowest res
#define MAP_LO_RES_SHIFT        (4)
```

```
// smallest size of a map square
#define MAP_SMALLEST_SQUARE_SIZE_SHIFT (8)
#define MAP_SMALLEST_SQUARE_SIZE      (1 << \
MAP_SMALLEST_SQUARE_SIZE_SHIFT)

// largest
#define MAP_BIGGEST_SQUARE_SIZE_SHIFT (MAP_SMALLEST_SQUARE_SIZE_SHIFT +
MAP_HI_RES_SHIFT \
                                    - MAP_LO_RES_SHIFT)
#define MAP_BIGGEST_SQUARE_SIZE       (1 << \
        MAP_BIGGEST_SQUARE_SIZE_SHIFT)

// The length of one edge of the map in actual game coordinates
#define MAP_SIZE            (1 << \
(MAP_SMALLEST_SQUARE_SIZE_SHIFT + MAP_HI_RES_SHIFT))

// The map.
class Map
{
public:

    // A array of pointers to the different resolutions of map.
    GameObject **Who[MAP_HI_RES_SHIFT - MAP_LO_RES_SHIFT + 1];

    Map();
    ~Map();

    // Fills in the who array and clears it
    bool Init();

    // deallocate
    void Reset();

    // fills in the object's map-related information given its
    //  position and radius
    void PlaceObject(GameObject &obj, const vector3 &pos,
    float radius);

    // given a map square at a certain resolution, returns the
    //  one at the next lower resolution, or NULL if that was the
    //  lowest res
    GameObject **GetLowerMapSquare(GameObject **current, int res);
};

//////////////////////////////////////////////////////////////////////
//
// implementation

void GameObject::ProcessCollisions(Map *my_map)
```

```
{
    // We loop through several resolutions of map, starting with
    //  the current.
    // First start with the objects in my map square
    GameObject *walker = NextInMap;
    int current_res = MapRes;

    GameObject **map_who = MapSquare;

    do
    {
        // Do one resolution's worth of collision
        ProcessOneLevel(my_map, map_who, walker, current_res);

        // Move to the next lower resolution
        map_who = my_map->GetLowerMapSquare(map_who, current_res);
        current_res--;
    }
    while (map_who); // until we're at the lowest resolution.
}

void GameObject::ProcessOneLevel(Map *my_map, GameObject **map_sq,
        GameObject *walker, int current_res_shift)
{
    int current_res_size = 1 << current_res_shift;

    // Do all the objects in the first list presented
    for (; walker; walker = walker->NextInMap)
    {
        process_collision(this, walker);
    }

    // Work out if you can go to the adjacent map squares
    int current_offset = map_sq - my_map->Who[current_res_shift -
    MAP_LO_RES_SHIFT];

    // Then do map squares to the east, southeast and south
    if ((current_offset & (current_res_size - 1)) !=
    current_res_size - 1)
    {
        // Square to the east
        for (walker = map_sq[1]; walker;
        walker = walker->NextInMap)
        {
            process_collision(this, walker);
        }
    }

    if (current_offset + current_res_size <
    (1 << (current_res_shift * 2)))
```

```
    {
        // Square to the south
        for (walker = map_sq[current_res_size]; walker;
        walker = walker->NextInMap)
        {
            process_collision(this, walker);
        }

        // and lastly, southeast.
        if ((current_offset & (current_res_size - 1)) !=
        current_res_size - 1)
        {
            for (walker = map_sq[current_res_size + 1]; walker;
                    walker = walker->NextInMap)
            {
                process_collision(this, walker);
            }
        }
    }
}

void GameObject::RemoveFromMap()
{
    // Search for myself.
    for (GameObject **pointer_to_me = MapSquare;
         *pointer_to_me != this;
         pointer_to_me = &(*pointer_to_me)->NextInMap)
    {
        assert(*pointer_to_me &&
        "Game object couldn't find itself in map");
    }

    // Remove myself.
    *pointer_to_me = NextInMap;

    // And for safety's sake, let's clear my pointers.
    NextInMap = NULL;
    MapSquare = NULL;
}

Map::Map()
{
    for (int res = MAP_LO_RES_SHIFT; res <= MAP_HI_RES_SHIFT; res++)
    {
        Who[res - MAP_LO_RES_SHIFT] = NULL;
    }
}

Map::~Map()
{
```

```
    Reset();
}

void Map::Reset()
{
    // You'd better have cleared all the objects out by now.
    //  I won't check.
    for (int res = MAP_LO_RES_SHIFT; res <= MAP_HI_RES_SHIFT; res++)
    {
        delete[] Who[res - MAP_LO_RES_SHIFT];
        Who[res - MAP_LO_RES_SHIFT] = NULL;
    }
}

bool Map::Init()
{
    Reset(); // just in case

    // allocate and clear everything.
    for (int res = MAP_LO_RES_SHIFT; res <= MAP_HI_RES_SHIFT; res++)
    {
        Who[res - MAP_LO_RES_SHIFT] = new GameObject*
        [1 << (res * 2)];

        if (!Who[res - MAP_LO_RES_SHIFT])
        return false; // alloc failed

        for (int sq = 0; sq < (1 << (res * 2)); sq++)
        {
            Who[res - MAP_LO_RES_SHIFT][sq] = NULL;
        }
    }

    return true;
}

// fills in the object's map-related information given its
//  position and radius
void Map::PlaceObject(GameObject &obj, const vector3 &pos,
        float radius)
{
    // input value checking.
    assert(radius >= 0.f && radius < MAP_SIZE);

    // If you want to allow positions off the map, change
    //  these asserts into assignments
    assert(pos.x >= 0.f && pos.x < MAP_SIZE);
    assert(pos.y >= 0.f && pos.y < MAP_SIZE);

    // Conversion into integer coordinate system needed for
```

```
        //  shifting/array maths later on.  Note that these conversions
        //  are often slow and may have to be replaced with faster
        //  versions in some compilers.  If you do replace them,
        //  preserve their rounding-down nature
        int iradius = int(radius);
        int ix      = int(pos.x );
        int iy      = int(pos.y );

        // Find which resolution level of the map the object should
        //  go in.
        obj.MapRes = MAP_HI_RES_SHIFT;
        for (int map_size = MAP_SMALLEST_SQUARE_SIZE; map_size <=
        MAP_BIGGEST_SQUARE_SIZE;
        map_size <<= 1)
        {
            // Does the object fit?
            if (iradius <= map_size) goto it_fits;

            // step on...
            obj.MapRes--;
        }

        assert(!"object too large for map - some collisions may not be detected");

    it_fits:

        // Put it in the map.
        int which_level = obj.MapRes - MAP_LO_RES_SHIFT;
        GameObject **which_who = Who[which_level];

        // Then add on the position
        which_who += ix >> (MAP_BIGGEST_SQUARE_SIZE_SHIFT - which_level);
        which_who += (iy >> (MAP_BIGGEST_SQUARE_SIZE_SHIFT -
        which_level)) << obj.MapRes;

        // Insert the object into the map square
        obj.NextInMap = *which_who;
        *which_who = &obj;

        obj.MapSquare = which_who;
    }

    GameObject **Map::GetLowerMapSquare(GameObject **current, int who_res)
    {
        // Top of map?
        if (who_res == MAP_LO_RES_SHIFT) return NULL;

        // Cunning bit-shifting.
        int current_offset = current - Who[who_res - MAP_LO_RES_SHIFT];
```

```
    // Extract the y part of the current offset.
    int y_mask = 0xffffffff << who_res;

    // The new offset is this:
    int new_offset = ((current_offset & ~y_mask     ) >> 1) +
                     ((current_offset & (y_mask << 1)) >> 2);

    return Who[who_res - MAP_LO_RES_SHIFT - 1] + new_offset;
}
```

4.7 计算到区域内部的距离

Steven Ranck

本文描述了一种确定一个点在一个 2D 四边形（或区域）内何处的简单快速算法。计算结果是一个浮点数，0 表明点位于前边（leading edge）上，而 1 则表明点位于对边（opposite edge）上。区域可以是任何 2D 凸四边形。

本文对任何需要计算一个对象或点到一个 2D 区域“内”有多远的游戏都是有用的。例如，3D 赛车游戏可以使用它的跑道的俯视 2D 区域化图来描述 AI 驾驶系统如何驶过该跑道。使用区域数据和一辆汽车在区域内的 *XZ* 位置，AI 系统可以利用本算法来确定汽车到当前区域内部距离有多远，或确定它的横向距离。本算法是非常快速的，而且如果必要，多数情况下易于为每帧的每辆车进行计算。

4.7.1 问题

图 4.7.1 显示了 *XZ* 世界空间中的一个四边形 2D 区域。对象在 *XZ* 平面上的位置以点 *P* 标记。我们想要寻找一个连续函数，它以 *P* 为参数。如果点 *P* 在区域的前边上就产生一个 0 值，且 *P* 位于后边（trailing edge）上的任意位置则产生一个 1 值，当 *P* 位于前边与后边之间时，产生一个 0 与 1 之间的值。图 4.7.2 显示了在一个区域内几个点的量值。

如图 4.7.2 所示，所有前边上的点都产生一个值 0，而且所有后边上的点都产生一个值 1。当点从前边到后边扫过区域时，点产生的值从 0 增大到 1。因为我们感兴趣的是计算代价低廉的算法，所以不要求插值是线性的。然而，它必须是一个平滑的插值，而且函数必须能被用于所有的凸区域形状。图 4.7.3 显示了几个例子。

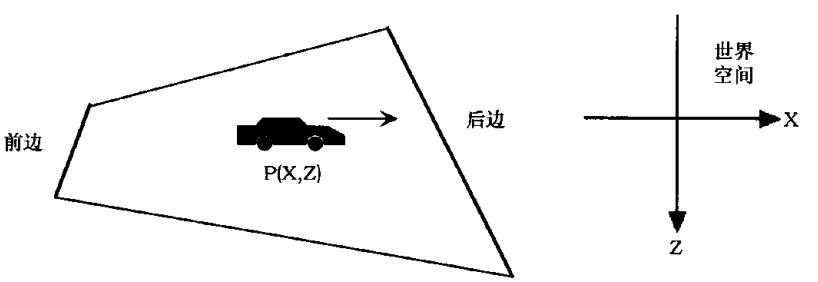


图 4.7.1 区域内的一个汽车的俯视图

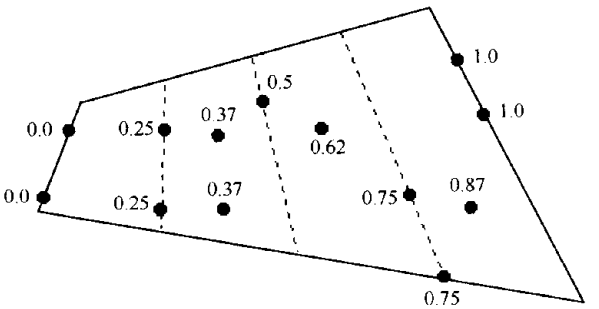


图 4.7.2 一个区域内想得到的不同点的量值

图 4.7.3 表明，不管区域的形状如何，当点在前边上时，我们期望的函数所得值总为 0，在后边上时总为 1，而且当点在两条边之间时要进行插值。

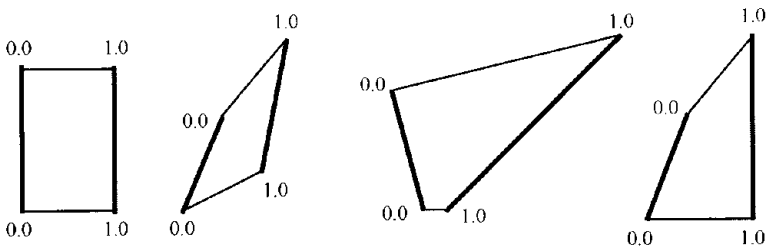


图 4.7.3 显示了相应量值的凸区域形状示例

4.7.2 算法描述

前节中提到的那种快速而简单的解决方案由下面的公式给出：

公式 4.7.1 计算穿过一个区域的距离值的公式

$$D = \frac{(V_{LP} \bullet N_L)}{V_{LP} \bullet N_L + V_{TP} \bullet N_L}$$

这里：

$V_{LP} = P - P_L$

$V_{TP} = P - P_T$

并且：

- P 是区域内感兴趣的点。
- P_L 是前边上的任何点。
- P_T 是后边上的任何点。
- N_L 是前边的指向内部的单位法线。
- N_T 是后边的指向内部的单位法线。

• D 是我们的结果：一个从 0 到 1 的浮点数。

图 4.7.4 显示了公式 4.7.1 涉及的变量及它们与问题的关系。注意 P_L 可以位于前边上的任何地方，而 P_T 可以位于后边上的任何地方。不过如果利用区域的对角则有几个优势。首先，由于区域是最可能由 4 个顶点定义的，利用这 4 个顶点是顺理成章的。其次，通过选择对角，公式 4.7.1 能既用于计算从前边到后边的距离，也能够计算横向的距离。这是因为 P_L 既位于前边上又位于一条横边上，而 P_T 既在后边上又在另一条横边上。

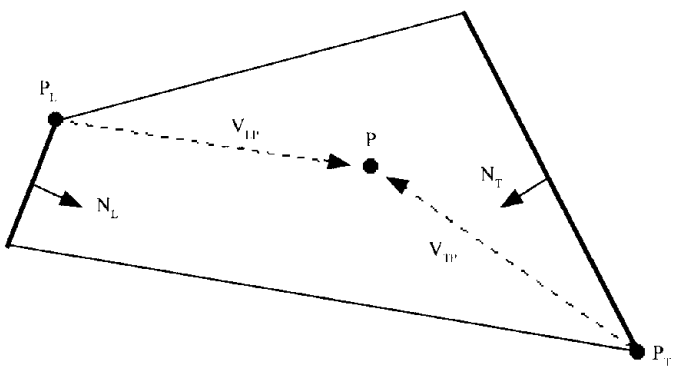


图 4.7.4 算法中使用的参数

公式 4.7.1 有几个性质：

（1）它产生一个从 0 到 1 的量值。

（2）如果 P 位于前边上则它产生的值为 0。

（3）如果 P 位于后边上则它产生的值为 1。

（4）虽然不是线性的，但它依赖于 P 在区域内的位置产生了一个从 0 到 1 的平滑插值。

（5）它很快，只需计算 2 个 2D 矢量减法、2 个 2D 点积、1 个标量加法，以及一个标量除法。

（6）世界空间中未经处理的点 P 能直接代入到公式中。不需要做任何区域空间变换。

（7）如果区域是一个静态的形状，那么 P_L、P_T、N_L 和 N_T 都可以被预先计算出来并存储在区域定义的数据结构中。

（8）如果区域是动态的形状，P_L 和 P_T 仍然可从区域的顶点得到，而且 N_L 和 N_T 可以很容易地从区域的顶点计算出来（虽然每条法线包含有求平方根的运算因而会比较慢）。

（9）如果为横边给出了两条指向内部的单位法线，它也可以被用于计算横向的距离。

该公式有几个要求：

（1）区域必须是凸的且有 4 条边，每边长都大于 0。

（2）区域必须有非零的面积。

（3）P 必须位于区域内部，或者沿着它的任何周边线段。如果点位于区域外，可以预料到其结果可能不是一个 0 到 1 之间的值。

4.7.3 应用

1. 沿一条跑道的距离

公式 4.7.1 在游戏开发中有许多应用。一个例子就是用于确定一辆汽车能够沿着一条跑道行驶多远。图 4.7.5 显示了一个由区域链覆盖的公路跑道的俯视图。

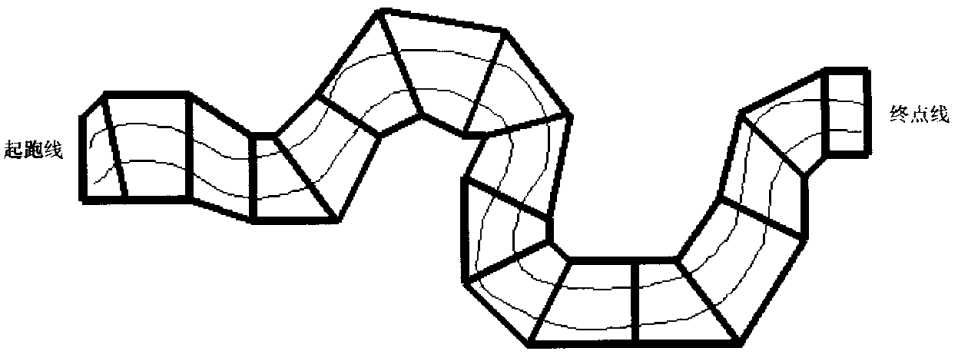


图 4.7.5 一条跑道的区域化

我们想知道的是每辆汽车能沿着跑道行驶多远。当汽车在起跑线上时，它的值应该是 0。当汽车到达终点时，想让它的值为 1。而且，我们想在它们之间进行合理的插值。

构造这些区域时完整地包含了跑道，所以汽车的位置 P 将总是在一个区域内。这也意味着相邻的区域使用了同样的顶点。在游戏初始化时，每个区域的前边和后边的指向内部的单位法线就被计算出来。注意一个区域的后边的法线等于它的相邻区域的前边的反向法线。但是如果存储器足够大，为每个区域存储两个法线将最快。

我们需要的另一个信息是在每一个区域的前边和后边之间的世界空间中的近似距离。一个粗糙但相当好的初始值是利用连接前边和后边的中点的矢量的量值。如图 4.7.6 所示。

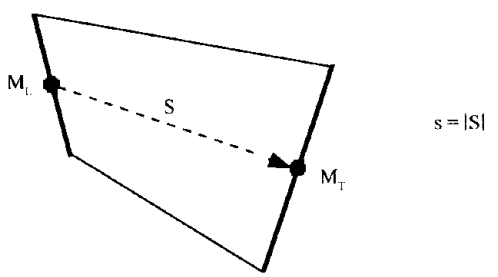


图 4.7.6 估计区域边之间的空间距离

和其他区域数据一起，我们为每一个区域预计算 s 并将其存储在区域的数据结构中。从这一点说，我们可以有一个类似下面的数据结构：

```
typedef struct {
```

```
    float fX, fZ;     // 2D XZ worldspace coordinate
} VecXZ_t;

typedef struct {
    // Clockwise, where [0] = left side of Leading Edge
    VecXZ_t aVertices[4];
    // Clockwise, where [0] = Leading Edge
    VecXZ_t aUnitNormals[4];
    // Distance from Leading Edge's midpoint to
    //   Trailing Edge's midpoint
    float fSectorDist;
    // = previous sector's fTotalPriorDist + fSectorDist
    float fTotalPriorDist;
} Sector_t;
```

注意 fTotalPriorDist 域，它是先前区域的 fTotalPriorDist 域和 fSectorDist 域的简单求和（对于第一个区域来说 fTotalPriorDist 是 0）。我们会在下面的讨论中发现它的作用。最后一个需要的信息是所有 s 的和，即将所有区域的 fSectorDist 加在一起。我们将预先计算它并存储它的倒数，并称其为 fOneOverTotalSectorDist。之所以存储倒数，是因为如我们看到的，实际上需要被所有 s 的和来除，而乘以倒数在大多数 CPU 上将比除法来得快。由于我们有了这些预先计算的丰富的信息，那么将不会错过做这个除法（求倒数）的时机并在运行时获得一个廉价的乘法。

有了前面这些信息的武装，我们现在能够确定一辆汽车能沿一个跑道行驶多远了。对每一辆汽车，我们需要下面的信息：

```
#include "mtxlib.h"

typedef struct {
    vector3 WorldPos3D;   // Vehicle's origin in 3D worldspace
    Sector_t *pSector;    // Points to the sector the
                          // vehicle origin is currently in
} Vehicle_t;
```

比赛开始的时候，汽车结构为每一辆汽车进行初始化，而且 pSector 指向包含了它的起点的区域。初始区域可以为包含汽车原始点的区域扫描全部区域的列表得到，或者将初始区域作为起跑线数据的部分进行存储而得到。

一旦比赛开始，我们需要跟踪汽车现在位于哪一个区域，因为公式 4.7.1 要求点是在一个区域内部的。为了做到这一点，每次我们移动汽车的时候，都使用一个简单的点在区域中（point-in-sector）的测试来看汽车是否仍旧在 pSector 内。如果不是，那么汽车就可能在下一个区域或在前一个区域中，然后我们再次应用点在区域中的测试。大多数时间只需测试一次；偶尔需要测试两次。如果汽车是向后倒的，将需要三次测试。如果允许汽车的速度快得从一帧变到下一帧时汽车跳过了一个完整的区域，就需要一个更完善的区域跟踪算法（sector-tracking）了。一个这样的解法将首先依次检验下 N 个区域。如果没有在其中任何区域中发现汽车，算法将接着检验前 N 个区域。如果包含汽车的区域仍未找到，这将被认为是一个异常条件，算法将采取扫描整个区域表的方法。

现在我们有了每辆汽车的一个有效区域，可以为公式 4.7.1 写一个函数以确定每辆汽车在其区域内行驶了多远：

```
float CalcUnitDistIntoSector( float fPointX, float fPointZ, const Sector_t *pSector )
{
    VecXZ_t VLP, VTP;
    float fDotL, fDotT;

    // Compute vector from point on Leading Edge to P:
    VLP.fX = fPointX - pSector->aVertices[0].fX;
    VLP.fZ = fPointZ - pSector->aVertices[0].fZ;

    // Compute vector from point on Trailing Edge to P:
    VTP.fX = fPointX - pSector->aVertices[2].fX;
    VTP.fZ = fPointZ - pSector->aVertices[2].fZ;

    // Compute (VLP dot Leading Edge Normal):
    fDotL = VLP.fX*pSector->aUnitNormals[0].fX +
VLP.fZ*pSector->aUnitNormals[0].fZ;

    // Compute (VTP dot Trailing Edge Normal):
    fDotT = VTP.fX*pSector->aUnitNormals[2].fX +
VTP.fZ*pSector->aUnitNormals[2].fZ;

    // Compute unit distance into sector and return it:
    return ( fDotL / (fDotL + fDotT) );
}
```

最后，我们可以像下面这样计算沿着跑道的距离：

```
// Pre-computed to be the inverse sum of Sector_t::fSectorDist
//   for all sectors.
float fOneOverTotalSectorDist;

float CalcUnitDistDownTrack( const Vehicle_t *pVehicle ) {
    float fUnitDistIntoSector, fDistDownTrack;

    // Compute how far vehicle is into its sector:
    fUnitDistIntoSector = CalcUnitDistIntoSector(
                          pVehicle->WorldPos3D.x,
                          pVehicle->WorldPos3D.z,
                          pVehicle->pSector
                      );

    // The distance down the track is the full distance
//  across all previous sectors, plus the partial
//  distance into our current sector:
    fDistDownTrack = pVehicle->pSector->fTotalPriorDist
                   + pVehicle->pSector->fSectorDist *
fUnitDistIntoSector;
```

```
    // Finally, our unit distance down the track is our
//  distance so far divided by the track's total distance:
    return fDistDownTrack * fOneOverTotalSectorDist;
}
```

如果汽车是在跑道的第一个区域的前边上，函数 CalcUnitDistDownTrack( )返回值 0，如果汽车是在跑道的最后一个区域的后边上，将返回值 1，而基于汽车沿着跑道跑了多远返回一个 0 和 1 之间的插值。

2. 平滑光照变化

公式 4.7.1 的另一个实际应用是穿过区域时对象上光照的平滑插值。如果我们将环境光和方向光与前、后区域的边联系起来，可以对光照参数进行插值，当对象从一个区域移动到另一个区域时将被照亮。

考虑一个封闭在一个跑道部分内的区域，区域的前边在阳光下，而区域的后边终止于一个昏暗的岩洞。如图 4.7.7 所示。

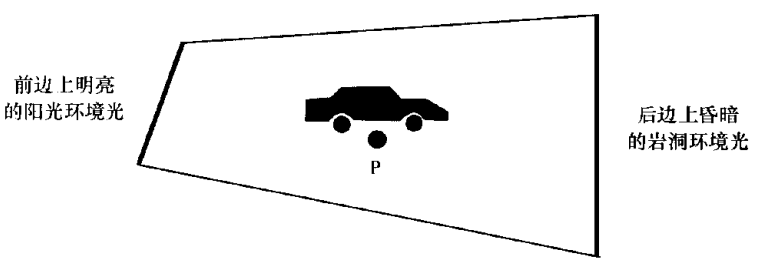


图 4.7.7 穿过一个区域时的环境光照的梯度变化

我们可以像这样存储每一个邻接的区域边上的环境 RGB 光的颜色：

```
typedef struct {
    float fR, fG, fB; // RGB ambient light
} Ambient_t;

typedef struct {
    // Clockwise, where [0] = left side of Leading Edge
    VecXZ_t aVertices[4];
    // Clockwise, where [0] = Leading Edge
    VecXZ_t aUnitNormals[4];
    // Distance from Leading Edge's midpoint to Trailing
    //  Edge's
    float fSectorDist;
    // = previous sector's fTotalPriorDist + fSectorDist
    float fTotalPriorDist;
    Ambient_t LeadingAmbient; // Ambient light at Leading Edge
    Ambient_t TrailingAmbient;     // Ambient light at Trailing Edge
} Sector_t;
```

邻接区域的环境值必须是一样的，以避免当汽车穿越不同区域时光照水平在视觉上的突变。在前述的实现中，我们简单地将每个区域前边和后边的环境值都存储起来，但是一个更有“存储意识”的实现可能会通过指针共享环境数据。在任何情形下，当对象穿行区域的时候，前述的方法都能满足平滑地制作从阳光到洞穴光照的环境光照水平的需要。下面的函数采用了这样的技巧：

```
#define LERP( fUnit, fV0, fV1 ) ( (1.0f-(fUnit))*fV0 + \
(fUnit)*fV1 )

Ambient_t CalcAmbientLightLevel( const Vehicle_t *pVehicle ) {
    float fUnitDistIntoSector;
    Ambient_t *pLeadAmbient, *pTrailAmbient, RetAmbient;

    // Compute how far vehicle is into its sector:
    fUnitDistIntoSector = CalcUnitDistIntoSector(
                          pVehicle->WorldPos3D.x,
                          pVehicle->WorldPos3D.z,
                          pVehicle->pSector
                      );

    pLeadAmbient = &pVehicle->pSector->LeadingAmbient;
    pTrailAmbient = &pVehicle->pSector->TrailingAmbient;

    RetAmbient.fR = LERP( fUnitDistIntoSector,
pLeadAmbient->fR, pTrailAmbient->fR );
    RetAmbient.fG = LERP( fUnitDistIntoSector,
pLeadAmbient->fG, pTrailAmbient->fG );
    RetAmbient.fB = LERP( fUnitDistIntoSector,
pLeadAmbient->fB, pTrailAmbient->fB );

    return RetAmbient;
}
```

前述函数计算了在汽车的世界位置的环境光照水平。一个图形引擎可能采用最终的环境RGB来产生总的环境光照去照亮汽车，如果需要，可以与其他更成熟的光照技术相结合。当汽车驶过区域的时候，无论该区域的形状如何，环境光照都被平滑地进行插值。

可以写一些类似于CalcAmbientLightLevel( )的函数，进行任何与一个区域的边界有关的平滑插值。例如方向光（插值方向和颜色）、区域高度、水流速率和方向、雾的特征、天空的外貌以及AI障碍等。

4.8 对象阻塞剔除

Tim Round

阻塞剔除（occlusion culling）是一种用于从视野中剔除不需要的几何体的技术。这是视野裁剪的一个扩展，它有助于减少与渲染一个网格相关联的（例如变换、光照以及光栅化）不必要的处理时间。阻塞剔除提供了一个能在任意动态几何体数据上工作的裁剪方法。这意味着网格数据不必包含任何关于潜在可见的数据集的信息。阻塞剔除并不限制于在室内场景，而是能用于标志任何可能从你的视野被遮挡的网格数据（请看图4.8.1 的一个例子）。

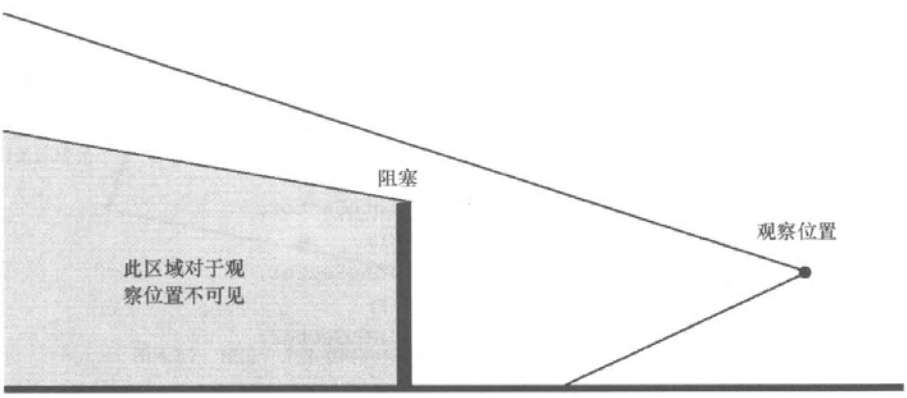


图 4.8.1 一个典型的遮蔽

使用 Z 缓冲器能使恰当地显示一个场景变得更容易，但是当测试每个像素的深度的时候，仍然需要进行变换、光照，并且需要描绘多边形。这种阻塞剔除的实现通过使用一种简单的预定义阻塞形状（即矩形），简化了建立阻塞数据的过程。这些阻塞矩形易于作为两个共面的三角形增加到原始的几何图形内，而且进行命名或染色时应使输入程序或装载程序可以把它们从原始网格数据中分离出来。

4.8.1 可视棱台裁剪

为了更好地理解阻塞剔除，有必要了解一种用于视野裁剪的技术。视野是在 3D 空间中从当前摄像机的观察点能够看到的范围。这个范围典型地用前后剪平面（clip plane）和视角来描述（参见图 4.8.2）。

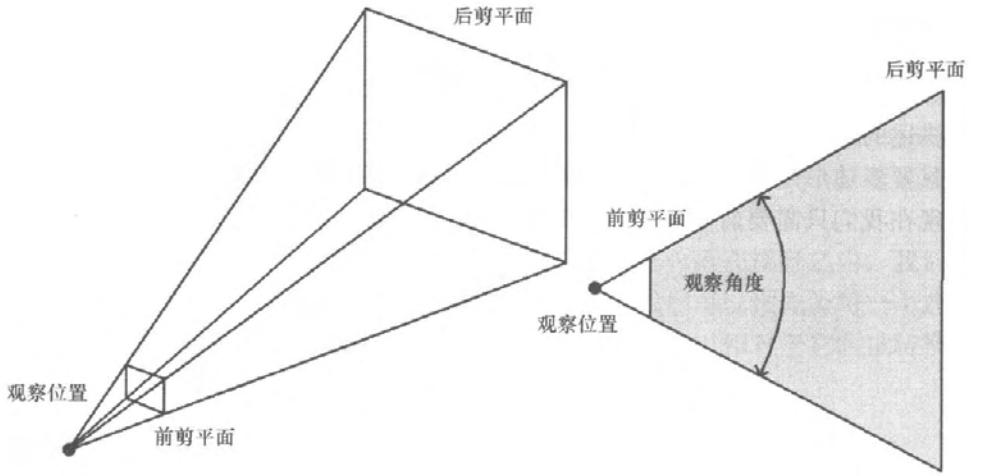


图 4.8.2 视野或观察截面

裁剪的过程使用了每个对象网格在世界空间的包围球（bounding sphere），而且测试是否它落在视野内。为了执行这个测试，将包围球的中心点转换到视域空间内（相对于摄像机而言），并且检验相对于近剪平面和远剪平面的新的 Z 值。然后你可以相对于左、右、顶和底剪平面测试中心点。视野的剪平面能在渲染循环开始时被预先计算。因为这些剪平面是在视域空间中，所以我们可以简化裁剪测试。前后剪平面将与 Z 轴垂直，所以相对于 Z 值的一个简单比较能迅速决定网格是在摄像机的前面或后面。左右平面的平面方程中的系数 B 和 D 为零（因为它们是垂直的），而且这些平面都经过原点（0, 0, 0）。这意味着相对于左平面或右平面测试一个点，可以使用下面的方程：

```
DistanceFromClipPlane = (x * Plane.a) + (z * Plane.c);
```

这也可以用于顶剪平面和底剪平面，只是现在系数 A 和 D 为零，故有下面的方程：

```
DistanceFromClipPlane = (y * Plane.b) + (z * Plane.c);
```

如果从平面到点的距离比包围球的半径要大，那么包围球就在视线范围之外了。通过相对于所有的剪平面测试包围球，我们可以发现网格是否位于视野中；这可以帮助我们移除所有不可见的网格。中心点相对于剪平面进行测试的次序可以为匹配几何体数据而更改；例如，如果一个风景更多地沿着 *X* 轴和 *Z* 轴而不是 *Y* 轴方向伸展，那么我们在测试顶和底剪平面之前要先测试左和右剪平面。而前后剪平面则应首先进行测试，因为它们需要较少的计算量并且通常能移除大部分几何体。请看程序清单 4.8.1 中实现了视野裁剪的示例代码。

你已经看到了，我们能使用平面方程来描述视域棱台的边界，而且通过首先将包围球的中心点转换到视域空间中，可以简单化点到平面的测试。故我们可以利用同样的原则描述一个遮蔽物的边界。

4.8.2 阻塞剔除

本文描述的遮蔽是由平面四边形或平面多边形构成，但是它们可以容易地使用更多或更少的边，只要多边形是平面的就行。我们可以用与视野裁剪同样的方式预先计算遮蔽的剪平面，只是现在我们只需要前平面和4个侧平面了（参见图4.8.3）。

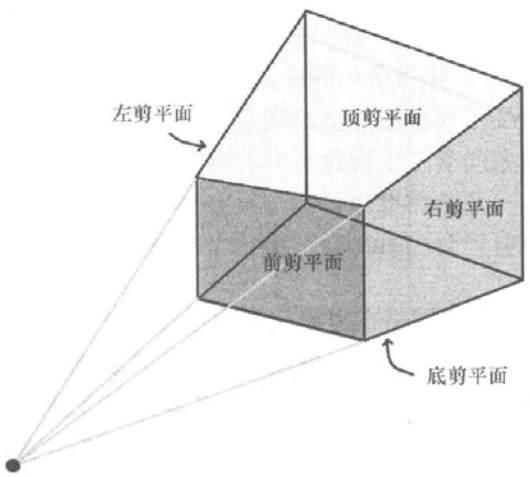


图4.8.3 一个四边形遮蔽被4个平面描述：前、左、右、顶和底平面

遮蔽区域与视域棱台的不同之处在于它描述了一个洞（hole）。任何位于这个洞中的网格将不会被渲染（参见图4.8.4）。

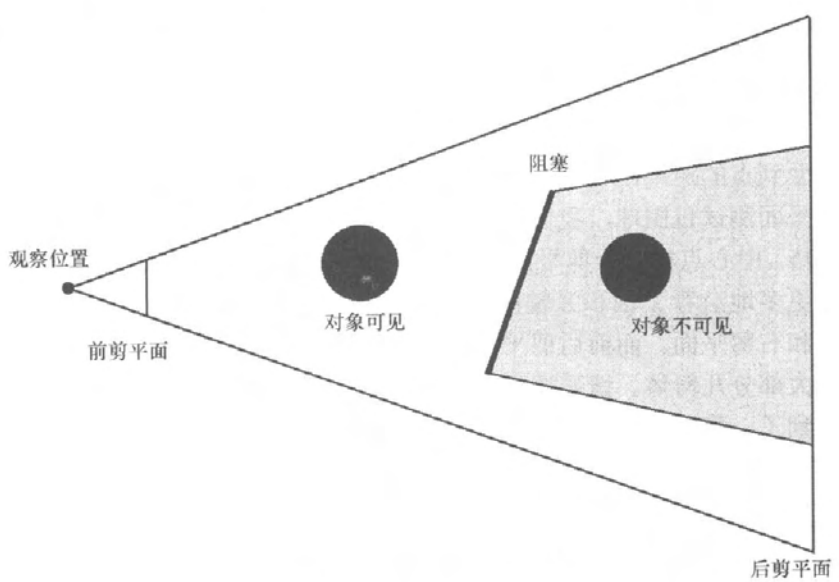


图4.8.4 遮蔽后面的对象是不可见的

遮蔽的前平面不会总与前剪平面垂直，于是我们不得不使用平面方程中的所有 4 个系数。我们也不能简化侧平面，因为它们也可能有任意的角度。前平面通过使用遮蔽多边形上的 3 个点（在它们被转换到视域空间之后）来计算。侧平面通过使用摄像机位置（在视域空间中是(0, 0, 0)）和沿着遮蔽多边形的边的两个点计算（参见程序清单 4.8.2 中的 SetupOcclusion( ) 函数）。通过测试前平面朝向哪一方及颠倒生成平面时所使用的点的次序，遮蔽能被做成双面的。

在相对于遮蔽测试一个对象的网格之前，需要测试它是否落在视野之内，我们已经将包围球的中心转换到视域中了。为了加快遮蔽测试的速度，我们可以预先为每一个遮蔽计算一个视域中的 Z 的最小值（即最接近于前剪平面的值），而且可以相对于它测试转换过的包围球的中心，以便迅速测试网格是否在遮蔽的前方（参见图 4.8.5）。

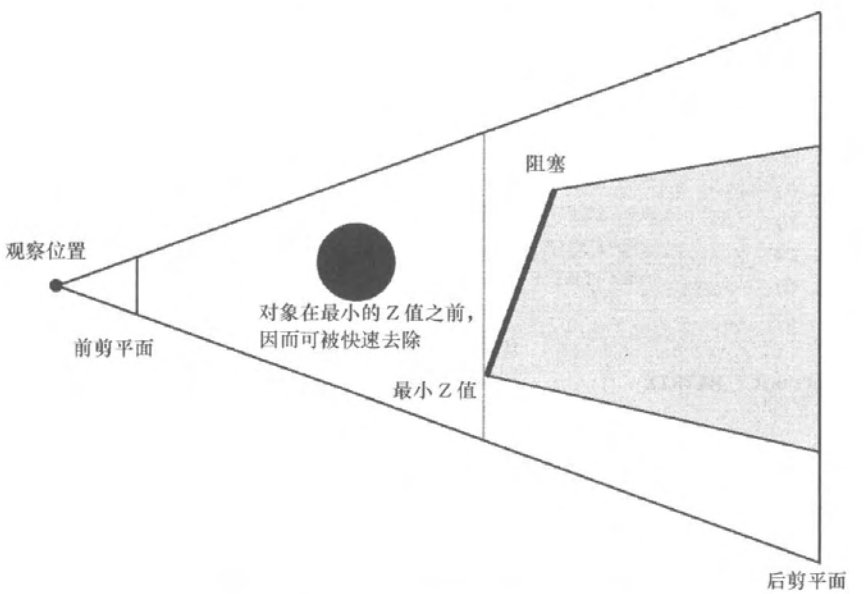


图 4.8.5 一个最小的 Z 值能用来加快测试速度

如果网格的包围球与一个遮蔽的边重叠，我们可以测试轴对齐的包围盒的所有点得到额外的精确度。如果一个物体沿着轴非常大（比如非常高），那么用一个包围球描述它的范围将是非常浪费的，但是用一个包围球相对于一个平面进行测试要比用一个包围盒快得多。于是我们首先使用包围球来快速去除一些完全落在一个遮蔽的内部或外部的网格。

当建立一个可见遮蔽的列表时，你也可以相互测试遮蔽以去除任何被遮住的遮蔽。如果一个遮蔽覆盖了整个视域棱台，我们可以将后剪平面移得更近些。遮蔽也可以有自己的包围球，可以用于去除任何视域中不可见的遮蔽。

4.8.3 总结

如你所见，这种阻塞剔除相当容易实现，但是它不一定被限制于网格裁剪——它也能够

用于裁剪声音。遮蔽也有助于防止耗时效果（比如有皮肤的动画（skinned animation））被应用于不可见的对象。

遮蔽也能通过将几个遮蔽缝接在一起而构成更为复杂的遮蔽带。但这就超出本文讨论的范围了。

下面的代码是用基于 DirectX 写的（即它是左手坐标的），但是它可以容易地修改为适合任何坐标系统。代码是为说明算法设计的，也易于改编为适合任何目标平台或应用。

程序清单 4.8.1 视野裁剪代码

```
typedef struct _VECTOR
{
    float x;
    float y;
    float z;
}VECTOR;

typedef struct _PLANE
{
    float a;
    float b;
    float c;
    float d;
}PLANE;

typedef struct _MATRIX
{
    float _11;
    float _12;
    float _13;
    float _14;
    float _21;
    float _22;
    float _23;
    float _24;
    float _31;
    float _32;
    float _33;
    float _34;
    float _41;
    float _42;
    float _43;
    float _44;
}MATRIX;

PLANE    g_FOVLeftPlane;
PLANE    g_FOVRightPlane;
PLANE    g_FOVTopPlane;
PLANE    g_FOVBottomPlane;
float    g_FOVFrontClip;
```

```
float     g_FOVBackClip;
MATRIX g_ViewTransform;


voidNormalize(VECTOR *pV)
{
    float Length , InvLength;

    Length = (float) sqrt( ( pV->x * pV->x ) +
( pV->y * pV->y ) + ( pV->z * pV->z ) );
    InvLength = 1.0f / Length;
    pV->x /= InvLength;
    pV->y /= InvLength;
    pV->z /= InvLength;
}


voidCrossProduct(VECTOR *pV0 , VECTOR *pV1 ,
VECTOR *pCrossProduct)
{
    pCrossProduct->x = pV0->y * pV1->z - pV0->z * pV1->y;
    pCrossProduct->y = pV0->z * pV1->x - pV0->x * pV1->z;
    pCrossProduct->z= pV0->x * pV1->y - pV0->y * pV1->x;
}


voidPlaneFromPoints(VECTOR *pP0 , VECTOR *pP1 ,
VECTOR *pP2 , PLANE *pPlane)
{
    VECTOR V0,V1,V2;

    V0.x = pP1->x - pP0->x;
    V0.y = pP1->y - pP0->y;
    V0.z= pP1->z- pP0->z;
    V1.x = pP2->x - pP0->x;
    V1.y = pP2->y - pP0->y;
    V1.z= pP2->z- pP0->z;
    CrossProduct( &V0, &V1, &V2);
    Normalize( &V2);
    pPlane->a = V2.x;
    pPlane->b = V2.y;
    pPlane->c = V2.z;
    pPlane->d = -( V2.x * pP0->x + V2.y * pP0->y + V2.z *
pP0->z );
}


//This function calculates the planes for describing the view
//frustum using 3 points. Because we are in view space the
//cameras position is at 0,0,0. We use the back clip position
```

```
//and the viewing angle to work out a point on the edge of
//the frustum.  The view angle is the angle between the top
//and bottom of the view frustum in radians.

void SetupFOVClipPlanes( float Angle , float Aspect ,
float FrontClip , float BackClip )
{
    VECTOR   P0 , P1 , P2;

//  Calculate left plane using 3 points
    P0.x = 0.0f;
    P0.y = 0.0f;
    P0.z = 0.0f;
    P1.x = -BackClip * ( (float)tan( Angle * 0.5f ) / Aspect );
    P1.y = -BackClip * ( (float)tan( Angle * 0.5f ) );
    P1.z = BackClip;
    P2.x = P1.x;
    P2.y = -P1.y;
    P2.z = P1.z;
    PlaneFromPoints( &P0, &P1 , &P2 , &g_FOVLeftPlane );

//  Calculate right plane using 3 points
    P0.x = 0.0f;
    P0.y = 0.0f;
    P0.z = 0.0f;
    P1.x = BackClip * ( (float)tan( Angle * 0.5f ) / Aspect);
    P1.y = BackClip * ( (float)tan( Angle * 0.5f ) );
    P1.z = BackClip;
    P2.x = P1.x;
    P2.y = -P1.y;
    P2.z = P1.z;
    PlaneFromPoints( &P0, &P1 , &P2 , &g_FOVRightPlane );

//  Calculate top plane using 3 points
    P0.x = 0.0f;
    P0.y = 0.0f;
    P0.z = 0.0f;
    P1.x = -BackClip * ( (float)tan( Angle * 0.5f ) / Aspect);
    P1.y = BackClip * ( (float)tan( Angle * 0.5f ) );
    P1.z = BackClip;
    P2.x = -P1.x;
    P2.y = P1.y;
    P2.z = P1.z;
    PlaneFromPoints( &P0, &P1 , &P2 , &g_FOVTopPlane );

//  Calculate bottom plane using 3 points
    P0.x = 0.0f;
    P0.y = 0.0f;
    P0.z = 0.0f;
    P1.x = BackClip * ( (float)tan( Angle * 0.5f ) / Aspect);
```

```
    P1.y = -BackClip * ( (float)tan( Angle * 0.5f ) );
    P1.z = BackClip;
    P2.x = -P1.x;
    P2.y = P1.y;
    P2.z = P1.z;
    PlaneFromPoints( &P0, &P1 , &P2 , &g_FOVBottomPlane );
}

BOOL MeshFOVCheck(VECTOR *pBSpherePos ,
float BSphereRadius,VECTOR *pViewPos)
{
    float    Dist;

//  Transform Z into view space
    pViewPos->z = g_ViewTransform._13 * pBSpherePos->x +
        g_ViewTransform._23 * pBSpherePos->y +
        g_ViewTransform._33 * pBSpherePos->z +
        g_ViewTransform._43;

//  Behind front clip plane?
    if( ( pViewPos->z + BSphereRadius ) < g_FOVFrontClip )
        return FALSE;

//  Beyond the back clip plane?
    if( ( pViewPos->z - BSphereRadius ) > g_FOVBackClip )
        return FALSE;

//  Transform X into view space
    pViewPos->x = g_ViewTransform._11 * pBSpherePos->x +
        g_ViewTransform._21 * pBSpherePos->y +
        g_ViewTransform._31 * pBSpherePos->z +
        g_ViewTransform._41;

//  Test against left clip plane
    Dist = ( pViewPos->x * g_FOVLeftPlane.a ) +
( pViewPos->z * g_FOVLeftPlane.c );
    if( Dist > BSphereRadius )
        return FALSE;

//  Test against right clip plane
    Dist = ( pViewPos->x * g_FOVRightPlane.a ) +
( pViewPos->z * g_FOVRightPlane.c );
    if( Dist > BSphereRadius )
        return FALSE;

//  Transform Y into view space
    pViewPos->y = g_ViewTransform._12 * pBSpherePos->x +
        g_ViewTransform._22 * pBSpherePos->y +
        g_ViewTransform._32 * pBSpherePos->z +
        g_ViewTransform._42;
```

```
//  Test against top clip plane
    Dist = ( pViewPos->y * g_FOVTopPlane.b ) +
( pViewPos->z * g_FOVTopPlane.c );
    if( Dist > BSphereRadius )
        return FALSE;

//  Test against bottom plane
    Dist = ( pViewPos->y * g_FOVBottomPlane.b ) +
( pViewPos->z * g_FOVBottomPlane.c);
    if( Dist > BSphereRadius )
        return FALSE;

//  Mesh is inside the field of view
    return TRUE;
}
```

程序清单 4.8.2 阻塞剔除代码

```
typedef struct _OCCLUSION
{
    VECTOR P0;
    VECTOR P1;
    VECTOR P2;
    VECTOR P3;
    float     MinZ;
    PLANE     FrontPlane;
    PLANE     FirstPlane;
    PLANE     SecondPlane;
    PLANE     ThirdPlane;
    PLANE     FourthPlane;
}OCCLUSION;

void VectorMatrixMultiply3x4(VECTOR *pNewVector ,
VECTOR *pVector , MATRIX *pMatrix)
{
}

void SetupOcclusion(OCCLUSION *pOcclusion ,
MATRIX *pViewTransform)
{
    VECTOR   P0 , P1 , P2 , P3 , Camera;

//  Transform points form world space to view space
    VectorMatrixMultiply3x4( &P0 , &pOcclusion->P0 ,
         pViewTransform);
    VectorMatrixMultiply3x4( &P1 , &pOcclusion->P1 ,
         pViewTransform);
    VectorMatrixMultiply3x4( &P2 , &pOcclusion->P2 ,
         pViewTransform);
    VectorMatrixMultiply3x4( &P3 , &pOcclusion->P3 ,
```

```
            pViewTransform);

        pOcclusion->MinZ = P0.z;
        if( P1.z < pOcclusion->MinZ)
            pOcclusion->MinZ = P1.z;
        if( P2.z < pOcclusion->MinZ)
            pOcclusion->MinZ = P2.z;
        if( P3.z < pOcclusion->MinZ)
            pOcclusion->MinZ = P3.z;

    //  The camera position in view space is 0,0,0
        Camera.x=0.0f;
        Camera.y=0.0f;
        Camera.z=0.0f;

    //  Create front plane from first three points
        PlaneFromPoints(&P0 , &P1 , &P2 , &pOcclusion->FrontPlane);

    //  Test the D co-effecient to find which way the
    //   occlusion faces
        if(pOcclusion->FrontPlane.d > 0.0f)
        {
            PlaneFromPoints( &Camera , &P0 , &P1 ,
                  &pOcclusion->FirstPlane);
            PlaneFromPoints( &Camera , &P1 , &P2 ,
                  &pOcclusion->SecondPlane);
            PlaneFromPoints( &Camera , &P2 , &P3 ,
                  &pOcclusion->ThirdPlane);
            PlaneFromPoints( &Camera , &P3 , &P0 ,
                  &pOcclusion->FourthPlane);
        }
        else
        {
            PlaneFromPoints( &P2 , &P1 , &P0 ,
                  &pOcclusion->FrontPlane);
            PlaneFromPoints( &Camera , &P1 , &P0 ,
                  &pOcclusion->FirstPlane);
            PlaneFromPoints( &Camera , &P2 , &P1 ,
                  &pOcclusion->SecondPlane);
            PlaneFromPoints( &Camera , &P3 , &P2 ,
                  &pOcclusion->ThirdPlane);
            PlaneFromPoints( &Camera , &P0 , &P3 ,
                  &pOcclusion->FourthPlane);
        }
    }

    BOOL TestIfOccluded(OCCLUSION *pOcclusion ,
            VECTOR *pViewPos , float BSphereRadius)
    {
        float    MinZ;
```

```
    MinZ = pViewPos->z - BSphereRadius;
    if( pOcclusion->MinZ < MinZ )
        return FALSE;
    if( ( ( pViewPos->x * pOcclusion->FrontPlane.a) +
        (pViewPos->y * pOcclusion->FrontPlane.b) +
        (pViewPos->z * pOcclusion->FrontPlane.c) +
        pOcclusion->FrontPlane.d) > BSphereRadius )
        return FALSE;
    if( ( ( pViewPos->x * pOcclusion->FirstPlane.a) +
        (pViewPos->y * pOcclusion->FirstPlane.b) +
        (pViewPos->z * pOcclusion->FirstPlane.c) +
        pOcclusion->FirstPlane.d) > BSphereRadius )
        return FALSE;
    if( ( ( pViewPos->x * pOcclusion->SecondPlane.a) +
        (pViewPos->y * pOcclusion->SecondPlane.b) +
        (pViewPos->z * pOcclusion->SecondPlane.c) +
        pOcclusion->FirstPlane.d) > BSphereRadius )
        return FALSE;
    if( ( ( pViewPos->x * pOcclusion->ThirdPlane.a) +
        (pViewPos->y * pOcclusion->ThirdPlane.b) +
        (pViewPos->z * pOcclusion->ThirdPlane.c) +
        pOcclusion->FirstPlane.d) > BSphereRadius )
        return FALSE;
    if( ( ( pViewPos->x * pOcclusion->FourthPlane.a) +
        (pViewPos->y * pOcclusion->FourthPlane.b) +
        (pViewPos->z * pOcclusion->FourthPlane.c) +
        pOcclusion->FirstPlane.d) > BSphereRadius )
        return FALSE;
    return TRUE;
}
```

4.9 永远不要让他们看到你的“抖动”——几何体细节层次选择问题

Yossarian King

对物体和角色在计算图形学中表现为几何体模型。模型能在不同的细节层次（level of detail，LOD）被创建，有更多细节的模型具有更多的多边形和更大的纹理，而有更少细节的模型具有更少的多边形和更小的纹理。为什么要这样做呢？为了改善渲染性能和视觉品质。当一个对象与摄像机相距很远的时候，绘制较少的多边形减少了场景中多边形的数目，从而加速了渲染。当一个对象与摄像机相距比较近的时候，应用有更多细节的模型将改善视觉品质。如果仅用一个单一的模型，就总会有一个性能与效果之间的权衡问题——多个细节层次将有助获得这两方面的效果。

为了实现 LOD 渲染，要在不同的细节层次创建多个模型，而且每帧要渲染的模型应基于与摄像机间的距离进行选择。作为一个粗略的规则，每个细节层次应该有大约 2 倍于前一层次的多边形的数目。模型是为了尽可能减少“抖动（popping）”而创建的——当角色或对象从一个细节层次变换到另一个细节层次的时候，几何体（尤其是轮廓边缘）和纹理的视觉变化必须被最小化。美工的任务是在变换将发生的范围内进行渲染时，创建尽可能相似的模型。程序设计师的任务就是当最小化 LOD 变换的数目时，确定何时改变 LOD 以获得理想的性能和品质。本文阐述了怎样完成这一任务。

注意与摄像机相距一个相对来说恒距离的对象或角色（如一个第三人称视角游戏中的英雄角色），细节层次选择是不必要的。也请注意本文不讨论地形渲染的细节层次问题。

4.9.1 LOD 选择

选择要渲染的细节层次的最简单方法是在从摄像机到对象的距离上应用一个阈值。例如，当对象距离 500 单位以内使用较高细节层次的模型，当模型距离在 500~1500 单位时使用中等细节层次的模型，而对象的距离远于 1500 单位时使用较低细节层次的模型。乍一看这是合理的——当对象更近时用更多的细节；当它远离的时候用较少的细节。然而，这种方法有两个问题：

首先，它没有考虑摄像机的视野。如果对象是远离摄像机的，但是视

野非常狭窄（比如变焦镜头），那么对象可能会在屏幕上显示得很大，因而选择一个有更多细节的模型将更适合。类似地，一个对象也可以与摄像机相对较近，但是如果视野非常广阔（比如微距透镜），那么对象就可能在屏幕上显示得比较小且应该使用一个低细节层次的模型。图 4.9.1 说明了同一个对象在与摄像机同样的距离处，在屏幕上显示的大小却并不总是相同。与其使用与摄像机的距离，倒不如使用对象在屏幕上的投影的大小来作为一个选择细节层次的依据。屏幕上的大小明显和与摄像机的距离相关，但是也要考虑视野。

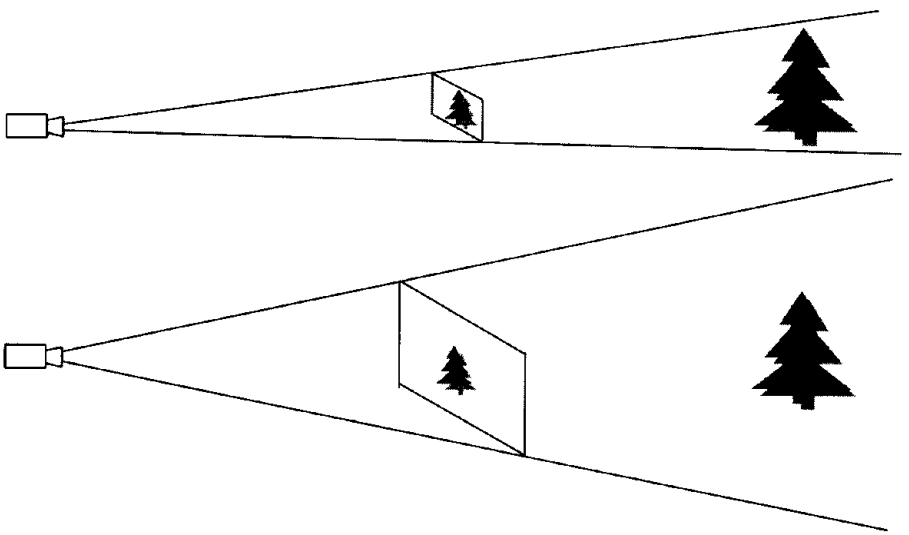


图 4.9.1 视野变化改变了对象在屏幕上的投影大小

上图：一个狭窄的视野在屏幕上产生了一个较大的图像。下图：一个广阔的视野产生了一个较小的图像。在两种情形中，树的大小和从树到摄像机的大小是相同的，证明了摄像机的距离对于选择细节层次来说并不充分

关于简单距离阈值方法的第二个问题是，如果对象保持与阈值接近的时候，细节层次间将会有快速的来回转换。当一个角色跑着穿过一个接近于阈值距离的视野时，就很有可能会发生这种情况。从一个细节层次到另一个细节层次的一次抖动会引人注意，而在层次间快速地循环往复将非常令人心烦意乱。

幸运的是，这两个问题都很容易解决。一个更好的替换摄像机间距离的方法是使用“放大率因子（magnification factor）”，它是物体的屏幕尺寸与其物理尺寸之比。当这个比率增加的时候（即当一个对象在屏幕上变大时），我们就选择更高的细节层次。屏幕大小要同时考虑摄像机距离和视野，于是第一个问题就解决了。放大率因子非常容易计算，将在下一节进行阐述。

快速往复抖动的问题可以使用滞变阈值（hysteresis thresholding）来解决。普通的阈值选择是在一个输入值上应用单个的阈值来选择输出。滞变阈值采用一个较高的阈值和一个较低的阈值，并且基于前先的输出值来决定应用其中的哪一个。只要输入值保持在较高和较低的阈值之间，输出值就不改变，这样就保持了 LOD 选择的稳定性。其细节随后描述。

4.9.2 放大率因子

一个对象的屏幕尺寸可以通过转换和投影对象上的最高点和最低点，并减去每一坐标的屏幕位置得到屏幕高度来确定。这种方法依赖于对象的朝向，并且需要处理两个点。放大率因子计算更为简单并独立于朝向。它能通过将对象的位置变换到视域中然后进行计算而得到：

$M = xscale / zview$

这里 *xscale* 是用于投影方程的换算参数：

$xscreen = (xview * xscale) / zview + xcenter$

由于视点坐标只是一个世界坐标的旋转或平移，*zview* 由世界单位量度。*xscale* 相对于摄像机视野，以像素为单位；因此，放大率因子 *M* 度量了每世界单位的像素。当 *M* 增加时，每世界单位有更多的像素——对象相对来说在屏幕上更大，因此应该使用一个更高的细节层次。注意 *M* 类似于用于插值贴图纹理的细节层次，在贴图的情形下，每纹理像素的像素值才是我们感兴趣的量度。

M 既考虑了摄像机距离（通过 *zview*）也考虑了视野（通过 *xscale*），而且给出了一个比简单摄像机距离更好的确定细节层次的选择依据。然而，对 *M* 应用一个简单的阈值将会有与先前描述过的相同的抖动问题。解决方案是应用滞变阈值。

4.9.3 滞变阈值

滞变阈值是一个有想像力的术语，它是对应于一个值的范围的阈值，而不仅是单个值的阈值。一个简单的阈值是这样应用的：

$$output = \begin{cases} 1 \text{ if } input >= T \\ 0 \text{ if } input < T \end{cases}$$

滞变阈值使用一个较高的和一个较低的阈值并且记录前一次的输出值。如果输入值是在较高的和较低的阈值之间，则输出值不变。

$$output(t) = \begin{cases} 1 \text{ if } input >= T_{high} \\ 0 \text{ if } input < T_{low} \\ output(t\text{-}1) \text{ otherwise} \end{cases}$$

如果输入是递增的，那么在输入达到 T_{high} 之前输出将为 0。如果输入是递减的，那么在输入降到 T_{low} 之下以前输出将保持为 1。无论值是递增还是递减，只要输入是在较高和较低的阈值之间输出就不改变。采用这种方式选择细节层次意味着物体不会在一个点上进行反复的细节层次变换——滞变阈值确保了我们所得到的是一个单一的抖动，永远不会是一个反复变换行为。应用简单阈值和滞变阈值的一个视觉比较如图 4.9.2 所示。

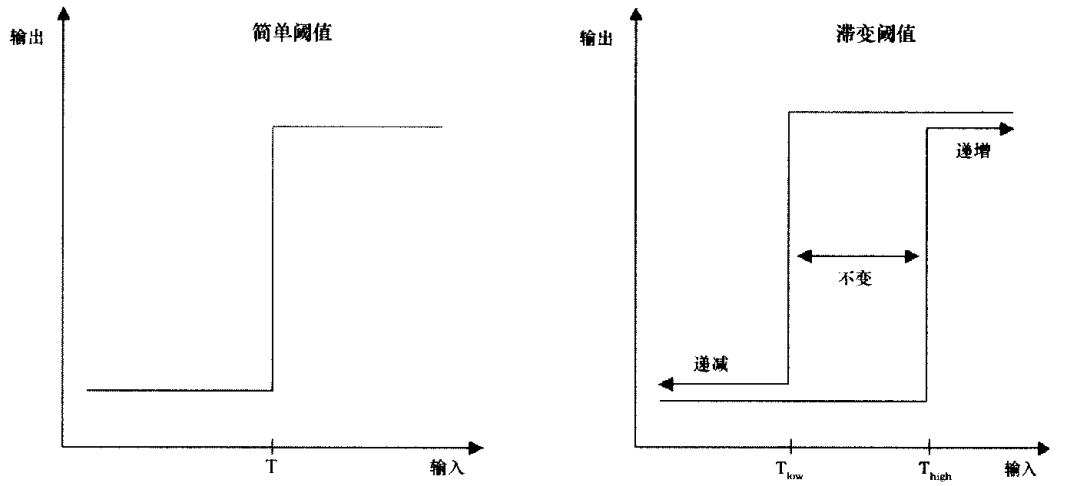


图 4.9.2 左：一个输入值针对单值阈值 T 的输出。右：一个输入值针对于阈值 T_{low} 和 T_{high} 的输出，有一个基于前面的输出值的适当的阈值选择。只要输入保持在 T_{low} 和 T_{high} 之间，输出就不做任何改变

4.9.4 实现

有了放大率因子和滞变阈值，我们可以创建一个细节层次选择算法了，它考虑了摄像机视野并避免了快速抖动的问题。假设我们有 3 个细节层次的模型：高、中和低。用于在高细节层次和中细节层次之间移动的滞变阈值为 T_{hupper} 和 T_{hlower}。在中细节层次和低细节层次之间移动的滞变阈值为 T_{mupper} 和 T_{mlower}。细节层次选择算法的伪代码如下：

```
int computelod:
worldpos world position of the object
lodprev  level of detail chosen in previous frame
{
    viewpos = transform( worldpos )
    M = xscale / viewpos.z

    if ( M < T_mlower )
        lod = low
    else if ( M < T_mupper )
        lod = lodprev          hysteresis range for medium/low
    else if ( M < T_hlower )
        lod = medium
    else if ( M < T_hupper )
        lod = lodprev          hysteresis range for high/medium
    else
        lod = high             M >= T_hupper

    return lod
}
```

注意如果 M 是在 T_{hlower} 和 T_{hupper} 之间，那么将总是返回前一个细节层次，即使它是低细

节层次。如果你期望对象迅速放大，则可以很容易地修改算法。

使用一个简单的距离阈值的等价算法将在每个细节层次之间仅使用一个阈值，而且将会如下所示：

```
int computelodwithpopping:
worldpos
{
    viewpos = transform( worldpos )

    if ( viewpos.z < T_m )
        lod = low
    else if ( viewpos.z < T_h )
        lod = medium
    else                    viewpos.z >= T_h
        lod = high

    return lod
}
```

正如我们已经看到的，解决视野依赖和抖动问题并没有明显增加细节层次选择的复杂度。

4.9.5 其他问题

阈值选择

对于任何阈值方法，滞变阈值或者其他，你都需要选择阈值。对于细节层次选择来说，选择阈值就是在性能与视觉品质之间进行权衡。如果阈值设置得过低，那么更高的细节层次就会被频繁地选择，每帧不得不渲染更多的几何体，这将使渲染速度慢下来。如果阈值设置得过高，那么更低品质的模型将会更经常地绘制，而层次之间的抖动将更为明显。

为了减少抖动，你可以执行选择算法，然后将对象朝着和远离摄像机移动，调整阈值直到抖动可以被接受。记住，在一个有着移动的对象、移动的摄像机和注意力分散的用户的玩游戏的情形，抖动将比在一个试验台环境显得不那么引人注目。为了得到更好的性能，需要美工和程序设计师共同努力来减少多边形的数目并调整阈值，以达到一个合适的平衡。

1. 用户注意力

到此为止，我们只是考虑了一个对象在屏幕上的大小来确定将渲染哪个细节层次。另外一个应该考虑的因素是用户会往哪儿看。通常，我们期望用户注意离摄像机近的事物，但是可能需要一些关于特殊游戏环境的额外的知识来帮助我们了解用户可能会朝哪儿看。例如，在一个运动游戏中，一个由用户控制的角色将可能是注意的焦点，如一个拿球的角色，或者卷入当前比赛的角色。当用户可能正在观察一个物体或角色时，这种“期望的注意焦点”能通过修改放大率因子被用于细节层次选择算法中。在运动游戏的例子中，当角色处于用户的控制之下时，我们可以用一个比例参数（比如 1.1）乘以放大率因子。

2. 修改放大率因子

修改放大率因子的思想也可以用于其他情形。如果在一个游戏环境中，渲染质量比性能更重要（例如一个非互动的渲染过的剪辑场景），所有对象的放大率因子都能被修改得更高以便在更高细节层次上渲染对象。或者放大率因子能根据帧速率动态地修改——当帧速率下降时，将放大率因子修改得更低以选择更低级的多边形模型并提高帧速率。

3．限制模型或多边形的数目

如果一个场景有同一对象的多个实例，那么可能会有关于在每个细节层次能够渲染的对象数目的限制，施加这样的约束是为了在对象表示中节省内存。或者你可能仅仅希望限制高细节层次的多边形模型的最大数目以便提高性能。每个细节层次上的模型数目的限制能容易地嵌入选择算法内——用放大率因子将对象分类，然后在每个细节层次取出N个最大的对象，将剩下的对象降为下一个较低的细节层次。选择算法稍做修改就可以被用于选择对象，而使所有对象的总的多边形数目低于某个目标值。

4．渐进的网格

最后一个值得提及的问题是使用渐进的网格（progressive mesh），或者其他动态的细节层次方法。渐增的处理器性能和游戏模型中渐增的多边形数目正在开始使这些技术成为可能。有了这些技术，对象的多边形的数目就能急速地在一个连续的范围内变化。仍然需要为每一帧选择一个合适的多边形数目，于是放大率因子仍然是有效的。如果多边形的数目连续地变化，那么就不再需要滞变阈值了。然而，当多边形连续地从模型中被去掉或添加，会引起令人心烦的抖动效果，于是可能仍然需要用滞变阈值来决定何时改变多边形的数目。

4.10 八叉树构造

Dan Ginsburg

几乎在每个 3D 引擎的开发过程中，都必须解决为可见性确定和碰撞检测进行几何体裁剪的问题。有无数的数据结构和方法可以解决这个问题。大多数的解决方案在是几何体上施加约束，并且常需要 3D 美工明确地为引擎提供信息，比如 Portal（孔洞）的位置。与上述不同的八叉树（octree）是一种较简单的数据结构，能被用于空间细分任何形状的几何体。

本文研究的是取一个多边形输入集合并构造一个八叉树所必需的步骤，该树空间地分割了几何体。八叉树最适用于静态地形，不过也可以用于为一个场景中动态运动的对象存储附属表。八叉树可以作一个完全解决方案用于可见性裁剪、碰撞检测裁剪以及对象管理。

4.10.1 八叉树概述

总的来说，一棵八叉树只不过是每个节点最多有 8 个子节点的一棵树（一个无环有向图）。它是表现一个被立方体封闭的三维世界的理想结构。一棵八叉树的根节点包含一个立方体，它封闭了世界中所有的几何体。每个节点的子节点是 8 个同样大小的立方体，它们将双亲细为八分体（参见图 4.10.1）。当用户指定的试探式被满足时，细分停止：典型地，要么包围盒达到一个特定的大小，要么每个节点内包含一些最小数目的多边形。

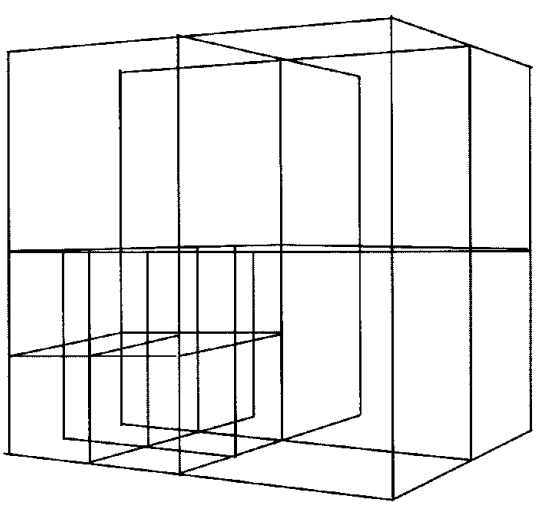


图 4.10.1 将一个立方体细分为八分体

位于每个节点的包围盒是为进行空间细分而应用一个八叉树的关键。每个节点包含了指向所有位于它体积内的多边形的指针。有了这个信息，你会开始看到这种数据结构的威力了。对于可见性确定，树根节点的轴对齐包围盒将相对于可视棱台而进行测试。如果它是完全可见的，它的所有几何体都将被渲染。如果只是部分可见的，将继续沿其子节点进行遍历。如果它完全落在可视棱台之外，遍历就可以结束了：它和它所有的子节点都是不可见的。关于使用八叉树的进一步的例子将随后给出。然而，首先检验一下构造一棵八叉树需要的特殊步骤是必要的。

4.10.2 八叉树数据

使用一棵八叉树分割几何体是一个典型地在预处理阶段所执行的步骤。一些工具将接受几何体的输入集合，并以产生八叉树数据作为输出，它们能够被应用程序实时使用。

八叉树中的每个节点至少要包含以下数据：

- **包围盒**——这是空间中八叉树的节点封闭的立方体。
- **几何体列表**——每个节点封闭了许多多边形，它们必须以某种方式存储在每个节点中。
- **子节点**——每个节点有最多 8 个子节点，必须在每个节点中存储指向每个子节点的指针。
- **相邻节点**——每个节点最多有 6 个相邻节点（每个立方体面一个）。为碰撞检测进行的树的遍历要求每个节点要有指向其所有相邻节点的指针。相邻节点允许碰撞算法沿着一个碰撞光线快速“走”过该树。这一点将会在稍后更详细地讨论。

4.10.3 建立树

建立八叉树的第一步是得到一个在世界中的所有多边形的列表。一旦这个列表创建好了，那么八叉树的根节点就可以被构造了。可以确定顶点列表中的任何分量 X，Y 或 Z 的最大绝对值 V。这个值被用来为该世界（它的跨度为从$[-V,-V,-V]$到$[V, V, V]$）创建包围盒。由定义，根节点的几何体的列表将包含世界中的所有的多边形。从根节点出发，世界现在可以使用一棵八叉树被递归细分。这里是建立八叉树算法的伪代码：

```
BuildOctree(Node N)
{
    if(NumPolys(N) > POLY_THRESHOLD)
        for(int i = 0; i < 8; i++)
{
            BuildNode(N->Child[i], i, N);
            BuildOctree(N->Child[i]);
        }
}
```

对于包含多边形数大于最低阈值的节点 BuildOctree()共产生了 8 个子节点。在产生 8 个子节点时，通过一个假设而简化了运行时代码，该假设为：若任何子节点存在于某节点内，

则全体 8 个子节点均在其内。若不建立这个假设，有的节点就可能没有相邻节点，这会使得碰撞检测的跟踪变得困难。

BuildOctree( )的核心是 BuildNode( )，它创建了节点数据，分两步实现：

1. 为节点创建包围盒
2. 确定哪些多边形位于节点的包围盒内部

为每个节点创建包围盒是一件繁琐的工作。索引 i 能被用于指定节点位于哪一个八分体内。那么包围盒将通过取父节点的包围盒并将其适当地细分而完全确定。包围盒的宽度、高度和深度将是其父节点包围盒的一半，并且以 8 个位置中的一个为中心，具体位置取决于 i。

确定哪些多边形位于包围盒内部有一点麻烦。在讨论这个问题之前，值得先来讨论一下如何在每个节点存储多边形列表。显然，在树的每个节点都存储一个多边形列表将使存储效率非常低。一个多边形有可能存在于多个节点内：一个父节点将总是包含其子节点中的多边形的一个超集。此外，多边形也有可能跨越节点的边界。解决方法之一就是沿着边界分裂这些多边形。然而，这会产生额外的多边形数据，可能对实时性能有不利的影响。相反地，多边形将有一个“帧数”值，而且实时渲染代码将负责保证每个多边形在每帧仅被渲染一次。

为每个节点存储几何体的一个可能的方法是存储一个区域 ID 的列表。然后，在编码程序的其他地方，对每个区域 ID 将存储共享多边形表中的一个索引表。这样在每个节点仅需极少的数据，而且确保了当多边形跨越多个节点时将不会被复制。

4.10.4 多边形重叠

给出了在每个节点存储多边形列表的方法，下一步就是创建一个算法来确定一个多边形是否存在于一个立方体中了。[Moller99]（第 10.9 节）提出了一种测试三角形是否与一个体素（voxel）相交的快速的方法。体素是一个以原点为中心的立方体，它的每一边的长度均为 1。这个算法易于被扩展到测试一个三角形是否与一个任何大小的世界对齐的立方体相交。应用的技巧是确定什么样的平移和比例缩放可以将立方体变换为一个体素。接下来在每个三角形上执行该变换。然后每个变换过的三角形对于一个体素进行相交测试。下面是算法的主要部分：

```
TriInCube(Tri T, Cube C)
{
    Vector Trans= C.Center;
    Vector Scale= 1.0 / C.Size;

    for (int i= 0; i < 3; i++)
        T.Vert[i]= (T.Vert[i] - Trans) / Scale;

    if (TriInVoxel(T))
        return true;

    return false;
}
```

4.10.5 相邻节点

每个八叉树节点的主要成分现在已经被填充了：包围盒、几何体列表以及子节点。这些是可见性裁剪需要的所有信息。然而，为了使用八叉树进行碰撞检测，立方体的6个面每个面的相邻节点都必须被确定。一个给定立方体面的相邻节点定义为与其邻接的尺寸等于或大于它的节点。相邻节点永远不会比它小，而且算法将搜索最适合的相邻节点（即最小的可能相邻节点也不会比该节点小）。

这个步骤需要在八叉树被完全构造好，所有的节点被创建之后才执行。取每个节点的每个立方体面并与树中同一层或更高层的其他立方体的面相比较来进行计算。两个立方体面相邻必须满足几个条件：

- 面的法线方向必须相反；
- 所有的源面上的顶点位于目标面上或其内部；
- 源面的尺寸小于或等于目标面的大小。

满足所有三个条件且尺寸最小的立方体被认为是相邻节点。

4.10.6 应用

正如先前所讨论的，构造好的八叉树的应用之一是对于静态几何体进行可见性判定。另一方面，八叉树也可以用于管理世界中的动态对象的可见性。实时代码中的每个八叉树节点将存储一个附属对象的列表，并且世界中的每个对象可以存储一个它所附属的节点的列表。然后，为了渲染场景，每个可见节点中的地形多边形以及所有附属于它的对象都被渲染。当一个对象移动的时候，它把所有附着的节点与自身相分离，并且重新附属于任何它位于其内部的新节点。惟一的技巧是对每个对象再存储一个“帧计数器”，以确保每帧只渲染该对象一次。（因为一个对象可以很容易地跨越多个节点）。

八叉树现在能用于在碰撞检测中进行裁剪了。考虑简单的光线碰撞测试的情形。两个点定义了一条碰撞光线：一个起点和一个终点。碰撞测试从寻找起点所位于的八叉树的叶子节点开始。线段在它相交的每个立体面被划分为子线段。新的子线段针对所有在它的节点内的几何体和对象进行测试。下一条子线段始于前一条子线段的终点，位于与它相交的相邻立方体面的节点内。这种遍历经过相邻节点继续进行，在每个节点与几何体及对象相碰撞，直到到达原始的终点。几个其他类型的碰撞测试（如轴对齐的包围盒以及球面测试），当使用八叉树时也作用得很好。

4.10.7 结论

八叉树是一种建立一个简单几何体裁剪系统的很有用的数据结构。本文试图对如何建立八叉树做了一个简要介绍。可以在这个结构上做更进一步的优化和增强，以便提高它的实时性能及其有效性（如增加阻塞剔除和深度排序）。请阅读参考文献中关于八叉树的进一步介绍和它们在3D图形学中的应用。

4.10.8 参考文献

[Foley87] Foley, van Dam, Feiner, and Hughes, "Computer Graphics: Principles and Practice 2nd Edition", 1987, p 550–555.

[Hoff] Hoff, Kenny, *"Fast ABBB/View-Frustum Overlap Test" www.cs.unc.edu/~hoff/research /vfculler/boxvfc/boxvfc.html*

[Moller99] Moller and Haines, "Real-Time Rendering", 1999, p. 206–211, 310–312.

[Suter99] Suter, Japp "Introduction to Octrees" April 13, 1999, www.flipcode.com/tutorials/ tut_octrees.shtml

4.11 松散的八叉树

Thatcher Ulrich

八叉树是将 3D 数据集划分到不同的包围体层次的一种典型有效的数据结构。对于有着众多物体的数据集，八叉树能够极大地加速可视棱台裁剪（frustum culling）、光线跟踪（ray casting）、接近度查询（proximity query），以及几乎任何其他的空间操作。

然而，普通八叉树的确有一些缺点。在本文中，我将集中详细讨论它的一个缺点，那就是较小的物体（依赖于它的位置）却可能被存储在一个具有非常大包围体积的八叉树节点中。当一个物体跨越两个较大的节点的分界面时，这种情况就会发生。这会在划分层次过程中产生“粘性（sticky）”区域，较小的物体却保持在树的较高层，而且降低了划分的效率。

有各种调整基本的八叉树数据结构的方法和算法，以减轻或避免这个问题，而且每种方法都有其独特的考虑。在本文中，我提出了这样的一种替换方法，即“松散的八叉树（loose octree）”。它与普通八叉树相比，主要优点是避免了物体划分中的粘性，产生了更为精确的空间数据库查询。对于某些应用，如在大量运动物体间进行相互碰撞检测，其效率的增加可能会非常显著。还有一个额外的较小的“副产品”，就是计算一个特定物体在树中的期望节点仅是一个简单的 $O(1)$运算。普通的八叉树也可以使用一个类似的技巧做到这一点，但是不如它直接。

它的主要缺点是，对一个特定的数据集，它倾向于使用比普通八叉树更多的划分节点。对树的深度进行限制能够缓和这种趋势，但我们必须意识到这一点。

4.11.1 四叉树

八叉树是一种 3D 数据结构。相对的 2D 数据结构是四叉树，它们有着同样的基本特性。对于松散四叉树也是如此，它只不过是松散的八叉树的 2D 版本。松散四叉树与普通的四叉树之间有着和松散的八叉树与普通八叉树同样的折衷，那么它在仅需在二维空间进行层次划分的应用中是有效的。

由于在 2D 空间形象化这些数据结构容易得多，所以在本文中我将使用以四叉树为基础的 2D 图示。而八叉树的原理与四叉树完全一样，且到 3D 的扩展也非常直接。

4.11.2 包围体

在一个传统的八叉树中，基本的节点包围体是一个立方体。所有与一个节点相关联的物体必须被完整地包含在节点的包围立方体中。每个节点最多可以有 8 个子节点，它们的包围立方体是将父立方体切割为 8 个相等的子立方体而形成的。图 4.11.1 说明了四叉树版本的包围体。

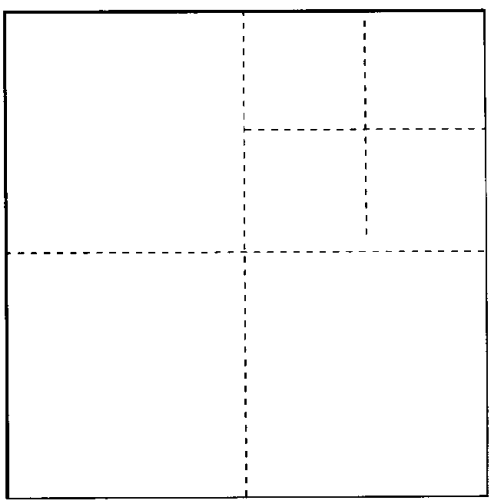


图 4.11.1 一个四叉树节点，其包围正方形以粗体显示，沿虚线被细分。每 1/4 区成为一个子节点。子节点也可以像图中右上角的 1/4 区那样被细分

子节点的包围体完全嵌套在父节点的包围体中，它们填充了整个父节点包围体且没有重叠。子节点也能以同样的方式被进一步细分。如果检验包围体的大小和间距，那么你可以看到它们遵循着一个有规律的模式。考虑包围立方体的边长：在树的根节点，立方体的边长等于它的世界尺寸。在每一个更深的层次，立方体的边长是其前一层立方体边长的一半。这样，包围立方体的边长计算公式为：

$$L(depth) = W / (2 \wedge depth)$$

这里 W 是世界尺寸，而 *depth* 是层数，代表一个节点由根节点分离的层数。根节点的 *depth* 值为 0。

在特定深度，包围立方体中心的间距也遵循同样的模式。在根节点只有一个节点，于是节点间距毫无意义。不过从 *depth* 为 1 开始，根节点的子节点的中心与它们相邻节点的间距就是 $W/2$ 单位了。每个随后的层次将节点间距减半。计算节点间距的公式为：

$$S(depth) = W / (2 \wedge depth)$$

所以立方体的边长和节点间距对于一个特定的深度是一样的。这很自然，因为在一个特定的树的层次，包围立方体被正巧装入世界体中，既没有间隙，也没有重叠。

4.11.3 划分物体

给定一个虚拟世界中的物体集，每个物体有着一些有限的包围体，八叉树能用于划分世界空间内的物体，以加速不同的空间操作，诸如可视棱台裁剪、光线造型、接近度查询等。可以使用不同的划分标准，但是典型的八叉树划分模式是：将一个给定物体与一个八叉树中的节点联系起来，该节点的包围立方体紧紧包含了整个物体的体积。这个节点可以通过树的递归遍历很容易地找到。这里是一些伪代码：

```
struct node {
    Vector3  CubeCenter;
    node*    Child[2][2][2];
    ObjectList Objects;
};


int Classify(plane p, volume v)
{
    if (v is completely behind p) {
        return 0;
    } else if (v is completely in front of p) {
        return 1;
    } else {
        // v straddles p.
        return 2;
    }
}


void    InsertObjectIntoTree(node* n, Object* o)
{
    int xc = Classify(plane(1,0,0,CubeCenter.x),
        o.BoundingVolume);
    int yc = Classify(plane(0,1,0,CubeCenter.y),
        o.BoundingVolume);
    int zc = Classify(plane(0,0,1,CubeCenter.z),
        o.BoundingVolume);

    if (xc == 2 || yc == 2 || zx == 2) {
    {
        // Object straddles one or more of the child
        // partition planes, and so won't fit in any
        // child node, so store it in this node.
        Objects.Insert(o);
    } else {
        // Object fits in one of the child nodes. Recurse to
        // find the correct descendant.
        InsertObjectIntoTree(Child[zc][yc][xc], o);
```

```
        }
    }
```

这是一种较好的、直接的层次划分方案，它能够应付任何可能的情况并且通常会产生相当好的划分。然而，它也有一个令人烦恼的怪异之处：如果一个物体跨在一个节点的任何一个划分平面的之上，那么物体就会存储在那个节点中。即使物体微小而节点巨大，这种情况也会发生；图 4.11.2 就是一个例子。实际上，如果你有很多较小的物体，那些位于根节点划分平面上的物体就通过用较小的、拙劣划分的物体填充根节点而“粘上”（clog up）它，并且降低空间操作的效率（如图 4.11.3）。

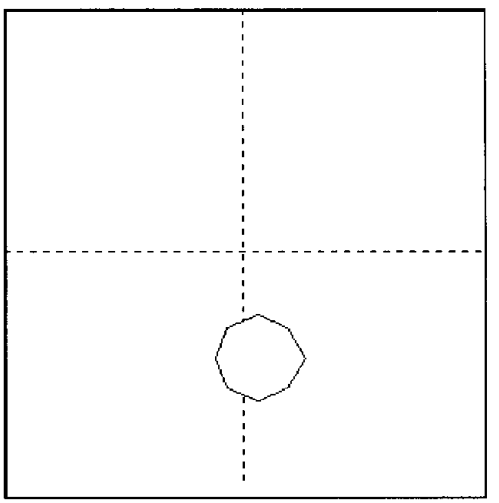


图 4.11.2 即使这个圆环与根节点（粗线条四边形）相比非常小，它也不能被放入一个子节点中，因为它跨越了一条（虚线的）划分线

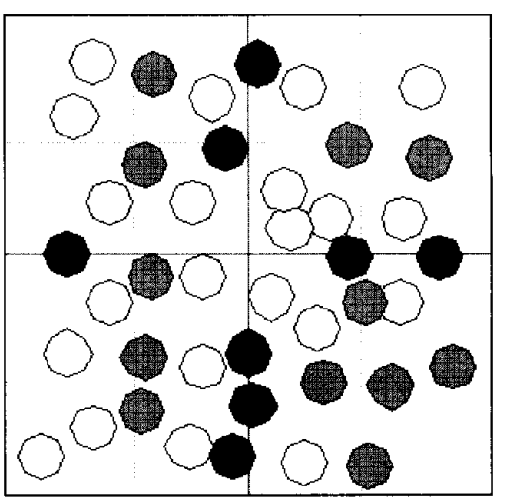


图 4.11.3 所有的物体都很小，但是那些以阴影表示的物体却由于它们跨在划分线上而粘在四叉树的较高层节点上

我把这个问题称为“粘性平面（sticky plane）”问题，意思是说树中较高层次的划分平面

吸附了与其节点相关的额外的物体，这就是“粘性”。有各种不同的方法能够解决这个问题。一种方法就是使位于划分平面上的物体分裂，然后对这些分裂碎片分别进行划分。一种方法是允许一个物体能够被不止一个节点所引用，于是一个物体就可以被粘性平面任何一侧的子节点所共享，而不是存储在父节点中。对于静态物体，这些方法都是有效的，但是它们处理动态物体时就不那么有效了。

4.11.4 使它松散

“松散的”八叉树方法采用一种不同的方针：它通过调整节点的包围体来解决粘性平面的问题。明确地说，就是“放松”包围立方体，但是令节点的层次和节点的中心不变。一个节点的包围体仍然是一个立方体，但是在传统的八叉树中立方体的边可能有长度 L，而在松散八叉树中立方体的边长将会是 kL，这里 $k>1$。这样，包围立体边长的计算公式就被修改为：

$$L(depth) = k * W / (2 \wedge depth)$$

另一方面，节点的间距仍与传统八叉树保持相同。这意味着一个节点的包围立方体现在与它的相邻节点的包围立方体相重叠了。图 4.11.4 显示了一棵松散四叉树的这种重叠。

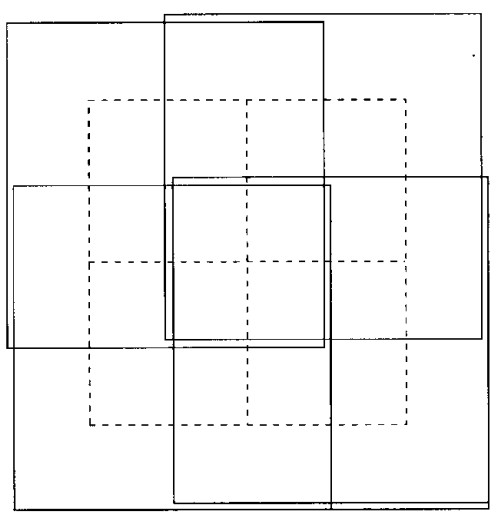


图 4.11.4 4个节点。传统的包围四边形以虚线显示。在松散四叉树中同样的4个节点有着以黑色显示的包围四边形。四边形被进行了轻微的偏移以使它们能够彼此区分

这种包围立方体的放松增大了被一个粘性平面吸附物体的最小尺寸。先前的一个有任意尺寸的穿过一个粘性平面的物体将被存储在平面的节点中，它有着更松散的包围立方体，较小的物体将会适合于其中的一个子节点（如图 4.11.5 所示）。一个物体必须多小才能避免被一个粘性平面粘住？它依赖于平面节点在树中的深度，以及我们所选择的 k 值。对于一个位于给定深度的节点，没有包围半径小于 $(k-1) * L/2$ 的物体会因跨在一条划分线上而被节点粘住。代替的是，由于子节点的包围体已经被扩大了，这样的物体可以适合在其中的一个子节点中。

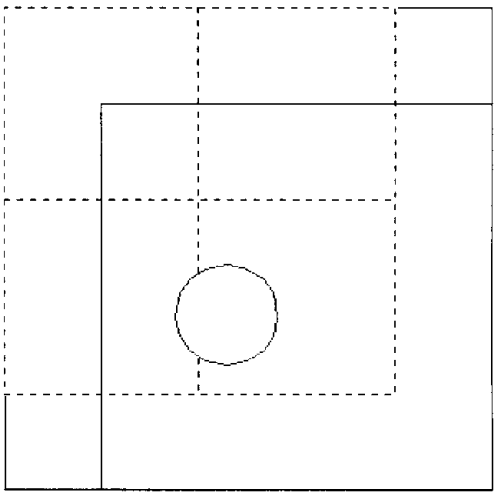


图 4.11.5 圆环不会适合于任何传统的子节点的包围正方形内，但是可以适合于右下角的子节点的松散包围正方形

那么，什么样的 *k* 值比较好？本文没有完全探索所有的可能取值，我建议以 *k=2* 作为一个有益的综合值。*k* 值比 2 小得多的树开始经历粘性平面的困扰，而 *k* 值远大于 2 的树会导致过分松散的包围体。

假定一棵八叉树的 *k=2*，我们可以写出一个极为简单的物体插入过程。基本原理是对于一个特定的物体，节点的深度可以被单独基于物体的大小而计算，而且树中那个深度的特殊节点完全根据物体的中心位置进行选择。为了得到深度的计算公式，注意在松散八叉树中一个给定的层次能够容纳任何其半径小于或等于包围立方体边长 1/4 的物体，而不管其位置如何。任何具有半径<= 1/8 包围立方体长度的物体应该放进树的更深的层次。例如，在图 4.11.5 中，注意不管物体是如何被放置的，它将适合于节点的一个包围正方形。

下面是层次选择（level-selection）公式的推导：

$L(depth) = 2 * W / (2 \wedge depth)$

令 $Rmax(depth)$=在 $depth$ 能被容纳的最大的物体半径。

$Rmax(depth) = 1/4 * L(depth) = 1/2 * W / (2 \wedge depth)$

令 $depth(R)$=第一个能够容纳一个半径为 R 的物体的树的层次。

$R <= Rmax(depth(R))$

$R <= 1/2 * W / (2 \wedge depth(R))$

$depth(R) >= log2(W / R) - 1$

$depth(R) == floor(log2(W / R))$

一旦深度知道了，在给定深度选择特殊的节点就很容易了——只要寻找距物体中心最近的节点就可以了。假设世界是以坐标系的原点为中心的，计算节点索引的公式为：

$index\{x,y,z\} = floor((object.\{x,y,z\} + W/2) / S(depth))$

注意这个过程并不是十分理想。它没有为所有情况实际找到可能的最紧密的包围节点(参见图 4.11.6)。为了得到最后的紧密信息，我们可以首先使用上述公式寻找候选节点，然后检验最接近于物体的子节点，看物体是否适合于它。

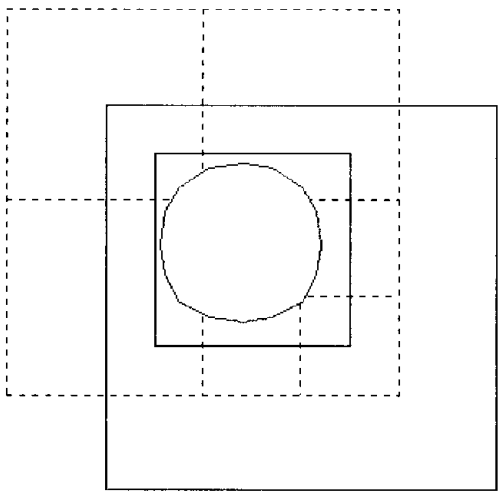


图 4.11.6 简单的放置公式会把圆环放入一个由大的黑色四边形包围的节点内，但是由于它的特殊位置，圆环更适合于由较小的黑色正方形包围的左上节点

在松散八叉树上执行空间操作非常类似于传统的八叉树。例如，下面是用视域四棱锥裁剪进行渲染的伪代码：

```
enum Visibility { NOT_VISIBLE, PARTLY_VISIBLE, FULLY_VISIBLE };

voidNode::Render(Frustum f, Visibility v)
{
    if (v != FULLY_VISIBLE) {
        v = ComputeVisibility(this.BoundingBox, f);
        if (v == NOT_VISIBLE) return;
    }

    this.ObjectList.Render(f, v);

    for (children) {
        child.Render(f, v);
    }
}
```

完全一样的算法也能用于传统的八叉树；惟一的区别是 this.BoundingBox 会更小。

4.11.5 比较

为了比较松散八叉树和普通的八叉树，我写了一个以松散四叉树与普通四叉树为基础的测试程序。该程序假定了一个 2D 正方形虚拟世界，每一边有 1000 个单位。生成一些圆形的物体填充该世界。每个物体有一个位置和一个包围半径，它们是被随机选择的以适应世界的边界。然后 2D 截面的某个数字（比如边）被生成，有着一个固定的视角，以及一个随机的位置和方向。物体首先用一个传统的四叉树来划分，然后对于每个截面，数据集被进行可见性物体查询。将潜在位于截面内部的物体数目和实际上在截面内部的物体数目进行统计。接下来，用一个松散四叉树对物体进行预划分，然后运行同样的截面测试并收集同样的统计数字。

表 4.11.1 总结了一些例子运行的结果。

表 4.11.1 一些例子运行的结果

测试参数	一般四叉树			松散四叉树		
tree max depth = 5 100 frusta FOV = 45°	物体可能可见	物体实际可见	考察节点	物体可能可见	物体实际可见	考察节点
500 个物体 obj min radius=30 obj max radius=30	18 859	6 883	2 976	9 442	6 883	7 024
1000 个物体 obj min radius=15 obj max radius=15	31 133	15 173	7 265	22 457	15 173	8 815
2000 个物体 obj min radius=5 obj max radius=100	55 451	29 935	9 102	45 209	29 935	9 075

注意在松散四叉树上的截面查询与在传统四叉树上的同样查询相比，一般返回了较少的“可能可见的”物体。另一方面，松散四叉树的查询通常不得不检验更多的节点。于是，对于截面裁剪，二者之间的差异是值得注意的，但是并不十分显著。

当观察一个物体之间查询时，情况会变得非常有意思。比如碰撞检测。我在测试程序中增加了一个测试，其中每个物体都为了与数据集中所有其他的物体相联系而被进行测试，而且要收集检验的统计数字。先前用过的同样的数据集的测试结果如表 4.11.2 所示。

正如你能看到的，对于这些数据集，松散四叉树需要为同样的查询做少得多的物体到物体测试。松散四叉树进行了更多的物体到节点测试，但是总的来说，松散四叉树对于这种类型的查询显然更加有效。

表 4.11.2 测试结果

测试参数	一般四叉树			松散四叉树		
tree max depth = 5	物体间的接触	物体对物体的检测	物体对节点的检测	物体间的接触	物体对物体的检测	物体对节点的检测
500 个物体 obj min radius=30 obj max radius=30	3 034	53 469	7 351	3 034	9 125	24 839
1000 个物体 obj min radius=15 obj max radius=15	2 730	113 989	18 040	2 730	24 609	45 658
2000 个物体 obj min radius=5 objmax radius=100	7 094	345 377	38 107	7 094	89 276	89 312

4.11.6 结论

八叉树是一种功能极强的工具。然而，在某些情况下，你可能想修改传统的八叉树方法以更好地适应你的问题。松散八叉树就是这样一种变形，它避免了传统的八叉树的粘性表面问题。在有着大量的相互作用的、动态物体的情况下（如一个特殊的有着粒子之间碰撞的粒子系统），松散八叉树是一个超过传统八叉树的特别好的选择。松散八叉树对于一般的空间划分任务，如截面裁剪，也执行得很好。

4.12 独立于观察的渐进网格

Jan Svarovsky

渐进网格（Progressive Mesh，PM）是一种能够实时改变其细节层次的基于三角形的网格，在每次做出得到或丢弃两个三角形的决定时，要同时尽可能地保存它的原始形状。它可以在任何细节层次上被绘制出来，从创造它的传统网格到一个用细节减小试探法确定的最低细节层次的“基网格”，甚至可以小到根本没有多边形的程度。

典型的情况是，在远处以较低细节层次渲染网格，更多的系统资源可用于在前景中绘制更高分辨率的网格。图形引擎的全局细节层次也以正在运行的计算机的能力为基础。

首先我将介绍一下渐进网格，并对关于这个主题的不同变种进行一些讨论。基于此，我将论述一个把传统的网格数据转换成渐进网格的算法，以及渲染这些网格的一些有效而简单的代码。

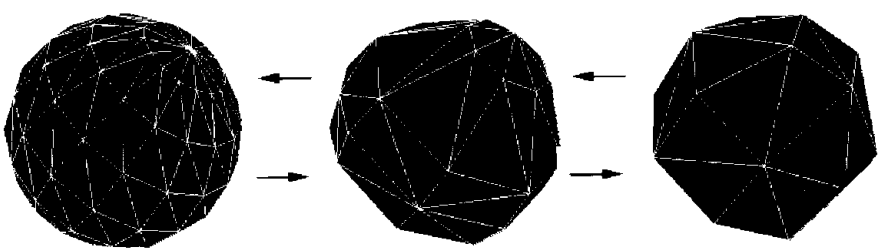


图 4.12.1 一个细节变化的渐进网格

4.12.1 渐进网格概述

渐进网格的基本原理可以简单地描述为：取一个网格，重复地决定网格中重要性最小的边，并且通过使位于它两端的两个顶点位置相等的方法删除这条边。这种边塌陷（edge collapse）操作典型地使两个共边的三角形合二为一。由顶点分裂（vertex splits）对塌陷的逆操作将恢复网格的细节。

在这方面有人做了很多工作，并有一些公开文献，特别是[Hoppe96，Hoppe97，Hoppe98]。图 4.12.2 概述了单元可逆操作和通用术语。

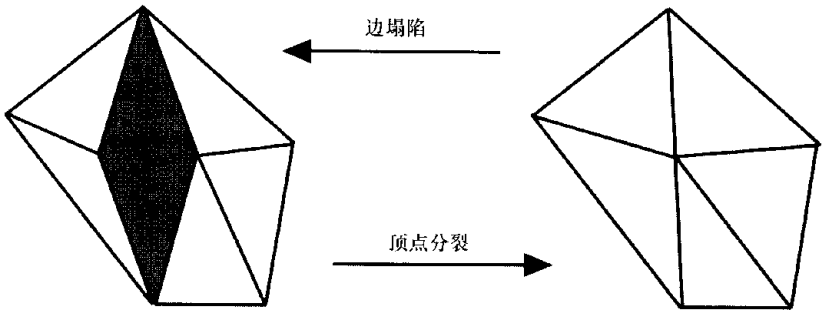


图 4.12.2 一个网格加细（顶点分裂）或约简(边塌陷)的步骤

4.12.2 关于这个主题的变种

有了这个基本前提，为了得到更好的实现细节可以进行各种改进。我在这里只是简要地提到它们；更详尽的讨论可参看[Svarovsky99]。

1. 何时进行顶点塌陷，塌陷到哪里?

当两个顶点塌陷成一个的时候，把顶点放在哪里就有一个选择问题。可以经过计算将它放在多边形网格所试图表现的虚构的平滑表面上。或者可以把点正好放在它替换的两点正中间，你可能会认为这样做更划算，因为你将不必储存一个预先计算的新点。最后，你也可以干脆选择保留某一个正在塌陷的原始顶点（图 4.12.3）。

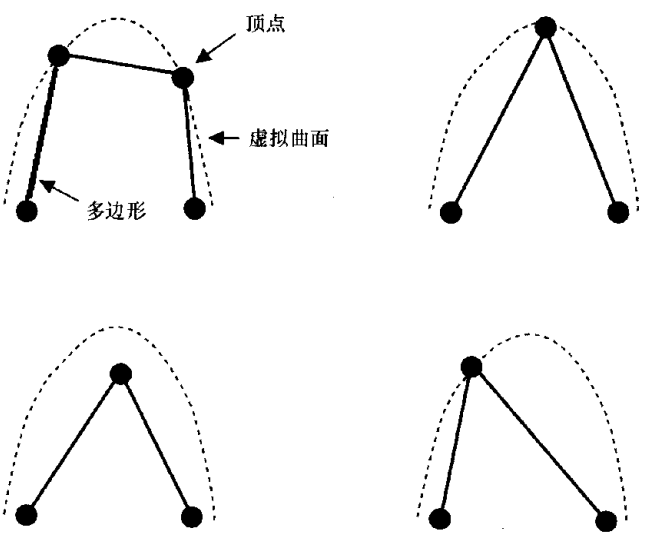


图 4.12.3 边塌陷所产生顶点的位置选择

a：较高细节层次的网格。b：新点位于虚构的表面上。c:选择两点的中点。d：挑选其中一个点 —— 简单

中点系统有着使凸边形物体在失去细节的时候变得更小的缺点。精确预计算点系统占用两倍的存储空间，或者占用额外的 CPU 时间来计算在线上的新点。保留某一个点最简单，占

用的存储空间最少，并且物体的表面积没有过多的损失，常常能较好地表现原始形状，例如特别是塌陷立方体的对象的角和沿着立方体的一个面的一个点的时候。虽然它缺乏计算新点的适应性，但是它的强大的优势是它不需要对顶点数据进行实时改变或产生新顶点。它也是我将在这里使用的系统。

2. 独立于观察的渲染与依赖于观察的渲染的对比

从基网格中的一个顶点开始，每个顶点分裂序列可以被形象化为一棵二叉树，每个顶点分裂成两个新的顶点（虽然在这个系统中我只是在一个原始的顶点上增加新顶点）。分裂既可以在树形结构的左边进行，也可以给定某一固定的次序（图 4.12.4）。

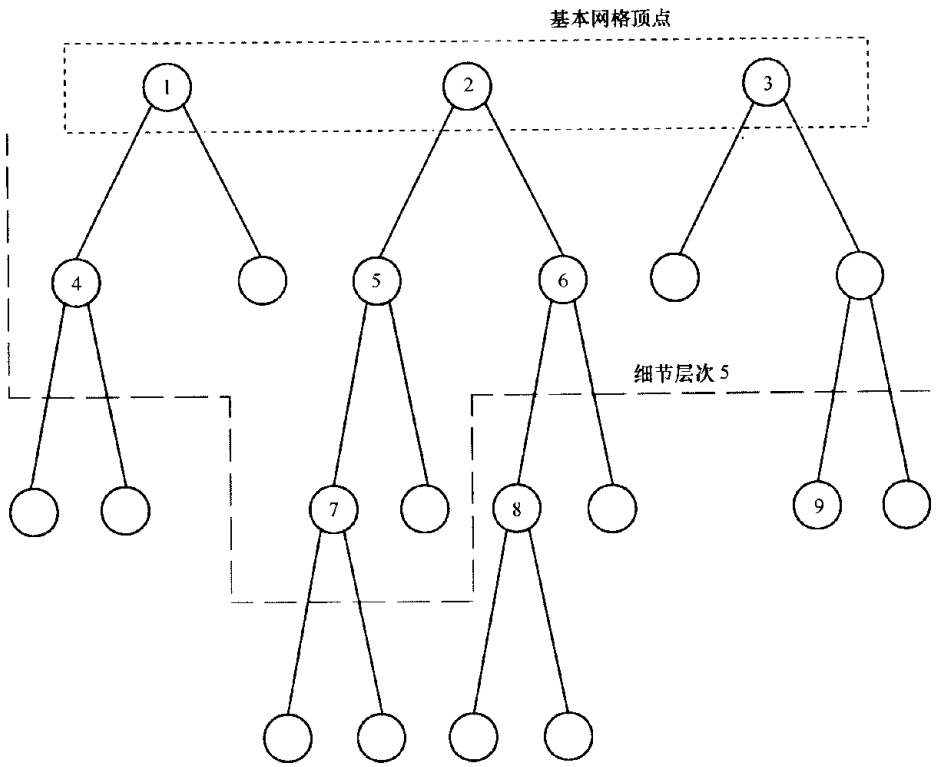


图 4.12.4 一个顶点分裂树的森林。虚线代表当渲染细节水平为 5 的网格时将会用到的顶点

独立于观察的网格使用一个固定的顺序进行边塌陷，因此可以进行离线计算，而且可以抛弃这种树形表示法。如果你以某种方式保持该树，你可以改变扩展的那个节点。这在为虚线定位时有效地给了你更多的灵活性[Hoppe97]。这种独立于观察的 PM 可用来在更靠近观察者的网格部分或轮廓边缘上给出更多细节。

依赖于观察的 PM（VDPM）能够更有效地使用三角形计数，因为它在边塌陷次序的选择上有更多的灵活性。然而在我看来，在新式系统中这是不恰当的，因为两种类型的渲染器在效率上存在巨大差距。对于一个在视觉上更令人愉快的风景来说，一个 VDPM 渲染器可以使用较少的三角形，但是与选择细节层次和对涉及的数据进行处理所必须增加的处理器时间

投入相比，这点节省就显得微不足道了。

一个独立于观察的PM（VIPM）网格中三角形和顶点可以被排序，这样首先消失的部分会排到表的尾部，因此当使用较低细节层次的时候就不被历经或不会“挡道”。这也可能导致有趣的渐进文件格式：读得越多，得到的网格细节层次就越高[Hoppe98]。

因为只有一个塌陷次序，对整个网格来说只有一个细节层次。如果你与网格的一个部分较接近，因此想要在较高细节层次上渲染这部分，那么其余的所有部分也将必须在较高的细节层次上被渲染。然而，在实际的游戏情形中，一个大的物体可以被细分成独立（但是可能互相交叉）的部分，它们可以被进行独立于观察的渲染。既然你有了这些分离的部分，你就可以为它们分配一些游戏代码，于是它们就变得具有一点交互性了，例如在一个空间站上的窗口、天线、雷达反射器，或者一个风景中单独的小屋、树木等等。

像起伏的山峦那样的连续网格能用一个定制渲染器来构造，这样的渲染器用的是另一种方法如ROAM算法[Duchaineau97]，在远处用到的多边形更少。关于这些用于特殊网格地形学的其他依赖于观察的系统的讨论超出了本文讨论的范围，本文把注意力放在一般三角形网格的渐进网格上。

4.12.3 边缘选择函数

我相信，一旦你有了一个可以给出相当满意结果的边缘选择系统，复杂的估价函数就没多少用处了。我把文献中讨论的一些方法的实现作为练习留给感兴趣的读者。我已经列出了一些有关不同工作的参考文献，其中[Lindstrom99]是一篇综述。最好是构造允许美工对塌陷序列的自动生成进行干预的编辑器。据我的经验，在花几天时间构造网格之后，美工相当乐意花一些时间调整，看看它在较低的细节层次上，特别是在非常低的细节层次上（在那里只有几个可以调整的多边形）看起来如何。正是在这些较低层次上，自动化的系统才会有最多的麻烦。

我已经描述了一个在CD上的示例代码中实现的非常简单的函数。它基本上是在围绕三角形的运动量的基础上建立起来的。

4.12.4 难处理的边

禁止某些特殊情形的边可以简化算法。这些情形源于三角形共享在三维空间中同样的点，但是不共享一些其他的顶点数据，例如顶点法线、纹理类型或纹理坐标。令三角形指向共享的纹理坐标和指向共享的顶点位置是件特别复杂的事情。为避免这一点，而且更适宜于当前的图形硬件，我们的顶点包含所有的纹理坐标、法线和位置信息。这样，网格将在相同的位置中包含多个顶点，不过带上了不同的材质信息。

如果一条正被删除的边包含了这些重复的顶点，且因此沿这个边的三角形不共享顶点，就把它当作同时发生的两个边塌陷进行处理。当一个附近的三角形只涉及一个塌陷的边的一个端点时，且这是即将要消失的那个端点的时候，就产生了问题（图4.12.5）。

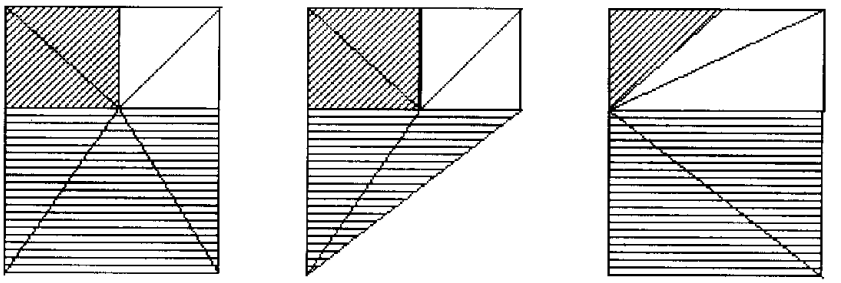


图 4.12.5 当一个额外的顶点必须被生成的时候。a：原始形状。b：轻微的塌陷。c：不同的复杂塌陷

可以为这个材质生成一个额外的顶点，它会保持冗余直到更低的细节层次需要它为止。这种效率上的降低非常轻微，因为极少的情况下才需要生成这些顶点。你可以通过禁止这些塌陷来避免这种情况，不过这会限制网格能够减少到的最低多边形数目。典型的情况是，当这些边在剩余边中占较大的百分比时，多边形数目将会很低，以致于渲染器的调用开销引起更多细节层次的丢弃并不能真正加快渲染速度。对象也可能是无关紧要的（就像由决策函数确定它们成为较低细节层次那样），那么你可以根本就不把它们画出来！在写本文的时候，多数商业化的 PM 系统不允许进行这些类型的塌陷。

为简明起见，这里对系统做进一步的简化。所给的程序不包含共享相同位置的多个边的工作区。这样，我们可以去掉所有检查边一致性的代码、禁止某些塌陷的代码、使一些边塌陷同时发生的代码。这隐含了所有的网格必须是平滑的和连续纹理的，但是，正如在所附 CD 上的例子中所看到的，细致地构造网格仍然可能产生许多更一般的形状。

4.12.5 实现

1. 渲染器

对于多数帧来说，网格细节层次将不改变，那么用于进行渲染的数据结构对图形系统来说要尽可能有效，这是实质性的要求。这里，我们可以像对一个标准网格渲染器那样排列数据：

```
struct PMMesh
{
    int NumMaterials;
    struct PMMaterial *Materials;
};
```

网格由一个材质数组构成：

```
struct PMMaterial
{
    PMTexture *Texture;
    struct PMVertex *Vertices;
    int *Indices;
```

```
    int NumVertices, NumIndices;
};
```

每种材质有一种纹理（或者在一个多通道（multi-pass）系统中可能有几种纹理，诸如一个几何（凹凸）映射，一个颜色映射和实际的基纹理），以及一个顶点数组。它也拥有一些三角形，它们简单地索引到顶点数组中。注意代替一个 PMTriangle 数组，有一个 3 倍于顶点数组中索引数目的数组。出于效率的考虑这会稍后在 EdgeCollapse 结构中进行，而且如果你想记录每个三角形更多的信息，要恢复到三角形数组是很繁琐的。

```
struct PMVertex
{
    vector3 Position, Normal;
    float U, V;
};
```

每个顶点包含位置、光照和纹理信息。用这种方法，材质彼此间是非常独立的，而且网格看起来像是一个连续的物体，因为在不同材质中的一些顶点的位置是相同的。

2. 渐变（Morph）顶点还是弹射（Pop）顶点?

在这种实现中不用进行任何顶点渐变——顶点只是弹入和弹出实体。这样做代价低廉，而且以我与游戏小组的经验来看，实际上看起来要比渐变更好。有这个令人吃惊的结果是因为对于一个给定的多边形数目，网格要与它特有的形状尽可能地接近，而不是在当前形状与下一个更低层次之间的某处被混合。实际上在实践中弹出比做渐变的额外代价要少（特别是必须编辑顶点数据）。

3. 渐进网格渲染仅仅影响三角形列表

因为边塌陷在两个相关顶点中保留了一个，这种渲染器仅仅修改了三角形列表，而对于顶点数据没有影响。这意味着顶点数组能保持不动（而且在一些新式的硬件中，被预处理到一些更有效的格式），而且也能在同一网格的多个关联之间共享。三角形列表将随时间而修改，而且必须对每个网格的每个有效关联复制一次。

这也意味着顶点位置调节（比如动画）可以相当独立于渐进网格发生，只要你不介意当顶点相对于彼此移动的时候，塌陷的顺序不会改变。动画系统只是必须处理顶点可以来去的事实，而不是必须被连续地使用。

4. 较低细节三角形和顶点优先

关于渲染器要注意的一点是顶点和三角形已经被脱机排好序了，而且总是位于表尾的三角形和顶点被一个塌陷变得冗余。渲染器将总是在同样的地方开始提交三角形和顶点表，只是变化长度而已。关于数据结构生成的讨论将表明这是如何成为可能的。

这的确意味着除非你用其他的方式建立三角形条带或扇片，否则你将总会为图形硬件提交一个三角形的索引表。有意思的是，表中的邻接三角形常常共享顶点，这在许多系统中和三角形条带及扇片有同样的好处。因为至少在一条塌陷边的两侧有多对三角形都会在三角形表中彼此相邻。

5. 可逆的边塌陷表

另外一种渲染器数据结构描述了可逆的塌陷边序列，它改变了网格的细节层次。每次边塌陷丢掉一个顶点、一个或多个三角形，并且改变了剩余的三角形使用的顶点。有一个所有对象的边塌陷表，尽管不同的个别塌陷影响了不同的材质。换句话说，每种材质可以有一个塌陷表，但是这些塌陷表将不得不以某种方式联系在一起，以使得对象的接缝不会裂开。

```
struct PMEdgeCollapse
{
    float Value;
    PMMaterial *Material;
    int NumIndicesToLose, NumVerticesToLose,
    NumIndicesToChange;
    int *IndexChanges;
    int CollapseTo;
};
```

CollapseTo 成员声明哪个顶点应该置换所有涉及被表尾丢弃的顶点。所有这些改变都被存储在 IndexChanges 数组中。当细节层次再次增加时，这个操作可以为顶点分裂进行简单的逆变换。

当一个塌陷发生时，一些三角形消失了（NumIndicesToLose），一个或更多的顶点被变成冗余的（NumVerticesToLose——丢弃一些三角形会使一些顶点变得毫无用处），并且一些剩余三角形的索引也将被改变（NumIndicesToChange），当然，在顶点分裂过程中相反的情况会发生。

由于材质是非常独立的，有时两条边的塌陷必须被一次执行，以保持网格的接缝尽可能多。如同先前讨论过的，这是考虑到这两条边在空间中其实是同一条边，于是必须塌陷在一起，即使它们涉及不同的顶点。引擎必须连续地比较下一条边塌陷或顶点分裂的值，它们可以在以它的位置和其他变量为基础的网格的查询层次上被执行。

在这里提出的简单系统中，并没有考虑这些边，不过通过简单地为所有的边赋予同样的位置和同样的优先级，它们可以被处理得很好，即使与数据结构并不相连。

6. 离线计算

这里我将假设网格数据已经被以某种方式装载到了一种适合的格式中。为了可读性更好，算法将写得最简单而不是最高效，尤其是因为我们并不太关心离线计算的代价。

算法过程能被概述为重复决定哪些是将被丢弃的数据，并将它从网格中移除，同时生成以后渲染需要的边塌陷数据。当决定了一个顶点将被移除时，所有涉及它的三角形都必须更改，而且任何退化的三角形都交换到数据表的尾部。类似地，顶点也移动到剩余的顶点表的尾部。

当然，将三角形和顶点交换到它们的表尾改变了在其他边塌陷结构以及剩余网格中所有涉及它们的一切。其他代码（例如一个动画系统将使用同样的网格）可能需要知道顶点的重新排序。

7. 建议的离线计算优化

观察那些代码，显然在网格中额外的临时连接性信息将是有用的。例如，代码经常强制查找“所有与这个顶点有关的三角形”。代码也为下一次边塌陷重复搜索所有的三角形。如果你把候选的边塌陷都放在一个优先堆中，将有可能得到巨大的性能改善。

8. 边选择改进

最有效的边选择改进是使那些网格中影响不连续性的边变得不太可能塌陷。这使得更多的网格形状成为可能，而且也允许对象被进一步细分为子对象。每个网格细分给出了一个在细节层次选择方面的额外的自由度——请看前述关于使小屋和树木与在它们下面的风景网格相分离的讨论。参见图 4.12.6。

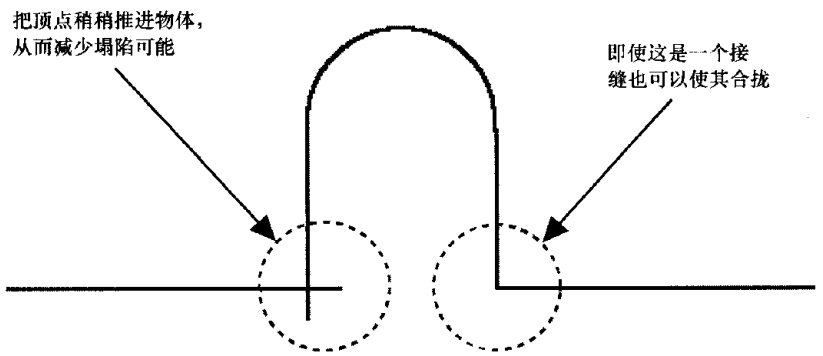


图 4.12.6 一个网格变为两个巧妙相交的网格，或者是两个重合为一个网格

9. 渐进网格进一步的变异

代替每次都改变顶点分辨率的细节层次，你可以存储几个预计算的不同分辨率的索引表。这些细节层次间的改变当然会变得更为明显。如果多边形数目或帧速率是非常高以致于弹出不成问题，这个系统就会变得更加有用。

它的另一个优点是你可以丢掉边塌陷表，与你用这种新方法存储的额外的索引表比较而言，塌陷表实际上是一个非常大的数据结构。你也可以丢弃塌陷/分裂代码，而且不需要为每个对象的每个有效实例建一个分离的索引表。渲染变得更简单了——回到了法线网格渲染，只不过是用代码为每个物体选择特定瞬间要用的索引表。

4.12.6 源代码

渐进网格的生成和渲染的代码包含在 CD 中。是以一种不依赖于系统的通用方式写的。

4.12.7 参考文献

[Duchaineau97] Duchaineau, M. et al, ROAMing Terrain: Real-time Optimally Adapting Meshes, 1997, http://www.llnl.gov/graphics/ROAM/roam.pdf.

[Garland97] Garland, M., and Heckbert, P.S., Surface Simplification Using Quadric Error Metrics, Siggraph 1997 Proceedings, pp. 209–216, August 1997.

[Hoppe96] Hoppe, H., Progressive Meshes, Siggraph 1996 Proceedings, pp. 99–108, August 1996.

[Hoppe97] Hoppe, H., View-dependent refinement of Progressive Meshes, Siggraph 1997 Proceedings, pp. 99–108, August 1997.

[Hoppe98] Hoppe, H., Efficient implementation of progressive meshes, Computers & Graphics, Vol. 22(1), pp. 27–36, 1998.

[Lindstrom99] Lindstrom, P., and Turk, G., Evaluation of Memoryless Simplification, IEEE Transactions on Visualization and Computer Graphics, Vol.5(2), April–June 1999.

[Ronfard96] Ronfard, R., and Rossignac, J., Full-Range Approximation of Triangulated Polyhedra. Eurographics 1996 Proceedings, in Computer Graphics Forum, 15(3), August 1996, pp. 67–76.

[Svarovsky99] Svarovsky, J., Extreme Detail Graphics, Game Developer's Conference 1999 Proceedings, http://www.svarovsky.freeserve.co.uk/ExtremeD.

4.13 插值的3D关键帧动画

Herbert Marselas

关键帧（Keyframing）是一种简单有效的创建3D动画对象的方法。然而，由于每个关键帧仅代表了对象运动的极端情况，这会使对象在不同位置间表现出跳跃。

4.13.1 线性插值

一种解决方案是增加更多的关键帧以使得关键帧之间的变化抖动得不那么厉害。另一种更经济的方法是使用插值法用程序创建动画的中间帧。

插值——也称为混合（blending）、变形（morphing）或者渐变（tweening），是在两个现有位置之间生成一个新的位置的过程。在现在的情形下，我们插值两个已知的关键帧位置p_0和p_1并得到一个新的位置$p(t)$。

最容易的插值法是线性插值。在这种情形下，在两个邻近的关键帧p_0和p_1的位置之间画一条直线，然后计算新的位置$p(t)$存在于直线上的何处（图4.13.1）。

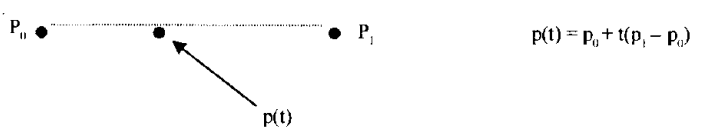


图4.13.1 线性插值的示例及公式

给出新的动画位置的期望时间、总的关键帧数目、动画的总时间，就可以计算两个邻近的关键帧之间的点了。

函数calculateFramePercentage说明了这一点。给出动画中总的关键帧的数目、动画的总时间，以及期望时间，新位置两边的关键帧和两帧间的百分比被计算并返回。

```
void calculateFramePercentage(long dwTotalAnimFrames,
    float fTotalAnimTime, float fDesiredTime,
    long &dwFirstFrame, long &dwSecondFrame,
    float &fPercentage)
{
    // determine which frames are involved
```

```
    float fTimePerFrame = fTotalAnimTime /
      (float) dwTotalAnimFrames;

    dwFirstFrame = 0;

    if (fDesiredTime > fTotalAnimTime)
      fDesiredTime -= fTotalAnimTime;

    for (float f = 0.0f; f <= fDesiredTime; f += fTimePerFrame)
      dwFirstFrame++;

    // set first frame

    if (f > fDesiredTime)
      dwFirstFrame--;

    if (dwFirstFrame < 0)
      dwFirstFrame = dwTotalAnimFrames - 1;
    else
      if (dwFirstFrame >= dwTotalAnimFrames)
        dwFirstFrame = 0;

    // set second frame

    dwSecondFrame = dwFirstFrame + 1;

    if (dwSecondFrame >= dwTotalAnimFrames)
      dwSecondFrame = 0;

    // calc the percentage

    fPercentage = (fDesiredTime - ((float) dwFirstFrame *
      fTimePerFrame)) * fTimePerFrame;
  } // calculateFramePercentage
```

首先，**calculateFramePercentage** 经过每帧时都增加，直到它找到了正好在期望时间之前的关键帧。这假定了每个关键帧都有同样的持续时间。如果关键帧没有设置为统一的时间间隔，该函数必须做相应的改变。

第一个关键帧找到以后，相对于动画中关键帧的数目再次对它进行检验。然后，第二个关键帧由第一个关键帧数字加 1 确定。第二个关键帧的数字也相对于动画中总的关键帧数目进行检验。这个代码假设了动画将在播完最后一帧后循环回到初始帧。

需要注意的是 **calculateFramePercentage** 函数，正如本文中的所有函数一样，是为了更好的可读性而不是性能而提出的。一种容易的改进性能的方法是预计算一些值，如 **fTimePerFrame**。

4.13.2 对顶点和法线进行插值

当两个关键帧和它们之间的百分比被确定之后，这些数据现在能被用于产生新的动画帧了。CombineVertices 函数说明了使用这些值将选择过的关键帧的顶点结合为新顶点。

```
void combineVertices(long dwVertexCount, float fPercentage,
    vector3 *pFirstFrameVertices,
    vector3 *pSecondFrameVertices,
    vector3 *pCombinedVertices)
{
  for (long i = 0;
    i < dwVertexCount;
    i++, pFirstFrameVertices++,
    pSecondFrameVertices++, pCombinedVertices++)
  {
    *pCombinedVertices = *pFirstFrameVertices +
      fPercentage * (*pSecondFrameVertices -
      *pFirstFrameVertices);

  }
}
```

calculateFramePercentage 计算出的百分比用于把两个关键帧中的顶点组合到一个它们之间的新位置中。

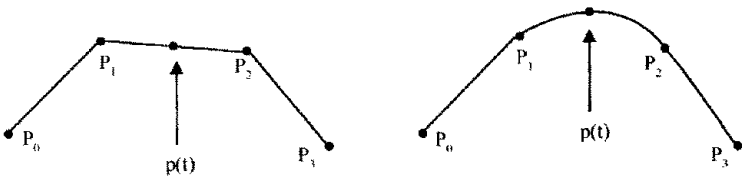
这种将两个关键帧的顶点相组合的方法同样也能应用于组合关键帧的法线。如果关键帧法线在插值之前被规一化了，那么组合的值将不必再被规一化，除非在两个法矢量之间有很大的差异。

如果在每个关键帧中分别存储面法线（为了进行背面裁剪）和顶点法线（为了光照）的列表，可以在性能上有所改善。面法线必须总是被进行插值，但如果你试图提高性能的话，顶点法线的插值可以跳过。虽然这意味着顶点的光照可能会不正确，但在许多情况下，用户不会注意到这些差别。

4.13.3 Hermite 样条插值

线性插值的一个缺陷是：一些插值过的动画帧可能有或大或小变形的倾向。为了解决这个问题，必须使用一个稍微复杂些的插值系统。下面的 Hermite 样条插值法考虑了在期望位置两侧的两个关键帧（参见图 4.13.2）。

与 calculateFramePercentage 类似，calculateFramePercentageSpline 确定了在期望的动画时间前后的关键帧。此外，在这两个帧前后紧挨着的关键帧也被计算出来。这些额外的关键帧被用于新位置的精确计算。



p(t) 线性插值　　p(t) 样条插值

图 4.13.2　线性插值与样条插值的对比

```
void calculateFramePercentageSpline(long dwTotalAnimFrames,
     float fTotalAnimTime, float fDesiredTime,
     long &dwFirstFrame, long &dwSecondFrame,
     long &dwThirdFrame, long &dwFourthFrame,
     float &fPercentage)
{
   // determine which frames are involved

   float fTimePerFrame = fTotalAnimTime /
     (float) dwTotalAnimFrames;

   dwSecondFrame = 0;

   if (fDesiredTime > fTotalAnimTime)
     fDesiredTime -= fTotalAnimTime;

   for (float f = 0.0f; f <= fDesiredTime; f += fTimePerFrame)
     dwSecondFrame++;

   // set second frame

   if (f > fDesiredTime)
     dwSecondFrame --;

   if (dwSecondFrame < 0)
     dwSecondFrame = dwTotalAnimFrames - 1;
   else
     if (dwSecondFrame >= dwTotalAnimFrames)
       dwSecondFrame = 0;

   // set frame before second frame

   dwFirstFrame = dwSecondFrame - 1;

   if (dwFirstFrame < 0)
     dwFirstFrame = dwTotalAnimFrames - 1;

   // set upper frame

   dwThirdFrame = dwSecondFrame + 1;
```

```
    if (dwThirdFrame >= dwTotalAnimFrames)
      dwThirdFrame = 0;

    // set frame after the third frame

    dwFourthFrame = dwThirdFrame + 1;

    if (dwFourthFrame >= dwTotalAnimFrames)
      dwFourthFrame = 0;

    // get the upper percent

    fPercentage = (fDesiredTime - ((float) dwSecondFrame *
      fTimePerFrame)) * fTimePerFrame;
} // calculateFramePercentage
```

4个关键帧的位置被用于下面的方程计算新的 $p(t)$：

$$p(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (t^3 - t^2)m_1 + (-2t^3 + 3t^2)p_1$$

$$m_i = \left(\frac{1-\alpha}{2}\right)((p_i - p_{i-1}) + (p_{i+1} - P_i))$$

第1个和第4个关键帧被分别用于计算第1个和第2个关键帧之间以及第3个和第4个关键帧之间的切线 m_i。

4.13.4 对顶点进行样条插值

CombineVerticesSpline 函数说明了计算切线和之后的 Hermite 样条插值的位置 $p(t)$。

```
void combineVerticesSpline(long dwVertexCount, float fPercentage,
                    vector3 *pFirstFrameVertices,
                    vector3 *pSecondFrameVertices,
                    vector3 *pThirdFrameVertices,
                    vector3 *pFourthFrameVertices,
                    vector3 *pCombinedVertices)
{
  float t = fPercentage;
  float t2 = t * t;
  float t3 = t2 * t;

  vector3 m0, m1;

  const float alpha = 0.0f;

  for (long i = 0;
      i < dwVertexCount;
```

```
        i++, pFirstFrameVertices++, pSecondFrameVertices++,
        pThirdFrameVertices++, pFourthFrameVertices++)
    {

      m0 = ((1 - alpha) / 2.0f) *
           ((*pSecondFrameVertices - *pFirstFrameVertices) +
            *pThirdFrameVertices - *pSecondFrameVertices);

      m1 = ((1 - alpha) / 2.0f) *
           ((*pThirdFrameVertices - *pSecondFrameVertices) +
            *pFourthFrameVertices - *pThirdFrameVertices);

      *pCombinedVertices = (((2 * t3) - (3 * t2) + 1) *
         *pSecondFrameVertices) +
         ((t3 - (2 * t2) + t) * m0) +
         ((t3 - t2) * m1) +
         (((-2 * t3) + (3 * t2)) *
         *pThirdFrameVertices);
    }
}
```

这个计算新增加了一个变量 alpha。alpha 控制正被计算样条的切线的张量。虽然可以改变 alpha 使张量更高（正值），或更低（负值），但 alpha 为零对大多数的动画来说已经足够了。

如果已经确定一个固定的 alpha 值对你的动画就足够了，那么可以预先计算切线方程 m_i 的第一部分((1－alpha)/2)，并且用一个常数来置换它，在目前的情形下可以取 0.5。

4.13.5 为什么用 Hermite 样条

乍一看，跳过一个著名的样条函数（比如 B 样条）而选择 Hermite 样条似乎是一个奇怪的选择。但是 B 样条提供了额外的连续性，这会缺少对一个插值曲线的趋势的控制。

4.13.6 总结

关键帧动画插值是提高动画质量的一个容易且代价低廉的方法。线性插值能以每顶点极小的代价来实现。Hermite 样条插值较之线性插值改善了插值关键帧的质量，但是每顶点付出了更大的代价。

4.13.7 参考文献

[Foley96] Foley, J., van Dam, A., Feiner, S., and Hughes, J. *Computer Graphics: Principles and Practice* 2nd Edition. New York: Addison-Wesley Publishing Company, Inc., 1996.

4.14 一种快速而简单的皮肤构造技术

Torgeir Hagland

本文描述了一种对较低多边形角色（少于500个多边形）最为适用的皮肤构造技术，这里美工和动画师需要对顶点的作用进行100%的控制。该技术可以简要地描述为对一种一个对象的顶点序列进行修改和排序，并相应地预映射其表面列表（face list）的巧妙方法。

4.14.1 为什么对低多边形有价值

当处理有较低数目的多边形模型时，每个顶点对于模型的轮廓都有着很大的视觉影响。看一下你的肘。影响它的骨头只有上臂和下臂。当你弯曲二头肌时，下臂将影响你的肘内侧的顶点的运动。它将那些顶点从下臂方向推起并把它与上臂的朝向平均。最后的结果就像你有了一个非常粗的肘部。这种技术仅考虑了每个顶点一个骨头的情况。

4.14.2 方法

本文建立了一种单一的皮肤模型，例如考虑一个空间勇士。他复制这种皮肤，接着按极小的比例缩小，并且继续将这些小的皮肤块切削到更小的部分（身体部分），它们被用作骨头。当每一块骨头创建完毕之后，就被给出与皮肤一样的有一个附加数字的名字，于是它能够作为骨头被程序识别出来。

一旦我们识别出了一块皮肤和它的骨头，我们就取骨头的几何体并将其顶点存储在一个大的列表中。该表中的每条记录包含顶点的位置和这个顶点属于的骨头。

现在对于皮肤上的每个顶点，我们寻找在骨头列表中与它相距最近的顶点。我们将皮肤顶点通过它最接近的骨头的逆矩阵进行变换。这将使皮肤顶点变换到骨头的局部坐标系（在绘制循环过程中，顶点又会被变换回来，于是即使骨头皮肤较小，它对于最终的结果也没什么影响，因为位置是相对的）。变换过的顶点存储在一个累积的临时表中，在那儿也可以存储原始顶点列表的索引以及一个指向影响它的骨头的指针。影响它的骨头有一个计数器用以记录它变换的顶点的数目。

当皮肤的所有顶点都有了一个影响它们的骨头时，我们来处理创建的临时表。然后这个表按骨头的顺序进行排序。对每一块骨头而言，它所影

响的顶点数目是存储在原始的皮肤顶点表中的，而且表面必须重新映射那些它们影响的顶点，因为我们刚才改变了所有的顶点索引。

程序清单 4.14.1 包含的示例代码解出了骨头影响并对表面相应地进行重映射。尽管这些代码使用了 3D Studio Max file 工具包，但该技术能很容易地同任何 3D 建模程序包一起使用。我只是用它来保持较小的源程序长度，而且确保主要关心的是影响的解决和绘制循环，而不是模型转换等。

执行完程序清单 4.14.1 之后，我们有：

- 一块皮肤，且每个顶点变换到了影响它的骨头的局部坐标系。顶点表以骨头的次序进行排序。
- 一个骨头的列表，以及一个用于记录每一块骨头应该变换多少个顶点的计数器。

皮肤的绘制循环就可以像程序清单 4.14.2 那样简单了。

4.14.3 总结

本方法快速而简单，并且尤其适用于低多边形角色。对于高多边形角色，其边缘更平滑，你可能会需要几块骨头来影响每一个顶点，那么也很可能为每个顶点存储 2 或 3 个指向影响它的骨头的指针。这意味你将不再能够预先存储逆变换的顶点了，而且每帧你都需要应用逆变换以及为每个影响它的骨头应用一个基于百分比的旋转。这引起的更是一个创建影响数据的工具的问题。输出骨头信息的商业工具包已经有了，而且你再也不用担心影响是如何施加的，只要关心如何来创建你的绘制循环就可以了。如果你决定自己创造一个影响工具，我强烈地建议你开发这样的一个工具：允许美工直接在几何图形上面“描绘”影响。这样他就不用再去揣度数学算法了。

程序清单 4.14.1

```
void SolveBoneInfluences(database3ds *db, Skin *skinptr)
{
    /* Allocate a big workbuffer */
    BonePoint *bonepointptr=
    (BonePoint*)malloc(30000*sizeof(BonePoint));
    BonePoint *curbonepoint=bonepointptr;

    long NrBoneVerts=0;

    /* Make all the bones' vertices into one big vertex
    list with information on what bone each point came from */

    MATRIX tmpmat;

    Bone *boneptr=skinptr->BonePtr;
    while(boneptr)
    {
        mesh3ds *bonemesh=NULL;
        GetMeshByName3ds(db,boneptr->Name,&bonemesh);
```

```
        assert(bonemesh);

        Copy3dsMatrix(tmpmat,bonemesh->locmatrix);
        InverseMatrix(tmpmat,boneptr->Matrix);

        point3ds *bonemeshpoints=bonemesh->vertexarray;

        NrBoneVerts+=bonemesh->nvertices;
        assert(NrBoneVerts<30000);

        for(int i=0;i<bonemesh->nvertices;i++)
        {
            curbonepoint->Point.x=bonemeshpoints->x;
            curbonepoint->Point.y=bonemeshpoints->y;
            curbonepoint->Point.z=bonemeshpoints->z;
            curbonepoint->BonePtr=boneptr;
            bonemeshpoints++;
            curbonepoint++;
        }

        RelMeshObj3ds(&bonemesh);
        boneptr=boneptr->NextPtr;
    }

    mesh3ds      *skinmesh        = skinptr->MeshPtr;
    point3ds     *skinmeshpoints  = skinmesh->vertexarray;
    BonePoint    *skinpointptr    = (BonePoint*)malloc(
skinmesh->nvertices*sizeof(BonePoint));
    BonePoint    *curskinpoint    = skinpointptr;

    /* Find the closest bone vertex to each skin vertex */
    for (int i=0;i<skinmesh->nvertices;i++)
    {
        curskinpoint->Point.x     = skinmeshpoints->x;
        curskinpoint->Point.y     = skinmeshpoints->y;
        curskinpoint->Point.z     = skinmeshpoints->z;
        /* need to store original vertex index, for
        face remapping */
        curskinpoint->Index       = i;
        /* no bone is influencing this bone yet */
        curskinpoint->BonePtr = NULL;

        curbonepoint=bonepointptr;
        float mindist=1e6;

        for(int j=0;j<NrBoneVerts;j++)
        {
            float dist=
            CalcDistNotSquared(skinmeshpoints,
            &curbonepoint->Point);
```

```
            if(dist<mindist)
            {
                mindist=dist;
                curskinpoint->BonePtr=
                curbonepoint->BonePtr;
            }
            curbonepoint++;
        }
        curskinpoint++;
        skinmeshpoints++;
    }

    /* Sort all the vertices of the skin by bone,
    and remap the faces accordingly */
    skinmeshpoints        = skinmesh->vertexarray;
    face3ds *skinfaces    = skinmesh->facearray;
    long CurIndex=0;
    boneptr=skinptr->BonePtr;
    while(boneptr)
    {
        curskinpoint=skinpointptr;
        for (i=0;i<skinmesh->nvertices;i++)
        {
            if(curskinpoint->BonePtr==boneptr)
            {
                Transform(boneptr->Matrix,
                (float*)&curskinpoint->Point,
                (float*)skinmeshpoints);
                RemapFaceList(skinmesh,
                curskinpoint->Index, CurIndex++);
                boneptr->NrVerts++;
                skinmeshpoints++;
            }
            curskinpoint++;
        }
        boneptr=boneptr->NextPtr;
    }

    /* Clean up after the remapping */
    CleanUpFaceList(skinmesh);

    free(skinpointptr);
    free(bonepointptr);
}
```

程序清单 4.14.2

```
void glDrawChar()
{
    mesh3ds       *meshptr      = SkinPtr->MeshPtr;
    Bone          *boneptr      = SkinPtr->BonePtr;
```

```
    point3ds      *vertptr =    meshptr->vertexarray;
    face3ds       *faceptr =    meshptr->facearray;

    /* For Each bone in the skin, transform X amount
  of vertices with the bone's current animation matrix*/
    point3ds *skinptr=SkinPtr->PointPtr;
    while(boneptr)
    {
        MATRIX mat;
        memcpy(&mat,&boneptr->AnimPtr[CurFrame],
            sizeof(MATRIX));
        for (int i=0;i<boneptr->NrVerts;i++)
    Transform(mat,(float*)vertptr++,
          (float*)skinptr++);

        boneptr=boneptr->NextPtr;
    }

    /* Then Simply draw the object using the facelist*/
    skinptr=SkinPtr->PointPtr;
    glBegin(GL_TRIANGLES);
        glColor3f(1,1,1);
        for(int i=0;i<meshptr->nfaces;i++)
        {
            point3ds *v1=&skinptr[faceptr->v1];
            point3ds *v2=&skinptr[faceptr->v2];
            point3ds *v3=&skinptr[faceptr->v3];

            glVertex3f(v1->x,v1->y,v1->z);
            glVertex3f(v2->x,v2->y,v2->z);
            glVertex3f(v3->x,v3->y,v3->z);

            faceptr++;
        }
    glEnd();
}
```

4.14.4 参考文献

[Lander98] Lander, Jeff, Game Developer Magazine, May 1998: *Real-time Skeletal Deformation.*

4.15 填充间隙——使用缝合和皮肤构造的高级动画

Ryan Woodland

随着硬件变得越来越快、功能越来越多，游戏开发者也在寻找使角色看起来更引人注目的方法。角色动画也许是众多改进中最重要的一种。

目前，大多数的三维游戏正在开始为他们的角色使用某种骨架表示法作为动画的拓扑结构。这些系统把几何体附在一个人物的“骨头”上。骨头是要被动画的，因此，所附的几何体就要承受骨头的动作来创造适当的动画。然而，通常用于表现角色的几何体在本质上是刚性的，但对于为自然界中绝非刚性的有机生物建立模型来说，它并不是最有效的表示法。

因为几何体完全是刚性的，任何两个假定互相连接在一起的部分（例如上臂和前臂）在它们连接的关节处就会显示出明显的不连续性。这显然会成为一个问题，因为我们试图表现的角色常常远非一具不显示任何裂纹和分隔的连续外壳。

在本文中，我将讨论的主题是把缝合和皮肤构造（stitching and skinning）技术用于产生更逼真的生物体动画。实际上缝合正是一种计算代价较小的皮肤构造的子集，因此首先讨论它。这两个技术都假定一个连续网格被附着在一个人物骨架结构上，而不像传统刚体动画中的许多网格附着在单个骨头上。这个连续的网格相对于角色骨架结构变形，产生的人物关节在运动时不再出现裂缝（这种裂缝常常很讨厌）。

在以下各节中，我将用一只手臂做例子说明缝合和皮肤构造技术的各种不同特征。所用的基本网格是图 4.15.1 中的图片。

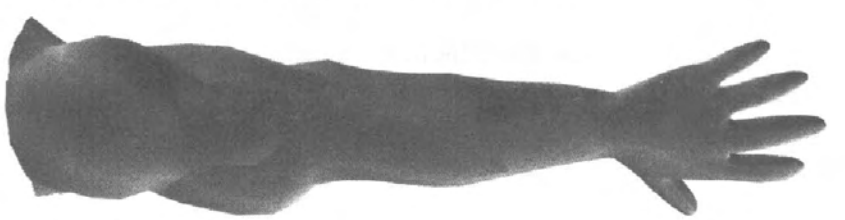


图 4.15.1 我们的基本手臂网格

4.15.1 缝合

正如前文所述，缝合是对附着在骨架结构上的一个连续网格进行操作。在刚体动画中，多边形用表现该多边形所附着的骨头的矩阵进行变换。使用缝合技术，多边形中每个顶点可以被其所附着的骨头的矩阵进行变换。这意味着通过在多边形中将不同的顶点附着到不同骨头上，我们可以创建一个把多个骨头简单地缝合在了一起的多边形。当骨头被操纵的时候，这个多边形将填充你在刚体动画中看见的间隙。

缝合和刚体动画之间主要不同之一是用于表现角色的数据拓扑结构。对刚体动画而言，骨头必定有一个指向某个将要进行动画的几何体的指针。然后，相应的骨头产生的矩阵对该几何体进行变换。对于缝合，角色皮肤中的每个顶点都必须追踪它所附着的骨头。

```
struct Vertex
{
    float s, t;
    float x, y, z;
    unsigned long color;

    unsigned long boneIndex;
};
```

在制作一个与这个数据拓扑结构恰当结合的动画角色之前，我们需要处理顶点不在要变换的正确空间中的问题。问题是这样的：一个用于为动画变换骨头的矩阵，假定骨头是以它的枢轴位于角色的空间坐标原点上开始的。如果我们考虑常人的一只手骨，这是有道理的。骨头应该从位于角色空间坐标的原点的枢轴点开始，这样我们易于使骨头围绕该点进行旋转。骨头动画后（被旋转），进而变换到前臂骨头的尽头。对前臂骨重复这个过程——手和前臂被动画，然后移到上臂骨头的尽头。这样继续向下经过各个数据层次直到整个骨架全部被适当变换。

给定角色的皮肤顶点和骨头之间的空间关系后，就要将矩阵变换之前就附在骨头上的皮肤顶点变换到骨头局部坐标空间。为了做到这一点，我们需要在每个骨头中保存一个矩阵，它告诉我们怎样把几何体变换回骨头的局部空间中。这个矩阵应该是将骨头从局部空间变换到角色网格的矩阵的逆，假设网格的朝向没有用到任何动画。请看图 4.15.2 中对手臂网格中的每个骨头的本地空间的一个描述。

因此，我们的骨头数据结构应该如下：

```
struct Bone
{
    Mtx orientation;
    Mtx animation;
    Mtx inverseOrientation;

    Mtx final;

    Bone *child;
    Bone *sibling;
};
```

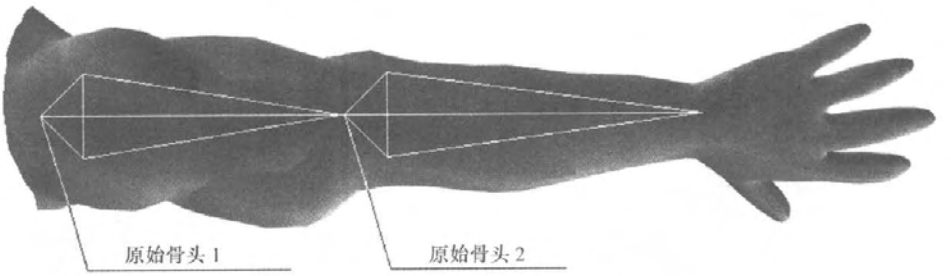


图 4.15.2 我们的手臂中骨头的一个描述

一旦有了这个数据结构，就将制作我们的角色了。为了做到这一点，我们必须简单地逐步遍历顶点数据，并且用朝向矩阵来变换每个顶点，然后计算对应骨头的动画矩阵。

所有这些变换都可以加快进行，通过处理骨头层次并将一个骨头的逆朝向矩阵、连接后的朝向矩阵，以及连接后的动画矩阵连接在一起，为每个骨头生成一个最终的变换矩阵，然后用结果矩阵转换几何体。

```
void BuildMatrices ( Bone *bone, Mtx forward, Mtx orientation )
{
     Mtx localForward;
     Mtx localOrientation;

     // concatenate the hierarchy's orientation matrices so
//   that we can generate the inverse
     concatenate(bone->orientation, orientation ,
     localOrientation);

     // take the inverse of the orientation matrix for this bone
     inverse(localOrientation, bone->inverseOrientation);

     // concatenate this bone's orientation onto the forward
     //  matrix
     concatenate(bone->orientation, forward, localForward);

     // concatenate this bone's animation onto the forward matrix
     concatenate(bone->animation, localForward, localForward);

     // build the bone's final matrix
     concatenate(bone->inverseOrientation, localForward,
     bone->final);

     if(bone->child)
          BuildMatrices(bone->child, localForward,
          localOrientation);

     if(bone->sibling)
          BuildMatrices(bone->sibling, forward, orientation);
}
```

在手臂网格上应用前述技术，前臂骨弯曲 45° 和 90° 产生了图 4.15.3 中所示的图像。

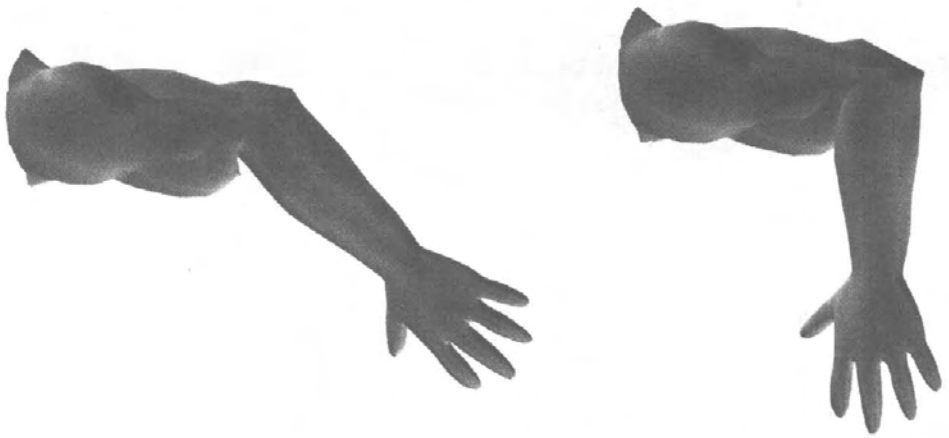


图 4.15.3 a：缝合的手臂网格弯曲到 45° 。b：弯曲到 90°

缝合是一种非常有效的技术，因为它容易利用任何提供了变换引擎的硬件的优点。在 CPU 上生成最终的缝合矩阵是必需的，但是硬件可以容易地用这些矩阵来变换我们传递给它的任何数目的顶点。

作为对这种技术的一种优化，我建议把连续的皮肤分解开，以使顶点存在于它们所附着的骨头的本地空间中。这使我们避免了必须对动画的每帧的每个骨头做一个额外的矩阵连接。

4.15.2 皮肤构造（Skinning）

虽然缝合是一种有效的技术，但它也有一些问题。在关节旋转的极端情况下，几何体倾向于进行大块剪切而显得不太自然。使用较早讨论过的技术，一个 120° 的前臂旋转在肘部显示出很糟糕的剪切效果。之所以会产生这样的结果，是因为我们只有一个多边形跨越了在上臂和前臂之间的整个间隙。间隙越大，结果就越糟糕，如图 4.15.4 所示。

图 4.15.4 难看的手臂网格弯曲到 120°

为了避免这种情况发生，我们可以实现一个完整的皮肤构造系统，其中一个顶点不限于被单一骨头所影响；而是可以替换成被多个骨头影响。如果我们观察一下人类身体的行为会发现这是合理的。一个人肘部的皮肤不是仅受一块骨头取向影响的，上臂和下臂的骨头的运动都会影响它。类似地，颈部和肩部的皮肤受手臂、颈部和胸部骨头取向的影响。

为使这点能够实现，在一个皮肤的网格中的每个顶点一定包含一个影响它的骨头的列表。每个顶点也一定带有每个骨头的一个权重信息，该权重告诉我们骨头对顶点的影响有多大。举个例子，我们将假定线性皮肤构造，这意味着一个顶点的所有权重加起来必须为 1.0。因此，给定影响一个顶点的 n 个骨头，我们需要储存 $n-1$ 个权重，因为剩余的权重应该是 $1.0-(weight_1 + weight_2 + ... + weight_{n-1})$。

```
struct Vertex
{
    float s, t;
    float x, y, z;
    unsigned long color;

    unsigned long boneIndex1;
    unsigned long boneIndex2;

    float weight;
};
```

如上所述，缝合是皮肤构造的一个子集，所以皮肤构造要遇到与缝合相同的本地空间变换的问题。因此，我们应该采用和前述相同的骨头表示法。

为了进行完整的皮肤构造，我们需要用影响每个骨头的每个矩阵来对它实施变换，然后用相应权重乘以结果，最后再累积结果。皮肤构造方程类似于：

*(vertex * matrix0 * weight0) + (vertex * matrix1 * weight1) + ... +*
*(vertex * matrixN * weightN)*

这里，所有权重的和 0..N = 1.0。

我可以在变换的顶点之间做一个有效的线性插值。下面是在给定的网格上执行这个操作的代码。

```
Vector3D TransformVertex ( Vertex *vert, Bone *boneArray )
{
    Vector3D  temp;
    Vector3D  final;

    temp = XFormVec(vert->position,
    bone[vert->boneIndex1]->final)
    final.x = temp.x * vert->weight;
    final.y = temp.y * vert->weight;
    final.z = temp.z * vert->weight;

    temp = XFormVec(vert->position,
    bone[vert->boneIndex2]->final)
```

```
        final.x += temp.x * (1.0F - vert->weight);
        final.y += temp.y * (1.0F - vert->weight);
        final.z += temp.z * (1.0F - vert->weight);

        return final;
    }
```

使用上述技术，我们可以对前臂转 45°、90° 和 120° 产生如下结果。甚至在转 120° 的极端情况下，肘部多边形仍保持连续性（见图 4.15.5）。

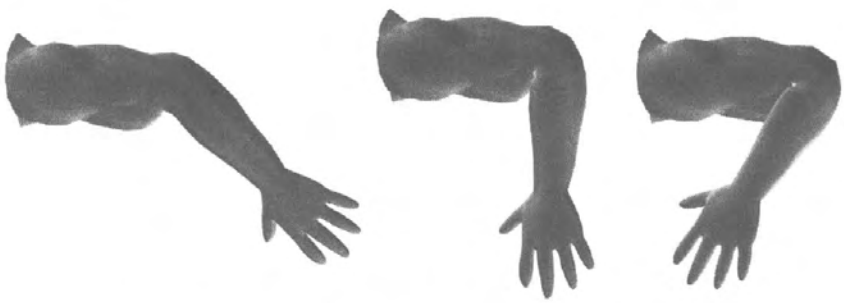


图 4.15.5 a：皮肤构造的手臂网格弯曲到 45°。b：弯曲到 90°。c：弯曲到 120°

正如你所看见的那样，与皮肤构造有关的一个主要问题是它计算代价昂贵。不幸的是，这些计算没有得到目前的硬件变换引擎很好的支持。实施插值计算的一个替代方法是利用现有硬件设备的一些潜能，生成一个传递给硬件的皮肤构造矩阵并完成最终的变换。为计算皮肤构造矩阵，可以仅仅基于权重对矩阵进行线性插值：

$$(matrix0 * weight0) + (matrix1 * weight1) + \ldots + (matrixN * weightN)$$

这里，所有权重的和 $0..N = 1.0$。

仅当相同的皮肤构造矩阵可以用于多个顶点时，这个方法才是有效的；换句话说，不同顶点在相同的骨头中权重是相同的。这种情况的真实性越小，该方法的效果就越差。

注意上面概述的皮肤构造技术不是一个在数学上完全正确的技术，这一点很重要。如果用该技术对法线进行变换，结果并不一定是规一化的。如果需要用这个技术对一个角色进行逐顶点的光照，那么，变换后的法线在光照运算之前必须被规一化。

4.15.3 进一步的问题

这个皮肤构造的例子假定所有影响一个顶点的权重加起来必须为 1.0。然而，用总和不为 1.0 的权重产生一些吸引人的特效也是可能的。举例来说，可以在一只模拟肱二头肌的手臂中放置一块额外的骨头。手臂皮肤中的所有顶点都应该在上臂、前臂和肩膀之间进行正常加权。然而，靠近肱二头肌的顶点也应该基于到肱二头肌的骨头的距离进行加权——较近的顶点应该有更高权重值。当手臂弯曲的时候，在肱二头肌的骨头上应用一个比例变换可以产生一个肌肉挠曲的外观。

本文所概述的皮肤构造技术在数学上并不正确，因为本质上我们是在对矩阵进行线性插值。代替用矩阵来表示骨头，也可以用四元数来表示它们。四元数之间的 SLERP 以每个顶点的权重为基础，然后从结果中产生一个矩阵。这会产生视觉效果更好些的皮肤构造网格。

4.15.4 参考文献

[Lander98] Lander, Jeff, "Skin Them Bones: Game Programming for the Web Generation," Game Developer Magazine (May 1998): pp. 11–16.

[Terzopoulos87] Terzopoulos, Demetri, et al, "Elastically Deformable Models," Computer Graphics, Vol 21, no.4 (SIGGRAPH 1987): pp. 205–214.

4.16 实时真实地形生成

Guy W. Lecky-Thompson

地形是许多游戏的中心装饰品，在一些游戏中它可能是重要的背景，而在其他游戏中可能仅仅用来填充空间。不论如何使用它，如果表现得不好，它会吸引不需要的注意力；同理，如果表现得好，它将会增加游戏的气氛、可玩性和生命力。

术语“地形（terrain）”会让大多数人在脑海中浮现出地貌、湖泊、山脉乃至不通风的荒凉的火山口。当作为创建游戏的一个重要方面的时候，“地形”这个词可以用于更广义的范围中。它可能包括物体、名称和建筑物、玩家将与之交互作用的游戏世界部分，以及仅对那些部分给予支持的片段。

本文的目的是为读者提供一些可生成逼真地形的算法，游戏可在其内运行。

这里强调的是生成，而不是存储。也就是说，算法是以一种倾向于用它们实时生成地形的方式提出的，而是不以日后重放这些内容为目的对生成的地形进行存储。使用本书中“4.0 可预测随机数”中定义的技术，可以生成一个强大的接近无限的世界。

4.16.1 风景设计

第一个能用来生成基本地形的技术是模糊风景设计（fuzzy landscaping）。从本质上讲，它以一种完全随机的方式简单地生成地形，不太顾及真实的世界。现在作为一个出发点提出这种技术，以此为基础我们可以构造更有用的新算法。

下面是生成一个有限栅格的伪代码：

```
y = -1;
while y < 100 {
    x = 0;
    y = y + 1;
    srand(y);
}
while x < 100 {
map(x, y) = rand (3);
    x = x + 1;
}
```

正如可以看到的那样，程序将一系列 0~3 的随机数分布到 100×100 栅

格中。接着我们可以把颜色分配到数字中去：比如 0 是黑色（水），1 是暗灰色（平原），2 是浅灰色（陆地），3 是白色的（山脉）。这个效果可以从图 4.16.1 中看到。请注意，随机数生成器是在和正方形有关的栅格部分播种的。这确保我们不必遍历整个栅格就能恢复该值[Lecky99]，而只需要穿越那条线即可。

这有点不尽如人意，因为我们更喜欢基于离散的正方形进行播种。为了做这一点，我们将需要创建自己的随机数生成器来去掉在图 4.16.1 中看到的那种令人讨厌的结果，这种结果起因于对每个栅格方块使用 ANSI srand 函数：srand (x + (x * y))。

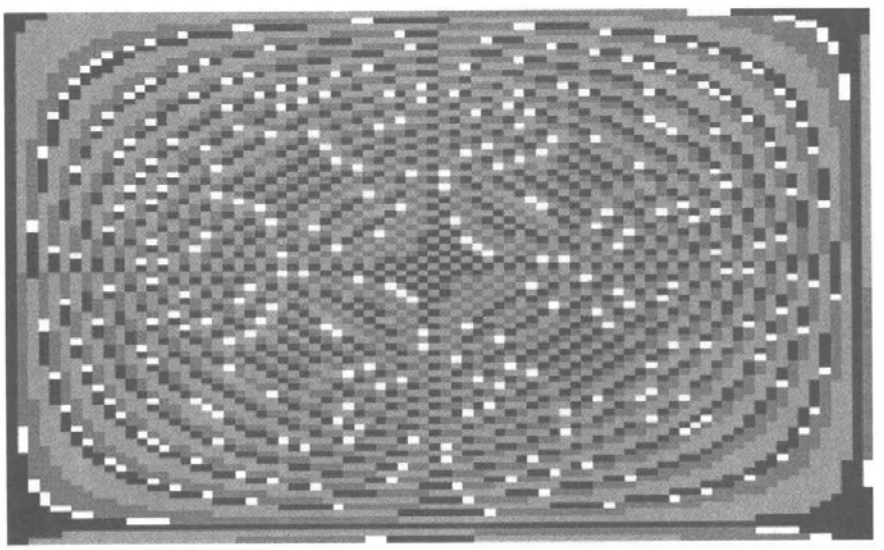


图 4.16.1 地形值的一个 100×100 随机栅格

漂亮的模糊地形比在图 4.16.1 中所示的更真实，因此我们需要在生成的“地图”上执行一些附加处理。我们将要使用的技术能用于这里的任何一种地形生成算法，当然能用于任何需要分组处理的随机数字的抽象集合。

驱动原理是要通过邻近的正方形保持相似值来减少地图的随机性，但是同时允许在正方形的特殊集合或地区之间存在差异。看看程序如何工作比解释更容易。其伪代码如下：

```
step = 4;
for y = 0; y < 100; y = y + step {
    for x = 0; x < 100; x = x + step {
        total = 0;
        for y_local = y; y_local <= y + step;
                y_local = y_local + 1 {
                for x_local = x; x_local <= x + step;
                     x_local = x_local + 1 {
                     total = total + map (x_local, y_local);
                }
        }
        average = total / (step x step);
        for y_local = y; y_local <= y + step;
             y_local = y_local + 1 {
```

```
                for x_local = x; x_local <= x + step;
                    x_local = x_local + 1 {
                    map (x_local, y_local) = average;
                }
            }
        }
    }
```

应用这个平滑算法的效果可以在图 4.16.2 中看到。

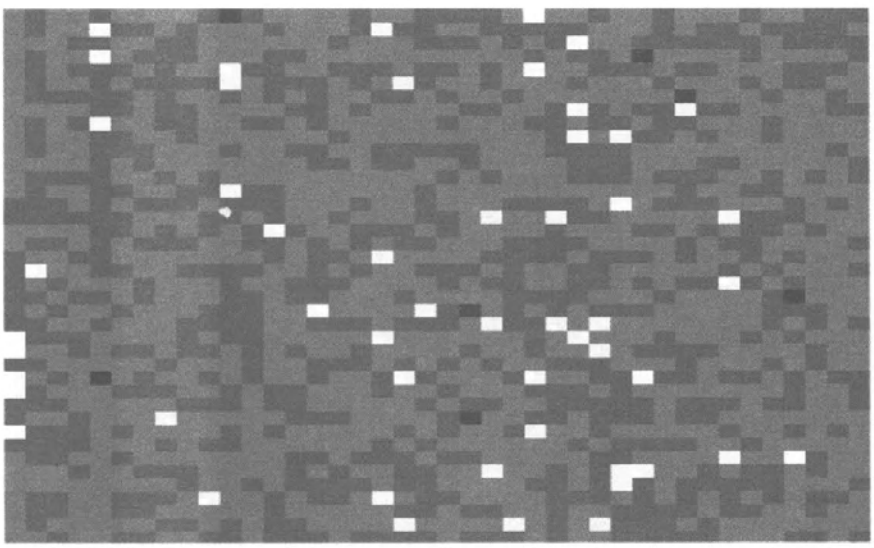


图 4.16.2 随机地形栅格的平滑版本

虽然最后的结果还远远谈不上完美，但最重要的感觉是地图的随机性已经变得比以前少得多了。

它的工作方式是：通过将栅格分成一系列较大的正方形，然后再细分它们；接着计算出子块的平均值并且推广到所有子块中去。最后达到一种平滑效果。

这里子块大小的选择也相当重要——太大的一个子块将会生成相似值的宽带区域，而太小的一个子块将生产不了期望的效果。

对这个算法的一个改进是从一个稍微不同的方向接近它。最终结果是一样的，基于周围点修正地形的离散点。然而，这次我们将为平均过程选择正方形的 4 角的点，而不是使用所有的点。我们也将只改变被选择的正方形的每个四分之一块的中心点，而不是所有的点。

下面的代码节选自 CD 中的地形生成软件。

```
for ( int square_size = width; square_size > 1; square_size /= 2 )
{
  int random_range = square_size;

  for ( int x1 = row_offset; x1 < width; x1 += square_size )
  {
    for ( int y1 = row_offset; y1 < width; y1 += square_size )
    {
```

```
        // Calculate the four corner offsets
        int x2 = (x1 + square_size) % width;
        int y2 = (y1 + square_size) % width;

        // Get the values
        int i1 = this->terrain[x1][y1];
        int i2 = this->terrain[x2][y1];
        int i3 = this->terrain[x1][y2];
        int i4 = this->terrain[x2][y2];

        // Create weighted averages, based on
        int p1 = ((i1 * 9) + (i2 * 3) + (i3 * 3) + (i4)) / 16;
        int p2 = ((i1 * 3) + (i2 * 9) + (i3) + (i4 * 3)) / 16;
        int p3 = ((i1 * 3) + (i2) + (i3 * 9) + (i4 * 3)) / 16;
        int p4 = ((i1) + (i2 * 3) + (i3 * 3) + (i4 * 9)) / 16;

        // Calculate the center points of each quadrant
        int x3 = (x1 + square_size/4) % width;
        int y3 = (y1 + square_size/4) % width;
        x2 = (x3 + square_size/2) % width;
        y2 = (y3 + square_size/2) % width;

        // Set the points to the averages calculated above
        this->terrain [x3][y3]   = p1;
        this->terrain [x2][y3]   = p2;
        this->terrain [x3][y2]   = p3;
        this->terrain [x2][y2]   = p4;
      }
    }

    // For the next row, move in slightly
    row_offset = square_size/4;
  }
```

图 4.16.3 显示了 4 角的点和用以上算法计算的 4 个中心点的包围矩形。

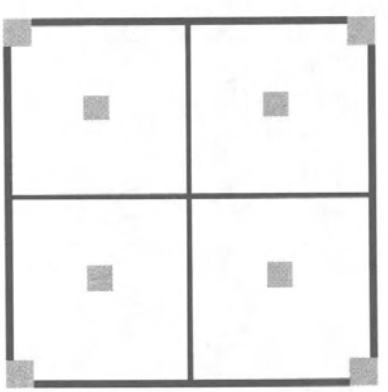


图 4.16.3 子块坐标

这个技术最初是由 James McNeill[McNeill95]介绍给我的，而且是我到目前为止看到过的最可信赖的“平滑”的例子之一。在其他技术中有着无穷的变化，包括增加随机的偏移量到计算出的点，这导致了更多样化的景观。

作为对这里讨论的两种技术的增强，我们可以介绍第三种机理，称为断层线风景生成。断层线风景生成的工作方式是随机选取两个点，并在它们之间的给定高度上画一条线。然后，再选取两个点，在它们之间画一条线，如此反复直到在屏幕上有了一定数量的线段，如图 4.16.4。

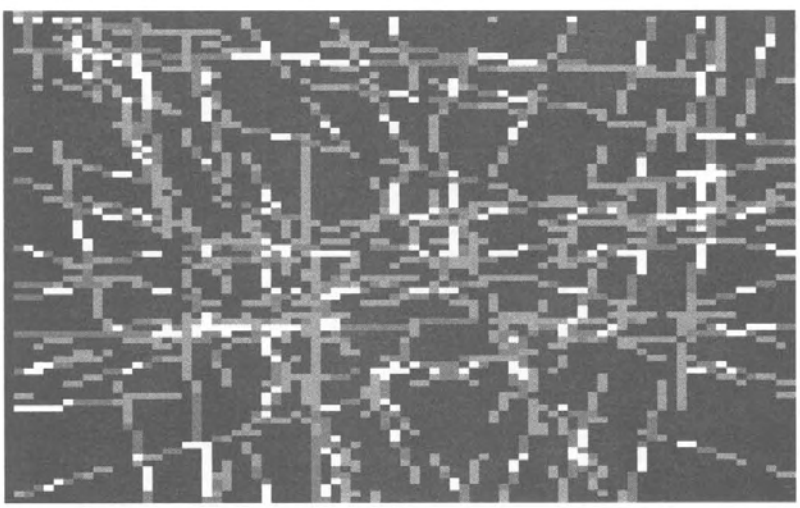


图 4.16.4 随机断层线

下一步只不过是用前面解释过的子块技术来对点之间的差异进行平滑了。这将生成一系列的“岛屿”，如图 4.16.5 中所示。关于这项技术的进一步的信息请参阅 Jason Shankel 的文章“4.17 分形地形生成——断层构造”。

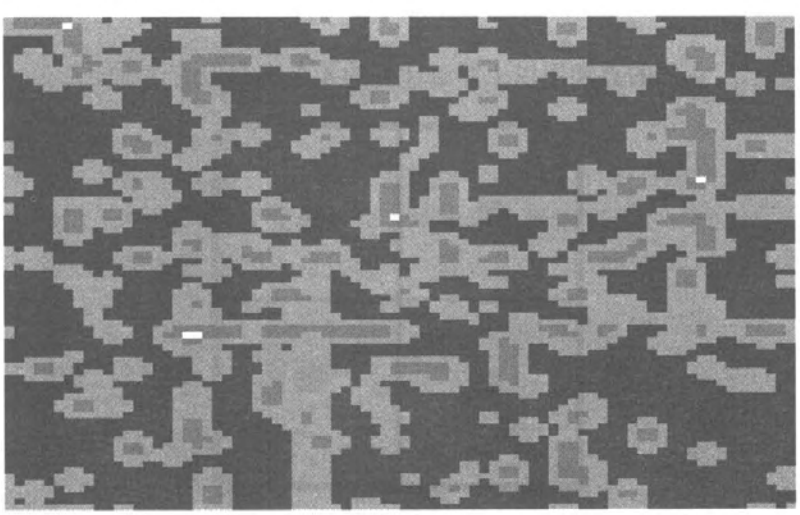


图 4.16.5 被平滑过的断层线形成了岛屿

虽然这可能在开始看起来很简单，但其实画线本身更复杂。正如细心的读者已经从图4.16.5中注意到的，那些线不是在一个常数“高度”上画的。即，沿着这条线上的每个点的值随着起点到终点的距离而变化。

用于决定每点“高度”的算法是一条正弦曲线，曲线的振幅以两点之间的距离为基础。下面的代码片段是核心的直线绘制的算法，节选自CD中的地形生成器软件。

```
do
{
  this->terrain[(int)x_start][(int)y_start] =
    nCurrentRandomValue;
  x_start = x_start + x_diff;
  y_start = y_start + y_diff;

  // Apply a sine function oscillating between 0 and 255
  // The sin function should be called with values from
  //   -pi/2 to pi/2

  if (x_diff < y_diff)
  {
    nCurrentRandomValue = (sin(x_start) * 128) + 128;
  }
  else
  {
    nCurrentRandomValue = (sin(y_start) * 128) + 128;
  }
} while (((y_start < (float)this->terrain_width) &&
    (y_start > 0.0)) &&
    ((x_start < (float)this->terrain_height) &&
    (x_start > 0.0)));
```

对每条线段执行它得到了类似山脉的效果，虽然有一个非常平滑的摆动。

这里讨论的有关技术需要注意的重点是生成的风景是可重复的。也就是说，使用同样的基本输入值，可以随意生成同样的风景。它们不需要在任何地方储存。一般而言，这是创建地形的根本原则，而且也是本文的剩余部分的核心主题。

和生成原理一起使用这些技术，可以说，既然理论上我们每一次播种随机数生成器将得到一个不同的随机数集合，那么生成地形的概率是无穷的。此外，既然可以随意地重复生成或重复计算任何地形的任何点，我们就可以只需要实时的存储量，剩下的输出空间就更大，不仅仅用于关卡文件了。

4.16.2 建筑物

在游戏播放的舞台（假设你允许双关语）中最常见的结构之一是迷宫。例如，游戏*Doom*就使用了好几个这样的结构，达到了较好的效果。另外，进阶和关卡类型的游戏中也使用迷宫的一个二维变种。这些迷宫时常会按照玩家的技巧增加复杂性，也可以被形形色色的财宝弄得乱七八糟。

可是这些结构常常会用如此多的存储空间（例如 *Doom* WAD 文件），使你不能在一张 CD 上装下它们以满足玩家的需要。一个真正的真实地形（从最广义上来说）需要无限的幻想，而且如果我们可以以某种方式实时生成这些容器，将会大有裨益。

如果不考虑生成物的“形状”，生成包含通道和路径但是没有房间的事物是极为简单的（作者还深情地记得几个早期的有拱廊的游戏，像……）。基本的目标是设想游戏区域被包含在一个有限的空间里，我们将称之为盒子。

然后以一个随机间隔从左至右地画一些水平直线对这个盒子进行细分。紧接着，再以一个随机间隔从上至下地画一些垂直直线对这些子盒子进行划分。图 4.16.6 显示了一种可能的结果。

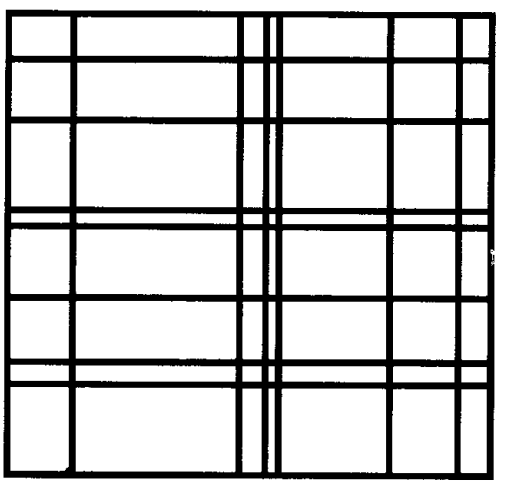


图 4.16.6 随机直线划分我们的盒子

从这个角度来看，这一点也不让人感到有趣，但是它完全是一个表现问题。可以把黑色的线想像成通道。现在想像玩家从第一人称的视点只看到了这个通道；他们将知道的一切就是联接和通道。

事实上，这仍然不能令人信服——它仍然缺少房间和尽头。这两者将增加体验的真实性。房间是相当容易生成的，因为所需的只是尺寸超过可以看作一个房间的最小的盒子的任何空间。计算出最小盒子的尺寸简直太容易了。

因为我们知道所有的水平线和垂直线一定在一些结合点上相遇，所以最小的盒子是水平间隔最小的两条垂直线和垂直间隔最小的两条水平线组成的区域。任何在两个方向上大于最小盒子的区域都可以看作一个房间；所有剩下的区域正好就是通道。基于此，显现出来的结果如图 4.16.7 所示（这里我们稍微加大了一点尺寸用于指出通道和房间）。

通过在墙壁上的不同地方设置入口，我们已经创建了一个可以被极快地完全生成的游戏区域。增加直线的数目，或者减小用于确定什么是通道和什么是房间的尺寸，都将影响游戏的复杂程度因此影响其玩的容易程度。

可这仍然不太真实。事实上，它看起来更像一些街道的十字路口，而不是一栋建筑物。那么让我们把它看成是建筑物，并确定如何将“房间”变成“建筑物”。通道仍然保持是街道。

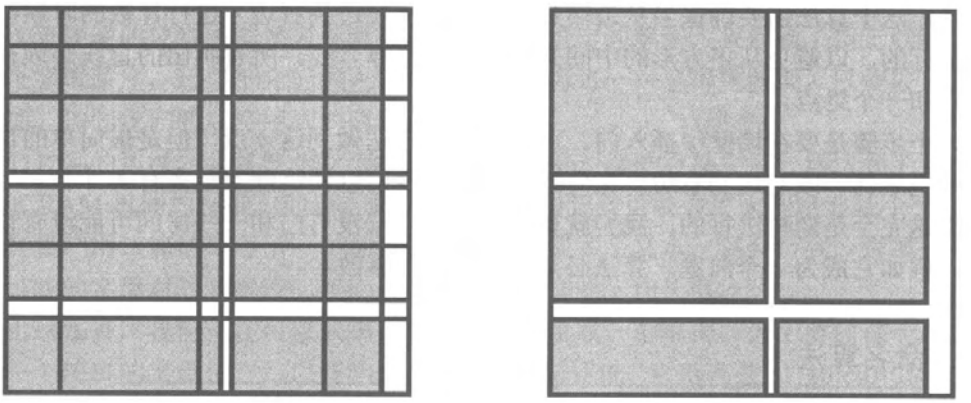


图 4.16.7 a:删除直线前 b:删除直线后

不是像我们先前所做的那样盲目地把“房间”切成碎块，而是通过细分把建筑物做得更逼真。取一个正方形，我们在一个随机的位置把它垂直地分成两块。然后我们把两块中的每一块水平地分为两块。接着可以选择把每个生成的块再垂直地分为两块。这个过程可以按照需要重复很多次。在图 4.16.8 中，我们取了前面的例子中的一个较大的正方形。

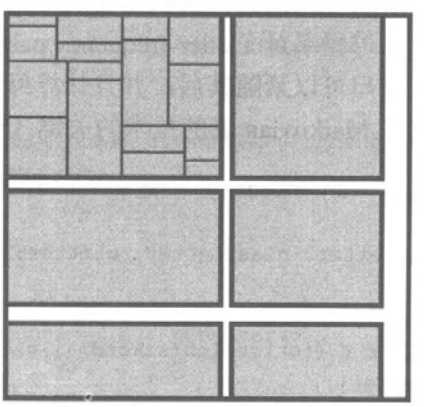


图 4.16.8

然而实现这个操作的算法相当复杂，这里有一个用于垂直线的可接受的改进算法：

（1）从最左边以一个随机的高度开始。

（2）计算到右边的四边形的数目，直到遇到墙壁为止。

（3）在从左边开始的一个随机数目的四边形内，从顶端到底部画一条线。

（4）重复第 1、2 和 3 步进行尝试。

画水平直线的等价步骤为：

（1）从最顶部以一个随机的宽度开始。

（2）向下计算正方形的数目，直到遇到墙壁为止。

（3）在从顶端开始的一个随机数目的四边形内，从左到右画一条线。

（4）重复第 1、2 和 3 步进行尝试。

当然，这个算法需要确保当从顶端到底部（或从左边到右边）进行计数的时候，是沿着一条线执行的，以避免从正方形的中间开始出现“漂浮”线。所有画出的直线必须在墙上有一个始点和一个终点。

下一个步骤是要在墙壁中插入门。可以用多种方法做到这一点，但是最简单的也许是再次从顶端到底部，或左边到右边，在内壁中插入一扇门以使墙壁上最多有 N 个门。

如果这完全是随机进行的，我们就要冒着一些墙壁没有门和一些房间可能没有直接通路的风险。假如它成为一个问题，算法必须进行相应的修改。

4.16.3 命名算法

对多数游戏设计来说，为物体、位置或特征起一个真实的名字是一件很繁琐的事情。由 David Braben 和 Ian Bell 开发的 *Elite*，为第一个恒星系统（第一个在近乎无限范围的宇宙中大得惊人的星系）提出了一些典范（Lave、Diso、Reidquat、Leesti、Orevre 等）。考虑到游戏所运行的机器（有 16K RAM）是资源有限的第二代微型计算机，所以这是一项极大的成就。很明显，没有办法储存这些东西（软盘还没有被发明），因此它们必须被生成。

我们可以从生成一个 6 个字母的单词开始，每个字母的选择是随机的。我们可能会得到类似于“ndpgbs”的单词，它不太令人满意。我们需要的是某种能确保相邻字母彼此自然适应的方法。第一步是生成一个字母频率对（letter-frequency pair）表，于是对于一个给定的字母，我们可以说有限个可能的字母可以紧随其后，并且计算每个可能字母紧随其后的几率。

下面的程序清单是一种称作 Markovian 表的技术的关键（也可参看[Dewdney90]）：

```
void AddLetters(char * szWord, unsigned long ulTable[28][28])
{
  int nWordLength, nFirstLetter, nLastLetter, nLetter;

  // Decapitalise the word
  for (nLetter = 0; nLetter < (int)strlen(szWord)-1;nLetter++)
      tolower(szWord[nLetter]);

  // Add the first, and last to the table
  nWordLength = (int)strlen(szWord);
  nFirstLetter = (szWord[0] - 'a') + 1;
  nLastLetter  = (szWord[nWordLength-1] - 'a') + 1;

  ulTable[0][nFirstLetter]++;  // Space followed by letter
  ulTable[nLastLetter][27]++;  // Letter followed by space

  for (nLetter = 0; nLetter < nWordLength-2; nLetter++)
  {
    nFirstLetter = (szWord[nLetter]   - 'a') + 1;
    nLastLetter  = (szWord[nLetter+1] - 'a') + 1;

    ulTable[nFirstLetter][nLastLetter]++;
  }
}
```

在这里描述的算法不需要多做解释，对于一个给定的单词，我们除了指定以哪个字母开始以哪个字母结尾，还要把它们加入表中。然后，为每一对字母更新表，以使被字母引用的位置增加，加固我们随后将用到的两者之间的关系。

正如看到的那样，这需要 28×28×4 字节（3 136）级的存储空间，这达到了我们要将其用于 16K RAM 机器的标准。的确，删除所有的空白记录可能会更大程度地减少空间需求。

一旦我们用该算法处理了一个选择好的文本，或多个文本，将会得到一张那些文本包含的所有可能的字母对的频率表。我们也应该储存单词的平均长度。

使用这张表，我们现在可以从第一个字母开始生成一个单词。为了要做这一点，我们应该选择一个随机的字母使它可以开始一个单词；换句话说，它落在被 ulTable[0]引用的表的一个列中，使 ulTable[0][x]比零大。此外，我们可以使用行中储存的值来确定一个被特殊单元选中的几率。

为了做到这一点，我们只是增加被 ulTable[0][i]（这里 i 从 1 到 26）引用的所有单元的值。接着，调用随机数生成器找回一个在 1 和我们已经计算出的值之间的值。然后我们再一次处理行，像前面那样对频率求和，直到我们已有的随机数小于运行的总数。这就是我们的字母；举例来说 ulTable[0][4]是一个 “d”。下面的代码显示了这个算法的一个一般形式，可以用于任何字母。

```
int GetLetterPosition(unsigned long ulWordTable[28][28], int nPrevious)
{
  int nCounter;
  unsigned long ulFrequencyTotal, ulFrequencyRunningTotal,
                ulRandomLetter;

  ulFrequencyTotal = 0;

  // Get the frequencies
  for (nCounter = 1; nCounter < 27; nCounter++)
  {
    ulFrequencyTotal = ulFrequencyTotal +
    ulWordTable[nPrevious][nCounter];
  }

  // Choose a 'target' frequency
  ulRandomLetter = rand() % (ulFrequencyTotal);

  // Move through the table until we hit the 'target' frequency
  ulFrequencyRunningTotal = 0;
  nCounter = 1;
  do
  {
    ulFrequencyRunningTotal = ulFrequencyRunningTotal +
                          ulWordTable[nPrevious][nCounter];
    nCounter++;
  } while (ulFrequencyRunningTotal < ulRandomLetter);
```

```
  return nCounter;
}
```

为了要构造一个长度为6个字母的单词，通用的算法将是：

```
Word[0] = GetLetterPosition (word_table,0)
x = 1
while x < 6 {
    word[x] = ((GetLetterPosition (word_table, word[x-1] - 'a'))
-1) + 'a'
    x = x + 1
}
```

这是CD上的软件NameGen的基本部分，而且CD上还提供了该软件的全部源代码。NameGen代码也用在示范程序UniGen中（为创建带有命名了的行星的恒星图）。

然而，按照现在的情况，单词生成算法并不预先排除像“fleeee”或“nooooo”这样悄悄混入的奇怪单词。问题是有一定数目的字母能够形成链。即“o”后可以紧随一个“o”，然后另一个“o”，如此下去几乎到无限。随机数生成器可以在一定程度上避免这种情况，但即使是3个“o”的重复也有点令人讨厌。

这样，我们的字构造算法应该变成：

```
Word[0] = GetLetterPosition (word_table,0)
x = 1
while x < 6 {
  word[x] = ((GetLetterPosition (word_table, word[x-1] - 'a'))
    -1) + 'a'
  x = RemoveChain(word,x)
}
```

如果一个字母在它本身的链中是第三个，RemoveChain函数就把它的索引返回到单词内。可以如下进行编码：

```
int RemoveChain( char word[MAX_LENGTH], int letter_position )
{
    int nPos = 0;
    int nOccurences = 0;

    while (nPos < strlen(word))
    {
        if (word[nPos] == word[letter_position])
            nOccurences++;
    }

    if (nOccurences > 2) return letter_position - 1;

    return letter_position + 1;
}
```

这将会清空单词，而且如果没有过多的链，字母位置计数器的值就加1。

最后采取的步骤用来确保为可信的单词的最后一个字母定位，而且确保它能普通地结束一个自然单词。举例来说，在英语中一些字母后面一般跟随一个元音，例如“j”。为了不混入以这样的字母的结尾的单词，我们必须执行至少两个操作。

第一个操作要确保字母可以跟在相邻的字母之后，而且确保被选择的字母可以结束一个单词（跟随一个空格）。此外，也必须警惕使我们可能会陷入一个无穷循环中的字母的存在。为达到这样的目的，下面的代码是从已经描述过的 GetLetterPosition 函数改编而来。

```
int GetEndLetter (unsigned long ulWordTable[28][28], int nPrevious)
{
  int nCounter;
  unsigned long ulFrequencyAdjacentTotal,
    ulFrequencyRunningTotal, ulRandomLetter,
    ulFrequencyEndingTotal;

  ulFrequencyAdjacentTotal = 0;
  ulFrequencyEndingTotal = 0;

  // Get the frequencies
  for (nCounter = 1; nCounter < 27; nCounter++)
  {
    ulFrequencyAdjacentTotal = ulFrequencyAdjacentTotal +
      ulWordTable[nPrevious][nCounter];

    ulFrequencyEndingTotal = ulFrequencyEndingTotal +
      ulWordTable[27][nCounter];
  }

  // Choose a 'target' frequency
  ulRandomLetter = rand() % ulFrequencyAdjacentTotal;

  // Move through the table until we hit the 'target' frequency
  ulFrequencyRunningTotal = 0;
  nCounter = 1;
  do
  {
    ulFrequencyRunningTotal = ulFrequencyRunningTotal +
                          ulWordTable[nPrevious][nCounter];
    nCounter++;

    if (ulFrequencyEndingTotal > 0)
      if ((ulFrequencyRunningTotal >= ulRandomLetter) &&
          (ulWordTable[27][nCounter] != 0))
        break;
    else
      if (ulFrequencyRunningTotal >= ulRandomLetter)
        break;

  } while (1 == 1);
```

```
    return nCounter;
}
```

选择字母函数的一个改进版本是作为与源字母相邻的目标字母和词尾字母的联合概率函数，作为练习留给读者。

另外需要注意的是这个方法只限于字母表（必须是罗马字母）的使用，而且没有将单词的开头大写。

4.16.4 参考文献

[Dewdney90] Dewdney, A. K., The Tinkertoy Computer, W.H. Freeman, 1990.

[Lecky99] Lecky-Thompson, Guy W., Algorithms for An Infinite Universe, Gamasutra, 1999.

[McNeill95] McNeill, James, SubDiv Applet, mcneja@wwc.edu.

4.17 分形地形生成——断层构造

Jason Shankel

在大自然中，诸如大陆构造板块的分离、块体运动（矿体废物），以及海岸线的腐蚀作用等引起了某些地形特征，如悬崖、台地和海滨峭壁。在本文中，我将展示如何使用一种断层构造（fault formation）算法来生成这些类型的地形。

4.17.1 断层构造

让我们从生成一个空的高地开始。画一条随机的直线穿过它并且为直线一侧的每个值增加一个偏移量值 *dHeight*：

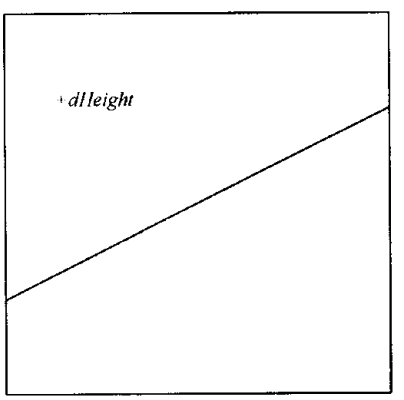


图 4.17.1 生成高地的第一步

然后减少 *dHeight*，画一条新直线，并且重复刚才的过程。继续生成直线并减少 *dHeight* 直到产生了充足的细节层次为止。

图 4.17.2 显示了 4 次、8 次、32 次和 64 次迭代所产生的地形高地（较高的海拔以白色表示）。

4.17.2 减少 dHeight

我们希望在每次迭代时都线性地减少 *dHeight*，但它不必非要降到 0。

令 $dHeight_{0..n}$ 为在每次迭代中 *dHeight* 的值。在第 i 次迭代时 *Dheight* 的值由下式给出：

$$dHeight_i = dHeight_0 + (i/n)(dHeight_n - dHeight_0)$$

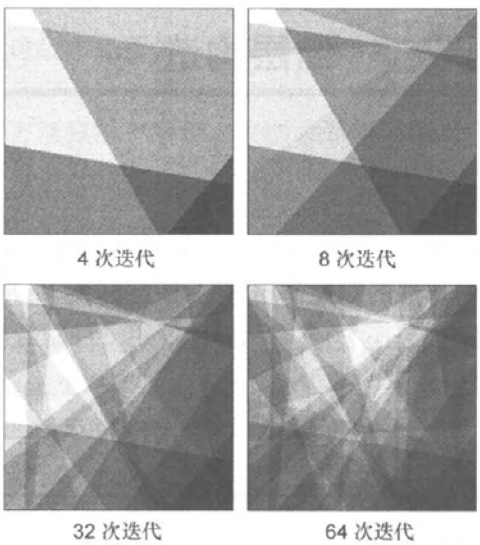


图 4.17.2 由这种过程产生的地形高地

4.17.3 生成随机直线

我们希望直线能与高地较好地相交，故为一个直线方程生成完全随机的值是不可取的，因为绝大多数直线将在单侧包含整个高地。

为了生成一条直线，最好选择两个位于高地内的随机点并用它们来确定一条直线。

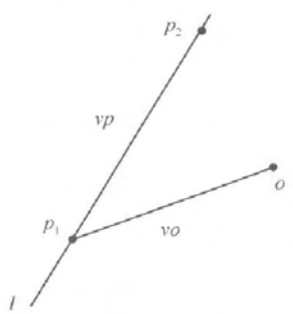


图 4.17.3 选择一条“随机直线”的说明

我们可以通过计算一个三维矢量叉积的 z 分量来确定一个点在直线的哪一侧（如图 4.17.3）：

令 l 为一条由点 p_1 和 p_2 定义的直线

令 vp 是一个由 p_1 指向 p_2 的矢量

令 o 为高地中的一个点

令 vo 是一个由 p_1 指向 o 的矢量

将 vp 和 vo 看作三维矢量，则它们的 z 分量为零。

令 $vx=vp\otimes vo$

如果 $vx.z>0$，则点 o 位于直线的左侧。如果 $vx.z<0$，则点 o 位于直线的右侧。如果 $vx.z=0$，则 o 在直线上。

4.17.4 腐蚀（erosion）

断层构造技术在高地的相邻单元间产生了显著的差异。对于一个较低的迭代次数，这会引起非常不真实的地形。即使是在较高数目的迭代中，地形看上去折痕依然非常明显，就像是一张用一个剪刀多次分割的纸。

问题出在我们的高地中有不真实的高频数据。在自然界中，相邻单元间尖锐的分界线将被腐蚀钝化掉。

为了模拟腐蚀，可以使这个高地通过一个低通图像过滤器（low-pass image filter）。

Robert Krten [Krten94]提出了一种简单的 FIR 过滤器。FIR 过滤器根据下面的公式将序列 $x_1, x_2, x_3...x_n$ 转换为 $y_1, y_2, y_3...y_n$：

$$y_i = ky_{i-1} + (1-k)x_i$$

这里 k 是 0~1 的过滤常数。较低的 k 意味着较少的腐蚀，较高的 k 意味着较多的腐蚀。典型地，一个大约 0.5 的 k 值对本应用作用得较好。

如果我们利用 FIR 过滤器函数，并在穿过高地的行和列两个方向上应用它，将得到一个经过精细腐蚀后的地形（如图 4.17.4）。

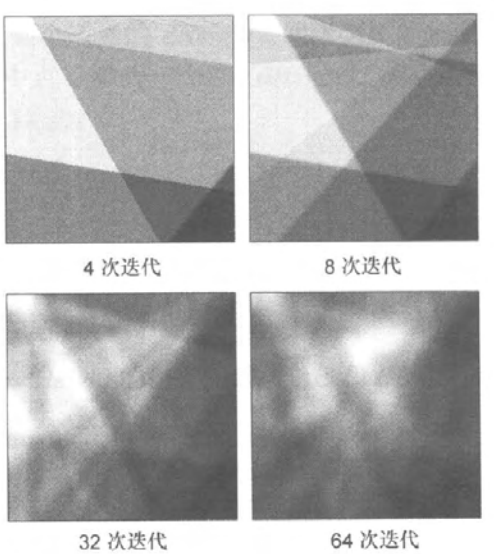


图 4.17.4 图 4.17.2 腐蚀后的版本

图 4.17.4 显示了图 4.17.2 中的地形经腐蚀后的同一地形。彩图 1 是一个腐蚀过的 64 次迭代级的 3D 渲染。

4.17.5 示例代码

代码中的算法允许你设置不同的迭代次数，第0次迭代至第n次迭代的值是*dHeight*，腐蚀因子是*k*，并且腐蚀间的迭代次数要进行传递。

4.17.6 参考文献

[Krten94] Krten, Robert, "Generating Realistic Terrain," Dr. Dobbs Journal (July 1994).

4.18 分形地形生成——中点置换

Jason Shankel

诸如洛矶山脉、Sierras、喜马拉雅山等山脉是由一种称为地壳隆起的地质过程形成的。构造板块运动的侧面压力引起了地球表面像织物一样起皱，就推挤出了山脉。在本文中，我将展示如何以一个递归的中点置换算法（midpoint displacement algorithm）模拟隆起，也称为等离子分形算法（plasma fractal）或菱形－正方形算法。

4.18.1 一维中点置换

在一维空间，中点置换是如下进行的。从一条线段 *AB* 开始（如图 4.18.1）：

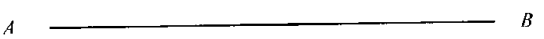


图 4.18.1 线段 *AB*

对于某个适当的 *dHeight*（线段 *AB* 的长度是一个不错的选择），取线段 *AB* 的中点 *C* 并用一个在− *dHeight*/2 和+ *dHeight*/2 之间的随机值置换它（如图 4.18.2）：

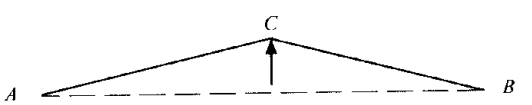


图 4.18.2 第一个置换阶段

减小 *dHeight* 的值并对线段 *AC* 和 *CB* 进行递归处理（如图 4.18.3）：

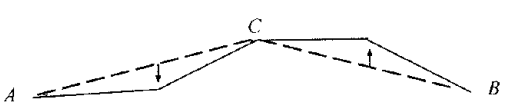


图 4.18.3 第二个置换阶段

重复执行上述步骤直到生成了足够多的细节为止（如图 4.18.4）：

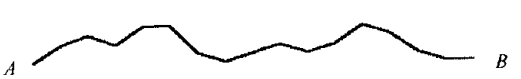


图 4.18.4 第 *N* 个置换阶段

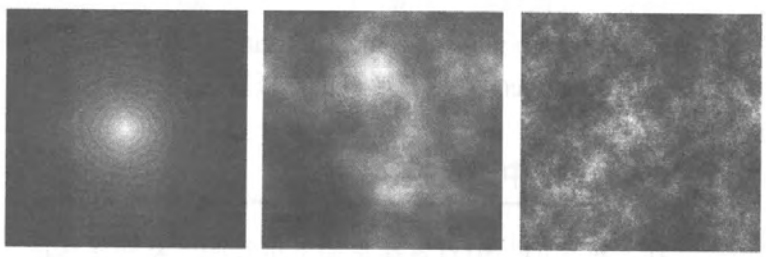
在每次迭代中，*dHeight* 被 2^{-r} 相乘，这里 r 是凹凸度常数（roughness constant）。

r 的魔术值是 1。如果 $r=1$，那么 *dHeight* 在每次迭代时都被 2 除，这也是水平线段减少的比率。当 $r=1$ 时，所生成的地形将是十分自相似的（self-similar）（小的部分将类似于大的部分）。

当 $r>1$ 时，*dHeight* 要比线段长度减少得快，于是较早进行的迭代对于地形有着不成比例的较大的影响。$r>1$ 有利于产生平滑的具有几个突出部分的地形（如山或山谷）。

当 $r<1$ 时，*dHeight* 要比线段长度减少得慢，于是较晚进行的迭代对于地形有着不成比例的较大的影响。$r<1$ 有利于产生混沌（chaotic）的地形。

图 4.18.5 显示了具有不同 r 值的 3 个地形（高海拔以白色表示）。



$r=4$ $r=1$ $r=1/4$

图 4.18.5 具有不同 r 值的地形实例

4.18.2 二维中点置换——菱形正方形算法

正如线段是一维中点置换的基本单位，矩形是二维中点置换的基本单位。

矩形要比线段稍微复杂一些，因为我们必须为每个矩形计算不是 1 个而是 5 个中点。也就是说，我们必须计算矩形自己的中点，以及构成矩形边的 4 条线段的中点。

在菱形－正方形算法中，矩形中点的计算过程被称为菱形步骤，而边的中点的计算过程则称为正方形步骤。

从一个矩形（*ABCD*）开始，选择 4 个顶点的高度值做为初始种子（如图 4.18.6）：

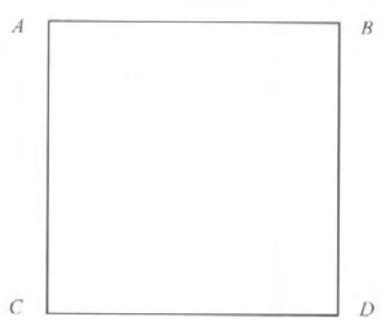


图 4.18.6 一个正方形 *ABCD*

通过计算 *A*、*B*、*C* 和 *D* 的平均值并加上一个 $-dHeight/2$ 与 $+dHeight/2$ 之间的随机值来计算中点 *E* 的高度（菱形步骤）（如图 4.18.7）：

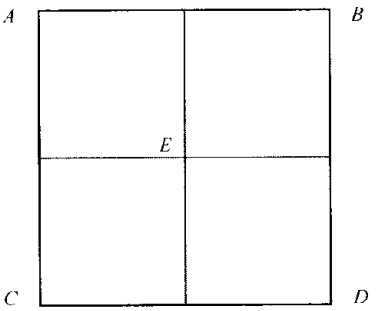


图 4.18.7 第一个置换阶段

$$E = (A+B+C+D)/4 + random(-dHeight/2, +dHeight/2)$$

现在，通过平均顶点的值和邻接矩形的中点的值，再加上一个 $-dHeight/2$ 与 $+dHeight/2$ 之间的随机值来计算每条线段的中点（*F*、*G*、*H* 和 *I*）的高度（正方形步骤）（如图 4.18.8）：

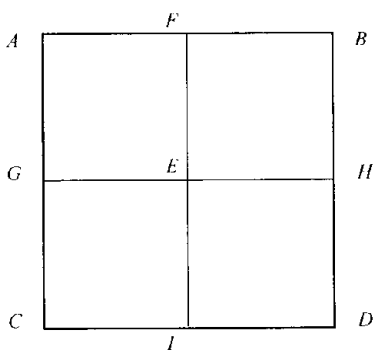


图 4.18.8 第一个置换阶段续

用 2^{-r} 乘以 *dHeight* 并对正方形 *AFGE*、*FBEH*、*GECI* 和 *EHID* 重复该过程（如图 4.18.9）：

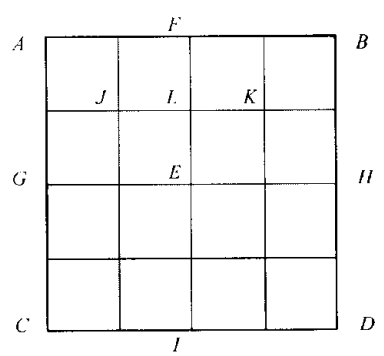


图 4.18.9 第二个置换阶段

重复上述步骤直到已经得到了充足的细节层次。

关于正方形步骤需要注意的一点是，正方形的值依赖于相邻正方形的菱形值。

例如：

$$L = (F+E+J+K)/4 + random(-dHeight/2,+dHeight/2)$$

所以，当算法在每一个细节层次进行迭代时，它必须在执行正方形过程之前为整个栅格首先执行菱形步骤。

为了在地形的边上计算一个正方形步骤值（比如说 H），可以将地形视为隐藏（wrap）。对于 H 的情形，取 E 作为 H 的左和右邻接点：

$$H = (B+D+2E)/4 + random(-dHeight/2,+dHeight/2)$$

4.18.3 高地中的菱形——正方形算法

当使用菱形正方形算法来填充一个高地时，最好选取一个宽度为 2^n（n 为整数）的正方形高地。这确保了每次迭代中矩形的大小将有一个整数的值。

彩图 2 是一个渲染到 256×256 高地的生成地形。

4.19 分形地形生成——粒子沉积

Jason Shankel

在大自然中，火山山脉和岛屿系统（如 Pacific Rim 的“火环”）是由熔岩流生成的。在本文中，我将用从分子束附生（Molecular beam epitaxy）领域借用的一种粒子系统来模拟熔岩流。

4.19.1 MBE 模型

分子束附生，或 MBE，是在一个晶体基层上沉积原子薄层的过程。我们能对 MBE 模拟中使用的模型进行修改来模拟熔岩流。关于 MBE 深入的数学分析，请参阅[Barabási95]。

4.19.2 粒子沉积

粒子沉积（Particle Deposition）的思想是使粒子序列下落并模拟它们在一个先前落下的粒子所组成的表面上的流动。下落足够数目的粒子将产生看起来像是粘性流体（熔岩）的流动图案（flow pattern）结构。

从一个空高度域开始并落下一个单独的粒子在其上（参见图 4.19.1）。

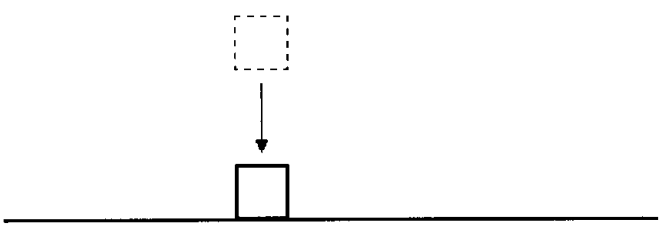


图 4.19.1 一个单独落下的粒子

现在，落下第二个粒子在第一个粒子的上面，并且摇动它直到它静止下来（即直到它的相邻粒子都不在一个更低的高度，参见图 4.19.2）。

继续落下粒子（周期性地改变下落点）直到有了一个大小合适的堆（参见图 4.19.3）。

你可以通过控制粒子落点如何移动来控制地形的外貌。使下落点保持在一个单一的位置将产生一个高高的山峰。周期性地移动下落点将产生一群小的连绵起伏的山峦。

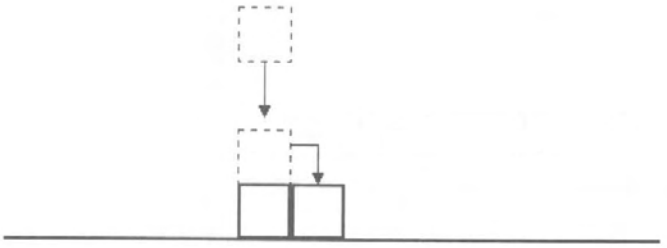


图 4.19.2 两个落下的粒子

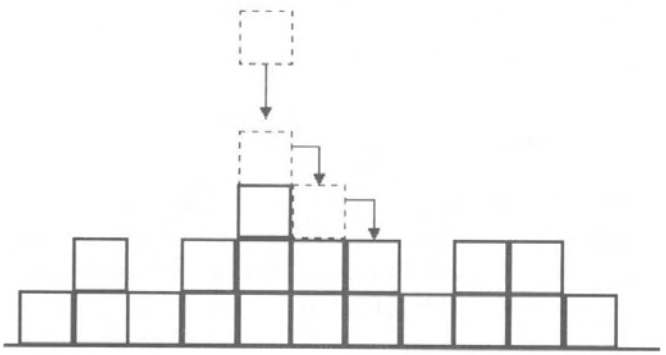


图 4.19.3 粒子的聚积

图 4.19.4 展示了用该技术生成的不同的地形（较高的高度以白色表示）。

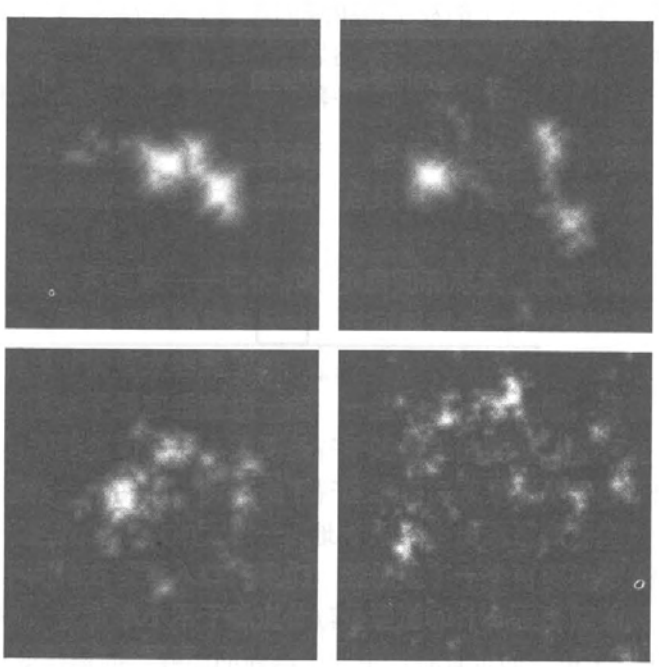


图 4.19.4 用该技术生成的一些高地

4.19.3 倒置火山口

真实世界的火山，尤其是活动的火山，有着非常与众不同的山顶。当岩浆停止流动时，火山顶部的熔岩冷却下来并且退回到地壳中，这样就在山顶产生了一个类似于碗状的区域，称为火山口。

可以通过对某个高度水平面以上的高地的值取反，为粒子山生成一个火山口。

假设一个高地由粒子沉积生成（如图 4.19.5）。

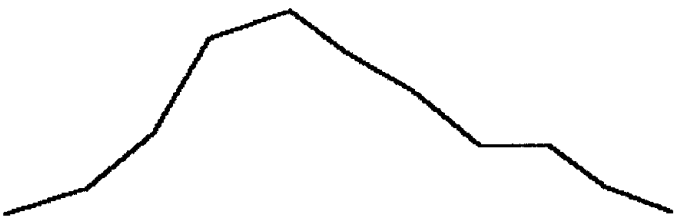


图 4.19.5 由粒子沉积生成的一个高地

通过一个任意的高度绘制一条直线（或一个平面）（如图 4.19.6）。

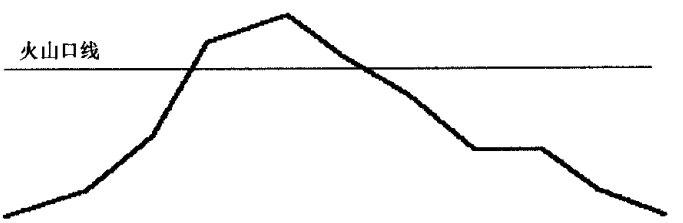


图 4.19.6 火山口线

然后对所有直线以上的高地值对该直线进行取反（如图 4.19.7）。

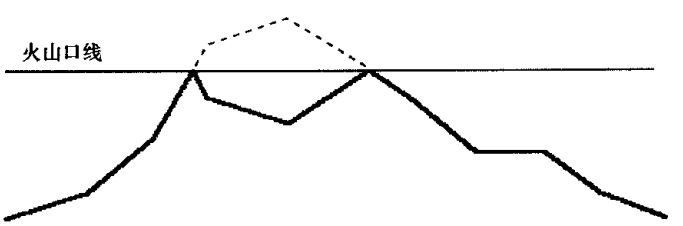


图 4.19.7 对火山口线以上的高地值取反

以这种方法削去峰巅可能导致产生一个折痕明显的火山口边缘。为了使得火山口看起来更真实，可以应用一个侵蚀过滤器（erosion filter）（参见“4.17 分形地形生成——断层构造”）钝化锋利的边缘。

在整个高地应用切掉火山口的技术可能会生成不合理的结果。如果地形有多个山峰，一个山峰的火山口线可能会干扰其他的山峰。

最好用一个倾撒填充技术（flood-fill）实现火山口的倒置。从一个初始点出发，将该点倒置并检验它的相邻节点。对于每个在火山口直线之上的相邻节点，倒置它并检验它的相邻

节点。就这样继续下去直到处理完所有的相邻节点。

彩图3是一个用粒子沉积方法生成的火山岛系统的3D渲染。

4.19.4 示例代码

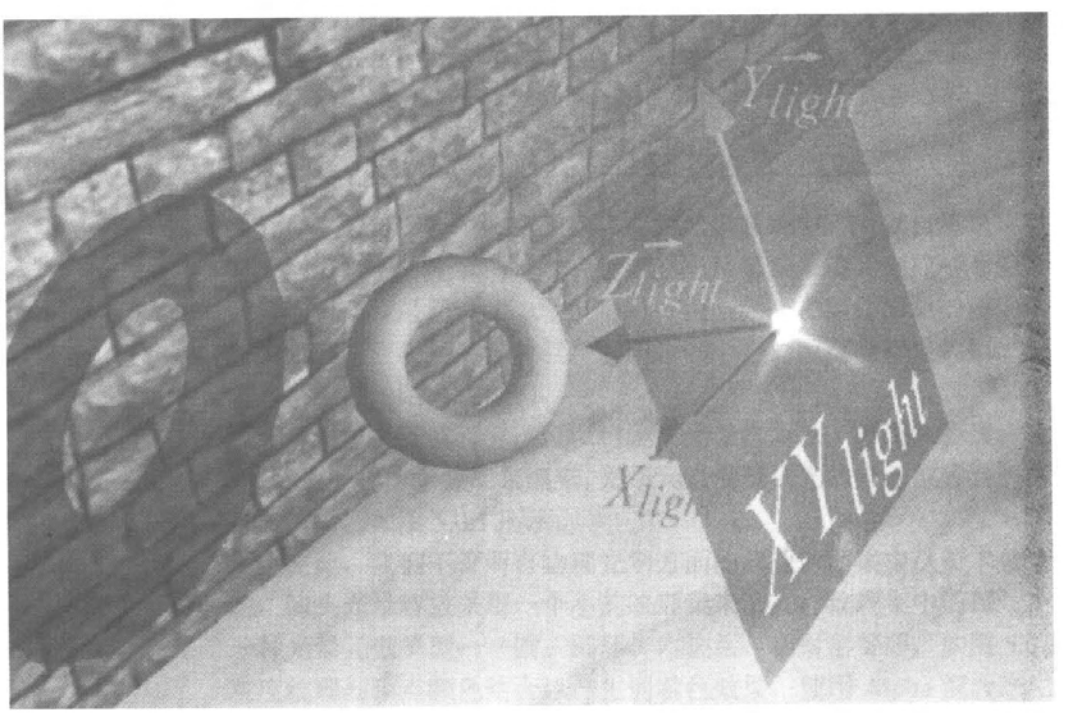
示例代码允许你控制堆的数目、每堆下落的粒子、粒子下落位置的移动，以及火山口深度。火山口深度被表示为一个山峰高度[0..1]的百分率。采用一个接近于1的深度值将产生灰岩坑（sinkhole）而不是山脉。

4.19.5 参考文献

[Barabási95] Barabási, A. L., and Stanley, H. E., *Fractal Concepts in Surface Growth* (Cambridge University Press, 1995)

第 5 章

像素特效



5.0 2D 镜头光晕

Yossarian King

镜头光晕是一种光学特效，它是当摄像机对向强光时，镜头的元件间相互反射而产生的。产生的最终效果就是从光源发散射来的一些透明的形状和颜色的变化图案。这种效果经常可以在电视中看到，当太阳进入到摄像机视野时就会出现。

在现实生活中，镜头光晕被认为是种缺陷，摄像机生产厂商通过使用特殊的镜头涂层来尽力避免光晕产生。不过视频游戏喜欢强调和夸大现实中所有让人觉得“酷”的地方，毫无疑问，镜头光晕就很酷。真实的镜头光晕靠光与摄像机光学系统的各个面复杂的交互作用产生。视频游戏中的镜头光晕只关注最终呈现的画面效果。本文讲述了一个迷人的镜头光晕特效的实现方法，此方法不用知道任何关于物理光学的知识，只需使用少量代码和美工。

5.0.1 方法

真实的镜头光晕产生于摄像机的光学系统中，所以自然呈现在所观看的景物“之上”。光晕的每个元素都是光从第二个镜片到前一个镜片上的反射。由于镜片是沿镜头中轴严格对齐的，因而在最终形成的画面上，这些反射成一条直线对齐，并且反射到画面中心的距离与相应的第二个镜片到前一个镜片的距离成比例。

基于以上观察研究，可以将镜头光晕作为 2D 问题来处理。光晕作为 3D 场景上的一个叠层来渲染，光晕元素沿着从光的投影位置到屏幕中心的直线渲染，如图 5.0.1 所示。

在此，我们不管所有物理光学方面的问题，而集中从美术效果上来考虑。镜头光晕特效采用一个小的纹理集来渲染，纹理集中的每一个元素为一种光晕元素类型——圆、圆环、六边形、辐射形等等，如图 5.0.2 所示。灰度纹理与顶点颜色结合以产生弱着色效果，使用 Alpha 混合产生半透明效果，元素以不同的大小渲染。

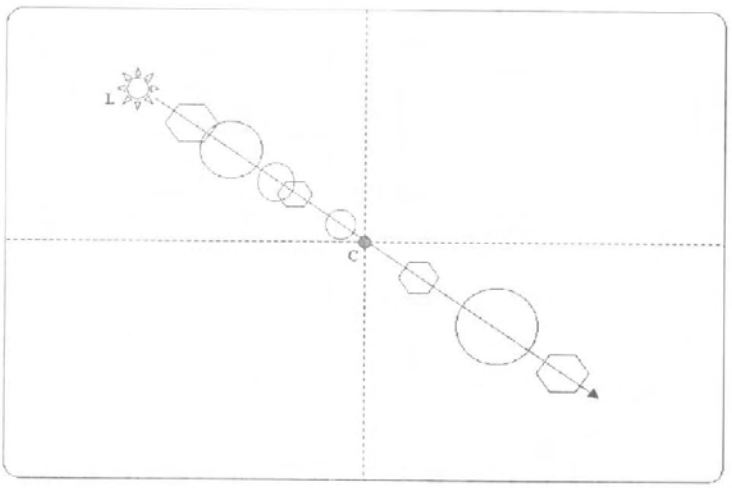


图 5.0.1 镜头光晕渲染是一个 2D 问题。镜头光晕的元素沿着从投影光到屏幕中心的直线渲染

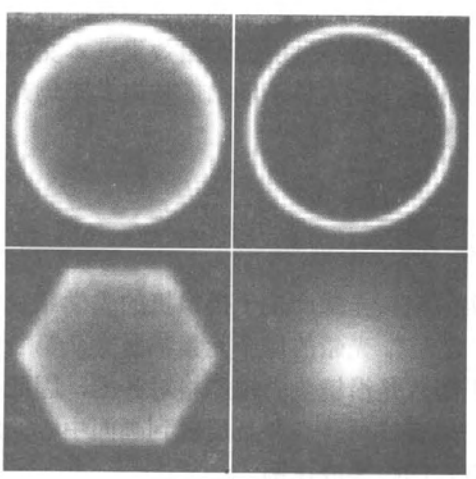


图 5.0.2 镜头光晕特效使用一个小型灰度纹理集来渲染

要达到真实的效果，镜头光晕还必须随着摄像机的移动产生相应的运动。总体上光晕的移动是由追踪从投射光位置到屏幕中心的直线来决定的。变化光晕元素的大小和透明度可以产生额外的细微效果。这个变化通过基于投射光位置与屏幕中心的距离缩放光晕元素的大小和 alpha 值来实现。光源距中心越远，元素越小越透明；光源距中心越近，元素越大越不透明。

上述方法的例子见彩图 4。

5.0.2 实现

上述方法可归结为对每个镜头光晕元素执行以下步骤：

（1）确定光晕元素的位置和大小；

（2）确定元素的纹理、颜色和透明度；

（3）采用计算出的属性，以 2D 子画面来渲染元素。

在这个实现方法中，光晕是元素的集合。每个元素都有以下静态属性：

- 纹理，可用的纹理（形状）。
- 距离，沿光源到屏幕中心直线的成比例的距离。
- 大小，（缩放前）元素的规格化大小。
- 颜色，在渲染时用于元素着色的 RGB 颜色。
- alpha，（alpha 缩放前）元素的透明度。

光晕还有个总体的缩放因子和最大尺寸，用于在渲染时控制元素的大小。这些属性都是在初始化的时候确定的。在演示代码中，属性可以随即确定，也可以从光晕描述文件载入。

在渲染过程中，每个镜头光晕的动态特性通过其静态特性与屏幕中光源的位置计算得出。光晕的纹理和颜色不会受影响，但位置、大小和 alpha 级别都将基于摄像机到光源的相对运动而变化。

镜头光晕特效的渲染伪码如下：

```
function renderflare:
    flare        // flare object to be rendered
    (lx,ly)      // projected position of light on screen
    (cx,cy)      // center of flare (normally center of screen)
{
    // Compute how far off-center the flare source is.
    maxflaredist = sqrt(cx^2 + cy^2)
    flaredist = sqrt((lx - cx)^2 + (ly - cy)^2)

    // Determine overall scaling based on off-center distance.
    distancescale = (maxflaredist - flaredist)/maxflaredist

    // Flare is rendered along a line from (lx,ly) to a
// point opposite it across the center point.
    dx = cx + (cx - lx)
    dy = cy + (cy - ly)

    for each element in flare
    {
// Position is interpolated between (lx,ly) and
// (dx,dy).
px = (1 - element.distance)*lx + element.distance*dx
        py = (1 - element.distance)*ly + element.distance*dy

        // Size of element depends on its scale, distance
// scaling, and overall scale of the flare itself.
        width = element.size * distancescale * flare.scale

        // Width gets clamped, so the off-axis flares keep a
// good size without letting the centered elements
// get too big.
        if (width > flare.maxsize)
```

```
            width = flare.maxsize

        // Flare elements are square (round) so height is
// just width scaled by aspect ratio.
        height = width * aspectratio

        // Alpha is based on element alpha and distance scale.
        alpha = element.alpha * distancescale

        // Draw the element's texture with computed
// properties.
        drawrectangle(   element.texture, element.colour,
alpha, px, py, width, height )
    }
}
```

5.0.3 源代码

镜头光晕演示示例包括 OpenGL 源代码和一个可执行的 Windows 程序。源代码分为一个 API 和使用该 API 的例子代码。API 包含定义光晕属性的结构和以下函数：

- FLARE_initialize：根据调用程序列出的属性，初始化光晕元素。
- FLARE_randomize：使用随即属性生成一系列光晕元素。
- FLARE_render：在给定屏幕位置渲染光晕。

演示示例使用上述函数，通过鼠标控制产生镜头光晕特效。鼠标光标作为光源在屏幕上的位置。镜头光晕可以随即生成，也可以从文件载入。演示示例中的 README.TXT 讲述了关于该演示界面的详细信息。

5.1 将 3D 硬件用于 2D 子画面特效

Mason McCuskey

图形卡的 3D 能力在过去的几年急速发展。在短短几年时间里，我们从 256 色、320×200 分辨率发展到了完全 3D 加速、1 600×1 200、32 位色。虽然当今大多数的图形卡都倾向于忽略 2D 而仅列出其针对 3D 的特性，但其实它们中还是有很多处理能力能用于实现炫目的 2D 特效。

本文介绍如何用 3D 硬件来实现 2D 特效。具体来说，我们将探讨如何进行 alpha 混合、子画面缩放以及子画面旋转。

5.1.1 进入 3D

本文所基于的是，显然，3D 场景必须渲染到 2D 表面来显示。OpenGL 和 Direct3D 的 Immediat 模式（在此模式下 3D 渲染设备必须附属于一个表面）都是如此。

在 3D 场景中见到的所有东西都是由图元（primitive）（通常是三角形或方形）组成。图元组以不同的方式组合，形成更为复杂的多边形。任何图元组都有“纹理”，该纹理掌管着该图元组的外形。我们不必关心纹理的细节，为了实现 2D 特效，只需要知道将纹理载入 OpenGL，然后将其指定到一个方形（四边形）图元即可。

5.1.2 建立 3D 场景

使用 3D 硬件显示子画面的技巧涉及如何建立你的 3D 世界。虽然有了 3D 卡，我们可以在 3D 空间中以任意旋转角度、任意位置来渲染多边形，但获得 2D 特效的秘密却在于建立 3D 场景，让所有东西都直接面对摄像机。总之，如果要建立 2D 游戏的真实 3D 模型，必须认识到 2D 子画面基本上是摄像机所直接指向的画面的固定片。换句话说，在 3D 世界中，2D 游戏看上去就像一个相当复杂的透视图。每一个子画面都是一个公告板，并且摄像机总是与子画面保持固定的距离而不移动（除非让它移动，这样的话可以产生有趣的特效）。

在阅读下面关于建立纹理和显示子画面的小节时，请牢记以上概念。

5.1.3 建立纹理

建立纹理需要用到几个 OpenGL 调用：

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glGenTextures(1, (GLuint*)&mTextureID);
if (mTextureID == 0)
  { GLenum gle=glGetError(); /* handle errors! */ }
glBindTexture(GL_TEXTURE_2D, mTextureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0, GL_RGBA,
  GL_UNSIGNED_BYTE, gpgtexture);
```

这个代码片段是示例程序中 C3DSprite::Init()方法的一部分，它在 OpenGL 中初始化纹理。gpgtexture 是一个指针，指向 RGBA 像素值数组（4 个字节，分别为红、绿、蓝和 alpha 的值）。glTeximage2D()调用使用 gpgtexture 像素数组建立纹理。glTexParameteri()调用建立纹理的各种属性，包括 wrap 模式（GL_CLAMP，表示“不循环使用纹理”）、用于纹理缩放的过滤器（在此我们使用线性过滤器）。

5.1.4 绘制 3D 子画面

现在纹理已经建立，我们可以开始绘制子画面了。要使用子画面，客户端应用程序需要首先建立该子画面的所有参数（屏幕中的位置、大小、透明度或 alpha 值等）。然后客户端调用 Display()方法渲染子画面。

以下代码源于 C3DSprite::Display()方法：

```
// Set up the rotation and translation matrices
glPushMatrix();
glTranslatef(m_iX, m_iY, 0);
glRotatef(m_fRotation, 0, 0, 1);
```

该代码做的第一件事是建立旋转和平移矩阵。OpenGL 使用矩阵栈，这样我们可以对所创建的顶点应用全局变换。所创建的任意顶点使用栈顶的矩阵进行转换。在创建顶点前，需要建立一个矩阵，它旋转顶点并将它们平移到子画面应为 (*m_iX, m_iY*) 的位置上。

为了得到该矩阵，我们将一个新的单位矩阵压入模型视图栈，然后将它乘以到子画面位置的平移矩阵（glTranslatef 的第三个参数是 z 坐标），再乘以旋转矩阵（使用 m_fRotation 变量）。glRotatef 的第二、三、四个参数告诉 OpenGL 相对哪个坐标轴进行旋转（在此是 z 轴）。z 轴垂直于显示器平面。

下一步是建立混合模式和纹理模式。前几行代码建立混合模式。我们将子画面与任意已经渲染到帧缓冲的图形进行 alpha 混合，使用的是 *srcColor*srcAlpha+destColor**(1−*srcAlpha*)。

然后，建立纹理模式。纹理模式表示，将要渲染的像素是源颜色和 alpha 通过纹理调解（相乘）而来。此后，调用 glBindTexture()命令，告诉 OpenGL 我们希望下面的图元使用 mTextureID 来纹理化，该 ID 在早先建立子画面的纹理时得到。

```
// Draw the sprite
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glBindTexture(GL_TEXTURE_2D, mTextureID);
```

现在建立 4 个顶点，每个顶点是方形图元的一个角。每一个顶点都有：

（1）颜色。通过 glColor4ub()函数设置颜色。在本例中，所有顶点的颜色都为纯白色，RGB(0xff, 0xff, 0xff)。

（2）相应的纹理坐标（通过 flTexCoord2f()函数设置），告诉 OpenGL 如何将纹理拉伸或缩小到方形图元上。纹理坐标（0.0, 0.0）表示最左上角的纹理像素，纹理坐标（1.0, 1.0）表示最右下角的纹理像素。事实上，本例中的纹理能够正好拉伸到该方形上。（请记住，方形图元自己可以为任意大小，纹理随之放大或缩小；我们在此讨论的是如何将纹理贴附到该方形图元上。）

（3）在 3D 空间中的位置，通过调用 glVertex3f()设置。在本例中，我们将子画面的中心作为局部坐标原点，方形的左上角即为(*−m_iWidth/2, −m_iHeight/2*)，右下顶点是(*m_iWidth/2, m_iHeight/2*)，其中 m_iWidth 与 m_iHeight 分别是子画面的宽度和高度。这个方法有效地将局部坐标原点定位在子画面的中心，这样当子画面旋转的时候就可以让它绕着中心旋转。

```
glColor4ub(0xff, 0xff, 0xff, m_iAlpha);
glTexCoord2f(0.0F, 0.0F);
glVertex3f(-m_iWidth/2, -m_iHeight/2, 0);

glColor4ub(0xff, 0xff, 0xff, m_iAlpha);
glTexCoord2f(1.0F, 0.0F);
glVertex3f(-m_iWidth/2, m_iHeight/2, 0);

glColor4ub(0xff, 0xff, 0xff, m_iAlpha);
glTexCoord2f(1.0F, 1.0F);
glVertex3f(m_iWidth/2, m_iHeight/2, 0);

glColor4ub(0xff, 0xff, 0xff, m_iAlpha);
glTexCoord2f(0.0F, 1.0F);
glVertex3f(m_iWidth/2, -m_iHeight/2, 0);
```

最后，以下代码结束场景，并将图形状态和矩阵栈置回到以前的状态：

```
glEnd();
glDisable(GL_TEXTURE_2D);
glDisable(GL_BLEND);

// Pop the matrix we set up above
glPopMatrix();
```

5.1.5 添加特效

现在，基本的绘制已经完成，我们可以添加一些特效。很幸运，最受欢迎的一个 2D 特效——alpha 混合，在 3D 中也很好实现。要获得 alpha 混合，只用简单地为每个子画面顶点指定 alpha 值即可。快速回顾一眼前面的代码，在调用 glColor4ub()中，第 4 个参数（本例中是 m_iAlpha 变量）就是顶点的 alpha 值。alpha 值与红、绿、蓝颜色值相似，可以为 0~255 中的任意值，0 代表完全透明，255 代表完全不透明。举个例子，如果要创建一个“半”透明的子画面，我们只用将子画面 4 个顶点的 alpha 值设为 128 即可。

当然，如果要产生不同的特效，我们可以单独改变每个顶点的 alpha 值。例如，如果将左边的两个顶点设为 0，右边的两个顶点设为 255，我们就得到一个从完全透明（左边）到完全不透明（右边）渐变的子画面。

要产生一些彩色特效，可以为方形图元的 4 个顶点设定不同的颜色。我们是通过在调用 glColor4ub 时放入不同的 RGB 值来做的。OpenGL 自动将这些颜色混合在一起。如果将左边设为蓝色（RGB(0, 0, 255)），右边设为红色（RGB(255, 0, 0)），那么将得到从蓝到红渐变的子画面。照此办法，我们可以快速添加高光色彩，省去了建立 OpenGL 光源的所有麻烦。请注意，如果不希望 OpenGL 在你的多边形上采用平滑的颜色/alpha 渐变，可以通过调用 glShadeModel(GL_FLAT)来改变渐变行为。

缩放 3D 子画面也很容易。3D 硬件会处理纹理的拉伸，我们所需做的只是建立子画面的顶点。拉伸或缩小子画面就和增加或减小方形图元的宽高一样简单。如果要沿 x 轴 2 倍拉伸子画面，只用将子画面的宽度值从 m_iWidth 改为 m_iWidth*2 即可。类似地，要沿 y 轴 2 倍拉伸子画面，只要将高度值从 m_iHight 变为 m_iHight*2。要缩小子画面到其一半大小，将宽度和高度都除以 2 即可。任意拉伸组合都可以得到，最妙的是，你用不着为子画面的单个像素操心。所有问题 3D 硬件都为你考虑到了。

旋转子画面同样很容易。同样，只用对顶点的位置进行操纵。可以指定一个角度，将其传递到 OpenGL 的 glRotatef 函数中，该函数将其应用于当前的转换矩阵。同样，单个像素的位置会由 3D 硬件自动计算得出。

要产生另一种有趣的特效，可以改变子画面旋转的坐标轴。glRotation 使用一个角度和三个参数，通过这三个参数我们可告诉 OpenGL 用于旋转子画面的坐标轴。绕着 x 轴或 y 轴旋转（其效果是在 3D 空间水平或垂直翻转子画面），而不仅仅是绕着 z 轴旋转（子画面自转），很可能会产生有用的效果。

5.1.6 结论

本文提出的 2D 子画面方法可能乍看上去没有什么必要，通常 2D 程序员通过 blitting 函数（如 DirectDraw 的 Blt()及 Win32 的 BitBlt()）绘制子画面更容易且更熟悉些。然而，学习如何用 3D 硬件“blit”从长远看是值得的，因为它最终更容易实现高级 3D 特效，如 alpha 混合及子画面缩放。此外，对于一般的计算机系统，将 alpha 混合及缩放所需的图形处理转移到图形卡中进行，可以节省 CPU 资源，这将最终给你更多空间去制作更好的游戏。

5.2 基于运动的静态光照

Steven Ranck

本文讲述了一种方法，将更多动态特性引入到预计算静态光照中。该方法能产生震撼的动态光照，其计算开销仅比传统的静态光照大一点点。

很多游戏通过预先计算各个顶点的光照颜色使用静态光照。这样做可以几乎不耗费计算开销而生成逼真的 Gouraud 光照。但是生成的光照看上去也是完全静态的。本文提出了一个算法，可以在静态光照中加入动态运动，而且执行速度几乎与传统的静态光照一样。例如，一面岩石墙上靠在一起的两个火把，会在墙上产生适当摇曳、相互辉映的火光。

5.2.1 传统的静态光照

传统的静态光照只是简单地将预先计算的 Gouraud RGB 值存储在每个顶点中，在渲染的时候以漫射方式使用。预计算 RGB 值可以由工具，也可以由游戏的初始化代码产生。在这两种情况下，都需要使用工具将光打在对象上，并且指定各项属性，如颜色、强度、照射半径、照射半径中光的减弱、光源类型（泛光灯、定向灯、聚光灯等）。由于处理的是静态光照，光源和其所有属性都是固定不变的。此外，光源在其照射对象的对象空间中的位置是固定不变的，也就是说，静态光源不能相对其照射对象移动。通过光源的位置和属性，可以使用任何光照方程计算出每个对象的顶点光。图 5.2.1 展示了该概念的一个简化 2D 表示。

图 5.2.1 表示的是一个有 6 个顶点的物体，静态地被两个光源照射。顶点 5 在光源 A 的影响范围中，顶点 3 在光源 B 的影响范围中，顶点 2 同时处于两个光源的影响范围中，其他顶点则均不在这两个光源的影响范围内。由于光源 A 和光源 B 是静态的，它们在物体坐标空间中是固定的；如果物体在世界空间移动和旋转，光源 A 和光源 B 将随着物体一起移动和旋转。因而，每个顶点的 RGB 光照只被计算一次（因为它从不变化）并作为顶点结构的一部分存储。基于以上原因，静态光照对于那些附属在物体上的光源对该物体的光照很有效。例如照亮街道的街灯，宇宙飞船外壳上照亮飞船的灯。

在图 5.2.1 中，顶点 5 的静态 RGB 可以简单地在传统光照方程中使用顶点位置和与光源 A 的法线计算得出。顶点 3 类似，不过使用的是光源 B。顶点 2 受到两个光源的影响，因而其结果 RGB 是这两个光照方程的和。

其余的顶点均不在这两个光源的影响下，所以它们的 RGB 是（0, 0, 0），照亮它们的责任就全在动态光和环境光上。

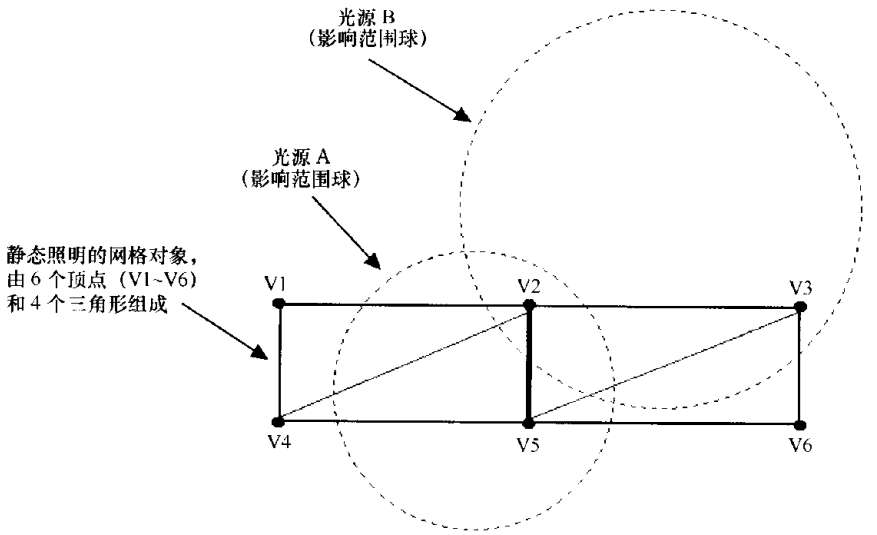


图 5.2.1 表态照明的网格对象

采用何种光照方程可完全根据设计者的喜好而定。因为是预先计算的，光照方程可以很复杂也可以有很大计算量。不管实际使用的方程，我们可以将光照函数写为：

$$I = f(L, V) \tag{5.2.1}$$

$$C_R = I \bullet L_R \tag{5.2.2}$$
$$C_G = I \bullet L_G$$
$$C_B = I \bullet L_B$$

其中：

L 代表光源的属性（位置、强度、照射半径、照射半径中光的减弱、光源类型等）。

V 代表顶点的属性（位置、法线）。

$f(L, V)$是光照方程。

I 是顶点上光的强度标量，通过光照方程计算得出。

L_R 是光的颜色（红色分量）。

L_G 是光的颜色（绿色分量）。

L_B 是光的颜色（蓝色分量）。

C_R 是最终的红色分量，将存储在顶点结构中。

C_G 是最终的绿色分量，将存储在顶点结构中。

C_B 是最终的蓝色分量，将存储在顶点结构中。

以下是一个简化了的泛光灯光照方程：

$$f(L, V) = (D_{VL} \bullet N_V)(R-D)/R \tag{5.2.3}$$

其中：

D_{VL} 是顶点到光源的单位向量。

N_V 是顶点的单位法线。

D 是顶点到光源的距离。

R 是光源的影响半径。

泛光灯是点光源，其光线发射向所有方向。光线的强度随着顶点到泛光灯位置的距离增大而减弱（衰减），在上面的方程中，$(R-D)/R$ 执行这个衰减。如果顶点位于光源所在的位置，D 是 0，$(R-D)/R$ 是 1，为最亮。如果顶点位于光源的影响范围之外，D 等于 R，$(R-D)/R$ 为 0。如果顶点位于光源的位置与其范围边界之间，$(R-D)/R$ 生成的数在 0 和 1 之间。更复杂的泛光灯可使用距离衰减方程以更接近现实世界模型。不过在本例中，方程中的线性衰减已经足够了。当然，若顶点位于光源影响半径之外时（$D>R$），要求 $f(L, V)$返回 0。

泛光灯方程中的第一项$(D_{VL} \bullet N_V)$，基于从泛光灯到顶点的光线与顶点法线向量的点积，执行了另一个衰减。它称为漫射（diffuse）光照因子，是一种常用的方法，当光线直接射在顶点上的时候加亮顶点，当光线从某个角度射向顶点时渐暗顶点。如果$(D_{VL} \bullet N_V)$得负值，则 $f(L, V)$返回 0。也就是说，如果光线从背面射向顶点，则阻止顶点被照亮。

方程 5.2.3 可产生一个好看的泛光灯，如果使用更复杂的光照方程，还能达到更高的品质。不管什么情况，方程 5.2.3 将得出一个标量，也就是该顶点总体光强度。方程 5.2.2 将用该强度乘以每个颜色分量（红、绿、蓝），生成最终的三个颜色分量，存储在顶点结构中。在渲染时，直接检索这三个颜色分量用于 Gouraud 光照。以下是一个顶点结构和渲染时绘制静态光照三角形的 OpenGL 代码片段。

```
typedef struct {
    float fR, fG, fB; // Static RGB color (0.0 -> 1.0)
} Color_t;

typedef struct {
    vector3 Pos; // This vertex's 3D position in model-space
    Color_t StaticColor;  // RGB to be used for this vertex
} Vertex_t;

void DrawMyTriangle( const Vertex_t *pV1,
const Vertex_t *pV2, const Vertex_t *pV3 ) {
        glBegin( GL_TRIANGLES );

    glColor3f( pV1->StaticColor.fR, pV1->StaticColor.fG,
pV1->StaticColor.fB );
    glVertex3f( pV1->Pos.x, pV1->Pos.y, pV1->Pos.z );

    glColor3f( pV2->StaticColor.fR, pV2->StaticColor.fG,
pV2->StaticColor.fB );
    glVertex3f( pV2->Pos.x, pV2->Pos.y, pV2->Pos.z );
```

```
    glColor3f( pV3->StaticColor.fR, pV3->StaticColor.fG,
pV3->StaticColor.fB );
    glVertex3f( pV3->Pos.x, pV3->Pos.y, pV3->Pos.z );

    glEnd();
}
```

传统的静态光照有个缺点，它只能生成静止的结果。也就是说，顶点的 RGB 值对于每一帧渲染都是固定不变的。对于阳光、月光以及其他许多不变化的光源，这没有什么问题。然而，对于较复杂的光源，如闪动的霓虹灯、火把的火焰、篝火、闪电，就不能使用传统的静态光照，因为它们要求帧与帧之间的 RGB 值有变化。如示例代码所示，传统的静态光照达不到这个效果。对此可使用基于运动的光照。

5.2.2 基于运动的静态光照

对于动态光照，传统静态光照的优势在于它的执行速度。在运行时，它几乎不需要执行任何计算。从顶点结构中检索出 RGB 值，直接加到用于渲染该顶点的颜色值中即可。不过它的缺点是只能产生静止的效果。每个顶点的 RGB 值预先计算、存储，在帧与帧之间从不改变。基于运动的静态光照可提供比传统的静态光照更具动态的效果，而性能改变微乎其微。

基于运动的静态光照仍属于静态光照。也就是说，与传统静态光照一样，照亮物体的静态光源在物体坐标空间中是固定的。不过，使用运动静态光照，可以实时地变化光的 RGB 分量。有一些动画，如墙上用于控制房间灯光的开关，可能处于玩家的控制之下。另一些动画，如火焰、电火花以及闪电，可能由算法控制。此外，有些光的 RGB 值可能固定不变。所有这些动画（包括固定不变的）都叫做光运动（light motif）。

要实现基于运动的静态光照，游戏需要一个运动表，它包含每个光运动的 RGB 表项。在后面我们将看到，该运动表充当了调色板的作用。图 5.2.2 展示了一个 9 表项运动表例子。每个表项是一个 RGB 颜色。基于不同的运动类型，RGB 颜色既可以在游戏初始化时预先计算，也可以每帧动态计算一次。不论怎样，在运动表中存储的 RGB 值是完全饱和（full-intensity）、未衰减的颜色。现在，在顶点结构中我们只存储了指向运动表的索引，而没有直接存储 RGB 颜色。我们马上将讨论此概念的细节。

对于帧与帧之间颜色保持不变的固定运动光，光的 RGB 值在初始化时直接存储在运动表中，保持在整个游戏或关卡过程中。在图 5.2.2 中，表项 7 和表项 8 包含两个固定运动的 RGB——暗红和浅蓝。

对由玩家控制的运动光（如灯开关），当事件发生时（如玩家按灯开关）RGB 值更新。在图 5.2.2 中，表项 4 和表项 5 包含两个由灯开关控制的静态光的 RGB。开始时，它们包括的可能都为 RGB=(0, 0, 0)（初始化时建立）。当玩家打开开关 0，游戏代码将灯光颜色存储进表项 4 中。例如，如果灯是绿色的光，在表项 4 中存储 RGB=(0, 1, 0)。如果玩家又将灯关掉，将 RGB=(0, 0, 0)存进表项 4 中。在图 5.2.2 中，表项 5 是另一个开关，控制另一个灯，它既可以在不同的房间中也可以在同一个房间中。完全有可能这两个灯很近，它们同时对同个房间的同一个顶点产生影响（类似图 5.2.1 中，光源 A 和光源 B 同时影响顶点 V2）。在这种情

况下，玩家摆弄开关，将期待看到房间中两个灯光互相影响的正确效果。稍后我们将看到运动静态光照如何处理这种情况。

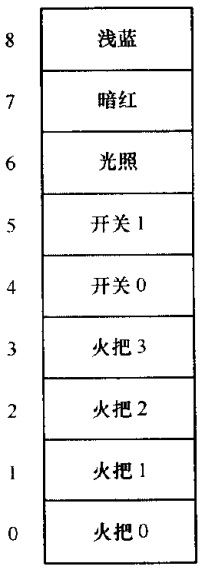


图 5.2.2 运动表示例

图 5.2.2 中表项 0 到表项 3 包含了一个摇曳火把的动画 RGB 颜色。这类运动是由算法实现的，每帧更新一次。以下是生成摇曳火焰动画的程序代码片断，每帧调用一次。

```
#define FLAME_SPEED   6.0f
// Constants determined via experimentation:
#define FLAME_K1  (0.093f * FLAME_SPEED)
#define FLAME_K2  (0.137f * FLAME_SPEED)
#define FLAME_K3  (0.195f * FLAME_SPEED)
#define FLAME_K4  (0.106f * FLAME_SPEED)
#define FLAME_K5  (0.170f * FLAME_SPEED)
#define FLAME_K6  (0.287f * FLAME_SPEED)

// Generates the RGB color for a particular frame of the flame
// motif and stores it in *pColor. nGameFrameCounter is simply
// the frame number of the game's current frame and is
// incremented once per frame.
void GenerateFlameMotif( unsigned int nGameFrameCounter,
Color_t *pColor ) {
    double dSinSum;
    float fIntensity, fAngle;

    fAngle = (float)nGameFrameCounter;

    dSinSum =
          sin( fAngle * FLAME_K1 )
        + sin( fAngle * FLAME_K2 )
```

```
            + sin( fAngle * FLAME_K3 )
            + sin( fAngle * FLAME_K4 )
            + sin( fAngle * FLAME_K5 )
            + sin( fAngle * FLAME_K6 );

        fIntensity = (float)dSinSum * 0.1f;
        fIntensity += 0.7f;

        if( fIntensity > 1.0f ) {
            fIntensity = 1.0f;
        }

        pColor->fR = fIntensity;
        pColor->fG = fIntensity * 0.4f;
        pColor->fB = 0.0f;
    }
```

在每次游戏循环的最上面，游戏调用 GenerateFlameMotif()生成火焰动画，并将结果 RGB 存储到运动表中的 0 道。很有可能房间中不只一个火把，如果所有的火把都采用同一个动画，它们看上去就将像在同步摇曳，效果很糟糕。所以，在运动表中，我们提供了不同的火把运动，以便设计者能将不同的火把运动分配给相邻的火把。通常并不需要为不同的运动编写不同的算法，简单地相移帧计数，并且调用同一个算法就足够了，对部分工作能产生很好的效果。

运动光算法通常是些简单的函数，不过就算需要使用运算量较大的函数，它也只需要每帧计算一次，因而对游戏的总体性能几乎没有什么影响。例如，在前面的代码中，相对于 CPU 执行一帧花费 1/60s 而言，每帧调用 6 次 sin()是微不足道的。即使这样，还是可以通过自己编写 sin()函数来降低总开销。可以牺牲一些精确度，将 sin()写为执行表查找。对于光照计算而言，精确度并不重要。不论怎样，运动生成函数的魅力在于它们每帧仅调用一次，而动态光照计算需要每帧每顶点执行一次！

现在已建立了运动表，让我们将注意力转向顶点结构。对于传统的静态光照，预计算的 RGB 颜色直接存储在顶点结构中。对于基于运动的静态光照，则需要定义不同的顶点结构。

```
typedef struct {
    vector3 Pos; // This vertex's 3D position in model-space
    int nMotifIndex; // Index into the motif table
    float fIntensity; // Intensity of the light motif's
// RGB at this vertex
} Vertex_t;
```

在顶点结构中，去掉了 StaticColor 域，使用了 nMotifIndex 和 fIntensity 域。这些域告诉渲染引擎使用运动表中的哪个光运动，以及在该顶点上光运动的 RGB 的强度。这两个值都可以实时或在游戏初始化时建立。运动索引代替了光的颜色属性，设计者不用再添置某个篝火并指定其为桔红色，而是为其指定一个运动即可。设计者放置光并指定光的属性，如强度、照射半径、照射半径中光的减弱、光源类型。不过，当指定颜色时，设计者可以从一些可能的运动值中选择一个运动。以篝火为例，设计者可以选择火把运动。这个运动不光描述了颜

色，还给出了静态光的动画。火把运动从颜色和颜色动画两方面模仿了摇曳的火焰。

fIntensity 域就是方程 5.2.1 中的 I，就是将传统的静态光照所使用的光照方程应用于这个顶点。设计者设置的光位置和光属性与顶点的位置和法线相互作用，产生 I，这是该顶点的光强度。参见方程 5.2.3，看看对于一个泛光灯，这些参数是如何相互作用得出 I 的。

现在有了强度（来自顶点结构中的 fIntensity 域）和 RGB 颜色（来自顶点结构中 nMotifIndex 域所指向的运动表项），就可以计算方程 5.2.2 来得到顶点最终的运行时 RGB 值了。

以下是 OpenGL 代码示例，用于渲染一个静态的运动光照三角形。

```
typedef enum {
    MOTIF_FLAME0,
    MOTIF_FLAME1,
    MOTIF_FLAME2,
    MOTIF_FLAME3,
    MOTIF_SWITCH0,
    MOTIF_SWITCH1,
    MOTIF_LIGHTNING,
    MOTIF_DARK_RED,
    MOTIF_BRIGHT_BLUE,

    MOTIF_COUNT
} Motif_e;

Color_t aMotifTable[MOTIF_COUNT];

void DrawMyTriangle( const Vertex_t *pV1,
const Vertex_t *pV2, const Vertex_t *pV3 ) {
    Color_t *pColor;
    float fR, fG, fB;

    glBegin( GL_TRIANGLES );

    pColor = &aMotifTable[ pV1->nMotifIndex ];
    fR = pColor->fR * pV1->fIntensity;
    fG = pColor->fG * pV1->fIntensity;
    fB = pColor->fB * pV1->fIntensity;
    glColor3f( fR, fG, fB );
    glVertex3f( pV1->Pos.x, pV1->Pos.y, pV1->Pos.z );

    pColor = &aMotifTable[ pV2->nMotifIndex ];
    fR = pColor->fR * pV2->fIntensity;
    fG = pColor->fG * pV2->fIntensity;
    fB = pColor->fB * pV2->fIntensity;
    glColor3f( fR, fG, fB );
    glVertex3f( pV2->Pos.x, pV2->Pos.y, pV2->Pos.z );

    pColor = &aMotifTable[ pV3->nMotifIndex ];
    fR = pColor->fR * pV3->fIntensity;
```

```
    fG = pColor->fG * pV3->fIntensity;
    fB = pColor->fB * pV3->fIntensity;
    glColor3f( fR, fG, fB );
    glVertex3f( pV3->Pos.x, pV3->Pos.y, pV3->Pos.z );

    glEnd();
}
```

以上实现仅支持每个顶点一个运动，这对于顶点位于不只一个静态光影响下的模型而言不大有用。要支持多于一个的运动，需要扩展顶点结构，对每一个影响它的运动光都有一个运动/强度对。通用的解决方法如下：

```
typedef struct {
    int nMotifIndex;      // Index into the motif table
    float fIntensity;     // Intensity of the light motif's
// RGB at this vertex
} MotifEntry_t;

typedef struct {
    vector3 Pos;          // This vertex's 3D position in model-space
    int nMotifEntryCount; // Number of motif entries
                          // pointed to by pMotifEntry
    MotifEntry_t *pMotifEntry;     // Pointer to an array of
                                   // MotifEntry_t structures
} Vertex_t;
```

以上实现方法支持任意数量的运动，但实现起来很复杂。还有一个更容易实现的方法，不过要牺牲一些灵活性，而且要消耗较多内存。

```
#define MAX_MOTIFS_PER_VTX     8    // Implementation-specific value

typedef struct {
    vector3 Pos;      // This vertex's 3D position in model-space
    int nMotifEntryCount;      // Number of motif entries in
                               // aMotifEntry[]
    MotifEntry_t aMotifEntry[MAX_MOTIFS_PER_VTX];
} Vertex_t;
```

为了清晰起见，我们将采用最后这个实现方法，不过高级开发人员应该考虑使用更通用的实现方式。

要支持一个特定顶点被多个光影响，执行前面所描述的运动/衰减计算，然后将颜色相加。

```
void ComputeVertexColor( const Vertex_t *pV, Color_t *pColor ) {
    Color_t *pMotifColor;
    float fR, fG, fB;
    int i;

    // Zero color components:
    fR = fG = fB = 0.0f;
```

```
    // Step through all the motifs affecting this vertex
    //   and sum their colors:
    for( i=0; i<pV->nMotifEntryCount; i++ ) {
        pMotifColor = &aMotifTable[
        pV->aMotifEntry[i].nMotifIndex ];
        fR += pMotifColor->fR * pV->aMotifEntry[i].fIntensity;
        fG += pMotifColor->fG * pV->aMotifEntry[i].fIntensity;
        fB += pMotifColor->fB * pV->aMotifEntry[i].fIntensity;
    }

    // Make sure final color is from 0 to 1:
    if( fR > 1.0f ) fR = 1.0f;
    if( fG > 1.0f ) fG = 1.0f;
    if( fB > 1.0f ) fB = 1.0f;

    // Store final colors in return variable:
    pColor->fR = fR;
    pColor->fG = fG;
    pColor->fB = fB;
}

void DrawMyTriangle( const Vertex_t *pV1, const Vertex_t *pV2,
        const Vertex_t *pV3 ) {
    Color_t Color;

    glBegin( GL_TRIANGLES );

    ComputeVertexColor( pV1, &Color );
    glColor3f( Color.fR, Color.fG, Color.fB );
    glVertex3f( pV1->Pos.x, pV1->Pos.y, pV1->Pos.z );

    ComputeVertexColor( pV2, &Color );
    glColor3f( Color.fR, Color.fG, Color.fB );
    glVertex3f( pV2->Pos.x, pV2->Pos.y, pV2->Pos.z );

    ComputeVertexColor( pV3, &Color );
    glColor3f( Color.fR, Color.fG, Color.fB );
    glVertex3f( pV3->Pos.x, pV3->Pos.y, pV3->Pos.z );

    glEnd();
}
```

5.2.3 结论

基于运动的静态光照，性能略低于传统静态光照而远高于动态光照，通过提供 RGB 动画给预计算的光照数据，能给我们带来生动的场景。无须使用昂贵的动态光照，篝火、摇曳的火把、电火花、灯塔……都可以实现了！

5.3 使用定点颜色插值模拟实时光照

Jorge Freitas

实时光照效果是当今 3D 游戏体验的重要组成部分，但是对于资源有限的系统来说，它的计算量非常大。在这些系统上，可以通过在预计算顶点颜色的集合之间插值来模拟实时光照效果。

这项技术最先应用于体育游戏中的人物。每一帧需要绘制 23 个人，都要使用多顶点权重的皮肤模型、骨骼层次。我们的基本目标是保持实时光照效果的同时，消除渲染流水线中耗时的光照计算。

全场的光照由阳光（或 4 个静态照明灯）和环境光组成。当人物跑到阴暗的地方时，环境光照值用于将人物变暗。还有一些特殊的光照效果：草的颜色会反射到球员的袜子和短裤上；手臂和腿的下方会有阴影；基于当时的天气情况和时间，会有不同的光照颜色。此外，对于夜间比赛，光照的最亮点应该位于最接近照明灯的地方。

需要找到一种低开销的方法计算人物身上变化的光照。预计算就是答案。根据光照，预先计算出尽可能多的信息。理想情况下，人物的每一个姿势及转动的光照都可以预先计算，但这并不实际，因为要存储这些信息，占用的内存是巨大的。

我们采用了一种替代的方法，预先计算出几个固定的转动，然后使用插值来生成所需人物转动的顶点颜色值。

5.3.1 光照方法

一般情况下，生成实时顶点光照的方法如下：

（1）转换对象上每个顶点的法线。

（2）确定光源面向顶点法线的角度。

（3）使用该角度，确定顶点上的光照强度。

（4）对照明该顶点的每个光源重复上述步骤。

（5）添加环境光强度。

使用以上方法的实时光照需要大量计算，使用插值法可以极大地降低计算量。

为了仿造实时光照，对 3D 场景需要作如下限制：

- 确定需要多少个光源位置用于插值。
- 确定插值计算所采用的旋转轴。

顶点颜色列表用于表示在场景光照中，人物在不同朝向的光照，不管

场景光照是来自一个还是多个光源。360° 的旋转效果都需要有，因而我们得决定使用多少个顶点颜色列表。至少需要三个位置，每个相差 120° 。请注意，增加顶点颜色列表可以减小插值的幅度，因而可以产生更真实的仿造光照。

5.3.2 美工创作

我们假定顶点颜色列表都是由一个 3D 内容创作应用程序生成的。当然，可以使用实时光照方法来生成顶点颜色列表，不过，对 3D 场景进行完美光照的美术任务还是应该交给美工吧？

在我们的人物例子中，用于确定光照角度的旋转轴假定为垂直方向（右手坐标系统中的 Y 轴），因为人物总是在平面（草地）上跑动。

程序清单 5.3.1 中的例子使用了 4 个顶点颜色列表，每个间隔 90° 。可使用二进制运算进一步提高整数的运算效率；可以使用移位来代替乘除，使用 AND 代替 MOD。不过为了清晰起见，示例代码中并没有使用。

根据比赛场上的灯光使用，将光源添加到 3D 场景中。可以使用任意数量的光源，因为结果数据是顶点颜色列表，所以不会对性能产生影响。

假定第一个顶点颜色列表代表人物旋转 0° 时的光照。将人物在 4 个旋转位置上打光，每个位置距前一个位置旋转 90° 。每个旋转位置的光照存储为一个顶点颜色列表，这 4 个顶点颜色列表将在实时计算时使用。

顶点颜色可由美工在 3D 创作应用程序中调整，以生成一些特殊效果。绿色可混合到腿的下部，以表示草的反射；在人物双腿间和胳膊下面可加黑，以产生阴影效果；可给人物的皮肤加上底色以表现不同肤色的人。

5.3.3 插值光照

以下讲解生成插值顶点光照的必要步骤。

1. 计算面对虚拟光源的角度

顶点颜色列表表示人物相对光源不同朝向的光照。光源的位置可以作为一个简单的添加值存储，表示在场景中的光照方向。面对角可以这样计算得出：将该偏移量加上人物的旋转角度，再对一次全旋转的度数（360）取余。

面对角＝（对象旋转角度＋光源位置）％ 360

请注意，通过让光源位置值在 0 到全旋转度数循环，可让光源绕着对象旋转。

2. 决定在哪两个顶点光照集合之间插值

基于面对角，必须决定表中的哪两个顶点颜色集合将用于插值（基本颜色列表和目标颜色列表）。在我们的例子中，每个顶点颜色集合都是预先生成的，采用 4 等分，每个间隔 90° 。将当前旋转角度除以每个顶点颜色集合之间的 4 等分值，得到基本颜色列表。目标颜色列表

就是表中的下一个顶点颜色列表。

```
基本颜色列表=面对角 / 4等分顶点颜色
if （基本颜色列表==前一个顶点颜色列表）
   目标颜色列表=第一个顶点颜色列表
else
目标颜色列表=下一个顶点颜色列表
```

3. 计算插值百分比

需要确定在两个顶点颜色列表间插值的位置。百分比数表示插值距离目标颜色列表的远近。该百分比通过将基本颜色列表取余，再比上4等分顶点颜色得出，其值在0.0和1.0之间。

百分比=（面对角 % 4等分顶点颜色）/4等分顶点颜色

4. 对每个顶点颜色进行插值

现在，我们计算表示人物当前光照的顶点颜色列表。要计算每个顶点颜色，每个颜色分量（RGB）都需要进行插值。我们对人物的每个顶点都重复此过程。

颜色=旧颜色+（新颜色-旧颜色）* 100%

5. 对每个顶点颜色，应用环境光RGB修改器并限定结果取值范围

可选的，可以应用环境光修改器来计算顶点颜色列表。修改器算出的颜色可能超过了可接受的颜色范围，即每个颜色分量都在0~255之间，因而需要对每个颜色分量单独进行限定。

```
颜色分量=颜色分量+环境光
if （颜色分量<0）
  颜色分量=0
else
{
   if （颜色分量>255）
      颜色分量=255
}
```

请注意，可以通过预先计算顶点颜色列表，使其确保在进行环境光修改后得出的顶点颜色不会超过可接受的范围。通过限制颜色分量的最小值和最大值，并限制环境光修改量的大小，就不需要限定程序了。

5.3.4 结论

这项技术能有效地减少表现实时光照所需的计算数量，已被用于PC和游戏控制台。生成一个顶点颜色只需进行3次减法、3次乘法和3次加法（如果使用环境光则为6次加法）

运算。即使你的 3D 游戏需要实时光照，也可以将插值顶点颜色光照结合实时光照，以减轻渲染场景时的计算负担。

如果要进一步减少计算量，可以不必在每帧渲染的时候都进行光照计算。通过为每个对象保留一个独立的顶点颜色缓冲区，就可以基于对象相对光源朝向的改变程度，每 2、3 或 4 帧进行一次光照计算。还可以将每帧都需要重新进行光照的物体，使用用较简单的表示方式来替换，以进一步减小计算量。

虽然此方法很简单，但是使用它可以预先计算非常复杂的数学模型。我们追求的目标是在 3D 游戏中产生实时光照的视觉效果。游戏玩家只关心最终的效果，而不会在乎使用的是什么数学方法。

程序清单 5.3.1：示例代码

在本示例代码中，假定人物每个顶点都有一个顶点颜色。请注意，数据结构中仅包含了顶点颜色插值所必要的信息。可以使用移位运算和 AND 代替除法运算和 MOD。为了清晰起见，此代码中并没有使用。

```
//
//------------
// defines
//------------
//
/* number of vertex color lists */
#define NUMBER_OF_ARGB_LISTS  4    /* number of radians in 360 degrees */
#define NUMBER_OF_RADIANS 1024
//
//---------------
// structures
//---------------
//
typedef struct
{
    float    alpha;
    float    red;
    float    green;
    float    blue;
}ARGB_DEF;

typedef struct
{
    /* angle used for simulated lighting */
    int      angleOfRotation;
    /* number of vertices in object */
    int      nVertex;
    /* pointers to vertex color lists */
    ARGB_DEF *pARGB[ NUMBER_OF_ARGB_LISTS ];
}OBJECT_DEF;
```

```
//
//--------------
// variables
//--------------
//
/* pointer to buffer used to store the calculated RGB's */
ARGB_DEF *gpVertexColorBuffer; /* global additive ambient light RGB */
ARGB_DEF gAmbientLight;   /* additive value used to offset the light "hot spot" */
int gLightOffset;
//
//---------------
// functions
//---------------
//
/***************************

Function : interpolateVertexRGBs
Linearly interpolates between two lists of vertex colors.

Input:
ARGB_DEF *pSrcA   - pointer to first source vertex color list
ARGB_DEF *pSrcB   - pointer to second source vertex
                    color list
ARGB_DEF *pDest   - pointer to storage for calculated
                    vertex colors
int      nARGB            - number of vertex colors to interpolate
float        percentage   - amount to interpolate between two vertex
                            tables (0.0 - 1.0)

Output:
Fills pDest with the calculated vertex colors

***************************/
void interpolateVertexRGBs(
    ARGB_DEF *pSrcA,/* pointer to source vertex color data */
    ARGB_DEF *pSrcB,/* pointer to source vertex color data */
    ARGB_DEF *pDest,/* pointer to dest vertex color data */
    int nARGB,            /* number of vertex colors to interpolate */
    float percentage )    /* interpolation amount, 0 to 1 */
{
        int  index;       /* index into arrays of ARGB_DEF's */
        float    red,     /* temporary storage for calculated RGB's */
          green,
          blue;

    for ( index = 0; index < nARGB; index++ )
    {
    //
    // calculate interpolated ARGB
    //
```

```
            red   =       pSrcA[ index ].red + ( pSrcB[ index ].red -
                          pSrcA[ index ].red ) * percentage;
            green =       pSrcA[ index ].green + ( pSrcB[ index ].green -
                          pSrcA[ index ].green ) * percentage;
            blue  =       pSrcA[ index ].blue + ( pSrcB[ index ].blue -
                          pSrcA[ index ].blue ) * percentage;
    //
    // add ambient light
    //
            red   += gAmbientLight.red;
            green += gAmbientLight.green;
            blue  += gAmbientLight.blue;
    //
    // clamp RGB's
    //
            if ( red > 255.0 )
                red = 255.0;
            else
            {
                if ( red < 0.0 )
                    red = 0.0;
            }

            if ( green > 255.0 )
                green = 255.0;
            else
            {
                if ( green < 0.0 )
                    green = 0.0;
            }

            if ( blue > 255.0 )
                blue = 255.0;
            else
            {
                if ( blue < 0.0 )
                    blue = 0.0;
            }
    //
    // store results
    //
            pDest[ index ].red = red;
            pDest[ index ].green = green;
            pDest[ index ].blue = blue;
        }
    }

    /*******************************

    Function : calculateSimulatedLighting
```

```
    Calculates the vertex colors used to represent the lighting at the given angle of
rotation.

    Input:
    OBJECT_DEF   *pObject - pointer to object structure

    ******************************/
    void calculateSimulatedLighting( OBJECT_DEF *pObject )
    {
            float        percentage;
            int      quadrant;
            int      angleOfRotation;

            ARGB_DEF *pSrcA,
                *pSrcB;

        angleOfRotation = ( pObject->angleOfRotation +
        gLightOffset ) % NUMBER_OF_RADIANS;

        percentage = (float)(angleOfRotation %
        NUMBER_OF_ARGB_LISTS ) /
        (float)( NUMBER_OF_RADIANS / NUMBER_OF_ARGB_LISTS );

        quadrant = angleOfRotation / ( NUMBER_OF_RADIANS /
        NUMBER_OF_ARGB_LISTS );

        pSrcA = pObject->pARGB[ quadrant ];

        if  ( quadrant == (NUMBER_OF_ARGB_LISTS - 1 ) )
            pSrcB = pObject->pARGB[ 0 ];
        else
            pSrcB = pObject->pARGB[ quadrant + 1 ];

        interpolateVertexRGBs( pSrcA, pSrcB,
        gpVertexColorBuffer,
        pObject->nVertex, percentage );
    }
```

5.4 衰减图

Sim Dietrich

人们所熟悉的顶点光照适用于很多应用。它有许多优点，比如可以正确地处理背对光的表面。但如果三角形相对于点光源光照范围很大的话，就会产生不自然的效果。

光照图（light map）是另一种计算光照的方法，它可以避免三角形出现棋盘格子形的不自然效果，但是需要耗费大量的 CPU 运算以更新动态光照，而且上载到视频卡也很慢。然而对于静态光照和阴影，光照图仍不失为一种很好的解决方法。

本文介绍一种新的技术，叫作衰减图（attenuation map）。使用多纹理操作，该技术能用于实现二次衰减的动态点光源光照。此外，该技术还可用于球形、椭球形、柱形、方形光照，或 CSG 运算，精确到像素级别，不用使用模板缓冲区。

5.4.1 讲解

用于光照的常用衰减方程如下：

```
X = lightPosition.X - vertexPosition.X;
Y = lightPosition.Y - vertexPosition.Y;
Z = lightPosition.Z - vertexPosition.Z;

D = sqrt(X*X + Y*Y + Z*Z);

Att = 1 / (C0 + C1*D + C2*D*D);
```

基于我们的具体目的，假定只使用二次衰减，因而假定 *C0* 为 1，且 *C1* 为 0。

```
Att = 1 / (1 + C2*D*D);
```

3D 纹理可简单地对该方程编码。直接存储将 *X*、*Y*、*Z* 作为 3D 纹理坐标构成的方程。然后建立纹理坐标系，计算出相对光源位置的 *dX*、*dY*、*dZ*，使用纹理矩阵在光的范围内按照比例 1 放缩 *dx*，*dy*，*dz*，并通过缩放、平移使得纹理的中心对应坐标（0, 0, 0）。

此情况下纹理矩阵计算的等式如下：

```
S = ((Light.X - Vertex.X) / LightRange) / 2.0f) + 0.5f;
```

```
T = ((Light.Y - Vertex.Y) / LightRange) / 2.0f) + 0.5f;
R = ((Light.Z - Vertex.Z) / LightRange) / 2.0f) + 0.5f;
```

不过，3D 纹理并没有广泛使用，因而我们需要处理 2D 和 1D 纹理。即使可以用 3D 纹理，寻找一种减少纹理内存计算衰减的方法也是值得的。

那么首先，我们只有 2D 和 1D 纹理可用，也就是说不能只用单个纹理来计算方程组（因为单个纹理不可能同时有 *X*、*Y*、*Z* 三个坐标）。这就意味着需要将上述方程组拆分成两个或多个。

使用 2D 和 1D 纹理，怎么来表达衰减方程呢？让我们用 *X* 和 *Y* 表示 2D 纹理，*Z* 表示 1D 纹理。

如果 *X* 和 *Y* 为一个纹理，*Z* 为另一个纹理，那么 *X* 和 *Y* 的函数 *f*(*X*, *Y*)以及 *Z* 的函数 *g*(*Z*)，都应表达为颜色。

换句话说，如果将函数作为两个纹理的合并来存储，则必须将最终的函数表示为两个颜色的和或积。颜色只能是 0～1 的正数，这将影响到我们所选取的衰减函数的形式。

前面的衰减函数如下：

Att = 1 / (1 + c2*D*D);

该函数的几个特殊点如下：

Att(0) == 1
Att(2) == 0.5
Att(Large D) approaches 0

衰减随着 D 的增大趋向 0。

让我们将函数针对两个纹理编码。首先，用其分量将 D*D 展开。

D = sqrt(X*X + Y*Y + Z*Z)
D*D = (X*X + Y*Y + Z*Z)

现在，用 *X*、*Y*、*Z* 来表示衰减方程。

Att = 1 / (1 + C2*(X*X + Y*Y + Z*Z))

到这里，我们好像被困住了。*X*、*Y*、*Z* 都在分母上，不可能将 Att 函数表达为 *f*(*X*, *Y*)和 *g*(*Z*)的和或积。必须找到一个可以拆分的方程。

使用倒数是不可能拆为两个颜色之和的，因而需要找到一个函数，效果相同，但不包含倒数。

大于 1 的数，其平方大于该数本身；而在（0..1）的数，其平方小于该数本身。例如，0.5*0.5 等于 0.25。这就是为什么我们在衰减函数中限制 *C0* 为 1——避免很近的光产生的亮度过高。

请记住，颜色的范围是[0..1]。这意味着 *f*(*X*, *Y*)和 *g*(*Z*)的结果必须也在范围[0..1]中。

以下方程既不使用倒数，产生的结果也在范围[0..1]中。

Att = 1 – D*D

Att = 1 – (X*X + Y*Y + Z*Z)

可以将(*X**X+*Y***Y*)作为 2D 纹理的 *f*(*X*, *Y*)，*Z***Z* 作为 1D 纹理的 *g*(*Z*)。图 5.4.1 和图 5.4.2 分别表示了这两种纹理。

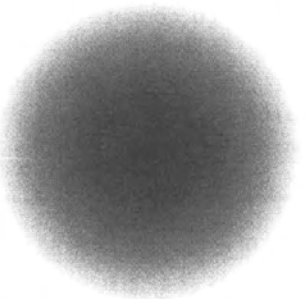


图 5.4.1 *f*(*X*, *Y*) = (*X***X* + *Y***Y*)

图 5.4.2 *g*(*Z*) = Z*Z

请注意图 5.4.1 的纹理边缘是如何正好达到 1 的。通过多重纹理操作将这两个函数加在一起，我们就可以逐像素计算出 D*D。使用逆混合，我们可以用 alpha 混合单元完成计算并算出 1−*D***D*。

使用衰减图实现点光源光照的过程如下：

```
Draw ambient light and/or global illumination in the scene

For each Point Light in the scene {
        For each Object that is approximately near or within the
        Light's range {
        For each Vertex in the Object {
        Subtract the Vertex Position from the Point Light
               Position
        Scale the Position by 1 / the Point Light's Range
        Scale and Bias the result to range from 0 to 1
               for points inside the Light's Range

        Store Position.X in the S texture coordinate of
               Texture 0
        Store Position.Y in the T texture coordinate of
               Texture 0
```

```
        Store Position.Z in the S texture coordinate of
              Texture 1
        }
        Set up the multitexture hardware to choose the Light Color
        in the color unit
        Set up the multitexture hardware to compute Texture 0 +
        Texture 1 in the alpha unit
        Set up the alpha-blender to compute SrcColor * InvSrcAlpha
        + FrameBuffer

        Optionally set the alpha test to reject pixels with an
        Alpha of 1. This avoids rendering pixels that are outside
        of the light range.

        Draw the Object
        }
}
```

也可以使用纹理坐标系和纹理矩阵来进行 GPU 上每个顶点的所有运算。

- 为第一层多纹理建立纹理坐标系，给出摄像机空间位置。
- 建立纹理矩阵，与光源位置的 *X*、*Y* 坐标相减，得出（*dX*, *dY*），存储在纹理 0 的 *S* 和 *T* 中。
- 使用纹理坐标系，建立第二层多纹理，同前，给出摄像机空间位置。
- 旋转 *Z* 到 *X* 轴上，建立纹理矩阵，减去光源的 *Z* 坐标，得到 *dZ*，存储在纹理 1 的 *S* 中。

这项技术可以修改以适应不同情况。对于图形硬件或 API（如 OpenGL）不允许区分颜色和 alpha 纹理混合的情况，可以将光颜色分解到自身的纹理中。这种修改也允许使用其他混合模式，如多帧缓冲区混合。

计算的衰减函数如下：

```
LightColor * (1- (X*X + Y*Y + Z*Z))
```

展开得到：

```
LightColor - LightColor * (X*X + Y*Y + Z*Z) =
LightColor - (LightColor * (X*X + Y*Y) +LightColor* (Z*Z))
```

这表明，需要预先将两个衰减图乘以光颜色以得到正确的效果，但实际只需一个衰减图。我们可以用 *LightColor*(X*X + Y*Y)*纹理中心来为 *LightColor*(Z*Z)*进行计算。取水平中心或垂直中心都可以，水平中心需要指定的纹理坐标较少而且可能纹理缓冲性能更高。

5.4.2 比较衰减图与光照图

光照图通常用于存储静态光照数据，如阴影和用全局照明方案计算出的光。在运行时为点光源更新光照图很复杂而且开销很大。用衰减图补充光照图，省去用动态点光源的麻烦，

可以得到很好的点光源性能。不用更新新光照图来反映点光源颜色、范围或位置的变化，只用对附近的场景简单地使用衰减图来混合该场景的光来渲染即可。

5.4.3 CSG 效果

通过使用 alpha 测试或模板（stencil），可以测试物体是否处于球形区域（如点光源的光照减弱范围）中。对于一个点光源光照需要绘制的每一个像素，可以将模板设为一个特定的值或将一个固定的颜色混合到帧缓冲区中，这样就可以处理其他的基于范围的效果。

5.4.4 基于范围的雾

此概念的一个应用是基于范围的逐像素雾。先简单地将场景渲染为没有雾时的样子，然后将摄像机位置作为“光源位置”应用衰减图来渲染场景。计算出纹理矩阵的单位矩阵，在光的范围中按照比例 1 缩放矩阵。这项技术能够生成正确透视效果的逐像素雾。

送到帧缓冲区混合单元的 SRC_COLOR 是雾的颜色与浓度的乘积。观察者位置的雾浓度为零，越远浓度越强在最大范围时雾浓度为 1。

渲染完雾以后，让 alpha 混合执行 SRC_COLOR * 1 + DST_COLOR * (1 - SRC_COLOR)。

5.4.5 其他形状

有时所需要使用的可能不是球形，如要使用类似长方形的形状（例如矩形或椭圆形），只用选定长、短两个轴并表示为纹理矩阵就可以。长轴和短轴需要分开缩放。

5.4.6 结论

明智地选择衰减函数，可以通过使用两个纹理图来执行逐像素的球形范围计算。计算结果可用于逐像素点光源光、雾以及 CSG 效果。

5.5 使用纹理坐标生成技术的高级纹理

Ryan Woodland

当今的图形处理器可以处理越来越多的多边形，人们也开始将注意力转向使用此能力来创建并行纹理效果。很多处理器都具备了额外的多纹理能力，人们开始思索如何有效地使用这些特性。当然，手工绘制纹理是我们最为熟知的，但我们很快发现，在运行时进行纹理映射可以产生一些有趣的效果。开发者开始使用纹理坐标生成（texture coordinate generation）技术执行动画、光照、反射、折射以及凹凸贴图等等。本文讨论了几个最常见的纹理坐标生成技术。

通过矩阵来变换数据（位置、法线、纹理坐标），是大多数人最乐意使用纹理坐标生成方法的地方。这种方法很容易被采用，因为大多数 3D 程序员都对矩阵变换很熟，而且矩阵变换也常常可由硬件加速。本文讲述的技术都是可以用矩阵运算执行的。

5.5.1 简单纹理坐标动画

游戏常使用简单的纹理坐标旋转或平移来模拟一些简单效果，如反射，或呈现水或一些运动介质。其思想就是：一个纹理坐标可以被简单地看作一个 2D 点。程序员习惯用矩阵来进行点变换，因而很容易理解，可以将纹理坐标转换成 3×3 矩阵来进行旋转、平移或缩放。就像几何体一样，必须将同质的坐标加到 s、t 对才能进行转换。因而，坐标生成方法如下：

$$(s, t, 1) * 3x3mtx = s', t'$$

图 5.5.1 中的演示图是使用旋转和缩放生成的。第一幅图是未经变换的纹理；第二幅是纹理坐标旋转 45° 的效果；第三幅是纹理坐标平移 0.5 的效果。

5.5.2 纹理投影

纹理投影（texture projection）对生成很多效果都很有用。它最常用于模拟如聚光灯或阴影的灯光效果。纹理投影的结果很直接：纹理从空间中的某个点投影到一些几何体上。例如，可以在场景中某点定义一个聚光灯，投影一个纹理（如圆形光）在几何体上，产生了聚光灯的照明效果。

再次重申，纹理投影的思想源于常规 3D 几何技术。当模拟摄像机时，

投影矩阵用于在摄像机空间投影顶点到摄像机的近裁剪面上。这些点的 X 和 Y 都映射到范围 −1~1 中，并且它们将通过视点变换转换到屏幕空间中（通常用到平移和缩放变换）。

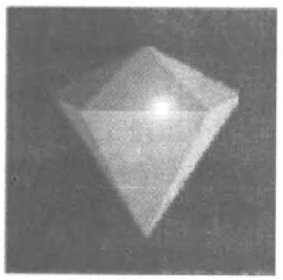
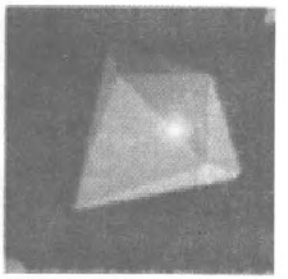
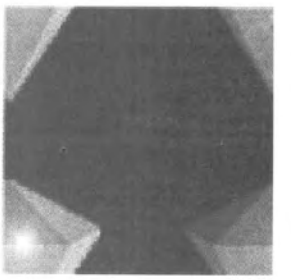




图 5.5.1

对于纹理投影，我们通常在空间中对光建模，而不是对摄像机建模。光空间的顶点投影到光的近裁剪面上，其 X 和 Y 值被用作 S 和 T 值，将纹理映射到投影几何体上。

对光建模和对摄像机建模一样。近裁剪面应该设定成反射投影纹理的形状。例如，方形的近裁剪面应该用于投影方形纹理。

正如所提到的，被投影在纹理上的几何体应该在光空间中，就如投影在屏幕上的几何体需要在摄像机空间中一样。要做到这一点，首先需要将几何体变换到世界空间中。然后，光矩阵（就如摄像机矩阵）必须转换该几何体。之后，几何体就可以通过光的投影矩阵投影了。

一旦几何体被投影，另一个问题就会产生。如前所述，投影几何体的 X 和 Y 都将落在−1~1 的范围中，原点（0, 0）为投影面相对光源的中心。纹理坐标的 S 和 T 通常都在 0~1 范围中，原点为纹理的左上角。要将投影坐标映射到纹理空间中，需要先以 0.5 的比例进行缩放，使取值范围变为−0.5~0.5；然后再平移 0.5，使范围变为 0~1。

这些矩阵可以集中在一起形成一个给定几何体的最终的投影矩阵。顺序如下：

*M_obj * M_light * M_proj * M_scale * M_trans **
{x, y, 0, z} = {s, t, r, q}

其中：

- *M_obj* = 对象的世界空间矩阵
- *M_ligh* = *t* 用于将几何体从世界空间转换到光空间的光矩阵
- *M_proj* = 光的投影矩阵
- *M_scale* = 以 0.5 的比例缩放矩阵
- *M_trans* = 平移 0.5 的矩阵

这个计算的结果是一个四维点。对于简单的纹理投影，r 坐标应该忽略，应该输出（s, t, q）三元组。如果硬件允许，可直接将这三个坐标传给光栅化处理。q 坐标用于执行透视矫正，只有在光栅化时执行才是正确的。

图 5.5.2 是使用纹理投影生成的。它展示了光的截面体，在球体上投影出了高光的圆。

以这种方式投影几何体，会产生一些意外的结果。首先，纹理坐标通常都是以平铺的方式使用的，这意味着范围是 0~1 的坐标与范围是−1~0 的坐标之间没有区别。然而投影在几何

体上的纹理则常常有限定范围，在该纹理边缘之外的应该为坐标小于 0 或大于 1 的任意纹理。因此，纹理边缘应该进行上色，以使得选择的纹理结合模式能够产生正确的效果。

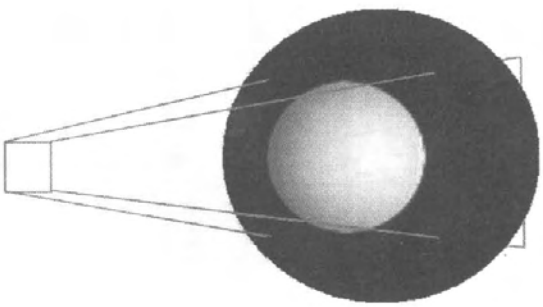


图 5.5.2 纹理投影示例

第二个问题更复杂，我把它叫做“照透（shine-through）”。例如，当我们投影纹理在球上时，球对着光的正、背两个面都出现纹理。这是因为球正面和背面上的顶点都投影在正确的纹理空间中。

图 5.5.3 展示了这个问题。可以看到，聚光灯纹理正确地投影在了球的正面上，但它也同时照透到了球的背面上。

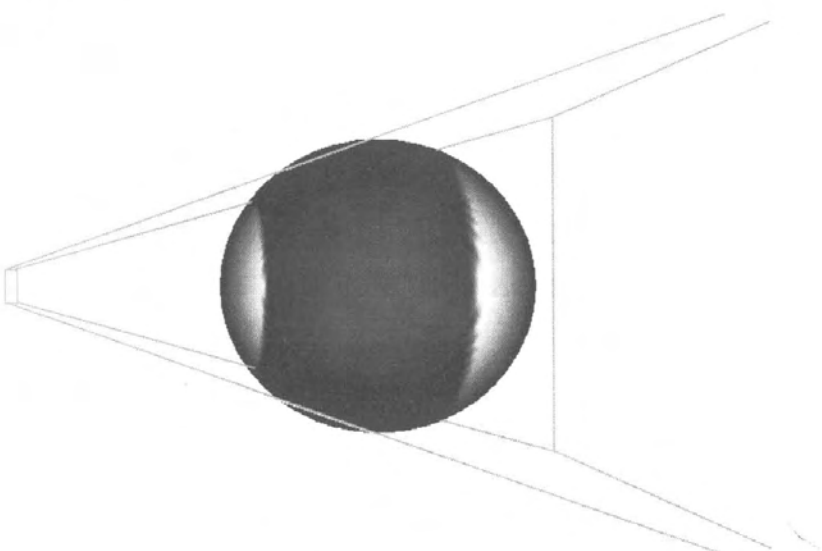


图 5.5.3 纹理投影中的照透

有两种方法可以修正这种照透问题。第一种是对顶点法线和光法线执行点积，以判断该顶点是否是背面。如果是背面，将该顶点的纹理坐标设为范围在 0~1 之外的纹理。

第二种是，可以使用标准光照方程的输出值来判断该顶点是否是背面。在纹理投影点的地方放置一个平行光，如果该平行光对于一个顶点的输出是黑色，你就知道这个顶点是背面（因为只有该顶点的法线没有对着光的时候它的输出才会是黑色）。

5.5.3 反射映射

要执行反射映射，我使用了一种简单的方法，叫做“球形映射（sphere mapping）”。该方法的基本思想有两个前提条件。

首先，假定不论什么形状和大小的物体，都像球形一样反射周围环境。这个假定很重要，因为，从逻辑上来说，人物手上的点与脚上相同法线的点，反射效果是不同的。使用球形映射，则这两个点的反射相同（因为它们法线相同）。

其次，假定物体被看作反射球是无限小的。这意味着，在场景中从视点到该无限小球上所有点的所有视线都相互平行。

通过以上的限制，球形映射方法符合基本反射法则。例如，在场景中有一条从视点看向球上某个点的视线，该视线射在球体上，绕着交点的法线被反射。不论什么情况，视线都应该在球上的交点处反射。图 5.5.4 为此概念的图示。

追踪球上的每个点弹出的射线是不可行的，因此我们创建了一个包含必要环境信息的纹理图，该图叫做球形反射图（spherical reflection map）或球形图（sphere map）。

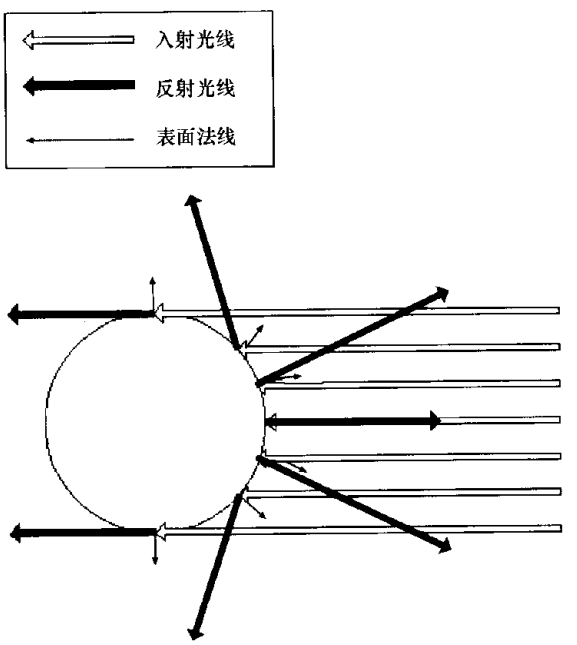


图 5.5.4 在可反射物体上，射线绕着交点上的表面法线被反射

球形图的基本定义就是，一个包含空间中某点 360° 全视角视图的纹理图。使用球形映射用于反射有一个很大的缺点——用作反射图的纹理是独立于视点的。这意味着，若要效果完全正确，每次摄像机移动时，纹理图必须动态创建。不过我发现，对一些常见效果，如在小汽车上生成特殊高光或创建人物的光照效果，不按照视点更新球形图常常不被觉察（要深入研究球形图生成，请参见[Blythe99]）。

一旦有了合适的球形图，纹理坐标生成就不算什么问题了。使用物体模型矩阵的逆变换，将物体的法线变换到世界空间。然后使用摄像机矩阵将法线变换到观察空间。最后，假设摄像机朝着 Z 轴反方向放置，分别用法线的 X 和 Y 作为 S 和 T 坐标，用于相应的顶点。

显然，使用这个技术，$-Z$ 坐标的法线与 $+Z$ 坐标的法线会生成相同的 S、T 对。这没有问题，因为法线 Z 坐标为负值的顶点是背面的点，不会被看到（这在观测空间中会进行计算）。

彩图 5 是使用一个室外环境的球形反射图映射到圆环上生成的。

使用该技术，很容易通过使用正确的纹理图，执行反射、镜面映射和漫射光照。

关于独立于视图的方法生成这些效果，请参见[Heidrich98]，它讲述了双抛物面映射。此外，如果目标硬件支持，立方体环境映射是个很好的方法生成动态光照和反射效果。要获取更多信息，请参见[Nvidia00]。

5.5.4 参考文献

[Blythe99] Blythe, David, *Advanced Graphics Programming Techniques Using OpenGL*, http://reality.sgi.com/blythe/sig99/advanced99/notes/node80.html, April 7, 2000.

[Watt92] Watt, Alan, and Watt, Mark, *Advanced Animation and Rendering Techniques*, ACM Press, 1992.

[Heidrich98] Heidrich, Wolfgang, and Seidel, Hans-Peter, *View-independent Environment Maps*, Eurographics/ACM Siggraph Workshop on Graphics Hardware 1998, www9.informatik.uni-erlangen.de/eng/research/rendering/envmap/, March 22 2000.

[Nvidia00] NVIDIA technical brief, *Perfect Reflections and Specular Lighting Effects with Cube Environment Mapping*, www.nvidia.com/Marketing/Developer/DevRel.nsf/WhitepapersFrame?OpenPage, March 10, 2000.

5.6 硬件凹凸贴图

Sim Dietrich

凹凸贴图，最早由 Jim Bilnn 在[Blinn78]中提出，该技术通过在光滑的多边形表面上应用纹理来模拟光照在粗糙或凹凸表面的效果。所使用的纹理叫做“凹凸图”，因而该技术得名“凹凸贴图”。

当今的硬件提供了一些凹凸贴图的方法。本文着重于阐明程序员在从“凹凸球”示例到实际的游戏实现中常常会遇到的问题。凹凸贴图实际上是光照计算，因而文中将使用一些照明的术语来讨论凹凸贴图。

凹凸贴图技术既可计算也可近似光向量 L 和表面法线 N 之间的点积，来计算漫反射光照。为了简单起见，假定点积运算是逐像素计算的。在本文写作的时候，一些硬件厂商已经在其硬件中提供了此功能，而且将来很可能继续保留此功能。

镜面光照可以通过计算半角向量 H 与表面法线 N 的点积，再对结果进行幂运算来实现。本文针对的是漫反射照明，但是请记住，文中讨论的所有技术也可同样用于镜面凹凸贴图或光照。

基于点积的凹凸贴图和逐像素光照都是复杂、庞大的主题，我们将着重解释手边的问题和一些实际的解决方案。本文不讨论纹理混合模式、立方体图、纹理格式等实现细节上的问题，而是集中讲述如何得到任意模型或网格的正确的凹凸图或光照。

5.6.1 如何将凹凸图应用于对象上

对此问题的简单回答是，使用代表凹凸图的纹理对对象进行纹理映射。要给出更有用的回答则需要知道关于具体的平台和待实现效果的更多信息。对于凹凸图纹理，不同的硬件凹凸贴图技术需要不同的源数据，从 alpha 高度图到 RGB 表面法线图到 RGB *dUdV* 图。因为 RGB 表面法线图和 RGB *dUdV* 图都可以从 alpha 高度图生成，所以假设凹凸图的原始源数据是 alpha（或灰度）高度图。

前面假定了我们可以进行逐像素点积运算，因而可以假设我们有一种方法生成合适的纹理格式，可以直接以 RGB 格式编码表面法线。这种格式将 X、Y、Z 向量分量从[−1..1]浮点范围映射到[1..255] RGB 色彩通道范围。

在 RGB 纹理中存储法线带来了一个有趣的问题：这些法线是在什么空间定义的？模型空间，世界空间，还是其他空间？

基本凹凸贴图和漫反射照明运算，$N \bullet L$，并不关心向量定义于哪个空间，但是必须保证 N 和 L 是用同一个坐标系表示的。

我们将看到，这个看起来无关紧要的问题，正是基于点积的凹凸贴图技术最重要的一个考虑因素。下面讲述各种选择以及它们用于什么地方最为合适。

5.6.2 为法线选择一个空间

让我们从这个简单的例子开始，有一个球体要在模型空间中进行凹凸映射。假设该凹凸图对球体是惟一化的纹理，也就是说，球上的每个点都有其对应的纹理部分，因而不需要平铺和镜像凹凸图。

在这种情况下，首先找出凹凸图中每个纹理像素在模型空间中球体上的对应位置，生成该位置的表面法线。下一步，在球体的该位置上生成一个 3×3 的坐标系，将表面法线作为 $+Z$ 轴。这个空间通常叫做“正切空间”，因为它表示的空间与表面正切。

要生成正切空间，需要两个向量，第三个向量可以由这两个生成。不过第二个向量的选择并不是惟一的，因为同一个 $+Z$ 轴可以对应无数个正切空间。

我们选择模型空间的 $+Y$ 轴作为第二个向量，然后对 $+Y$ 和表面法线叉积，得到第三个向量，作为 $+X$。还可以继续对 $+Z$ 和 $+X$ 叉积，再生成一个新的 $+Y$ 轴。对这三个向量规一化，得到表示球面上该点正切空间的 3×3 矩阵的三个列。

现在，我们得到了该点的基本矩阵，可以从凹凸图中取出凹凸向量（世界空间表示的），然后用该矩阵将其旋转到局部正切空间中。这样就得到了球面上特定点的凹凸向量。现在可以用正切空间的凹凸向量替换纹理中世界空间的凹凸向量。

在运行时，使用世界到模型空间转换矩阵，将世界空间中的光向量 L 旋转为 L'。由于 L' 是相对与模型空间的，因而可以计算光向量和凹凸图向量的点积 $L' \bullet N$。

L' 对于特定的模型空间矩阵是常量，因而它仅对当前的模型层可用。这很方便，因为 L' 可以对于进行凹凸贴图的整部分纹理都保持不变。

在运行时，建立纹理混合单元进行计算：

Texture • ConstantColor

其中 Texture 对应用 RGB 对法线编码的表面法线图；ConstantColor 对应常量 L' 向量，转换为 RGB 形式。该技术叫做对象空间或模型空间凹凸贴图。它的优点是，除了每个对象层一次单个向量旋转，运行时没有开销。它的缺点是，需要惟一化的对象纹理，这会占用大量的纹理内存。此外，对象的蒙皮或变形需要每帧都重新生成表面法线纹理。

为了克服上述缺点，引入下述技术。

5.6.3 另一种方法：使用正切空间凹凸贴图

我们已经为球体定义了正切空间，也就是说，我们有了从模型空间到每个顶点上定义的局部空间的转换。可以利用这个信息来避免进行惟一纹理化，以及避免模型运动时必须重新生成表面法线图。我们保留了法线图，没有重新生成对应于模型空间的法线图；但是生成了

一个矩阵，将光向量从模型空间旋转到正切空间。

假设原始的高度图表示的是“拔离该表面”的高度。在数学上，高度图“海拔”的方向对应于球面每个点上正切空间的+Z 轴。我们使用这个关系创建了一个数学方法，将凹凸图空间转换到球面每个点的局部正切空间。

一种方法是，完全按照前面所讲述的在模型空间中的凹凸贴图来计算正切空间，但是仅在球体的每个顶点进行计算，并将正切空间矩阵存储在对应其顶点的数据结构中。当对该顶点进行光照或凹凸贴图计算的时候，使用这个 3×3 正切空间矩阵，将光向量转换到局部正切空间中。不是像前面那样，使用正切空间矩阵转换表面法线；而是用该矩阵对光向量进行转换。记住，只要两个向量在同一个坐标系中，点积运算是在什么空间中进行的并没有什么关系。

在前面所讲述的模型空间凹凸贴图中，为了生成覆盖球体的惟一凹凸图，正切空间矩阵的生成是在球体每个点上进行的预处理步骤。在正切空间凹凸贴图中，只用在球体每个顶点上生成正切空间矩阵。

在运行时，将每个顶点的 L 向量进行旋转（先使用“世界到模型”矩阵，然后使用在预处理中生成的，存储在顶点中的局部正切空间矩阵），得到 L'，即局部正切空间中的光向量。当然，L'仅对特定的顶点是正确的。在模型空间凹凸贴图中，L对整个模型层是常量；现在 L'对于不同的顶点是不同的。

因而，对于之前作为常量颜色进行存储的 L'，现在需要通过三角形进行插值。

要执行正切空间凹凸贴图，可建立纹理混合单元执行。

Texture • DiffuseColor

其中 Texture 是表面法线图；DiffuseColor 是代表 L'的迭代颜色值。

我们利用硬件的颜色插值能力将 L'向量从一个空间“旋转”到另一个空间。实际上，我们在各顶点的 L'向量间执行的是线性插值，而没有真正地旋转或球形插值，不过这对于大多数的用途都能产生很好的结果。请注意，透视纠正（perspective-correct）颜色插值对这些情况非常有效。

在多重纹理硬件上，需要使用立方体图（cube map）或抛物面图（paraboloid map）对 L'插值，不过这已经超出了本文的讨论范围。我只想说，对 L'向量使用线性插值，当对接近大型三角形的局部光进行凹凸贴图时，会产生变暗的不自然效果。产生这个效果的原因是，当从三角形内部线性插值时，反面或接近反面的 L'向量被变短了。立方体图通过各项异性地（anisotropically）缩放法线图，可有助于正确处理光向量插值。

正切空间凹凸贴图相比模型空间凹凸贴图有很大的改进，凹凸图不必是惟一的，可以平铺在表面上。此外，对象蒙皮或变形只需对每个运动顶点重新生成 3×3 的正切空间矩阵，不必每帧重新创建表面法线图。

正切空间凹凸贴图比模型空间凹凸贴图的 CPU 效率要低，因为在运行时它需要将 L 旋转到局部正切空间中。它还需要在运行时更新模型数据，这可能产生 GPU 拖延或需要额外的数据拷贝。

正切空间凹凸映射还有个不太明显的缺点，当创建凹凸球例子的时候这个缺点可能看不出来，但在实际的游戏环境中进行凹凸映射的时候却可能产生问题。这就是，正切空间凹凸

贴图需要定义凹凸图的应用方式与对象表面之间的关系。

要生成局部正切空间矩阵，仅知道高度图的“海拔”方向对应于正切空间中的+Z是不够的。这与模型空间凹凸贴图例子中的问题一样。我们必须有两个向量来定义一个三维坐标系。当今（2000年左右）网上大多数的凹凸贴图例子都是采用任意的映射，简单地选取世界或模型空间坐标（如+Y）来生成正切空间矩阵。

以下是对该问题的描述。一个程序员和一个美工一起使用凹凸图对球体进行凹凸贴图。第二天，美工在模型上将凹凸图旋转了90°，并将其导入游戏。应该从上方照射下来的光现在变到侧边去了！程序员来了，检查该问题，将这个90°旋转考虑进来，更新正切空间矩阵，修正了问题。

该问题能够像这样进行修正，说明程序员没有在正切空间到纹理图空间的贴图中自动考虑美工提供的信息。

美工知道凹凸图打算如何应用，程序应该能遵从她的选择。凹凸图可以像其他任意纹理一样应用，可以拉伸、扭曲、投影等等。正切空间方法过于简单，不能考虑凹凸图实际应用的模型上的方式，并假设在凹凸图和所贴附的表面之间有简单的对应关系。正切空间凹凸映射需要知道纹理是如何应用的——平面、盒状、球形或柱形映射。

正是这种假设使得正切空间凹凸贴图难以应用在游戏中，尤其当游戏有现成的手绘图时，我们又不想重新纹理化以生成受限的凹凸图。

5.6.4 解决方案：纹理空间凹凸贴图

纹理空间凹凸贴图与正切空间凹凸贴图类似，主要的区别在于为每个顶点生成局部矩阵的方式不同。

纹理空间贴图在每个顶点生成的不是正切空间矩阵，而是创建一种矩阵，我称之为“纹理空间矩阵”，该矩阵考虑到了美工是如何将纹理应用于表面的。

要生成该矩阵，需要考察每个三角形以及它是如何与纹理进行映射的。它可以旋转、缩放S或T、翻转或投影。要考虑纹理的应用方式，我们所需知道的是纹理梯度（texture gradient）。纹理梯度有9个梯阶值，代表S和T到X、Y、Z轴的方向。

按如下方法计算纹理梯度。首先使用三角形的平面等式：

$$Ax + By + Cz + D = 0$$

可以在平面等式中使用任意3个不相关的变量。我使用X、S、T，而不是X、Y、Z。

$$Ax + Bs + Ct + D = 0$$

现在使用该等式计算相对于X的纹理梯度。

假设有两个不同的x和两个不同的s值，将其分别相减得到dx和ds。在此假设t为常量。

$$Ax1 + Bs1 + Ct + D = 0$$
$$Ax0 + Bs0 + Ct + D = 0$$
$$Ax0 + Bs0 + Ct + D = Ax1 + Bs1 + Ct + D$$

$$Ax1 + Bs1 + Ct + D - (Ax0 + Bs0 + Ct + D) = 0$$
$$Ax1 - Ax0 + Bs1 - Bs0 = 0$$
$$A(dX) + B(dS) = 0$$
$$A(dX) = -B(dS)$$
$$dX/dS = -B/A$$

X 相对 S 的梯度为 $-B/A$。我们得到了相对每个 S 变化，X 变化的度量。可同样计算得出 dS/dT。Y 和 Z 的梯度也可类似地生成。如果 A 为 0，可将梯度设为 0，意思是 S 相对 S 完全没有变化。

现在有了 6 个梯阶值，分别代表模型空间中 X、Y、Z 沿着纹理 S 和 T 方向的变化。

可以使用此信息导出 S、T、$S\times T$ 轴与模型空间的 X、Y、Z 之间的矩阵。

$$\begin{bmatrix} S.x & T.x & (S\times T).x \\ S.y & T.y & (S\times T).y \\ S.z & T.z & (S\times T).z \end{bmatrix}$$

在运行时，可以使用该模型的层次矩阵栈，将光向量移动到局部模型空间中。使用这个矩阵对每个顶点上的光向量进行转换，并将结果向量存储为漫反射或镜面反射迭代的颜色。现在光向量定义在了与法线图相同的空间中，可以对它们进行点积运算了。

5.6.5 纹理空间问题

以纹理为基础进行计算会产生一些问题。最显著的问题是关于翻转纹理（flip texture）。由于纹理没有“正反面”但是法线有，因而我们需要把三角形的面法线和纹理的 $S\times T$ 轴进行比较以检测这种情况。如果它们间的点积为负，就可以简单地翻转 $S\times T$ 轴。有时候美工在对称的对象上将部分纹理镜像。在这种情况下，边界两边的两个三角形的 $S\times T$ 轴相反。这时，程序员或美工必须复制顶点。

有时因为纹理坐标系不同，程序员不得不复制顶点。这种情形表明纹理应用到表面的方式是不连续的。解决这个问题的办法是为每一个三角形创建一个独立的纹理空间基础矩阵。然后找出被三角形的面共享的顶点——换句话说，共享 X、Y、Z 位置的顶点。现在，对每个顶点，取共享该顶点的每个三角形的 S、T、$S\times T$ 轴并相加。将结果 S、T、$S\times T$ 规一化。这和从面法线创建顶点法线类似，只是在此需要每次计算 3 个向量。这 3 个结果向量组成了一个新的“平均纹理空间”存储在模型的每个顶点中。

规一化结果列向量的代价是不能正确处理各向异性缩放的纹理。如果使用立方体图或其他方式规一化向量，就可以跳过规一化步骤而且能轻松处理各项异性缩放的纹理。

5.6.6 结论

已经介绍了基于点积的凹凸贴图和逐像素光照，随之带来了一些有趣的参数化问题。若

想将平面纹理解释为 3D 表面法线向量有两种方法：在模型空间凹凸贴图中，需要重新生成匹配模型的表面法线纹理；在正切或纹理空间凹凸贴图中，需要为每个顶点生成一个基础矩阵。

5.6.7 参考文献

以下是三个关于逐像素光照的重要参考资源。

[Blinn78] Blinn, James, "Simulation of wrinkled surfaces," *Computer Graphics (SIGGRAPH '78 Proceedings)*, vol. 12, no. 3, pp. 286-292, August 1978.

[Everitt00] Everitt, Cass, "High-Quality, Hardware-Accelerated Per-Pixel Illumination for Consumer Class OpenGL Hardware," www.r3.nu/~cass/thesis, May 2000.

[NVIDIA] NVIDIA's Developer Relations site at www.nvidia.com/Developer.nsf, May 2000.

5.7 底面阴影

Yossarian King

实现任意物体在其他物体或任意地形上的阴影投射是很困难的。然而，如果你在平坦底面上投射阴影，则有很简单的解决方法。显而易见的方法就是在角色的脚边绘制某种类型的子画面，该子画面可以在将角色轮廓渲染到离屏缓冲区的时候生成。使用这种方法对象和阴影间可能会错位。本文提出了一种简单的底面阴影技术，能够生成正确的物理投影效果，而且计算开销很小。

在此技术中使用了一个额外的变换矩阵，用于将对象的顶点“压扁”到一个任意高度的水平表面上。阴影投影矩阵由光源相对对象的方向和高度决定。有了这个矩阵，对象就可通过这个变换渲染出阴影。

用半透明的灰色多边形代替对象的纹理多边形产生阴影效果。用于生成阴影的对象不必与原始对象一模一样，通常可以使用较少数量的多边形来表示对象以提高效率。阴影多边形投影到底面时会重叠，如果用半透明效果渲染就会产生不自然的重叠效果。可以使用硬件 Z 缓冲消除重叠效果。

本技术不仅可用于坐标平面，还可以推广到 3D 平面阴影投影。使用两次，本技术还可以在墙角投射阴影。

5.7.1 阴影数学

当光源和表面之间有固体对象的时候就会产生阴影。对于表面上的某个点，如果光到该点的光线与对象相交，则该点处于阴影中。在渲染系统中，对象是由一个多边形集合表示的。从光源射出的光线，通过模型的每个顶点，投射到表面上，将勾勒出该表面上的阴影区域。对于该对象的一个多边形，此方法将得出在表面上的一个多边形阴影投影。

不幸的是，此方法并不能直接用于实际的实时渲染。计算光线与表面的交集需要很大开销。投影的阴影多边形需要针对投影表面做进一步的分割，这项操作的复杂性是不确定的，依赖于表面的复杂程度。渲染投影多边形阴影有一种替换办法，那就是修改表面的顶点颜色，让在阴影中的顶点颜色变暗。但是这样做的缺点是，降低了阴影的精确度而且使得阴影依赖于表面的光强度。相对于顶点着色，动态阴影图纹理可以以更丰富的细节和更统一的方式进行着色计算，但其开销也很大。

如果阴影只需投射在水平底面上，则问题可得以简化。对于很多游戏应用，如模拟体育运动或走廊（corridor）游戏，水平表面都是其基准，因

而可以使用这种简化。

图 5.7.1 以 2D 图表示了底面投影。从 *L* 射出的光线将 *P* 点的阴影投在底面（*X* 轴）的 *S* 点上。我们想要得到 *S* 点的 *X* 坐标值 *sx*。

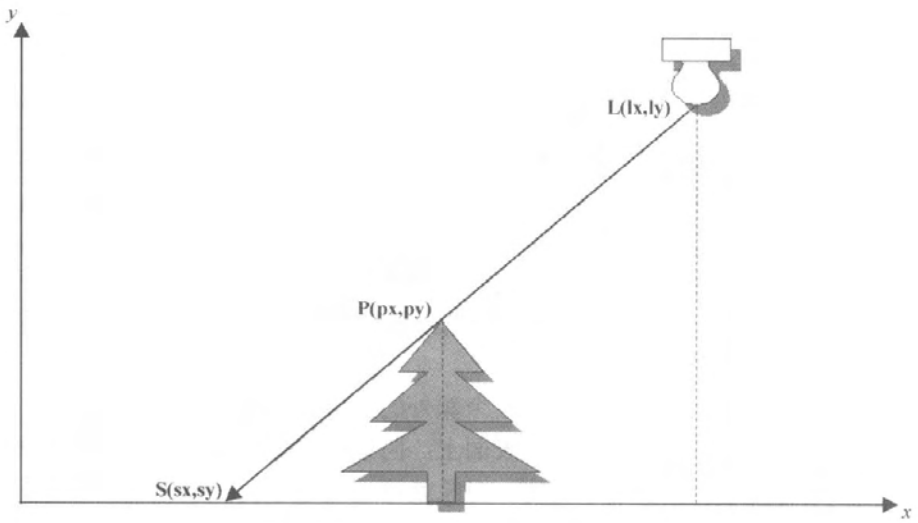


图 5.7.1 从 *L* 射出的光线将 *P* 点的阴影投在底面（*X* 轴）的 *S* 点上

使用相似三角形，得到：

$$\frac{(sx-px)}{(sx-lx)}=\frac{py}{ly}$$

求解 *sx*：

$$sx=px+py*\frac{(sx-lx)}{ly}$$

表达式右边还有 *sx*，并没有实际求得解。不过，在表达式右边惟一包含 *sx* 的是（*sx*−*lx*），它是阴影点到光源的水平距离。假设相对于阴影的长度，光源到对象的距离非常远，可以将（*sx*−*lx*）近似为（*ox*−*lx*）（对象到光源的距离）。代入得到：

$$sx=px+py*\frac{(ox-lx)}{ly}$$

或：

$$sx=px+k*py$$

其中

$$k=\frac{(ox-lx)}{ly}$$

远光源的假设等同于将光看作方向光（direction light）而不是点光源。

扩展到三维并将阴影点 S 投影到 xz 平面，得到：

$$\begin{aligned} sx &= px + k1 * py \\ sy &= 0 \\ sz &= pz + k2 * py \end{aligned} \tag{5.7.1}$$

其中

$$k1 = \frac{(ox - lx)}{ly}$$

$$k2 = \frac{(oz - lz)}{ly}$$

通过将对象的每个顶点投影到底面上，我们得到了一个投影多边形集合，从而得到该对象的阴影。进行投影，仅需计算 $k1$ 和 $k2$ 的值，这两个值完全由对象和光源的相对位置决定。下一节将进一步讲述此技术的实现。

5.7.2 实现

在典型的渲染流程（rendering pipeline）中，模型顶点 m 先变换到观察空间中的点 v，再投影到屏幕空间中。此操作可写成：

$$v = C * T * m$$

其中 T 是模型变换的矩阵表示，它将模型顶点 m 转换到世界空间中；C 是摄像机变换，它将世界空间坐标转换到观察空间。要渲染一个对象，先要将它每个顶点变换到观察空间，再投影到屏幕空间并渲染结果多边形。

将阴影投影过程简单地加入到渲染流程中，方程 5.7.1 可以矩阵形式写为：

$$s = S * P$$

其中：

$s = (sx, sy, sz, 1)$ 是世界空间中的投影阴影点。

$p = (px, py, pz, 1)$ 是世界空间中对象上的点。

S 是阴影投影矩阵，可写为：

$$S = \begin{bmatrix} 1 & k1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & k2 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

现在，要渲染一个对象的阴影，只用简单地将阴影投影矩阵插入到顶点变换中：

$$v_s = C * S * T * m$$

该表达式使用阴影投影矩阵将对象上的点变换到底面上的投影阴影点。阴影投影矩阵 S 仅依赖于常量 $k1$ 和 $k2$，这两个值由对象和光源的相对位置决定。

这个技术可用于渲染对象扁平地投影在地面上。如果阴影矩阵 S 简单地插入流程中并渲染对象，产生的结果是对象的“扁平版本”——带有纹理、着色等等。要将对象渲染为阴影，可以将每个顶点作为平的半透明灰色来渲染，忽略模型中的纹理信息。在实践中，使用比原对象少的多边形可以降低渲染阴影的开销。

使用这个方法有个问题，投影在底面上的多边形会重叠，如果重叠多边形使用半透明色绘制，则重叠的地方要比没重叠的地方黑。可使用硬件 Z 缓冲消影——第一个阴影多边形绘制完后，后面多边形与它重叠的部分从 Z 缓冲中删除。如果 Z 缓冲不精确导致不自然的效果，那么当阴影多边形被渲染的时候，每一个多边形在 Z 中的深度都会更深一些，因而 Z 比较能够消除重叠区域。

图 5.7.2 展示了一个渲染有阴影的茶壶使用 Z 缓冲消影前后的效果。

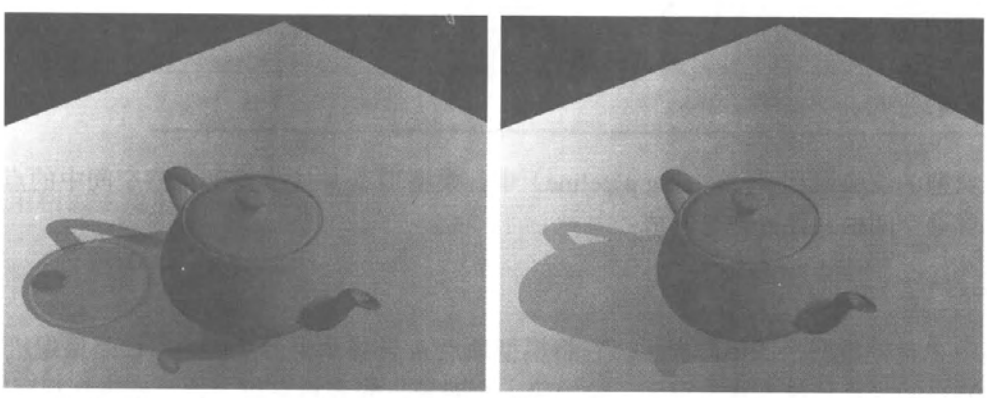


图 5.7.2 将茶壶几何体投影在底面上渲染茶壶对象的阴影。使用硬件 Z 缓冲消除了半透明多边形重叠的现象，如右图所示

5.7.3 扩展

如上所述，本文提出的方法可以方便地在水平底面上投影阴影几何体。本方法还可以很容易地扩展到其他对齐于坐标轴的竖直平面。再做一些改进，它还可扩展到任意平面。任意平面可看作是水平底面经旋转、平移后的平面。在任意平面投影阴影可以这样做：将平面旋转、平移到水平位置，投影阴影，再将平面变换回原来的位置。光源也必须进行变换。通过多次使用投影技术，可模拟阴影投射在多重平面上，如墙和地板之间的角落。

5.8 复杂对象上的实时阴影

Gabor Nagy

本文提出了一种高效算法，可在实时应用中创建真实感阴影。该算法利用了现在的高速纹理贴图和 3D 变换硬件。

5.8.1 介绍

阴影是人类视觉感知深度最重要的一个线索。在计算机图形中，它们能给图像带来最终的真实感。没有阴影，即使使用真实感光照和纹理效果，计算机生成的图像看上去也不能给人真实感。即使对象在一个平面上，它们看上去也像是漂浮在空间中。当摄像机不移动时（没有视差信息），这种不能让人在计算机图形中感知相对位置和深度的缺陷显得尤为明显。

直到现在，只有大计算量的算法（如光线跟踪和辐射度）才能产生准确的阴影，其中不论投射阴影的对象还是受到投影的对象，其复杂度都是不确定的。

本文提出的算法对实时应用进行了优化。它在真实感和渲染性能之间做了很好的权衡，同时易于扩展到所有的情况。考虑到游戏程序员总是“渴求着性能”的需要，本文着重指出可以使用硬件特性优化性能的地方。

本文的一些基本思想在很多文献中都已出现，但是我们提供了它们一般都没有讲述的重要实现细节。

5.8.2 光源、遮挡物体和接收物体

思考图 5.8.1 中的简单示例。圆环（遮挡物体或遮挡物）遮挡了一些从光源而来的光，并在墙上投下阴影。墙接收阴影，或“缺少光”，因而叫做接收物体或接收物。

如果光源是点光源（无限小），遮挡物体将光源的光遮挡在一个轮廓清晰的体积之外，这个体通常叫做“阴影体”（见图 5.8.2）。在接收物体上，其表面与阴影体相交的地方产生阴影。如图 5.8.2 所示，阴影体的形状是一个截去了头的圆锥，从遮挡物体开始到无限远。阴影体确实始于遮挡物体的轮廓处，圆锥的顶点则是光源。

让我们来查看一下，随着光源变远，阴影体的横截面是如何变化的。将遮挡物体表面离光源最近的点叫做 P_n，最远的点叫做 P_f。

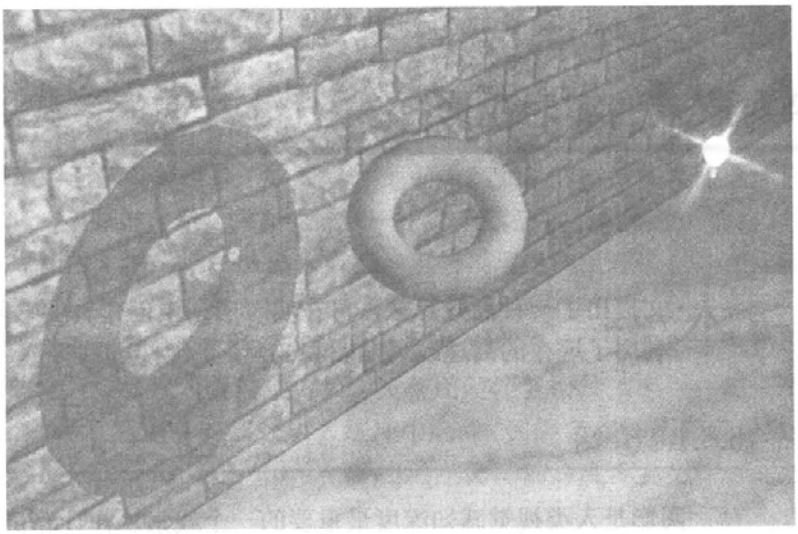


图 5.8.1 阴影、接收物、遮挡物和光源

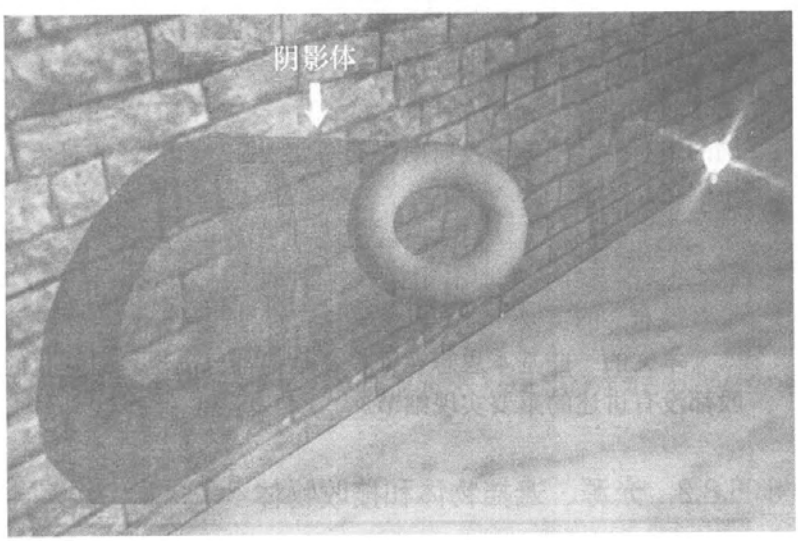


图 5.8.2 阴影体

可将阴影体分成 3 个区域：

（1）在光源和 P_n 之间的部分；

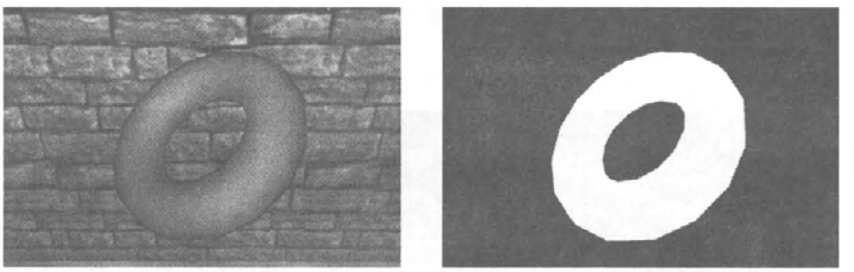
（2）在 P_n 和 P_f 之间的部分；

（3）从 P_f 到无限远的部分。

很容易看出，在区域 3 中，阴影体横截面形状是固定的，离光源越远尺寸越大。

因为有这个现象，所以除非在区域 2 中有接收物体，否则可以通过使用一个二维遮光板，让光源对其投影，得到阴影体的精确建模。然后，使用同一个投影，可以将这个 2D 遮光形状映射到接收物体上得到阴影区域。这个 2D 遮光形状叫做阴影图，它可以简单地通过绘制从光源看向遮挡物体的轮廓图得到。

请注意，在图 5.8.3a 中看不到圆环的阴影投射，因为圆环正好挡住了它。这正好表明，我们可以简单地使用合适的投影 2D 图像或遮光形状（见图 5.8.3b）来定义阴影体。这个方法通常叫做投影阴影映射。



(a) (b)

图 5.8.3 a 从光源看到的遮挡物体；b 遮挡物体的轮廓

5.8.3 本文的目的

使用上面介绍的方法绘制阴影，需要做以下事项：

（1）为每个光源/遮挡物对创建阴影图。

（2）计算用于接收物体顶点上的阴影图（纹理）坐标。

（3）将阴影图作为 2D 纹理使用来渲染接收物体。

5.8.4 创建阴影图

渲染阴影图的第一步是建立一个从光源出发的透视投影。这个投影将遮挡物体投影到光源和遮挡物体之间的虚拟屏幕平面上，参见图 5.8.4。

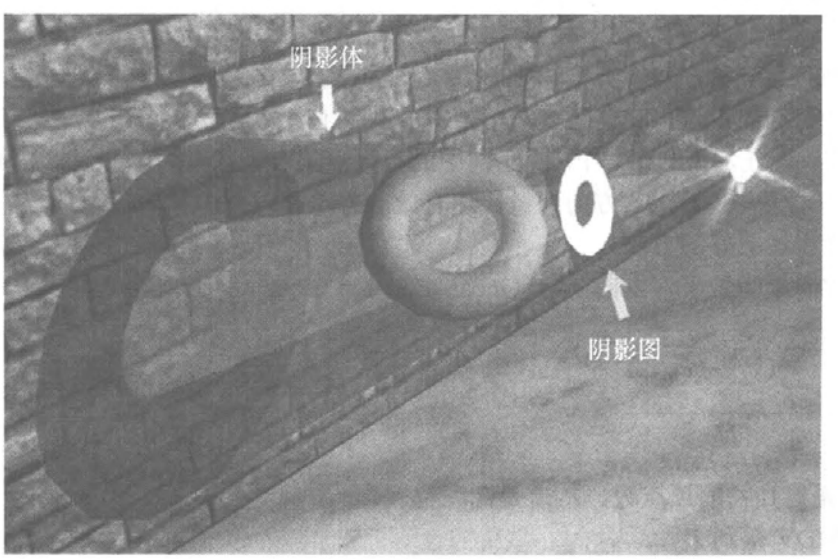


图 5.8.4 阴影图投影（同彩图 6）

1. 光坐标系统

首先，让我们定义一个新的坐标系统，原点位于光源处，Z 轴指向遮挡物体。这个坐标系统的 Z 轴决定了透视投影的中线；XY 平面定义了屏幕平面的方向，我们在该平面上投影阴影图。如果将遮挡物体变换到这个光坐标空间（如图 5.8.5 所示），就可以简单地将它投影到该平面上。

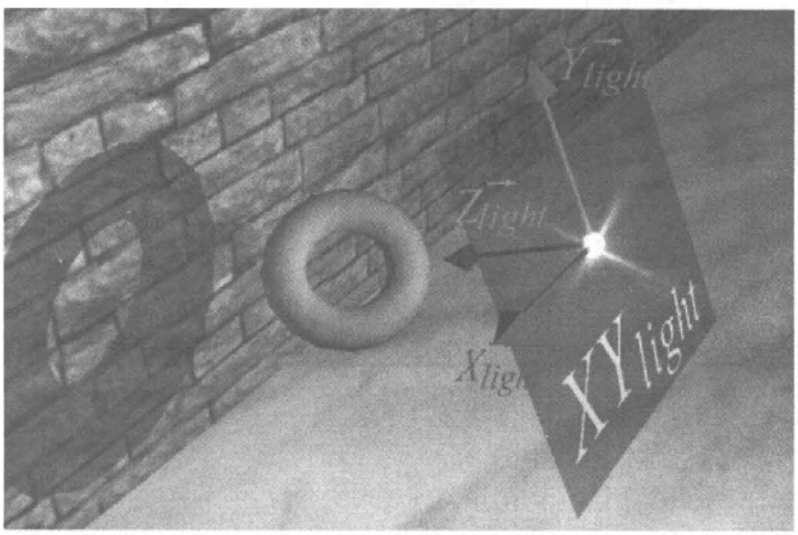


图 5.8.5 光坐标系统（同彩图 7）

要定义一个坐标系统，需要知道原点的位置和坐标取向。原点位置已经知道了——光源的位置。光坐标系统的取向则可以用它三个坐标轴的方向描述：X_{light}、Y_{light} 和 Z_{light}（都是世界坐标中的 3D 单位向量）。

2. 寻找 Z_{light}

从 Z_{light} 轴开始，我们可以很容易地找到 X_{light} 和 Y_{light}。Z_{light} 是一个方向向量，从光源指向遮挡物体。让我们假设遮挡物体是多边形，并且我们有一个包含所有用于渲染该物体的多边形顶点的数组。现在，我们有一个 3D 空间中的目标点集合（遮挡物体的顶点）和一个点（光源的位置）。一种获取“好的”方向向量快速有效的方法是取光源指向每个顶点的向量的平均值（后面将会讲述更好的方法）。

计算出的向量叫做平均方向向量（mean direction vector），或 MDV。

$$\vec{MDV} = \frac{\sum_{i=1}^{N_v}(V_i - P_{light})}{N_v}$$

其中 N_v 是遮挡物的顶点数，P_{light} 是光源位置。

规一化 MDV 得到 Z_{light}。

$$\vec{Z}_{light} = \frac{M\vec{DV}}{\left|M\vec{DV}\right|}$$

3. 优化窍门 #1

因为要规一化 *MDV*，所以不用将光到顶点的向量和除以 Z_{light}，这可以省去一次除法运算。还可以预先计算 $P_{light}*N_v$ 以避免 $-P_{light}$ 出现在顶点循环中，因为：

$$\frac{\sum_{i=1}^{N_v}(V_i - P_{light})}{N_v} = -P_{light} + \frac{\sum_{i=1}^{N_v}V_i}{N_v}$$

以下是计算的 C 代码。

```
typedef struct
{
  E3dType     X,Y,Z;
  short  Flags;
} E3dVertex;

void ShadowMatrix(Matrix LBlockerLocalToWorldMatrix)
{
  unsigned long   LVn, LVC, LN, LC;
  float           Mx, My, Mz, LPlightX, LPlightY, LPlightZ,
  float           LMDVX, LMDVY, LMDVZ,  // Mean Direction Vector
  float           LZlightX, LZlightY, LZlightZ,  // Zlight vector
  // Initialize Mean Direction Vector to (0.0, 0.0, 0.0)
  //
  LMDVX = LMDVY = LMDVZ = 0.0;

  Lvertex = LMesh->Vertices;

  // Average vertex-to-light vectors
  //
  LVn = LMesh->NumOfVertices;

  LMDVX = LPlightX * LVn;
  LMDVY = LPlightY * LVn;
  LMDVZ = LPlightZ * LVn;

  for(LVC = 0;L VC < LVn; LVC++, LVertex++)
  {
    Mx=LVertex->X; My=LVertex->Y; Mz=LVertex->Z;
    E3dM_MatrixTransform3x4(LBlockerLocalToWorldMatrix, LX, LY, LZ);

    LMDVX -= LX;
    LMDVY -= LY;
    LMDVZ -= LZ;
```

```
    }

    // Normalize Mean Direction Vector (MDV)
    //
    LVF = sqrt(LMDVX*LMDVX+LMDVY*LMDVY+LMDVZ*LMDVZ);
    LVF = 1.0 / LVF;     // We can save 2 divisions by doing this in advance...

    LZlightX = LMDVX * LVF;
    LZlightY = LMDVY * LVF;
    LZlightZ = LMDVZ * LVF;
    …
}
```

E3dM_MatrixTransform3x4 是一个宏函数，它用一个 3×4 矩阵（实际上是一个 4×4 矩阵的左上部分）对 3D 向量（*Mx*、*My*、*Mz*）进行变换。

源代码的其他部分请参见本书附带 CD-ROM 的示例程序。

4. 寻找 X_{light} 和 Y_{light}

将阴影图纹理映射到接收物体上的投影与用于绘制阴影图的是同一个投影，因而阴影图的朝向（绕着 Z_{light} 轴的旋转角度）无关紧要。也就是说，绕着 Z_{light} 轴旋转 *XY-light* 平面没有什么区别。这意味着，可以使用任何垂直于 Z_{light} 的单位向量作为 X_{light} 轴（参见图 5.8.5）。可以将 Z_{light} 与任意不与其平行的向量做叉积运算，得到 X_{light}。将这个做叉积运算的向量叫做 *V*。

我们知道，世界坐标系统的 *X*、*Y*、*Z* 轴至少有两个符合条件。简单起见，使用单位向量 *V*(*x*, *y*, *z*)，一个坐标为 1 其余为 0。

向量的最大分量（*X*、*Y* 或 *Z*）决定其主要方向；因而，要获取一个远离 Z_{light} 的向量，我们将 *V* 的分量中在 Z_{light} 方向上绝对值最小的分量设为 1。这消除了在运行 Z_{light} 和 *V* 的叉积运算时，对浮点数精确性的担心。

例如：

如果 Z_{light}=(0.381, 0.889, 0.254)，则 *V* 为：(0.0, 0.0, 1.0) = Z_{world}

如果 Z_{light}=(− 0.889, 0.254, 0.381)，则 *V* 为：(0.0, 1.0, 0.0) = Y_{world}

等等。

将 Z_{light} 和 *V* 做叉积运算得出第三个向量，它与这两个向量都垂直。该向量规一化后得到 X_{light}，即光坐标系统的 *X* 轴。

$$\vec{X}_{light} = \frac{\vec{Z}_{light} x \vec{V}}{\left|\vec{Z}_{light} x \vec{V}\right|}$$

有了 Z_{light} 和 X_{light}，Y_{light} 轴就是再一次的点积运算：

$$\vec{Y}_{light} = \vec{X}_{light} x \vec{Z}_{light}$$

请注意，这一步得到的是单位矩阵，所以不用再规一化了。

$$\left|\vec{X}_{light}\right| = 1 \text{ 且 } \left|\vec{Z}_{light}\right| = 1 \text{ 且 } \vec{X}_{light} \perp \vec{Z}_{light} \Rightarrow \left|\vec{X}_{light} \times \vec{Z}_{light}\right| = 1$$

知道了 X_{light}、Y_{light}、和 Z_{light} 及 P_{light}，简单地代入下面这些值，我们就可以得到用于将点从直接坐标变换到光坐标的变换矩阵了。

$$M_{WorldToLight} = \begin{bmatrix} Xof\vec{X}_{light} & Xof\vec{Y}_{light} & Xof\vec{Z}_{light} & 0.0 \\ Yof\vec{X}_{light} & Yof\vec{Y}_{light} & Yof\vec{Z}_{light} & 0.0 \\ Zof\vec{X}_{light} & Zof\vec{Y}_{light} & Zof\vec{Z}_{light} & 0.0 \\ -Xof\vec{X}_{light} & -Yof\vec{Y}_{light} & -Zof\vec{Z}_{light} & 1.0 \end{bmatrix}$$

这就是为什么我们使用 X_{light}、Y_{light} 和 Z_{light} 来描述光坐标系统方向的原因。

下一步，将这个矩阵和遮挡物体的“局部—世界”矩阵预先做乘法运算。

$$M_{BLocker\,LocalToLight} = M_{Blocker\,LocalToWorld} * M_{WorldToLight}$$

顾名思义，这个矩阵将遮挡物上定义在局部坐标系统中的点变换到光坐标系统。这样变换的 X 和 Y 坐标，定义了从遮挡物体到阴影图平面（与光坐标系统 XY 平面平行）的平行或正交投影。

5. 定义透视投影

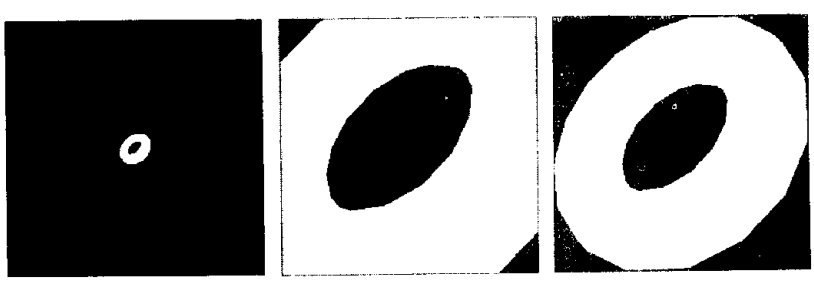
要定义透视投影，需要一个视域，或 X 和 Y 的“投影比例”。通过使用矩阵 $M_{BlockerLocalToLight}$ 变换顶点，并将结果 X 和 Y 坐标除以 Z 坐标，可以得到遮挡物每个顶点的投影比例（R_X 和 R_Y）。

$$R_Y = \left| \frac{V_{tx_Y}}{V_{tx_Z}} \right|$$

6. 自适应投影

可以使用相同的投影比例，但是如果将光源移远或拉近，阴影图中遮挡物的轮廓大小会变换。如果改变遮挡物大小，或者如果光源换一个角度进行照射，都会出现同样的问题。这些变化将导致遮挡物在阴影图中的图像过小或过大（见图 5.8.6）。在前一种情况下（图 5.8.6a），在接收物上只能得到一个低分辨率、不自然的阴影图。这是一种阴影图内存的浪费。

后一种情况会导致错误的阴影形状（图 5.8.6b），而且可能会“阴影泄漏”（参见“纹理坐标和阴影图坐标”小节）。在这种情况下，阴影图不够大。



(a) (b) (c)

图 5.8.6 非自适应（a 和 b）和自适应（c）的遮挡物投影

我们没有使用一个固定的值，而是将最大的 R_X（R_{Xmax}）作为投影的水平比例，将最大的 R_Y（R_{Ymax}）作为投影的垂直比例。这使得透视投影在 X 和 Y 方向上都能自适应。也就是说，遮挡物的轮廓总是刚好适应阴影图，这样最大程度地利用了阴影图中的所有像素。

因为我们希望使用最小的必要纹理尺寸，所以这个概念非常重要。原因如下：

纹理内存总是紧缺资源，并且最大纹理尺寸可能会受其他因素的限制。

在一些硬件上，绘制完阴影图图像后，需要将其从帧缓冲区转移到专用纹理内存，而转移的速度受到总线和内存带宽的限制。

现在我们可以得出遮挡物体的标准投影矩阵：

$$M_{Blocker\,Projection}=\begin{bmatrix}\frac{0.98}{R_{X\,max}}*SMapWidth & 0 & 0 & 0\\ 0 & \frac{0.98}{R_{Y\,max}}*SMapHeight & 0 & 0\\ 0 & 0 & \frac{Z_{far}+Z_{near}}{Z_{near}-Z_{far}} & -1\\ 0 & 0 & 2\frac{Z_{far}Z_{near}}{Z_{near}-Z_{far}} & 0\end{bmatrix}$$

其中 *SMapWidth* 和 *SMapHeight* 分别是阴影图水平和垂直的像素分辨率。Z_{near} 和 Z_{far} 分别是从光源出发的视锥的近裁剪面和远裁剪面。

将这个矩阵预乘可以得到一个 4×4 矩阵，该矩阵是从遮挡物局部坐标到阴影图坐标的透视投影。

$$M_{BlockerLocalToShadowMap}=M_{BlockerLocalToLight}*M_{BlockerProjection}$$

在 OpenGL 中，只用简单地将单位矩阵装载到 PROJECTION 矩阵，将 $M_{BlockerLocalToShadowMap}$ 装载到 MODELVIEW 矩阵，就可以开始使用遮挡物的局部坐标绘制阴影图了。

7. 优化窍门 #2

要减少创建阴影图所需的时间，可以使用两、三个不同几何体版本的遮挡物体，用于不同的渲染阶段。

- 建立阴影图投影，可用使用最少顶点的遮挡物几何体。此处不需要连接性数据（如多边形）或法向量。所关注的是，不论观察角度如何，遮挡物不应该有多边形在阴影体的投影形状之外。因为如果这样的话，多边形就出了阴影图的边界，会导致“阴影泄漏”。我们甚至可以使用一个“好”的包围体（bounding volume），如遮挡物的包围盒，这可以避免计算 MDV（只使用从光源到包围盒中心的向量就可以了）。
- 绘制阴影图使用的遮挡物几何体可以仅包含必要的轮廓细节，且很少或不包含表面细节。
- 当然，绘制遮挡物体的几何体则需要包含所有表面细节和表面属性（如法向量），这样才能绘制出漂亮的对象。

8. 优化窍门 #3

如果渲染引擎是可编程的，可以使用很简单（且快速）的渲染器代码来绘制阴影图。

- 不需要光照，简单的“平色”渲染器就可以。
- 不需要裁剪（遮挡物的图像总是刚好与阴影图适应）。
- 不需要深度测试（Z 缓冲）。

5.8.5 在接收物体上投影阴影图

现在我们有了与遮挡物体和光源相关联的阴影图。这个阴影图可投影到任意数量的接收物体上，而且由于它是作为纹理应用的，接收物体可以为任意复杂的形状（曲线形、小洞、脉纹等等）。

如前面所提到的，我们使用了与创建阴影图相同的投影将阴影图投影到接收物上。惟一的区别是图像的偏移量和缩放系数不同，因为坐标范围是［0..1］而不是[0..SmapWidth]或[0..SmapHeight]。

$$M_{Receiver\,Projection}=\begin{bmatrix}\frac{0.49}{R_{X\,max}}*SMapWidth & 0 & 0 & 0\\ 0 & \frac{0.49}{R_{Y\,max}}*SMapHeight & 0 & 0\\ -0.5 & -0.5 & \frac{Z_{far}+Z_{near}}{Z_{near}-Z_{far}} & -1\\ 0 & 0 & 2\frac{Z_{far}Z_{near}}{Z_{near}-Z_{far}} & 0\end{bmatrix}$$

下一步，将此矩阵预乘 $M_{WorldToLight}$。

$$M_{WorldToShadowMapST}=M_{WorldToLight}*\ M_{ReceiverProjection}$$

结果矩阵可将一个点从世界坐标变换到阴影图纹理坐标（结果 X 和 Y 分别给出 S 和 T）。

将 $M_{WorldToShadowMapST}$ 预乘接收物的“局部到世界”矩阵所得出的矩阵，可从接收物局部空间直接变换到纹理图坐标空间。

$$M_{WorldToShadowMapST}=M_{WorldToLight}*\ M_{ReceiverProjection}$$

纹理坐标和阴影图坐标

阴影图是一个有着有限像素和整数坐标值（如 256×256）的图像。然而，纹理坐标常常规一化得到浮点值，就是说，[0..1]的范围对应像素坐标水平[0..255]，垂直[0..255]的范围。那么在[0..1]范围之外的该怎么办？必须保证接收物上的纹理像素是没有阴影时的颜色（图 5.8.6 中的黑色区域）。

在大多数硬件上使用纹理贴图至少有两种选择：

- 纹理重复。在[0..1]范围之外，简单地重复纹理——例如纹理坐标范围[-1..0]的纹理与[0..1]的纹理图像相同。

- 纹理钳位。在[0..1]范围之外的地方都使用纹理图像边界像素的颜色，或定义一种专门的“边界颜色”。

显然，我们需要使用纹理钳位，因为我们希望在[0..1]纹理坐标范围之外的效果是统一的。

有了纹理钳位，就不用测试接收物体与阴影体的交集了。因为不是所有的3D硬件和API都提供单独的纹理边界颜色，所以必须在纹理图周边空出一个像素的宽度。如果不小心在这个边界绘制了颜色，则会散布到整个接收物上，产生泄漏效果（“纹理泄漏”）。为了确保边界上不绘制任何东西，我们在投影矩阵中（元素(0, 0)和(1, 1)）稍微减小了 X 和 Y 的缩放系数。这就是为什么在 $M_{BlockerProjection}$ 中使用了值 0.98（而不是 1.0），在 $M_{ReceiverProjection}$ 中使用了值 0.49（而不是 0.5）。注意，这些值依赖于阴影图的分辨率（计算公式请参见随书光盘中的示例程序）。

5.8.6 渲染接收物体

绘制接受阴影的物体有几种不同的方法，其中最常用的两种是：

单通道渲染。如果在接收物体上没有其他的纹理，可以在一个通道中进行绘制，使用白底黑图的阴影图作为纹理，用光源来照明物体。

使用减法混合的多通道渲染。如果接收物已经有了纹理且硬件不支持多重纹理，则需要多个通道。

（1）绘制一般情况下的接收物体。

（2）使用黑底白图的阴影图，用减法像素混合绘制阴影。这可以减小有阴影投射地方的表面色彩强度。使用“GREATER-OR-EQUAL”或“LESS-THAN-OR-EQUAL”Z 比较函数来绘制多重通道。在这种方式下，如果传递的是相同的图元，则当前通道会覆盖或混合前一个通道。

关于像素混合，请参见本书 5.10“游戏中的玻璃效果”。

5.8.7 对基本算法的扩展与改进

简单与高性能常常都是需要代价的，这个投影阴影贴图算法也不例外——它有一些限制条件。不过，这些限制中的大多数都很容易克服，可以扩展算法处理这些情况。

1. 背面阴影消除

投影阴影贴图的一个负作用是，它会把阴影贴图在接收物背对光源的那一面。

以下两种方法可以修正这个问题。

（1）确定三角形是否背对光源，如果是，则将其所有顶点设置到阴影图的范围之外（随书光盘中的示例代码使用的就是这个方法）。

（2）将接收物背对光源的面渲染为全黑（没有环境光照）。这种方法是正确的，它与真实的情况更接近。然而，如果场景中不只一个光源，遮挡物的“背面”可能被其他的光照亮。在这种情况下，需要使用多通道渲染，要在不同的通道中加进环境光及从其他光源而来的光。

2. 接收物在光源背后

需要明确地检测这种情况，不要在接受物体上映射阴影。

3. 多重光源

这种情况，需要在接收物体上使用减法混合的多通道渲染，每个通道用于一个阴影图。多通道可成功地减少接收物表面阴影区的强度（RGB 值），即使是阴影交叉的区域也能得到正确的效果。

4. 多个遮挡物

这种情况也需要多通道，惟一的区别是：在阴影交叉区累加阴影的效果是不正确的，因为两个遮挡物遮挡的是同一个光源的光。使用模板缓冲，不要在已经绘制有阴影的屏幕区域再进行绘制。

5.8.8 参考文献

[Blinn88] Blinn, James, "Me and My (Fake) Shadow," *Jim Blinn's Corner,* pages 53–61, January 1988.

[Foley90] Foley, et al., *Computer Graphics Principles and Practice,* second edition, pages 745–753, Addison-Wesley, 1990.

[Blythe96] Blythe, David, and McReynolds, Tom, *Programming with OpenGL: Advanced Rendering,* SIGGRAPH '96 Course Notes, August 1996.

[Heckbert96] Heckbert, Paul, and Herf, Michael, *Fast Soft Shadows,* SIGGRAPH '96 Visual Proceedings, page 145, August 1996.

[Heckbert97] Heckbert, Paul, and Herf, Michael, *Simulating Soft Shadows with Graphics Hardware,* CMU-CS-97-104, Computer Science Department, Carnegie Mellon University, January 1997.

[Woo97] Woo, Mason, Neider, Jackie, and Davis, Tom, *OpenGL Programming Guide,* second edition, Addison-Wesley Developers Press, Silicon Graphics, 1997.

5.9 使用光滑预过滤和Fresnel项改善环境映射反射

Anis Ahmad

为了渲染逼真的场景，我们必须能处理反射周围环境的表面。环境贴图（environment mapping）技术可实现实时近似反射。最近，视点独立（view-independent）的实现技术（如用于老硬件的双抛物线图和用于新硬件的立方体图）对环境图又作了提高。

环境图将一个特定点（环境图的原点）周围的场景映射成一个或多个纹理图，每个纹理像素关联单位球上的一个向量。每个纹理像素的值为从其对应的单位向量到环境图原点的光。然后，可以通过生成并将单位向量转换到纹理坐标（或者，如果是立方图，将向量本身作为纹理坐标），来对环境图进行索引。这样，通过将反射视图向量转换到纹理坐标，就可以检索出环境图中正确的点来模拟反射。

虽然简单且精致，但是这些环境图只能模拟光滑表面的效果——不管视角如何，这些表面的反射都完全遵守镜面反射。对该方法一种常用的改进方式是用一个漫反射纹理图来混合环境图。虽然这种改进确实提高了图像质量，但它并没有解决反射模型过于简单这个根本问题。本文讨论了使用该简单模型的前提条件，并讲述了两种技术（光滑预过滤和Fresnel项权重），可改进并扩展现有方法对环境图的使用。

5.9.1 第一个不正确的假设

传统环境图最主要的错误是，它假设所有表面对光完全反射——也就是说，假设所有打到表面的光子都从一个方向完全反射出去。这个简化的反射仅适合于镜面或反射率非常高的表面。对于其他类型的反射表面（无光泽金属、桔面等），则需要使用更普通的反射。

如图5.9.1所示，表面常常将光分散到各个方向。在一个特定方向反射的光的能量依赖于该表面的属性，尤其是光洁度。最终的反射效果是某种程度上“模糊了”的反射（因为产生出你所看到的图像的那些光，并不是沿同一简单路径从光源传入你眼睛中的）。

为了描述在反射中光是如何发散的，计算机图形研究者们提出了一种双向反射分布函数（BRDF），用于抽象表面反射模型。BRDF计算一个沿给定（入射）方向到达表面的光子以特定（出射）角度反射出去的概率。

BRDF 可以有任意数量的参数（包括表面属性、光的波长等），其中光的入射方向和所期望的出射方向是两个必要参数。BRDF 常常在全局照明求解中用于建模反射比。现在让我们看看，BRDF 是如何用于提高环境图的。

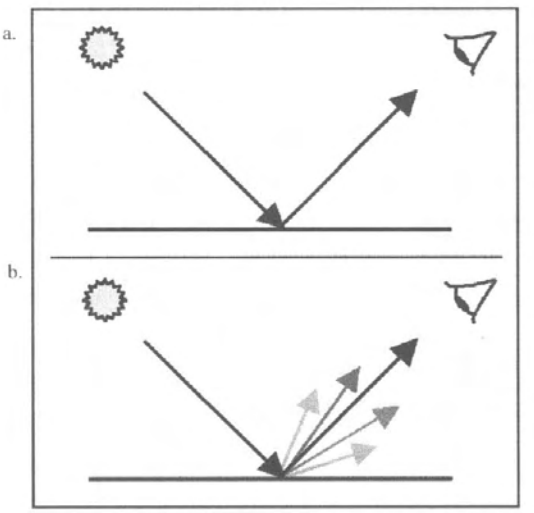


图 5.9.1 a.简化的光线反射；b.更真实的光线反射

Wolfgang Heifrich 和 Hans-Peter Seidel 提出了一项技术，叫做光滑预过滤（glossy prefiltering），用 Phong BRDF 对环境图进行适当的模糊。过滤的环境图中每一个纹理像素是通过找到其对应的单位向量（用于出射方向）产生的，该纹理像素使用以下积分运算计算颜色：

$$pref(\mathbf{o}) = s \cdot c \cdot \int_{\Omega} p(\mathbf{o} \cdot \mathbf{i})^{r} \cdot orig(\mathbf{i}) \cdot d\omega(\mathbf{i})$$

其中：

orig 是纹理图原点。

pref 是预过滤的纹理图。

r 是光洁度参数（Phong 指数的倒数）。

s 是反射系数。

c 是 Phong 修正系数。

i 和 **o** 分别是入射和出射方向。

p(x)是一个函数，当 x >= 0 时返回 x，否则返回 0。

Ω是积分域，单位球。

dω(**i**)是立体角在 **i** 方向上的度量。

为了能实际操作，光滑预过滤需要输入具有高动态范围的纹理，也就是说，纹理的值超过[0..1]的范围。这个范围要用于仿效更高的入射能量强度（相对于不反光表面的反射能量强度而言）。要将普通纹理转化成具有高动态范围的纹理，可以将其中对应于一个光源的所有纹理像素乘以一个扩大系数。请注意，前面的方程是用于将 Phong BRDF 应用于环境图中的。Jan Kautz 和 Micheal McCool[Kautz00]提出了一项技术，可使用任意各项同性的 BRDF 于过滤环境图。

光滑预过滤有许多优点：预过滤只需进行一次、很灵活、易于实现、不需要改变渲染通道（因为它过滤传统环境图只是作为一项预处理）。光滑预过滤的缺点是：计算速度慢（并且不能用于动态环境图）、增加了用于纹理的内存需求（因为每个表面类型都需要一个全新的图的集合）、要求输入纹理具有高动态范围、需要对球体积分。

5.9.2 第二个不正确的假设

在使用环境图的时候又引入了另一个假设，那就是假设反射表面是金属的。当非金属表面反射时，反射率依赖于入射方向与表面法线的夹角。Fresnel 项[Foley90]通过调解反射率模拟这种依赖关系。它使用入射光的角度和表面的折射指数计算合适的权重。Fresnel 项的公式如下。

$$F = \frac{(g-k)^2}{2(g+k)^2}\left(1+\frac{[k(g+k)-1]^2}{[k(g-k)+1]^2}\right)$$

其中：

$k = \cos\theta$

θ是入射方向和表面法线的角度。

$g = \eta_\lambda^2 + k^2 - 1$

η_λ是表面折射指数与传输介质折射指数（波长的作用）的比。因为空气的折射指数是 1，所以通常可以直接使用表面折射指数。

在游戏中，通常都是处理折射指数为常量的表面和大气，所以上述方等式中只有一个变量 k。因而，Frensel 项可以写成 k 的函数，其中 k 的范围是[0..1]。如 Heidrich 和 Seidel 所指出的，可以预计算 Fresnel 项并将其作为一维纹理存储。通过将 Fresnel 项渲染到 alpha 通道中，可使用以下两种方法，将其合并到渲染流程中。

$C_f = C_m * F + C_d \quad or \quad C_f = C_m * F + C_d * (1 - F)$

其中：

C_f、C_m和 C_d分别是最终（传输）、镜像和散射颜色值。

F 是 Fresnel 项。

以这种方式使用 Fresnel 项，只需消耗一点内存就可以提供更真实的反射。缺点是需要一个额外的通道。

5.9.3 结论

有了高性能的纹理贴图硬件和更大的纹理内存空间，就可以使用纹理贴图技术来提高渲染图像的质量。本文提出了两种改进技术，它们都很简单，而且很容易合并到已有的渲染流程中。

5.9.4 致谢

非常感谢 Micheal McCool、Michael Anttila、Sim Dietrich 和 Mark DeLoura 对本文进行审阅。

5.9.5 参考文献

[Blinn76] Blinn, J., and Newell, M., "Texture and Reflection in Computer-Generated Images," *Communications of the ACM*, 19:542-546, 1976.

[Heidrich99] Heidrich, W., and Seidel, H.-P., "Realistic, Hardware-Accelerated Shading and Lighting," *SIGGRAPH '99 Proceedings*, pages 171–178, August 1999.

[Foley90] Foley, J., van Dam, A., Feiner, S., and Hughes, J., *Computer Graphics: Principles and Practice*, pp. 766–770, 1990.

[Kautz00] Kautz, J., and McCool, M., "Approximation of Glossy Reflection with Prefiltered Environment Maps," *Proceedings Graphics Interface*, www.graphicsinterface.org/,May 15–17, 2000.

5.10 游戏中玻璃的效果

Gabor Nagy

本文对常用的实时玻璃物体渲染算法提出了几种扩展方法。

5.10.1 介绍

在交互式的情况下渲染出好看的玻璃物体一直以来就极具挑战性。在计算机硬件有足够的速度进行实时光线跟踪之前，效果都不理想。

5.10.2 透明物体

玻璃有3个主要的视觉特性：

- 透明。玻璃可让一些照射在其表面的光通过，使得在它后面的物体能够部分可见。
- 折射。穿过玻璃物体的光会折射，透过玻璃显示的环境会被扭曲。
- 反射。玻璃会反射一些照射在其表面的光，因而会在其表面反射出周围的环境。

本文主要关注玻璃透明和反射的特性。

5.10.3 光栅化程序、帧缓冲、Z缓冲和像素混合

用当今的3D硬件绘制透明物体，通常使用像素混合（或简单地叫做混合）。像素混合在渲染流程的最后阶段实现（在光栅化之后的像素渲染中）。

光栅化程序将图元（三角形、线段等）转换成用X和Y屏幕坐标和深度（Z）值表示的点。

使用Z缓冲的简单的像素渲染如下：

- 计算待绘制像素的Z（深度）值。
- 比较该值与存储在Z缓冲区中相应位置的Z值。
- 如果该待绘制的像素距离观察者更近（它在已绘制在该位置的物体前方），则直接覆盖，将像素的颜色和Z值写入各自的缓冲中；如果不是，则不改变帧缓冲和Z缓冲的值。

在OpenGL中，Z值较小表示该像素离观察者较近。在绘制场景前，

每个像素的 Z 缓冲被初始化（清除）为最大的 Z 值。该值依赖于 Z 缓冲的位深度。请注意，对于不同的 3D API 和硬件，该值的意思可能完全相反。

5.10.4 不透明物体与透明物体

因为标准的 Z 缓冲技术只是简单地将距离观察者更近的点覆盖到前一点上，所以它只适合绘制完全不透明的表面。要绘制透明表面，不能简单地在帧缓冲覆盖像素颜色，而需要对两种颜色进行某种程度的混合。

在 OpenGL 中，可以使用混合函数来执行这个任务。可通过以下调用定义混合函数：

```
glBlendFunc(sfactor, dfactor);
```

输入到帧缓冲的颜色通常按如下计算：

$$RGB_{result} = RGB_{source} * sfactor + RGB_{destination} * dfactor$$

其中 RGB_{result} 代表输入到帧缓存中的红、绿、蓝分量；RGB_{source} 是新的像素分量；$RGB_{destination}$ 是已在帧缓冲中相应像素的值。

在不同版本的 OpenGL 中，sfactor 和 dfactor 可以有很多不同的预定义常量。例如：

```
glBlendFunc(GL_ONE, GL_ZERO);
```

执行简单的覆盖，因为：

$$RGB_{result} = RGB_{source} * 1 + RGB_{destination} * 0$$

再例如，如果想要将当前像素颜色与帧缓冲中已有的颜色相加，可以调用：

```
glBlendFunc(GL_ONE, GL_ONE);
```

这将得到如下的混合公式：

$$RGB_{result} = RGB_{source} * 1 + RGB_{destination} * 1$$

要在 OpenGL 中执行像素混合，可调用：

```
glEnable(GL_BLEND);
```

对于完整的像素混合描述，请参考 OpenGL 用户手册[Woo97]。

因为需要考虑 Z 值，所以我们将像素称作 RGBZ。

在使用透明物体渲染 3D 场景的时候，可能有如下情况之一。当前渲染的像素（$RGBZ_{source}$）可属于不透明物体，也可属于透明物体，并且：

- 它的 Z 值表明它比已经在帧缓冲中的对应像素（$RGB_{destination}$）距离观察者更近（它在已绘制在该地方的物体的前面）；
- 它比 $RGB_{destination}$ 离观察者远；

- 它与 $RGB_{destination}$ 的距离一样。

5.10.5 绘制不透明物体

让我们来看看绘制不透明物体的情况。如果 RGB_{source} 比 $RGB_{destination}$ 离观察者更近，可以用它简单地覆盖帧缓冲。然而，如果 RGB_{source} 比 $RGB_{destination}$ 远，但如果它是在透明物体的“后面”，则仍需要绘制。通常，可以通过先绘制所有不透明的物体来避免这个问题。

5.10.6 绘制透明物体

我们使用值 A_{source}（Alpha 或 Opacity 值）来定义当前绘制像素不透明的程度。A_{source}=0.0 表示该像素完全透明；1.0 表示它完全不透明。如果待绘制像素（RGB_{source}）在帧缓冲中相应像素（$RGB_{destination}$）的前面，需要使用如下公式决定结果颜色。

混合公式 A：$RGB_{result} = RGB_{source} * A_{source} + RGB_{destination} * (1.0 - A_{source})$

如果 RGB_{source} 在 $RGB_{destination}$ 后面，需要使用如下公式。

混合公式 B：$RGB_{result} = RGB_{source} * (1.0 - A_{destination}) + RGB_{destination} * A_{destination}$

该公式使用帧缓冲中存在的 alpha 位平面来追踪每个像素的透明度，详细内容参见[Woo97]。我们也需要更新已绘制像素的 alpha 值。

根据 RGB_{source} 是在 $RGB_{destination}$ 的前面还是后面，我们有两种不同的混合函数，或两种不同的途径。由于只能在一个时刻定义一个混合函数，因而我们需要找到一种解决办法。

1. 深度复杂性（depth complexity）

此问题的核心是“深度复杂性”：有可能多个像素（属于不同图元）以不同的深度值占用同一个平面位置。

深度复杂性 1 表示在平面上没有图元重叠。在这种情况下甚至不需要使用 Z 缓冲。当绘制透明图元的时候，只用将它与背景颜色混合，使用混合公式 A。

深度复杂性 2 表示在每个像素上重叠图元的数目是 2。在这种情况下，一个不透明的像素，或者挡住背景或透明像素，或者在一个不透明或透明像素的后面。

请注意，当摄像机移动的时候，3D 场景的深度复杂性可能会改变。

很幸运，在 OpenGL 中，我们可以对 Z 缓冲的执行进行一些控制。尤其是，可以禁用 Z 覆盖，所以当绘制像素的时候，只改变了帧缓冲中的 RGB 值。结合其他一些特性，我们可以找到大多数情况的解决办法。

2. 一个简单的解决办法

要在同一个图像中绘制不透明和透明的物体，我们可以使用一个不是完全正确但是很简单的方法：

- 清除 Z 缓冲。
- 启用 Z 测试和 Z 覆写，绘制所有的不透明图元（三角形、线段等）。
- 启用背面剔除（以最小化透明像素间的复杂性）、禁用 Z 覆写、使用混合公式 A 绘制

透明图元。

这个方法可以保证不透明的表面总是遮挡透明的表面，而且在透明物之后的不透明表面能够显示出来。它还保证如果两个透明表面的 alpha 值都是 0.5，它们总能被正确地混合，因为 0.5＝1－0.5，所以混合公式 A 和混合公式 B 是等价的。因而，当前绘制的透明像素在帧缓冲中像素的前面还是后面无关紧要。对于 0.5 以外的 alpha 值，或不只两个透明物体，效果不太精确，不过在很多情况下还是可以接受的。

3. “简单”的解决办法 #2

这个办法用于解决前一个小节中提到的问题，它保证当前绘制图元总是在帧缓冲中的图元的前面。

- 清除 Z 缓冲。
- 绘制所有不透明图元（三角形、线段等）。
- 按照深度对所有透明图元排序，并依从远到近的顺序进行绘制。

在此我们仅需使用混合公式 A。

使用此方法的一个主要问题是，深度排序可能带来极大的性能影响，尤其当有很多透明物体在不同的层次节点中时。有时还有可能一个图元既不完全在另一个图元的前面，也不完全在它的后面（如图 5.10.1 所示），因此需要进行逐像素深度排序，计算量非常大。

只要正确执行了深度排序，本方法对深度复杂性没有限制。请注意，使用这个方法，不必先绘制不透明的物体，但是这样做可以简化处理过程。

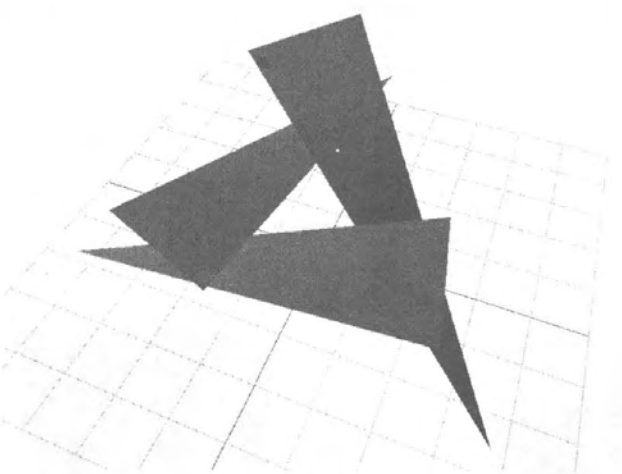


图 5.10.1 不确定的三角形深度顺序

4. 一个略微不同的方法

在 OpenGL 中可以选择 Z 缓冲的深度函数（depth fuction）。深度函数决定那些 Z 值进行 Z 比较。在 OpenGL 中采用的是“小于”（less-then）方式，也就是说如果像素的 Z 值比 Z 缓冲中的小，则绘制该像素。这使得我们可以分两步，使用两个不同的混合公式绘制图元。在此方法中，要求帧缓冲中有 alpha 值。

步骤如下：

（1）清除 Z 缓冲。

（2）清零 alpha 缓冲（透明）。

（3）将 Z 函数设为“大于”（greater-than）（绘制后方）、使用混合公式 A、禁用 Z 覆写，绘制第一个透明图元。此外，将该图元的 alpha 值写入帧缓冲以便它后面的像素可以正确地混合。

（4）将 Z 函数设为“小于等于”（less-than-or-equal）、使用混合公式、启用 Z 覆写，重新绘制该图元。将该图元的 alpha 值写入帧缓冲以便它后面的像素可以正确地混合。

（5）对每个透明图元重复上面两步。

（6）将 Z 函数设为“大于”（greater-than）（绘制后方）、使用混合公式 B、禁用 Z 覆写，绘制所有不透明图元。写入帧缓冲的 alpha 值为 1.0（或最大的整数值）。

（7）将 Z 函数设为“小于等于”（less-than-or-equal）、启用 Z 覆写、禁用混合，绘制所有不透明图元。

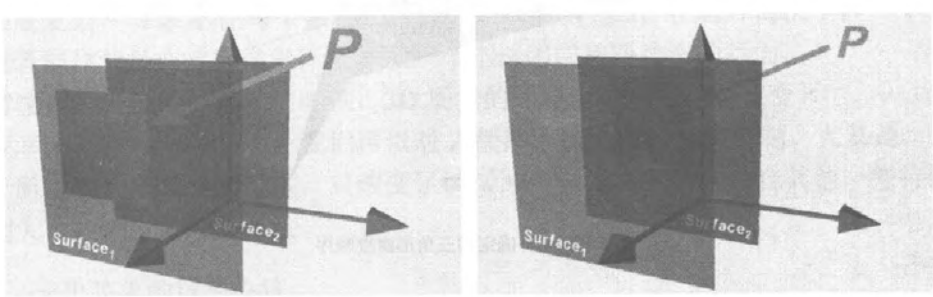
不幸的是，如果在帧缓冲的给定位置绘制了不只一个像素（不同的深度），该像素的透明图不再能由一个值表示，它依赖于该像素的深度，或在它之前所绘制的像素有多少。这个问题是 Z 缓冲的本质所引起的：它只能为一个像素存储深度值，后续像素将覆盖以前的值。也就是说，Z 缓冲只有固定为 1 的深度复杂性。

以图 5.10.2a 和图 5.10.2b 为例，假定在帧缓冲中有两个表面：

- $Surface_1 - Alpha: A_1 = 0.5$
- $Surface_2 - Alpha: A_2 = 0.75$

如果 P（待绘制像素）在 $Surface_1$ 和 $Surface_2$ 之间，$A_{destination}$ 为 0.5（只有 $Surface_1$ 在 P 的前面）。然而，如果 P 在 $Surface_1$ 和 $Surface_2$ 的后面，$A_{destination}$ 为 $Surface_1$ 和 $Surface_2$ 累积不透明度（cumulative opacity），是 $A_1 * A_2 = 0.375$。

本方法比前面讲述的简单方法更具灵活性，不用对透明图元进行深度排序，但比解决方法#2 的限制条件更多且更复杂。而且，每个图元经常变化的深度函数可能导致性能损失。建议根据实际的应用，稍微对本方法做一点“调整”（尤其可以考虑修改帧缓冲中的 alpha 值）。



a b

图 5.10.2 a：一个点在一个透明表面之后的效果 b：多个透明表面的累积效果

5. 非平面的玻璃物体

如果观测玻璃瓶和玻璃杯，我们将发现边缘的地方看起来更黑，在那里表面法线开始偏

离观察者。这是因为射入物体的光以更高的角度发生了折射，因而到达观察者的光变少了。可以在摄像机的位置设置一个光源（前灯）对物体进行照明来模拟这个效果。表面法线越偏离观察者，该光源的照明越少。简单的漫反射前灯很容易实现，而且计算量也不大。

5.10.7 反射

模拟反射，可以使用球形和立方体环境贴图。OpenGL 支持球形环境图（使用鱼眼图像作为环境图）。初始化纹理后，可使用下列调用启用球形贴图：

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

关于此主题有很多优秀的论文可参考（参见参考文献）。

5.10.8 有色玻璃

到现在为止，我们一直将表面的透明度作为单一的值。如果一个表面在一个透明表面的后面，前面的表面会均匀地减少后面表面的 R、G、B 颜色分量。我们可以对 R、G、B 分量使用不同的透明度值，以描述玻璃中的颜色。这需要使用多重绘制通道，下面的小节将进行讲述。

5.10.9 将它们放到一起

1. 单通道渲染

使用下列几项，可以在单通道中渲染玻璃物体：

- 当作 2D 纹理使用的环境图，纹理坐标通过球形贴图算法计算。
- “调节的”纹理算法和前灯。
- 合适的像素混合和 Z 测试，以将它作为透明物体进行绘制（如在前面的解决办法中所讲述的）。

2. 多通道渲染

要对最终效果获得更多控制，我们可以执行两个渲染通道：

（1）建立纹理渲染和 Z 测试，以将它作为透明物体进行绘制（如前面所讲述的）。也可以在物体上进行光照，模拟毛玻璃表面效果。

（2）在最上层渲染反射，使用附加的混合（如在单通道中的情况）。

有了两个通道，可以通过分别改变每个通道的混合因子来同时定义物体的透明度与反射性。使用多通道，我们还可以应用更复杂的公式。

5.10.10 实现

关于在 OpenGL 中的实现细节，请参见随书光盘中的示例代码。

彩图 8~11 是运用本技术在 Sony PlayStation 2 上渲染的效果。

5.10.11 参考文献

[Greene86] Greene, Ned, "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications,* volume 6. number 11, pp. 21–29 November 1986.

[Woo97] Woo, Mason, Neider, Jackie, and Davis, Tom, *OpenGL Programming Guide,* second edition, Addison-Wesley Developers Press, Silicon Graphics, 1997.

5.11 用于容器中液体的折射贴图

Alex Vlachos，Jason L. Mitchell

本文提出了一种简洁可行的方法，用于在实时用户 3D 加速环境中，对不透明容器中的液体进行折射贴图。为了以可交互的方式模拟水，需要计算折射、反射和 Fresnel 项。本文还给出了一些可以提高真实感的方法，包括刻蚀效果和颗粒物。

5.11.1 介绍

本文的目标是提出一种渲染解决方案，而不局限于模拟水。在提供的示例代码中，我们使用了基于 Erik Larsen 的 newave 例子的表面模拟。

此渲染方法中用到的照明方程非常典型，它包括了折射、反射和 Fresnel 项。Fresnel 项用作折射和反射项间的混合因子[Ts'o87]。

$$Result = Fresnel * Refraction + (1 - Fresnel) * Reflection$$

这也是 RenderMan 着色器中常用的式子。

此外，我们还阐述了一些用于容器内部照明和刻蚀效果计算的技术。本文讲述的所有技术都假定观察者站在容器外，且容器是不透明的。

5.11.2 折射项

1. 斯涅耳法则

计算从眼睛到水中每个顶点的射线，就是图 5.11.1 中的视线。斯涅耳法则用于为每个顶点折射视线。这个多边形网格表示了空气和水之间的界面。因为水对空气的折射指数的比是 1.33，所以我们将用这个值作为水法线函数，在给定顶点上计算折射视线，如图 5.11.1 所示。

参见图 5.11.1，水法线和视线之间的夹角（Θ_i）叫作入射角；折射线和反向水法线之间的夹角（Θ_r）叫作折射角。斯涅耳法则给出了这两个夹角和两个介质（空气和水）折射指数比之间的关系。

$$n_i \sin(\Theta_i) = n_r \sin(\Theta_r) \quad 或 \quad (n_i / n_r) \sin(\Theta_i) = \sin(\Theta_r)$$

对于空气到水的界面，n_i / n_r 是 1.333，这样我们就可以简单地从 Θ_i 计算出 Θ_r。

$\Theta_r = arcsin[1.333\ sin(\Theta_i)]$

到此，很容易从视线计算出折射视线，也就很容易得到视线与容器的交点。

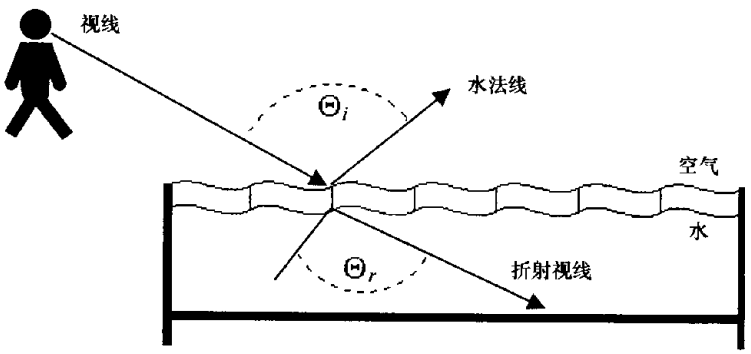


图 5.11.1 斯涅耳法则实践

2. 与容器相交

计算出了折线，下一步就需要确定折线与容器的交点。这一步对于给出容器形状的视觉呈现是至关重要的。抛物面或半球相交测试虽然简单有效，但总是一成不变地给出碟形容器的视像，尤其当观察者交互地在场景中移动时也是。Microsoft X-Box 汽艇中使用的水示例就是一个这样的例子[McQuade2000]。为了实现更复杂且更真实的容器效果，我们对很多几何形状简单的容器进行了试验，结果非常不错。在本文中，为了简单起见，我们使用了一个简单的平行六面体容器。

再参见图 5.11.1，现在考虑折射线与容器壁的相交。在这种情况下，容器由 5 个矩形面组成。计算每个面与射线的交集，得到容器与射线的交点。知道了这一点后，我们把容器内的位置转换到单个折射图的纹理坐标中，该图同时包含容器的 5 个面，如图 5.11.2 所示。

在图 5.11.1 中，折射线与池的底面相交。这就生成了折射图（图 5.11.2）中相应区域的纹理坐标。

3. 容器内部照明

可以预先对容器内部进行光照，再进一步将容器整合到场景中。在大型场景中使用水模拟，这是个很重要的视觉提示。随书光盘上的源代码演示了这项技术，不过水池周围没有添加几何体。演示代码使用了如图 5.11.2 的折射图进行预光照，并加入了 *radiATIon* 图形引擎中，效果参见彩图 13～16。

5.11.3 反射项

任意普通参数都可作为照明中的反射项。我们为静态场景选择了单抛物面环境图，因为它可被广泛的硬件支持。立方图可用于动态场景。在一些情况下，动态更新的二维反射图也很适合。

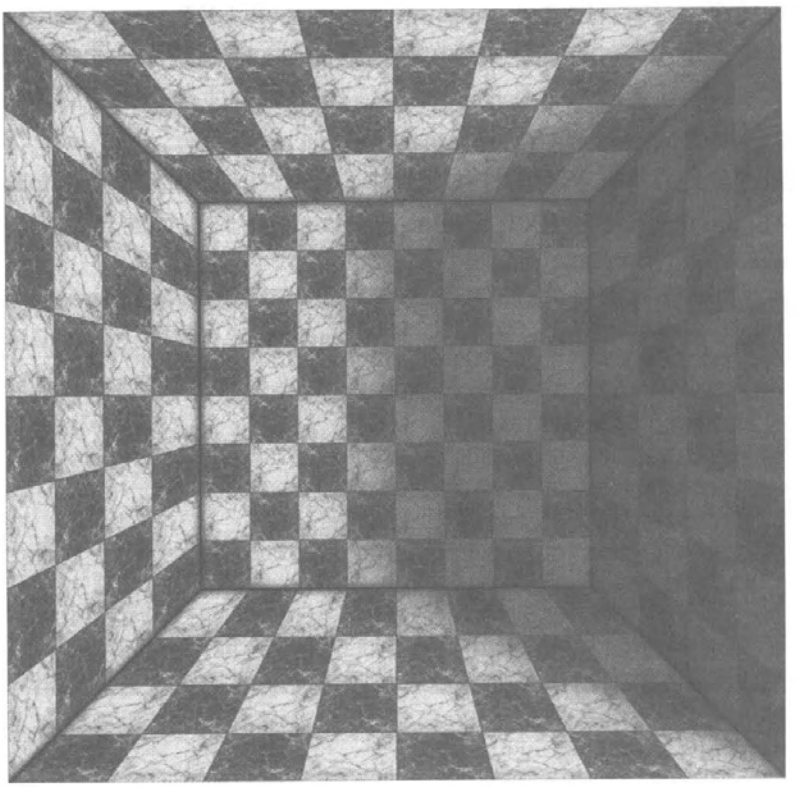


图 5.11.2 折射图（同彩图 12）

5.11.4 Fresnel 项

现在反射项和折射项都已经计算出来了，可以使用 Fresnel 项将它们进行结合。关于 Fresnel 方程请参见本书 5.9 节，“使用光泽预过滤和 Fresnel 项改进环境图反射”。Fresnel 方程决定了在水面特定点上，反射光与折射光的比例（它是观察视线与水面夹角的函数）。在计算机图形学中，该项通常被用作折射项与反射项的混合因子，不过仅将它乘以反射图也可以[Ts'o87]、[Apodaca99]。

此外，该函数本身也可以模拟在 0 到 1 范围内变化的正弦曲线或正弦平方曲线。随书光盘上的示例代码计算了每个顶点上水法线与视线的点积。得出了向量间的余弦，可直接用作 Fresnel 项或将其平方以产生不同的效果。

该方程可以直接在每个顶点上进行计算并在每个多边形上线性插值（如光盘中的示例代码），也可以作为纹理查找表逐像素进行计算[Bastos99]。

5.11.5 使用硬件渲染

以上的计算得出了网格中顶点的纹理坐标，该网格表示空气和水之间的界面。使用的纹理图是静态的，在用户级硬件上网格可以在单通道中渲染，该硬件应能对单抛物面环境图支

持至少两个纹理的多重纹理，EXT_texture_env_combine 和 EXT_texgen_reflection。Fresnel 项用作两个图之间的混合因子，存储在主颜色插值器的 alpha 通道中。再次重申，我们的目标是计算下面的方程：

*Result = Fresnel * Refraction + (1−Fresnel) * Reflection*

有了反射图在纹理 0 中，折射图在纹理 1 中，Fresnel 项在主颜色的 alpha 通道中，我们就可以使用 ARB_multitexture 和 EXT_texture_env_combine 来进行混合了。

```
glActiveTexture(GL_TEXTURE0_ARB);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB_EXT, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT,
        GL_PRIMARY_COLOR_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT, GL_SRC_ALPHA);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_MODULATE);
glTexEnvf(GL_TEXTURE_ENV, GL_RGB_SCALE_EXT, 1.0f);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_ALPHA_EXT,
        GL_PRIMARY_COLOR_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_ALPHA_EXT, GL_SRC_ALPHA);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT, GL_REPLACE);
glTexEnvf(GL_TEXTURE_ENV, GL_ALPHA_SCALE, 1.0f);

glActiveTexture(GL_TEXTURE1_ARB);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB_EXT, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT, GL_PREVIOUS_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_ADD);
glTexEnvf(GL_TEXTURE_ENV, GL_RGB_SCALE_EXT, 1.0f);
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_ALPHA_EXT, GL_PREVIOUS_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_ALPHA_EXT, GL_SRC_ALPHA);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT, GL_REPLACE);
glTexEnvf(GL_TEXTURE_ENV, GL_ALPHA_SCALE, 1.0f);
```

因为该方法只是对两个静态图进行混合，所以对渲染系统造成的负担最小，它只是一个计算纹理坐标的方法。虽然很简单，但该方法产生的视觉结果非常真实。

5.11.6 该技术的扩展

我们概述了水的一个简单方程，其中包括反射、折射和 Fresnel 项。其他的视觉现象，如颗粒物质带来的刻蚀与分散，也可以很好地模拟。

1. 容器内的刻蚀

本技术最重要的扩展就是用来实现容器内的刻蚀效果。将刻蚀效果加入场景中的一种方法是，模拟来自给定光源的折射光线与容器相交（与前面的模拟观察类似）。沿这些射线传来

的光聚集成一个动态“刻蚀图”，它可以与折射图进行多重纹理合成[Stam96]。对于一些应用，使用简单的静态刻蚀图并将其在池子内部反复滚动就足够了。请参见本书 5.5 节“使用纹理坐标生成的高级纹理”中，关于如何使用纹理矩阵，在不同的帧之间滚动纹理坐标的讨论。在游戏《赛尔达传说：时光之笛》(*The Legend of Zelda: The Ocarina of Time*) 的 Zora's Domain 关卡中使用了这个技术描绘湖面在周围洞穴墙上映出的刻蚀效果。

2. 模拟颗粒物质

模拟水中存在的颗粒物质是一个重要的视觉感受，可以很容易地加入系统中。实际上，折射线在容器中的介质中穿过的距离，是在进行相交检测时附带计算出来的。使用该项，就可以在顶点上混合“水色”，其方式类似于在地形引擎和飞行模拟中混合雾色模拟大气效果。请注意，只有折射项受水中颗粒影响。

5.11.7 结论

我们提出了一个简单的方法，可以在用户 3D 硬件上实时模拟简单几何体容器中液体的折射、反射和 Fresnel 效果。该技术的设计初衷是能在用户 3D 硬件上高效地运行。因而，在现在的状态下，该技术仅为反射图和折射图计算新的纹理坐标，而不是对纹理进行更新。

我们还对该技术提出了一些可扩展的领域，包括刻蚀效果和颗粒物质，并提出了一些实现思想。

5.11.8 参考文献

[Apodaca99] Apodaca, A., and Gritz, L., *Advanced RenderMan: Creating CGI for Motion Pictures*, Morgan Kaufmann, 1999.

[Bastos99] Bastos, R., Hoff, K., Wynn, W., and Lastra, A., “Increased Photorealism for Interactive Architectural Walkthroughs,” *ACM Symposium on Interactive 3D Graphics,* pp. 183–190, 1999.

[Heidrich98] Heidrich, W., and Seidel, H.-P., “View-independent Environment Maps,” Eurographics/ACM SIGGRAPH Workshop on Graphics Hardware, 1998.

[McQuade00] McQuade, L., *Personal Communication,* 2000.

[Ts'o87] Ts'o, P., and Barsky, B., “Modeling and Rendering Waves: Wave Tracing Using Beta-Splines and Reflective and Refractive Texture Mapping,” *ACM Transactions on Graphics* (6), pp. 191-214, 1987.

[Stam96] Stam, J., “Random Caustics: Natural Textures and Wave Theory Revisited,” SIGGRAPH , www.syntim.inria.fr/syntim/research/stam/caustics.html, 1996.

第 6 章

附 录

6.0 矩阵工具库

Dante Treglia II, Mark A. DeLoura

矩阵库是游戏编程的一个组成部分，而且对于很多应用（如高级图形、物理、碰撞检测等）至关重要。不可避免地，本书很多文章中需要使用的矩阵运算，OpenGL 矩阵函数都不能提供。因而，在很多提供者的帮助下，我们为本书示例软件的使用编写了一个 C++矩阵库。我们的意图是将其作为教学工具，让它易于阅读且基于通用的用途；因而没有进行优化。我们建议你将该库基于你的游戏进行修改，并基于你的环境进行优化。

6.0.1 说明

本库包括 5 个主要的类：vector2、vector3、vector4、matrix33 和 matrix44，分别表示相应规格的向量和矩阵（如，matrix44 是 4×4 矩阵）。vector 有公有成员 x、y、z 和 w；matrix 由 vector 数组组成。本库提供的功能和格式与 OpenGL 相同，不过矩阵维持“列一行”顺序。大多数标准矩阵和向量运算都被重载；叉积和点积运算则作为实用函数提供，以提供更一致的书写形式。

```
vector3 vec;
vector3 vec1(0.0, 1.0, 0.0);
vector3 vec2(1.0, 0.0, 0.0);
matrix44 mtx = RotateRadMatrix44('x', DegToRad(45.0));

vec = mtx * CrossProduct(vec1, vec2);
```

程序清单 6.0.1 答案：vec = [0.0, 0.707107, -0.707107]

使用该库时，需要记住几个事项。首先，缺省的构造函数并不初始化每个类的成员。这样做消除了对那些成员已被及时设定的向量或矩阵的冗余运算。正确的初始化构造函数和 set 方法由类提供。其次，没有提供点类。需要确保为作为点的向量设定了同质分量（通常设为 1.0）。最后，如果需要将矩阵载入 OpenGL 矩阵栈，安全的做法是将指向 matrix44 的指针转换为浮点指针。

```
vector4 point3;
matrix44 tranMtx = TranslateMatrix44(-10.0, 0.0, 5.0);

point3.set(0.0, 0.0, 0.0, 1.0);
point3 = tranMtx * point3;
```

程序清单6.0.2 答案：point3 = [-10.0, 0.0, 5.0, 1.0]

6.0.2 源代码

本书的附带光盘中有矩阵库的完整代码。关于更多使用信息，请参见源代码。

6.0.3 致谢

感谢此库的所有提供者们！尤其感谢Stan Melax、Miguel Gomez、Pete Isensee、Gabor Nagy、Scott Bilas、James Boer和Eric Lengyel！

6.1 文本工具库

Dante Treglia II

只要需要有输出，所有游戏开发都会面临一个问题。为此，能将输出显示在屏幕上会很方便，尤其是当游戏使用全屏模式时。当今很多游戏都有“控制台”模式，该模式一般在游戏开发期间供程序员使用。你将会发现，编写这样一个库并将其用于你的游戏中会带来很多好处。该文本工具库是 OpenGL 文本输出库的一个基本实现。它很小、易于使用且最重要的是，它很易于被切入。用于创建 8×8 字符的贴图仅 16K 字节，因而该库非常易于调试和剖析。

6.1.1 说明

文本工具库由一个名叫 TextBox 的类组成。这个类提供了两个方法向屏幕绘制文本。第一种是通过提供屏幕坐标和字符串来做的。该库会自动推入在屏幕空间中绘制文本所必要的正交投影，还会弹出保存了前一个矩阵状态的矩阵。第二个方法是第一个方法的自定义版本，需要你初始化屏幕上“文本框”的区域。然后，所有输入该框中的文字（可以为多行）将限制（或滚动）在框中显示，就和标准外壳一样。用于打印文本的是 printf() 函数，与标准 C 函数类似。与第一种方法不同，文本存储在内存中，直到下一屏进行绘制，因此 printf()函数可在游戏期间用于任意点。可使用任意颜色在透明或不透明的背景上打印文字。

6.1.2 源代码

本书的附带光盘中有文本工具库的完整代码和一个演示程序。请参见代码以了解详细的使用信息。

6.2 关于随书光盘

Mark A. DeLoura

在本书附带的光盘中，包含了书中所有的代码。我们非常希望本书能真正对您有的帮助，所以将源代码放在光盘中以方便您使用。

以下列出了光盘中的部分内容：

- 每篇文章中的所有源代码。
- 本书所讲述的一些技术的完整演示程序。演示程序需要在Windows或Linux下运行。
- glSetup单机版。
- GLUT（OpenGL实用工具包）。
- 矩阵工具库。
- 文本工具库。
- 一些有用的游戏编程网站。

完整的安装和使用指导请参见光盘中的AboutThisCD.htm文件。

请查阅Web站点www.gameprogramminggems.com，以获取本书和游戏编程的更多信息。

作者索引

Ahmad, Anis, a3ahmad@undergrad.math.uwaterloo.ca

Bilas, Scott, Gas Powered Games, scottb@aa.net, http://www.aa.net/~scottb

Bore, James, Lithtech, Inc., jimb@lith.com

DeLoura, Mark A., Nintendo of America Inc., madsax@satori.org, http://www.satori.org/madsax

Dietrich, Sim, NVIDIA Corporation, sim.dietrirch@nvidia.com

Dybsand, Eic, Glacier Edge Technology, edybs@ix .netcom.com

Edwards, Eddie, Naughty Dog, Inc., eddie@naughtydog.com

Freitas, Jorge, Electronic Arts Canada, Inc., jorgef@ea.com

Ginsburg, Dan, ATI Research, Inc., ginsburg@alum.wpi.edu

Gomez, Miguel, Lithtech, Inc., miguel@lith.com

Hagland, Torgeir, Shiny Entertainment, torgeir.hagland@powertech.no

Isensee, Pete, Emerald City, PKIsensee@msn.com, http://www.lucasarts.com

Kaiser, Kevin, velderon@warp3000.com

King, Yossarian, Electronic Arts Canada, Inc., yking@ea.com

Kirmse, Andrew, LucasArts Enterainment Company, akirmse@lucasarts.com

LaMothe, André, Xtreme Games LLC, ceo@xgames3d.com

Le Chevalier, Loïc, Infogrames, llechevalier@fr.infogrames.com

Lecky-Thompson, Guy W., LeckyT@GameBox.net

Lengyel, Eric, lengyel@C4Engine.com

Marselas, Herbert, Ensemble Studios, hmarselas@ensemblestudios.com

McCuskey, Mason, Spin Studios, mason@spin-studios.com, http://www.spin-studios.com

Melax, Stan, Bioware Corp., melax@cs.ualberta.ca, http://www.cs.uallberta.ca/~melax

Mitchell, Jason L., ATI Research, Inc., JasonM@ati.com

Nagy, Gabor, Sony Sony Computer Entertainment America, Gabor_Nagy@Playstation.sony.com, http://www.equinox3d.com

Olsen, John, Microsoft, Inc., infix@xmission.com

Paull, David, Tanzanite Softaware, cosmodog@tanzanite.to, http://www.tanzanite.to

Peasley, Mark, Gas Powered Games, mpeasley@gaspowered.com, http://www.pixelman.com/
Rabin, Steve, Nintendo of America Inc., stevera@noa.nintendo.com
Ranck, Steven, Midway Home Entertainment, stanck@home.com,
http://www.members.home.net/sranck
Round, Tim, timround@hotmail.com
Shankel, Jason, Maxis, shankel@pobox.com
Snook, Greg, Mighty Studios, greg@mightystudios.com
Stout, Bryan, bstout@mindspring.com
Svarovsky, Jan, Mucky Foot Productions, jan@svarovsky.freeserve.co.uk,
http://www.svarovsky.freeserve.co.uk/
Treglia, Dante II, Nintendo of America Inc., danttr01@noa.nintendo.com
Ulrich, Thatcher, Slingshot Game Technology, Inc., tu@tulrich.com, http://www.tulrich.com
Vlachos, Alex, ATI Research, Inc., avlachos@ati.com
Woodcock, Steven, Raytheon Technical Services Company, ferretman@gameai.com,
http://www.gameai.com/
woodland, Ryan, Nintendo Technology Devlopment, ryanwo01@noa.nintendo.com