THOMSON

Direct3D 游戏编程入门教程（第二版）

[德] Wolfgang F. Engel 著
周惟迪 徐 翎 张璐意 译

附光盘

人民邮电出版社
POSTS & TELECOM PRESS

Direct3D

游戏编程入门教程(第二版)

使用 DirectX 创建精美的 3D 图形程序!

www.thomsonlearningasia.com

光盘内容
ATI 的 RenderMonkey
NVIDIA 的 CG 工具包
微软 DirectX 9 SDK
安装演示与教程
书中的示例代码

ISBN 7-115-12963-0

9 787115 129635 >

人民邮电出版社网址 www.ptpress.com.cn

ISBN7-115-12963-0/TP·4368
定价:46.00 元(附光盘)

第九艺术学院游戏开发系列

Direct3D 游戏编程入门教程（第二版）

[德] Wolfgang F. Engel 著

周惟迪 徐翎 张璐意 译

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

Direct 3D 游戏编程入门教程 /（德）恩格尔（Engel,W. F.）著；周惟迪，徐翎，张璐意译. —2 版. —北京：人民邮电出版社，2005.1
ISBN 7-115-12963-0

Ⅰ. D... Ⅱ. ①恩... ②周... ③徐... ④张... Ⅲ. ①多媒体－软件工具，Direct ②游戏－应用程序－程序设计 Ⅳ. ①TP311.56 ②G899

中国版本图书馆 CIP 数据核字（2004）第 139953 号

第九艺术学院游戏开发系列

Direct3D 游戏编程入门教程（第二版）

◆ 著 [德] Wolfgang F.Engel
译 周惟迪 徐 翎 张璐意
责任编辑 李 岚

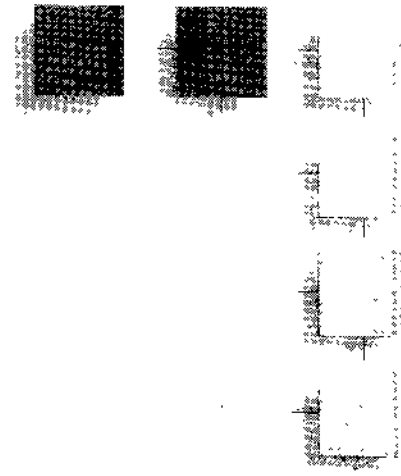
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
读者热线 010-67132687
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1000 1/16
印张：22.75
字数：501 千字 2005 年 1 月第 1 版
印数：1-4 000 册 2005 年 1 月北京第 1 次印刷

著作权合同登记 图字：01－2004－2592 号
ISBN 7-115-12963-0/TP・4368

定价：46.00 元（附光盘）

本书如有印装质量问题，请与本社联系 电话：（010）67129223



内 容 提 要

本书是基于DirectX 9.0编写的，是介绍Direct3D技术的入门教程。它详细介绍了Direct3D的背景、基本知识以及3D程序设计的相关概念与技术。全书由浅入深，分为四个部分，第一部分介绍Direct3D的发展历史和基本知识，对3D程序设计中的术语、概念进行了深入讲解；第二部分着重讨论纹理映射技术，包括多重纹理等；第三部分通过多个示例来探讨高级shader编程，并对.MD3文件格式做了详细分析；第四部分是附录，对本书中用到的概念和技术背景进行了简单补充。

本书适用于略有DirectX基础的C/C++程序员学习Direct3D游戏开发，也适用于具有Direct3D开发经验的程序员参考。

译　序

作为以游戏开发为主题的书籍，本书介绍了许多很 cool 的 3D 编程技术。这也促使我决定写一篇与之相当的开场白。但大约 3 分钟前的一次断电葬送了先前那篇很 cool 的译序。然而也正是由于这个插曲，让我在从头再写的时候想到了一些其他的话题，这些内容可能比单纯地向你形容那些 3D 技术会使你的游戏变得多 cool 更有意义。☺

3D 技术可谓是势不可挡！它在当今计算机游戏中的应用，几乎可以用“遍地开花”来形容。不信？随手打开一本喜爱的游戏杂志，留意一下有多少款游戏是采用 3D 技术来呈现的。很多，不是吗？所以，如果你想在当今的游戏程序开发行业中继续生存和发展下去，学习 3D 编程技术是势在必行的。如果你还是一个满怀憧憬、天天向上的“新兵”，那么掌握 3D 编程技术也将是你进入这个行业的一块有力的“敲门砖”。在众多 3D 编程技术中，“显示”、“渲染”显然是最重要的几个技术词汇。而本书也将就这些技术进行介绍。

就 3D 图形编程而言，当今主要流行着两套不同背景的 API：OpenGL 与 Direct3D。

中国人素有的“尊老美德”促使我将 OpenGL 排在前面。它是上个世纪 90 年代问世的，因此 OpenGL 具有来自于各个软/硬件厂商的广泛支持和认可。同时作为使用 OpenGL 的开发者，他们可以很方便地找到大量关于 OpenGL 的学习参考资料，诸如各种书籍、文章、实用程序、示例代码和网络资源。并且由于 OpenGL 采用了 C 语言的函数调用形式，所以学习起来也相对容易。

随着 DirectX 8.0 的问世，Direct3D 这个后生小辈开始第一次超过了 OpenGL（虽然还有很多人不愿意承认这点）。但是，对于某些初学者来说，Direct3D 依然是较难掌握的。例如，在初学时可能会有两只“拦路虎”——C++和 COM，以及一些 Windows SDK 编程的经验。从另一方面来说，Direct3D 相关的图书相对较少，不过这种情况在这几年开始已经有所改善。国外相关领域的专家们陆续出版了许多与 Direct3D 相关的著作，从基础的 3D 数学&图形学、Direct3D 中固定功能的流水线，直到高端/专项的 shader 编程、动画技术、地形渲染、用户界面、游戏引擎等。这批著作大大充实了 Direct3D 和游戏编程方面的教材及文献参考。而与之

形成鲜明对比的是国内相对“贫瘠”的 Direct3D 图书资料。所以，在国内对于 Direct3D 技术以及游戏编程技术的普及和教学，仍然“任重而道远”。

本书定位于 Direct3D 初学读物，但这并不意味着它会从零开始教会你所有的东西。因此你最好具备一些基础的 C/C++编程、线性代数以及空间解析几何方面的知识。如果你还不太了解这方面的内容，可以先参看本书的附录，里面相关章节提供了对这些内容的简要介绍。但毕竟那么几十页的内容不能完全替代这些领域的权威著作。所以，在深入阅读和学习本书之前，你还是应该对这些基础知识多投入一点精力。而这些投入也将会在你漫长的职业生涯中转变为你所具备的几项最有价值的技能。

由于译者水平有限，翻译过程中难免会有一些错误。笔者向几位知名的前辈译/作者们学习，决定继续跟踪中文版的出版与发行，同时征集广大读者对书中翻译错误的指正或改进建议，以求进一步完善此中文版。你可以给我发电子邮件指出书中的错误，也可以访问我的个人主页来检索一些相关资料。在我的主页上你可以很方便地找到本书相关页面的入口，我会将已证实的勘误和相关资料发布在上面。另外，如果你也是一位致力于 3D 图形或游戏开发的程序员，我也很希望结识你们，届时还请不吝赐教。

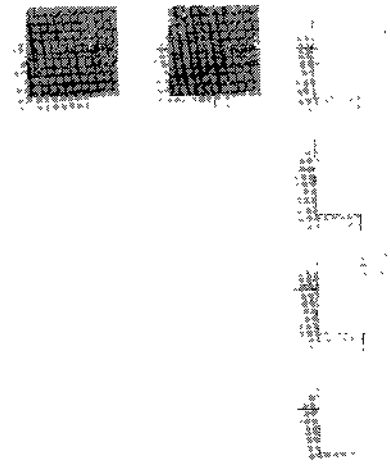
最后是再传统不过的感谢之词。这似乎已成为了“序”中不可或缺的一部分。而我直到现在才真正体会到，这样的言词不是为了去符合某种形式，而是为了能让大家知道、也为了记录下那些为本书中文版的翻译、编辑和发行付出过努力的人们。我要感谢的这些人，他们是：张璐意，我的表姐，她在有限的业余时间内完成了本书第一部分～第三部分（第 1 章～第 12 章）的初译稿；徐翎，我的好友兼同事，随着交稿日期的临近，他在百忙之中抽出时间分担了我的翻译压力，基于初译稿完成了对第 1、3、4 章的修正和附录 B～F 的翻译；

人民邮电出版社的编辑和其他工作人员，他们给本书带来了更通畅的文字和更美观的版式，特别感谢责任编辑李岚对我的耐心和指引。

再要感谢的是那些没有直接参与本书出版工作的幕后支持者们，他们或是在精神上给予了我很大的鼓励，或是在技术上给予我很多的支持。他们是：我的女友 — 张骥；我的领导 — 齐向中；我的好友兼“技术支持” — 史晓明（顺便说一句，他正在翻译的 *Massively Multiplayer Game Development* 也即将由人民邮电出版社出版）；最后是本书的原作者 — Wolfgang，感谢他在翻译过程中对我的耐心解答，并感谢他为我们带来这本书！

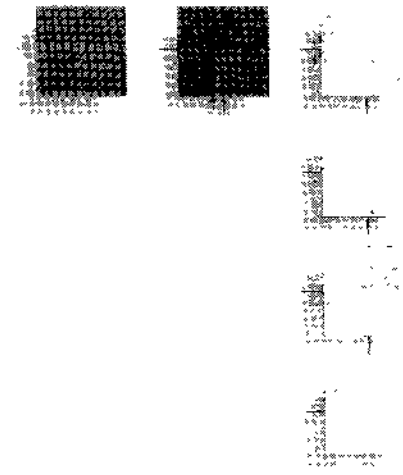
（左起周惟迪、张璐意、徐翎）

周惟迪 2004 年 11 月 12 日于上海
zhouweidi@hotmail.com
http://www.zhouweidi.name



关于作者

Wolfgang F. Engel 是几本编程书籍的合著者和编著者。他还在 http://www.gamedev.net、http://www.direct3d.info 及一些其他网站上撰写过一些在线教程。他曾在 GDC 2003 和在哥本哈根举行的 Vision Days 上作过演讲。Wolfgang 还是“游戏娱乐技术学院”（Academy of Game Entertainment Technology，http://www.academyofget.com/html/advisors.html）的一位指导教师。



来自系列书编辑的话

本书的第一版获得了很大成功，这促使我们考虑去跟进推出一个新的版本，使其再多涵盖一些高级主题，如 shader 编程和顶点编程，同时将内容更新到 DirectX 9.0。第二版仍以缓速切入，解释一些 Direct3D 编程的关键元素。不过后续内容被进一步加深了，将向你介绍如何基于 shader 编程来创造一些更高级的效果，shader 编程使用到了新的 shader 编程语言，即微软的高级 shader 语言（High-Level Shader Language，HLSL）。另外，我们考虑了为动画网格加入一些新的关于 *Quake 2* 和 *Quake 3* 文件格式的内容。

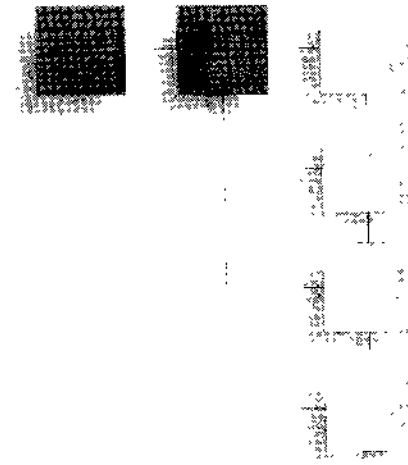
如果你正在寻找一本 Direct3D 方面的入门书籍，又不想被淹没在众多一般的 DirectX 问题中或 Windows 编程中，那这本书就是为你准备的。

André LaMothe
“游戏开发系列”编辑

致　　谢

没有许多人的帮助本书是无法写成的。特别我要感谢我的父母，他们给了我一个美好舒适的童年；我的妻子 Katja，感谢她对一个忙碌作者的容忍和耐心；还有我们快 2 岁的女儿 Anna，她让我明白了许多在人生中所要经历的事情。

另外，我要感谢那些协助本书完成面世的人们，他们是：Heather Hurley、Mitzi Koontz、Emi Smith、Cathleen Snyder、Heather Talbot 和 André LaMothe。谢谢你们对我的耐心以及在 GDC 2003 上的那段美妙时光。



前　言

在 1993 年获得了第一个学位（法律专业）后，我感到非常自豪，但也有些厌倦长时间的学习生活，于是我便决定放松一下自己，去玩一下 NovaLogic 公司开发的一个新的游戏 *Comanche*（卡曼奇）。

从 1 月 11 日晚上开始，我一连玩了近三天，中间只睡了几个小时，并因此萌生了从事计算机游戏编程的想法。我的目标是编写一个类似于 *Comanche* 的地形引擎（terrain engine）。我当时的女朋友（现在的妻子）非常诧异——一个刚毕业的年轻律师竟然要去当游戏程序员！

大约两年后，我成为了 CompuServe 上 Gamedev 论坛的一名成员，阅读了 André LaMothe 写的一些关于游戏编程的书籍和 Peter Freese 的一篇关于“高度映射”（height-mapping）[①]引擎的妙文。我完成了那个引擎运行在 OS/2 上。还为一些德语刊物写了一些关于 OpenGL 和 OS/2 游戏编程的文章，与他人合著了一本德语书籍，并开始从事 Windows 上的游戏编程。

在 1997 年，我写了第一篇关于 DirectX 编程的在线指南，并将其发布在自己的网站上。在与 John Munsch 以及其他 http://www.gamedev.net 的管理员协商后，我决定让更多人也能通过他们的网站看到我的在线指南。在 1998 年夏天，作为一名 DirectX 6.0 SDK beta 版的测试者，我决定撰写自己的第一篇关于 Direct3D 立即模式（Immediate Mode）框架下的操作指南。那时我用 http://www.netit.net 作为我的 URL，那里的一个邮件列表中聚集了很多对此感兴趣的人，许多人都通过 E-mail 给予了肯定的反馈。

事情开始变得十分有意思了。在 1999 年，我建立了自己的新网站 http://www.direct3d.net（现为 http://www.direct3d.info），专门用于给大家提供易于理解并具指导性的 Direct3D 编程指南。

这也是我写本书第一版的目的——帮助读者理解和学习 DirectX 图形编程。自本书第一版问世已有两年多时间了。两年就实时图形行业以及我个人而言都是相当漫长的一段时间。

译注①：一种根据位图中的每个像素信息来产生具有高低起伏的 3D 地形的有趣技术。

现在，我和我的妻子已有了一个女儿，这个小家伙甚至在交稿日追在眉睫时还缠着她的父亲。我写了一系列更有深度的关于顶点 shader 和像素 shader 编程的文章，发布在 http://www.gamedev.net 上。我还与 26 位作者合著了一本名为 *ShaderX—Vertex and Pixel Shader Tips and Tricks* 的书，本书着重于讲述高级 shader 编程。*Beginning Direct 3D Game Programming* 和 *ShaderX* 的巨大成功给我带来了新的挑战。其中之一便是在“2003 游戏开发者年会”（2003 Game Developers Conference）上就“介绍使用 HLSL/Cg 进行 Shader 编程”这一主题的演讲。另外，我也经常把 shader 技术用于几个正在进行的软件项目中。不过现在让我们回到正题……

如今，像 ATI RADEON 9800 PRO 和 NVIDIA GeForce FX 这些显卡的速度已远远超过两年前的显卡。顶点和像素 shader 已成为了每一个即将到来的游戏不可或缺的部分，由此它们也是游戏开发过程中不可或缺的一部分。这一事实以及大量的 E-mail 使我重新思考本书第二版的教学结构。首先，许多人认为在本书第一版中对于如何安装开发环境和 DirectX SDK 的解释不够详尽。此外，顶点和像素 shader 中那些新的必备特性也会使得哪怕最简单的 3D 例程的开发变得复杂化。因此，此版安排了一个对准备开发系统的扩展介绍。并且前几章主要讲解关于“固定功能的流水线（fixed-function pipeline）”的各项话题，以使讨论不会过于复杂。我将在后面才开始讲解顶点和像素 shader 编程。不幸的是 Amir Geva 在此版中没有参与编写关于物理和碰撞检测（collision detection）的章节。如果你有兴趣使用他的 DirectX 碰撞检测库（ColDet），我建议你阅读本书的第一版或访问他的网站 http://www.photoneffect.com。

我希望你们能像我编写本书一样愉快地阅读它，并随时发送 E-mail 给我。正如你们所知，本书的许多改进和新特色都归功于你们在 E-mail 中的想法，因此你们 Wolf（wolf@direct3d.net）将使我在将来新一版中更进一步地改进本书。

你将学到什么

本书涵盖了所有在 Windows95/98/Me/NT/2000/XP 或基于 Windows，为 PC 编写 Direct3D/DirectX 图形游戏所需要的元素，包括：

- 3D 图形和算法；
- 游戏编程技术和数据结构；
- 使用 3D 文件来构造游戏世界；
- 用角色动画系统（character animation system）来编写你自己的角色引擎；
- DirectX 图形编程；
- 还有更多！

你需要知道些什么

本书假设你会用 C 语言和少量的 C++编程。我只会运用一些简单的 C++特性，就像微软的工作人员在 DirectX SDK 中编写 Direct3D 立即模式的示例代码一样。如果你需要复习一下 C++，在附录 B 中有一个不错的 C++入门介绍可供参考。

针对图形/游戏编程的数学并不难掌握。只要你懂得加减乘除，最好还有平方，那你就可以理解本书中出现的 90%的数学知识。在附录 C 中有数学的初级介绍，它可确保你能看懂书中的一切数学。

本书是如何组织的

本书共四个部分。第一部分阐述了 Direct3D 游戏编程的本质，此部分主要介绍编程约定、基本算法、纹理贴图（texture-mapping）基础、3D 数学、变换流水线（transformation pipeline）、光照（lighting）以及深度缓冲（depth buffer）。

第二部分，你将学习如何用变换和光照流水线将纹理以不同的效果贴图到对象上。对所有的专业词汇，如凹凸贴图（bump mapping）、环境贴图（environment mapping）、过程贴图（procedural mapping）都会进行详细解释，并在示例程序中展示其效果。

在本书的第三部分，你将了解到文件格式以及如何将它们整合到你的游戏引擎中去。使用的文件格式是已经过显著增强的.X 文件和 MD3 文件，前者是由 DirectX 8.0 SDK 引入的，后者则是一种被许多游戏所采用、由 *Quake 3* 引擎驱动的文件格式。另外，还将在一个角色引擎中展示一种增强型的 MD3 文件格式。

这部分还着重于从头讲述顶点和像素 shader 编程。将告诉你如何使用 DirectX 9 中新增的高层 Shader 语言（High-Level Shader Language，HLSL）。

第四部分是附录，将有助于你温故 Windows 编程、C++和数学知识。

使用 CD-ROM

附录中有关于 CD-ROM 的使用介绍，但在此我先做个简要介绍。本书附带光盘中包括了书中的所有代码、微软 DirectX 9.0 的 C/C++ SDK 以及 DirectX 9 开发者运行时库(runtime)，这些在你编译和运行书中所讨论的示例程序时都需要先安装（见下一段的安装指导)。DirectX SDK 包含运行时（run-time）文件、头文件和库（library）文件，这些文件都是你编译书中示例所必须的，另外 SDK 还为每个 DirectX 组件提供了许多示例程序。

你可以在以每章编号命名的目录下找到书中的示例代码。在每个示例目录下，可以看到

图形及源文件。里面还有一个 readme.txt 文件，提供了一些编译这些应用程序的附加信息。

你需要一个能支持 DirectX 运行示例程序的系统，最低要求如下：

- 带有 Service Pack 2 或更高版本的 Windows 2000。（我推荐带有最新 service pack 的 Windows XP 专业版，因为只有专业版才允许安装用于开发 DirectX 应用程序的 Debug 运行时库）。
- 带有 Service Pack 5 的 Visual C/C++6.0（我推荐带有最新版本 Service Pack 的 Visual C++ .NET）。
- 有至少 256MB 的 RAM。
- 有至少 600MB 剩余硬盘存储空间（越多越好）。
- 有一块硬件加速的 3D 显卡。为了获得示例程序的最佳视觉效果，你需要相对较新的图形硬件。像素 shader 示例只能在与 DirectX 8.1 兼容的硬件上运行。有些示例需要显卡能支持某些 DirectX 9 的特性。
- 有最新的显卡设备驱动。

请下载最新的显卡设备驱动。通常，较新版本的驱动程序 bug 较少，运行性能更好。我的三台电脑都使用了 Windows XP 专业版，内存分别为 256MB、512MB 和 1GB，处理器分别为 Pentium III 1GHz、Pentium IV 2GHz 和 Athlon 1700+。这些示例是由带 Service Pack 5 的 Visual C/C++ 6 和 Visual C++ .NET 开发的。我在 NVIDIA GeForce3 和 GeForce4 TI、ATI RADEON 8500、9000 Mobility 和 9700 PRO 显卡上测试过所有示例。

安装 DirectX SDK

要安装 DirectX，先要找到本书附带光盘中一个名为 DirectX 的文件夹，该目录下的两个文件如图 I.1 所示。

图 I.1　安装 DirectX 9 所需的两个文件

当你点击 dx90_sdk_devruntime.exe 时，将会把一些文件解压缩到指定的临时目录，如图 I.2 所示。

我选择了桌面上一个名为 dxruntime 的文件夹，在解压缩文件至相应的文件夹后，便会在此文件夹中看到一个名为 SDKDev 的文件夹，如图 I.3 所示。

SDKDev 文件夹中包含两个名为 Debug 和 Retail 的文件夹。你应该选择 Debug 文件夹以尽可能多地获得调试信息。Retail 版本中去掉很多调试检查，因此能编译出运行速度更快的程序。如果你只是要玩 DirectX 游戏，而不是自己编写一个游戏的话，Retail 版本是最佳选择。Debug 版本则是帮助编码人员来了解和运行 DirectX 应用程序的版本，它通过给出调试信息

来帮助你跟踪发现错误。然而这也是有得有失的，你必须为此付出性能代价，运行要比 Retail 版本慢。

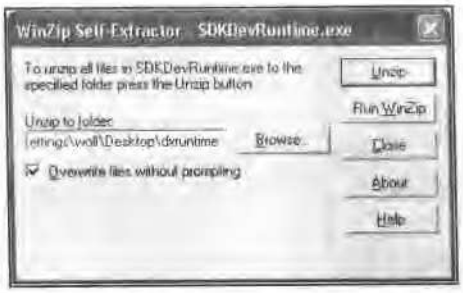


图 I.2　创建临时文件夹 dxruntime

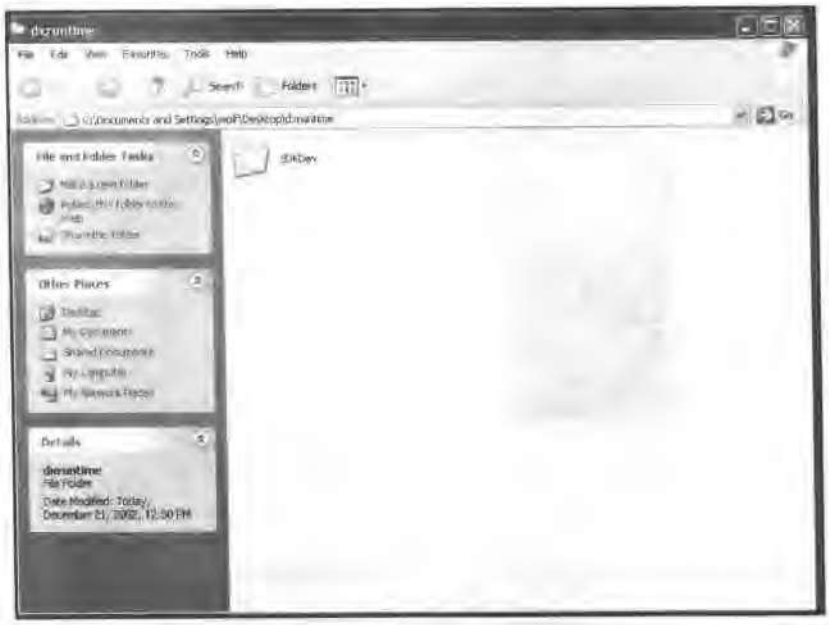


图 I.3　SDKDev 目录

在 Debug 文件夹中，你会看到一个名为 dxdevrtm.exe 的文件。运行这个文件就会开始安装过程，如图 I.4 所示。

接下来，你没有太多别的选择，只有选择接受常规许可协议并开始安装过程。最后，当你点击对话框中的“Finish”按钮后，将被强制要求重启计算机，如图 I.5 所示。

在重启后，最好从桌面上把临时文件夹 dxruntime 删除掉。

在将 DirectX 9 安装到你的计算机后，下一步就是安装 SDK。点击第二个名为 dx9sdkcp.exe 的文件，再次选择一个临时文件夹，我建议使用 dx9sdktemp 作为文件夹名放在桌面上，如图

I.6 所示。

图 I.4　DX9 运行时库安装过程的第一步

图 I.5　安装程序中的重启对话框

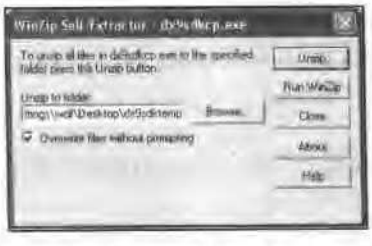


图 I.6　解压缩 SDK 文件到 dx9sdktemp

然后点击临时目录下名为“SKD(C++)”的文件夹，如图 I.7 所示。

这个文件夹的内容如图 I.8 所示。

点击 setup.exe 开始安装程序。在安装过程中，你要接受许可协议，此外还可选择不同的

组件来进行安装。我建议安装全部组件并指定你的安装目录（见图 I.9）。默认目录应该是“C:\DXSDK”。选择这个目录可以更容易地编译其他人开发的示例程序。

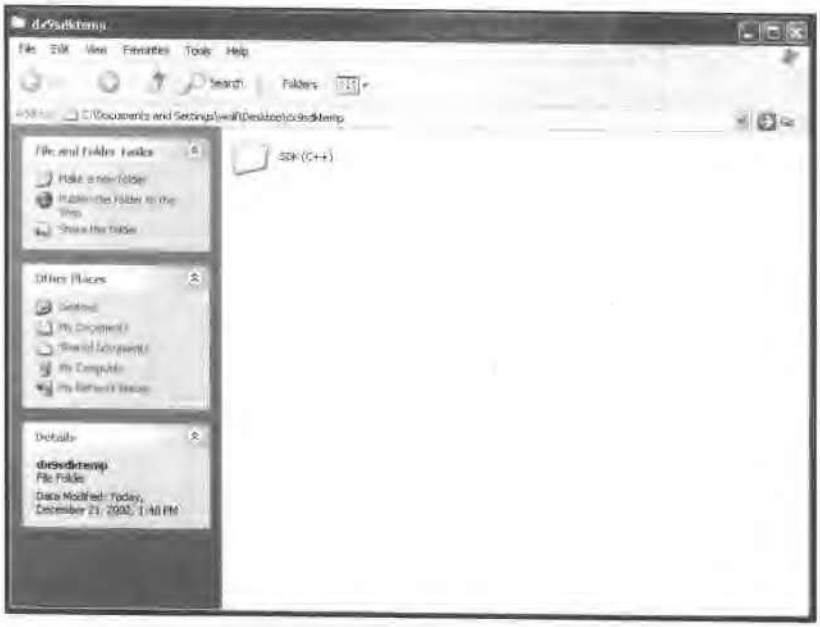


图 I.7 点击 SDK（C++）

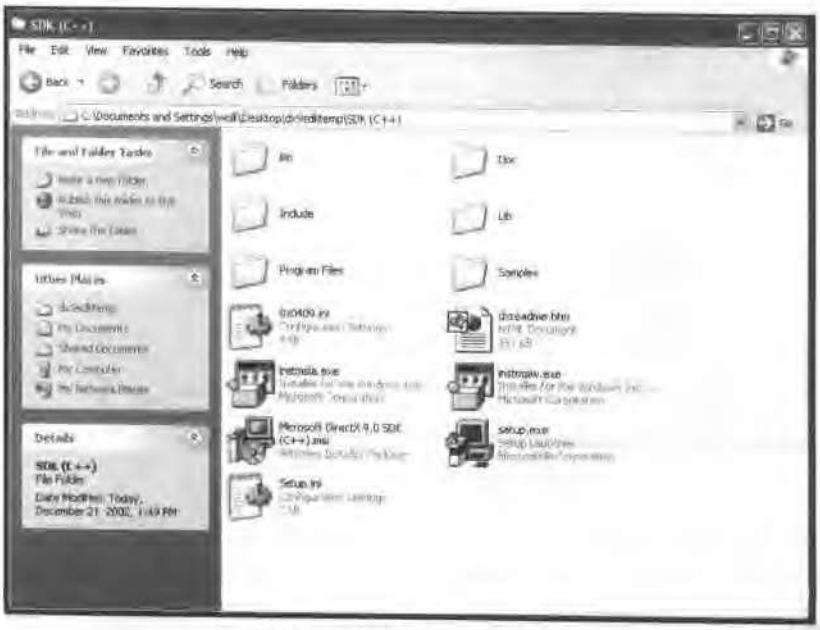


图 I.8 SDK（C++）的内容

在你点击安装向导的最后一个对话框中"Finish"按钮完成了安装程序后，你可以从桌面上把临时目录 dx9sdktemp 删除掉。

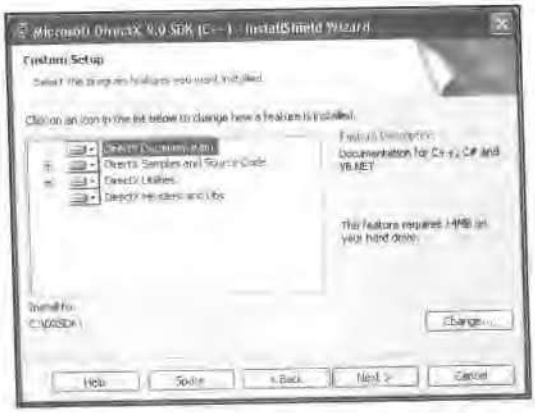


图 1.9　选择合适的部件和 DirectX 9 SDK 的目录

在你完成 DirectX SDK 安装后，看一下所有目录并熟悉一下库（library）文件、包含文件（include file）、帮助和示例的位置（见图 1.10）。你需要在设置 Visual C++编译器时用到这些位置。

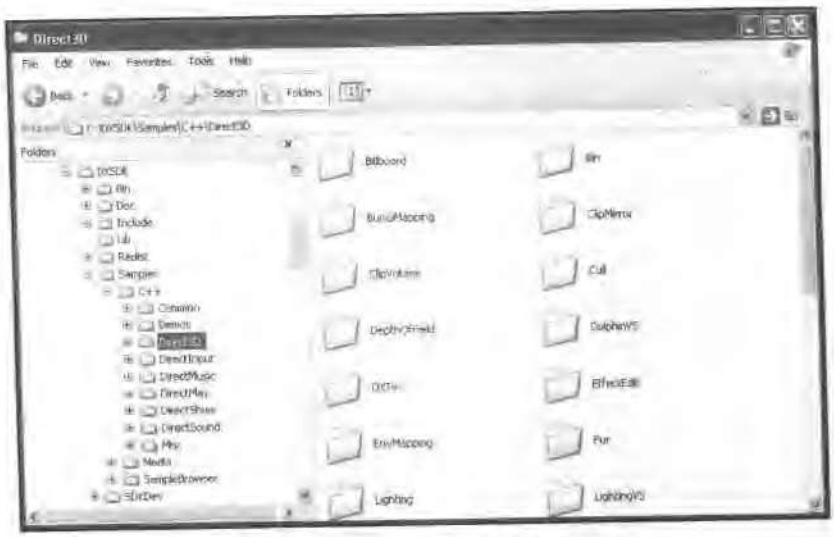


图 1.10　DirectX 9 SDK 中的重要目录

你可在"C:\DXSDK\Samples\C++\Direct3D"里找到 Direct3D 的示例程序文件。包含文件位于"C:\DXSKD9\Include"和"C:\DXSDK9\Samples\C++\Common\Include"，库文件在"C:\DXSDK9\Lib"。

上述内容和概念对于将来所有 DirectX 版本也同样适用，请时常留意一下

http://msdn.microsoft.com/directx 上的更新。

你还需为你的显卡安装最新的驱动程序，可以到显卡厂商的网站上找。最新的驱动程序通常有更高速度和稳定性，有时还会提供些新的特性。看一下厂商总会提供的 readme 文件以获取正确的安装指导。

设置 Visual C++ 6.0

安装完 DirectX 后，你需要考虑其余软件的设置。其中最重要的一项便是你的 IDE（Integrated Development Environment，集成开发环境）或你的编译器。我建议你使用 Microsoft Visual C/C++6.x 或更好的编译器。它可生成非常优秀的 Windows 代码。你可以在当地书店或软件商店购买到它，这是最省力的途径。

如果你是在安装 Visual C++ 6.0 后再安装 DirectX 9 SDK 的话，安装程序会为你正确地设置所有路径。尽管如此，以防万一，你应能检查正确的路径。接下来将逐步介绍对一个已安装的 Visual C++ 6.0 的配置过程。

首先，选择“Tools—>Options”，选择“Directories”标签页（见图示 I.11）。

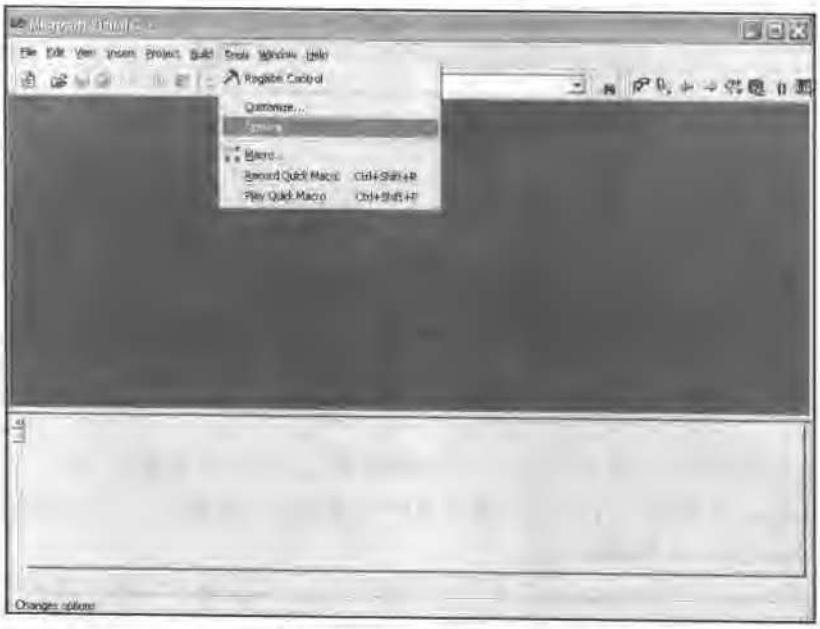


图 I.11 点击“Tools—>Options—>Directories”

你应该看到如图 I.12 所示的对话框。

include 文件目录为“C:\dxsdk\include”和“C:\DXSDK9\Samples\C++\Common\Include”。你可以点击“New”按钮来指定 Visual C++ IDE 为以上目录（见图示 I.13）。

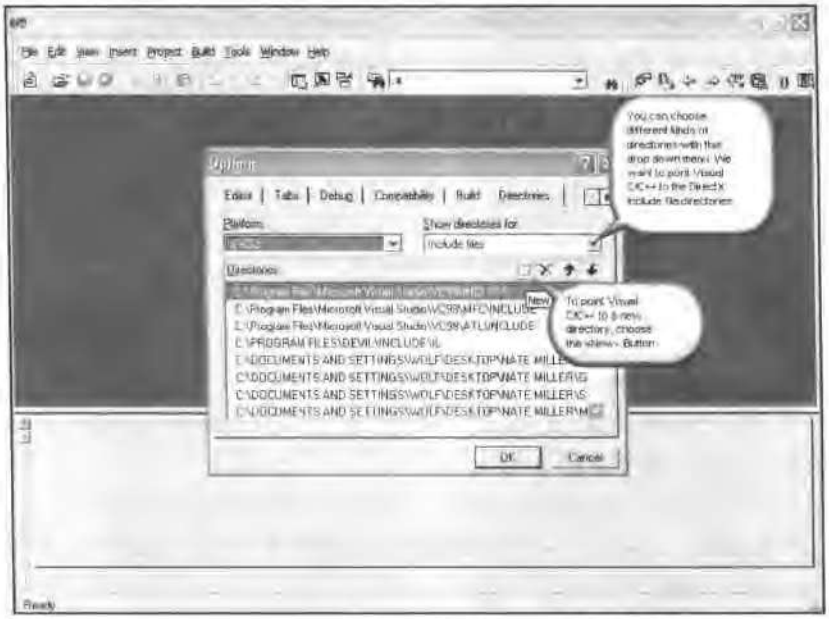


图 1.12　选项对话框

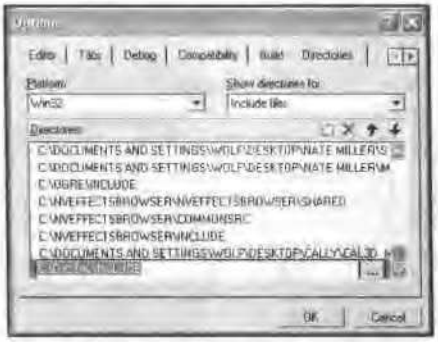


图 1.13　点击"New"按钮，并输入路径

这些目录应该始终位于列表的顶端，因为编译器将从顶端开始搜索文件。

在提供 include 文件路径后，你必须指定 IDE 链接器的搜索路径。这些库文件（library file）通常位于"C:\dxsdk\lib"（见图示 1.14）。

在配置完 Visual C/C++ 6 后，为确定配置正确，可以编译"C:\DXSDK\Samples\C++\Direct3D"目录下的任意一个示例。若你没有正确配置 include 路径，你将会看到类似下面的信息：

d:\book\source\part 1\chapter6\animated objects\objects.cpp(68): error C2146: syntax error: missing ';' before 'g_Keyboard_pDI'

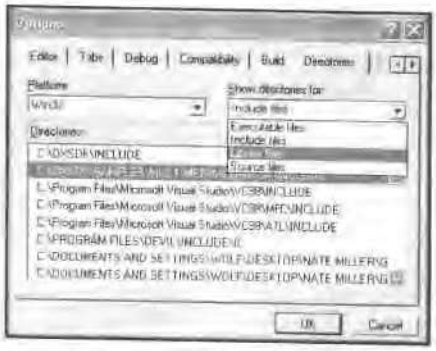


图 I.14 配置 library 文件的路径

即使提供了正确的头文件和库文件的路径，你可能仍需向开发环境的链接器提供文件的名字。这些 object/library modules（对象/库模块）的正确路径会在你的链接对话框中列出。要看到这个对话框，选择"Project—>Settings"，然后选择"Link"标签页。在 Category 列表的 General 中，有一个叫做"Object/Library Modules"的输入栏。它包括了所有链接至你当前开发的程序的库文件。它看上去应该类似于图 I.15。

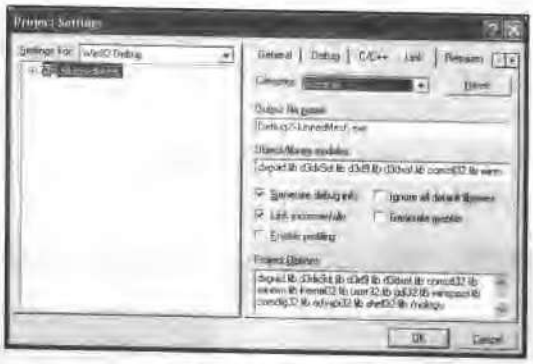


图 I.15 配置链接器路径

在这个输入栏中，你至少需要以下这些文件名：

- d3dx9.lib
- 3dxof.lib
- d3d9.lib
- winmm.lib
- dxguid.lib

如果你漏掉了其中的一个，将会出现类似于以下错误信息：

d3dapp.obj: error LNK2001: unresolved external symbol _Direct3DCreate9@4 Debug/SkinnedMesh.exe : fatal error LNK1120: 1 unresolved externals error executing link.exe.

在此，d3d9.lib 漏掉了。无法解析的外部符号是 Direct3D 的 COM（Component Object

Model，组件对象模型）接口的一部分。（我将在接下来的章节里详细介绍 COM）

你还需要检查另一个 include 路径，它囊括了对 C/C++编译器而言的通用包含文件（common include file）目录的路径。若你更换了机器或是在另一台使用了不同 DirectX 目录的机器上工作，通常就需要通过“Project—>Add Files to Project—>Files”来添加这些通用文件。

现在来编译我们的第一个项目：

（1）启动已配置好的 Visual C++开发环境；

（2）点击“Open Workspace”；

（3）选择 basics.dsp；

（4）检查是否配置好了如前所描述的目录路径；

（5）选择 Build/basics.exe 生成；

（6）一切正常运作后，选择运行 basics.exe。

这样就可以了。若出现错误，重新阅读一下安装 DirectX SDK 和设置 Visual C++两个小节，并查阅所提供的文档。

设置 Visual C++ .NET

类似 Visual C/C++ 6，若你安装了 Visual C++ .NET 以后再安装 DirectX 9 SDK 的话，DirectX 9 SDK 会自动为你设置所有路径。不过，也可能出现意外，本节就告诉你，当需要手动设置路径时，如何在 Visual C++.NET IDE 中配置 DirectX 9。

首先在 Visual C++ .NET IDE 中点击“Tools—>Options”，如图 1.16 所示。

图 1.16　选择“Options”菜单项

这里有一个 Project 项，包含了所有相关目录的路径，如图 I.17 所示。

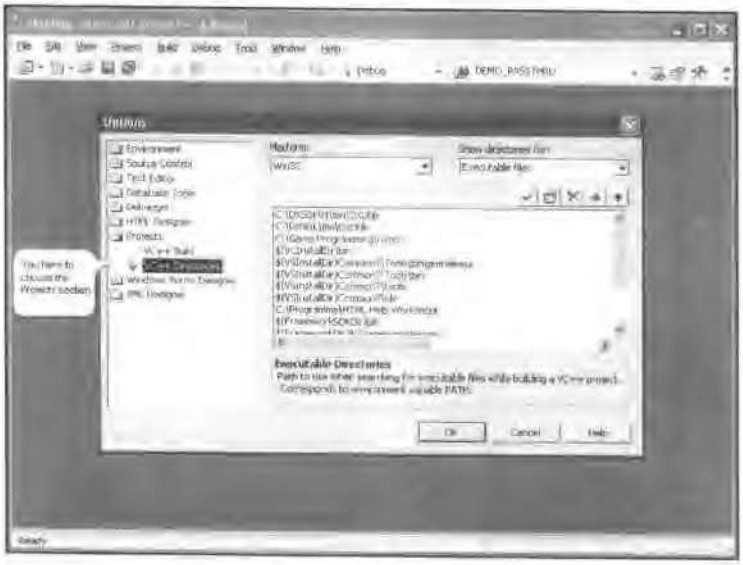


图 I.17　选择"Project"，VC++各种目录项

若你熟悉 Visual C/C++ 6 Studio 中使用的对话框，你就会觉得很轻松了，因为这个对话框使用相同的逻辑。你可在对话框右上角看到"Show directories for"下拉菜单，它提供了你可以引用的不同类型的文件。（见图 I.18）

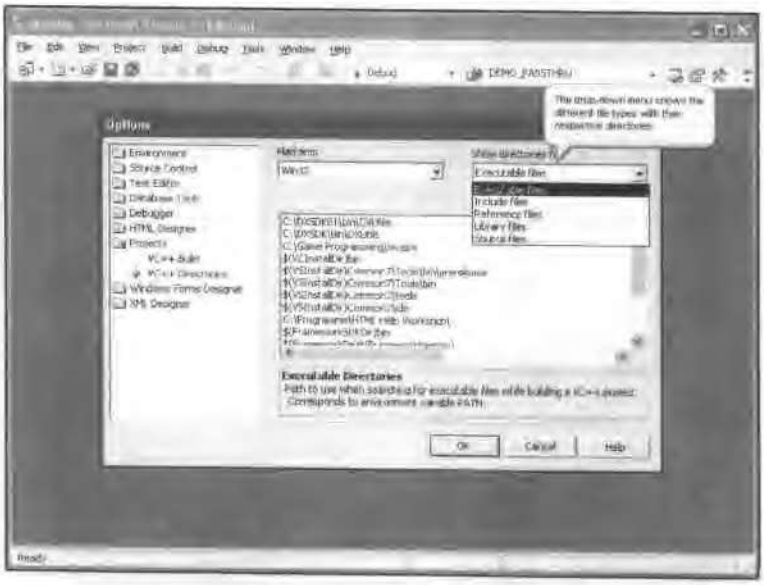


图 I.18　选定引用的文件类型

在你选定一种文件类型后，引用目录就会显示在对话框中央的那个大区域中。

include 文件目录为“C:\DXSDK9\Include”和“C:\DXSDK9\Samples\C++\Common\Include”。点击“New”按钮，然后按“省略号”按钮，便可指定 Visual C++ .NET IDE 的目录。接着会打开一个默认对话框，允许你选择一个目录。（见图示 I.19）

图 I.19　选择一个目录

现在用箭头按钮将此目录置于列表顶部。在定位了 include 文件的目录之后，选择路径“C:\dxsdk\lib”给 library 文件正确定位。勿忘用箭头按钮将它们提到列表顶部。

附加依赖文件的正确路径应列在“Project—>Properties—>Linker—>Input—>Additional Dependencies”字段中，在这个记录字段中，你至少需要输入以下几个文件（见图示 I.20）：

- d3dx9.lib
- d3dxof.lib
- d3d9.lib

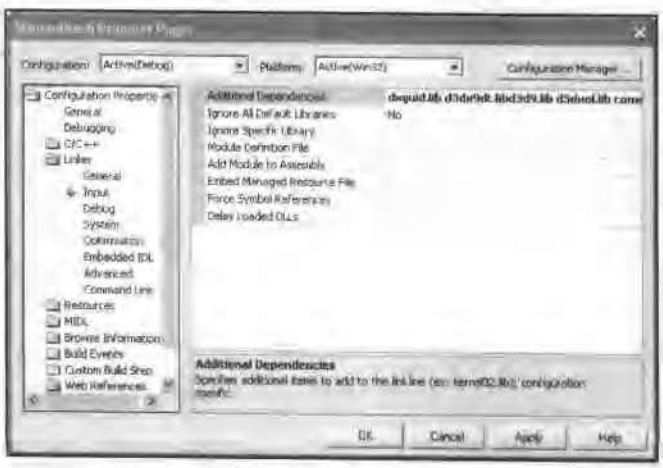


图 I.20　选择附加依赖文件

- winmm.lib
- dxguid.lib

若出现任何问题，翻阅“设置 Visual C++ 6.0”尾部段落来解决常见问题。解决 Visual C++ .NET 问题的方法与其相同。

现在让我们编译第一个项目：

（1）启动已配置好的 Visual C++ .NET 开发环境；

（2）点击“Open Solution”；

（3）选择 basics.sln；

（4）检查是否配置了如前所述的目录路径；

（5）选择“Debug—>Start Without Debugging”；

（6）当一切运作正常时，basics.exe 将会启动。

这样就可以了。若出现错误，重新阅读“安装 DirectX SDK”和“设置 Visual C++ .NET”部分，并查阅所提供的文档。

DirectX Shader 调试器（Debugger）

为调试顶点和像素 shader，你需安装 DirectX SDK Shader 调试器。这个调试器只适用于 Visual C++ .NET。你可以在 DirectX 安装程序对话框中找到，如图 I.21 所示。

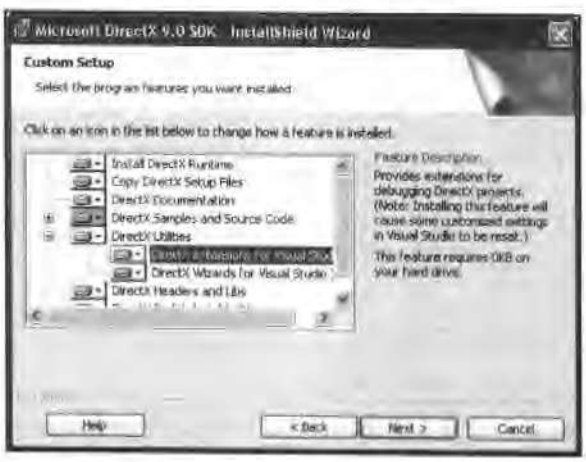


图 I.21 点击“DirectX Extensions for Visual Studio 7”来选定 shader 调试器

默认情况下，shader 调试器是未选中的，因此你需要显式地选择安装 shader 调试器。若计算机中只安装了 Visual C/C++ 6，你根本就不会看到这一项，对话框看上去应如图 I.22 那样。

请注意只有 Visual C++ .NET 用户才能使用 shader 调试器。

图 1.22　Visual C/C++ 6 下可安装的"DirectX Utilities"

其他工具

有两个即插即用型（plug-and-play）编译器（由 Intel 和 Codeplay 提供）可用于替换 Visual C/C++ 6 或 Visual C++ .NET IDE 的默认编译器。它们可为游戏程序员提供比 IDE 的编译器更好的性能。然而，我不建议初学者使用这些编译器，因为它们会使某些事情变得复杂化。你可以在 http://www.intel.com/software/products/compilers 和 http://www.codeplay.com 上找到更多相关信息。在 Intel 的网站上，不要忘记下载 VTune，它是一个可以帮助你优化游戏的分析器（profiler）。VTune 及微软 Visual C++编译器提供的这些分析器是如今大多数 PC 游戏程序员使用的基本开发工具。

附加资源

本书附带 CD-ROM 包括了一个名为"Movies"的目录。这些影片交互地向你展示了如何配置 Visual C/C++ 6 和 Visual C++ .NET 的默认路径。

你应该每隔一段时间就去访问一下微软 MSDN 网站（http://msdn.microsoft.com/directx），和邮件列表（http://discuss.microsoft.com/SCRIPTS/WA-MSN.EXE?SI=DIRECTXDEV）。你还可在 http://www.gamedev.net 或 http://www.flipcode.com 上找到关于游戏开发者社区的每日新闻。我还在 http://www.direct3d.info 上提供了其他的信息。

3.2 命名约定 …… 18
3.3 调试 DirectX …… 20
3.4 返回码 …… 21
3.5 小结 …… 22

第 4 章 3D 基础、Gouraud 着色、纹理映射基础 …… 23

4.1 3D 基础 …… 23
4.2 理解顶点 …… 25
4.3 使用朝向 …… 25
4.4 理解面（face） …… 26
4.5 理解多边形（polygon） …… 26
4.6 理解法线（normal） …… 27
4.7 理解法线和 Gouraud 着色 …… 27
4.8 纹理映射（Texture-Mapping）基础 …… 28
4.9 小结 …… 30

第 5 章 基础 …… 31

5.1 编译示例 …… 31
5.2 DirectX Graphics 通用体系结构 …… 33
5.3 Basic 示例 …… 34
5.3.1 ConfirmDevice()、OneTimeSceneInit()和 InitDeviceObjects()函数 …… 37
5.3.2 RestoreDeviceObjects()方法 …… 37
5.3.3 FrameMove()函数 …… 45
5.3.4 Render()函数 …… 46
5.3.5 InvalidateDeviceObjects()函数 …… 49
5.3.6 DeleteDeviceObjects()函数 …… 50
5.3.7 FinalCleanup()函数 …… 50
5.4 Basic2 示例 …… 51
5.4.1 InitDeviceObjects()函数 …… 53
5.4.2 RestoreDeviceObjects()函数 …… 54
5.4.3 Render()函数 …… 56
5.4.4 InvalidateDeviceObjects()函数 …… 58
5.4.5 DeleteDeviceObjects()函数 …… 58
5.4.6 FinalCleanup()函数 …… 58
5.5 Basic3 示例 …… 59
5.6 Basic4 示例 …… 60

5.7 Basic5 示例 …… 66
5.8 小结 …… 66

第 6 章 动画初步 …… 67

6.1 理解变换与视口 …… 67
6.1.1 世界变换 …… 68
6.1.2 观察变换 …… 88
6.1.3 投影变换 …… 94
6.1.4 使用视口 …… 96
6.1.5 深度缓冲（Depth Buffering） …… 99
6.2 附加资源 …… 102
6.3 小结 …… 103
6.4 第一部分提问 …… 103

第 2 部分 深入 DirectX 图形编程

第 7 章 纹理映射基础 …… 109

7.1 纹理的要点是什么 …… 109
7.2 使用纹理坐标 …… 112
7.3 使用纹理寻址模式（texture-addressing mode） …… 114
7.3.1 包装纹理寻址模式 …… 114
7.3.2 镜像纹理寻址模式 …… 116
7.3.3 夹持纹理寻址模式 …… 117
7.3.4 边框颜色纹理寻址模式 …… 118
7.3.5 一次镜像纹理寻址模式 …… 118
7.4 纹理包装（Texture Wrapping） …… 118
7.5 纹理过滤和抗锯齿（Anti-Aliasing） …… 120
7.5.1 Mipmap …… 121
7.5.2 最近点采样 …… 122
7.5.3 线性纹理过滤 …… 122
7.5.4 各向异性过滤 …… 123
7.5.5 抗锯齿（Anti-Aliasing） …… 125
7.5.6 alpha 混合（blending） …… 126
7.6 小结 …… 128

第 8 章　使用多重纹理 …… 130

8.1　多次渲染（Multipass Rendering） …… 130
8.2　颜色操作 …… 133
8.2.1　黑暗映射（Dark Mapping） …… 134
8.2.2　黑暗贴图动画 …… 136
8.2.3　混合纹理与材质漫反射颜色 …… 137
8.2.4　混合黑暗贴图与材质漫反射颜色 …… 139
8.2.5　发光映射（Glow Mapping） …… 140
8.2.6　细节映射（Detail Mapping） …… 142
8.3　alpha 操作 …… 145
8.4　多重纹理支持 …… 147
8.5　纹理管理 …… 147
8.6　附加资源 …… 148
8.7　小结 …… 148
8.8　第二部分提问 …… 149

第 3 部分　DirectX 图形编程核心

第 9 章　使用高级 Shader 语言进行 Shader 编程 …… 155

9.1　开始 HLSL 编程前需要先做些什么 …… 156
9.2　顶点/像素 Shader 的任务 …… 157
9.3　通过 HLSL 实现常用光照公式 …… 158
9.3.1　环境光照 …… 158
9.3.2　漫反射光照 …… 160
9.3.3　镜面反射光照 …… 163
9.3.4　自身遮蔽阴影项（Self-Shadowing Term） …… 167
9.3.5　凹凸映射（Bump Mapping） …… 168
9.3.6　点光（Point Light） …… 170
9.4　小结 …… 171

第 10 章　更多高级的 Shader 效果 …… 172

10.1　使用立方体贴图 …… 172
10.1.1　生成立方体贴图 …… 173
10.1.2　访问立方体贴图 …… 173
10.2　实现 …… 174

10.3　折射/反射环境映射 …… 176
10.3.1　动态折射/折射环境映射 …… 178
10.3.2　凹凸的动态折射/反射环境映射 …… 181
10.4　使用阴影 …… 183
10.4.1　阴影体（Shadow Volume） …… 184
10.4.2　使用阴影体所需考虑的问题 …… 196
10.5　小结 …… 196

第 11 章　使用文件 …… 197

11.1　3D 文件格式 …… 197
11.2　X 文件格式 …… 198
11.2.1　首部（Header） …… 199
11.2.2　网格 …… 199
11.2.3　网格材质列表 …… 201
11.2.4　法线 …… 203
11.2.5　纹理 …… 204
11.2.6　变换矩阵 …… 209
11.2.7　动画 …… 212
11.3　使用 X 文件 …… 215
11.4　扩展 X 文件 …… 217
11.5　附加资源 …… 218
11.5.1　X 文件格式 …… 218
11.5.2　蒙皮网格（Skinned Mesh） …… 218
11.6　小结 …… 218

第 12 章　使用*.md3 文件 …… 219

12.1　文件格式探索 …… 219
12.1.1　Animation.cfg …… 223
12.1.2　.skin 文件 …… 225
12.1.3　纹理与 Shader 文件 …… 226
12.1.4　自定义声音 …… 231
12.1.5　.md3 格式 …… 232
12.2　CharacterEngine 示例程序 …… 235
12.3　更多的改进 …… 258
12.4　附加资源 …… 258
12.5　小结 …… 258

第 4 部分　附　　录

附录 A　Windows 游戏编程基础 …… 261

A.1　如何看待窗口 …… 261
A.2　Windows 如何与你的游戏交互 …… 261
A.3　窗口的组成部分 …… 262
A.4　窗口框架 …… 262
A.4.1　步骤 1：定义一个窗口类 …… 266
A.4.2　步骤 2：注册窗口类 …… 269
A.4.3　步骤 3：创建一个该类的窗口 …… 269
A.4.4　步骤 4：显示窗口 …… 272
A.4.5　步骤 5：创建消息循环 …… 272
A.4.6　窗口过程 …… 275
A.5　为游戏而优化的窗口框架 …… 276
A.6　窗口资源 …… 280

附录 B　C++入门 …… 284

B.1　什么是面向对象程序设计 …… 284
B.1.1　抽象 …… 285
B.1.2　类 …… 287
B.1.3　封装 …… 288
B.1.4　声明一个类 …… 289
B.1.5　构造函数 …… 292
B.1.6　析构函数 …… 292
B.1.7　类层次和继承 …… 294
B.1.8　虚函数 …… 298
B.1.9　多态性 …… 299
B.1.10　内联（inline）函数 …… 300
B.1.11　C++相对与 C 增强的内容 …… 301
B.2　附加资源 …… 306

附录 C　数学入门 …… 307

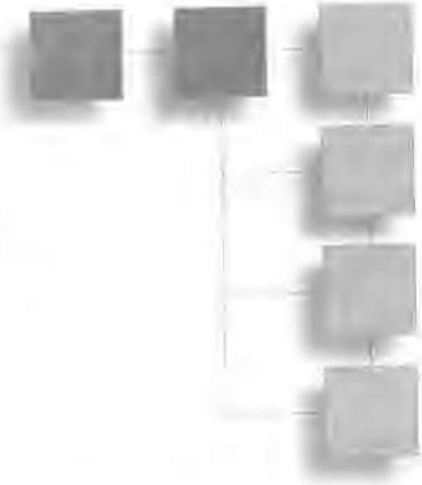
C.1　3D 中的点（Point） …… 307
C.2　向量（Vector） …… 309
C.2.1　受限向量（Bound Vector） …… 309
C.2.2　自由向量（Free Vector） …… 310

第 1 部分

DirectX 图形：Don't Hurt Me

- 第 1 章　Direct3D/DirectX 的历史
- 第 2 章　HAL 和 COM 概述
- 第 3 章　编程约定
- 第 4 章　3D 基础、Gouraud 着色、纹理映射基础
- 第 5 章　基础
- 第 6 章　动画初步

第 1 章



Direct3D/DirectX 的历史

本章将介绍 Direct3D 以及它长久以来功能发展的历史，这要从 DOS 和 PC 的历史说起，还讲述了三个英国学生是如何改变 3D 编程世界的故事。

在 Windows 之前，DOS 是 PC 上最流行的操作系统，曾经独霸游戏开发领域许多年。由于最初 Windows 的图形和音频性能让人难以接受，所以游戏开发者一直不愿在 Windows 系统下开发游戏。

DOS 提供了对硬件直接访问的能力，但同时也给自身带来了相应的复杂性。DOS 下的游戏必须支持所有的视频和音频硬件，这迫使游戏开发者仅仅为了在所有的 PC 之间提供一致的图形和声音，就要编写复杂的代码以支持数十种不同的配置。

在本章中，你将会了解到：

- DirectX 的历史，从较早版本直至当前最新版本的 DirectX；
- DirectX 8 中 point sprite（点精灵）以及顶点/像素 shader 的介绍；
- 3D 纹理的基础。

1.1 DirectX 2.0

自从 1995 年 DirectX 诞生以来，微软便能在 Windows 中提供以前只有 DOS 才具有的运行性能，并且不必为支持不同制造商的特定硬件而发愁了。现在，每个硬件制造商都会在交付他们的产品时附带 Windows 驱动程序。

作为 DirectX 的一部分，Direct3D 于 1996 年出现在 DirectX 2.0 中，它被设计为用于访问 3D 硬件加速器的高级图形功能，通过提供给程序员一个通用接口来促进设备无关性和硬件抽象。适当编写出来的 Direct3D 代码无论是现在还是将来都可以在 Direct3D 上工作。

让我们更深入地了解一下这段历史。在 20 世纪 90 年代初期，许多 PC 上的 3D 引擎都是

在英国设计的。其中有著名的 Renderware（http://www.renderware.com）、来自 Argonaut（http://www.argonaut.com）的 BRender（在 1994 年被移植到了 OS/2）以及一家叫作 RenderMorphics 的英国小公司。RenderMorphics 是在 1993 年由 Servan Keondjian、Kate Seekings 和 Doug Rabson 共同创立的，他们开发了一个叫作 Reality Lab 的产品。Keondjian 晚上在乐队里弹钢琴，白天则为他的 3D 引擎编程；Seekings 通过后来在 Middlesex 大学得到的计算机图形速成硕士学位提升了学历。有趣的是，当她将一个在 Keondjian 和 Rabson 帮助下开发的 3D 渲染库作为学校项目提交时却被评予了不及格，原因只是没有严格地遵循指定说明的要求。该小组在首次参加的一个交易展会上（SIGGRAPH 94）便引起了微软的关注，后来 RenderMorphics 在 1995 年 2 月被微软收购了。

收购了 RenderMorphics 之后，微软将 Reality Lab 集成到它的 DirectX API（Application Programming Interface，应用程序编程接口）家族。Reality Lab 的立即模式组件吸纳了当时的标准 3D Windows API——3—D—DDI。3—D—DDI 是由 Michael Abrash 创建的，后来他还在 id Software 协助下创建了 *Quake 1* 引擎。

1.2 DirectX 6/7

在 DirectX 8.0 出现以前，Direct3D 一直由两个截然不同的 API 组成：保留模式（Retained Mode）和立即模式（Immediate Mode）。当时使用立即模式 API 是比较困难的，但它是一个灵活的底层 API，能以最高的效率来运行。保留模式建立在立即模式之上，它提供了附加的服务，例如框架体系和动画等。保留模式相对于立即模式而言更易于学习和使用，但是由于程序员们需要立即模式所提供的更好的性能和灵活性，因此随着 DirectX 6.0 的发布，便停止了保留模式 API 的发展。

Direct3D 立即模式 7.0 比之 6.0，最大的变化就是支持了硬件加速的变换和光照，还有就是重新组织了光照、材质和视口对象，现在这些对象都可以调用 IDirect3DDevice7 的方法来直接设置。Direct3D 7 取消了在 Direct3D 6 中用于访问纹理的特殊接口，而用 IDirect3DrawSurface7 接口提供一种更为简单的管理纹理方法。

1.3 DirectX 8

DirectX 8.0 SDK 的到来带来了 Direct3D 历史上最大的改进。Direct3D 8.0 有一个全新的体系结构：DirectDraw 与 Direct3D 被集成到了一个叫作 DirectX Graphics 的接口中，由此促成了一个更小的内存占用量和一个更简单的编程模型，简化了数据的初始化、分配和管理。可以肯定地说，这些变化使 DirectX 8 第一次比 OpenGL 更易于使用、更具一致性。

DirectX 8 包括了以下这些新特征：

- Point sprite（点精灵，硬件支持的精灵对象）；
- 3D 体纹理（volumetric texture，三维纹理）；
- 对 Direct3DX 库的改进（提供许多有用且高度优化的例程）；
- N-patches（在模型上应用一种算法，并添加一些顶点以产生一个更为光滑、细致的模型）；
- 顶点 shader 和像素 shader（作为直接对图形处理器编程的接口）。

1.3.1　Point Sprite

DirectX 8 的一个新特征就是可以在粒子系统中运用硬件精灵（hardware sprite）生成火花、爆炸、雨、雪等效果。所谓的 point sprite 是由其自身编程接口支持的，用以帮助你实现这些效果的功能。

1.3.2　3D 纹理

通过 3D 体纹理，可以完成精确到每个像素的光照效果、点光和聚光效果及大气效果。

1.3.3　Direct3DX 实用程序库

DirectX 8 还增强了 DirectX 7 所引入的 Direct3DX 实用程序库，这个库提供了如下辅助功能：

- 枚举设备配置；
- 设置一个设备；
- 统一处理全屏模式或窗口模式；
- 处理调整大小的操作；
- 向量和矩阵的计算操作；
- 简化图像文件的加载和纹理的创建；
- 绘制简单的形状、精灵和立方体贴图（cube map）。

在 DirectX 8 发布中，实用程序库还提供了对蒙皮网格（skinned mesh）、多分辨率的细节级别（Level-Of-Detail，LOD）几何体、.X 文件的高阶表面数据（higher-order surface data）的支持。D3DX 图像文件加载函数现在可以支持 BMP、TGA、PNG、DIB、PPM、DDS 文件格式。DirectX 8 还引入了一些辅助方法来将 OpenGL 应用程序移植到 DirectX Graphics，以及仅在当今游戏中运用到的高阶表面——N-patches。这些辅助方法可以在图形处理器的帮助下提高模型精度，也就是说，当你传入一个低精度多边形模型的数据后，该模型会在图形卡

中完成镶嵌操作[1]（加入更多的多边形）。

1.3.4 顶点 Shader 和像素 Shader

DirectX 8.0 中的一个能改变现代游戏整体发展进程和外观的重磅炸弹就是引入了顶点 shader 和像素 shader。尽管 DirectX 8.0 已支持了顶点 shader 标准 vs_1_1 和像素 shader 标准 ps_1_1，但 DirectX 8.1 还是增加了对像素 shader 标准 ps_1_2、ps_1_3、ps_1_4 的支持。尤其是最后一个像素 shader 标准（ps_1_4）展现了 ATI RADEON 8500—9200 卡的高级功能，而且它也确立了在 DirectX 9.0 中即将到来的新像素 shader 标准（诸如 ps_2_0、ps_2_x、ps_2_a、ps_3_0）的语法方式。（第 9 章"使用高级 Shader 语言进行 Shader 编程"中的大多数例子中都会用到 ps_2_0 像素 shader。）

1.4 DirectX 9

从 DirectX 8.1 到 DirectX 9.0，API 整体上没有太多改变。新版本引入了新的、经过很大改进的顶点 shader 和像素 shader 标准，其中包括 vs_2_0、vs_2_x、vs_3_0 版本的顶点 shader 标准和 ps_2_0、ps_2_x、ps_3_0 版本的像素 shader 标准。就像现在被所有主要图形卡制造商所采纳的 ps_1_1 标准最初就是专为 NVIDIA GeForce 3 设计的那样，这三个版本的 shader 标准分别为三种硬件而设计。现在还有一些与顶点/像素 shader2_0 或 3_0 版本等价的软件实现，这些软件实现称作 vs_2_sw、vs_3_sw、ps_2_sw、ps_3_sw。

为了能以更高效的方式来编写 shader，DirectX 9 中引入了 HLSL（High-Level Shader Language，高级 Shader 语言）。HLSL 可让你以一种类似 C 语言风格的语法来取代原先汇编风格的语法，从而可以更快、更容易地编写出 shader。本书中所有的 shader 示例都使用了 HLSL。

为了赶上 OpenGL，DirectX 9.0 中引入了剪裁测试（scissor test）和行抗锯齿（line anti-aliasing）。剪裁测试在 alpha 测试之前、片断颜色（fragment color）计算之后进行，它用于确定一个值的屏幕坐标是否在由 4 个值定义的剪裁矩形中，并且丢弃位于矩形外的部分。另外，还有一个新的渲染状态用于支持 OpenGL 风格的球形映射 texgen（Texgen 是"自动纹理坐标生成（automatic texture-coordinate generation）"的别称）。还实现了对双面模板（two-sided stencil）的支持，允许程序员给封闭几何体添加阴影体（shadow volume）（请参见第 10 章"更多的高级 Shader 效果"中的双面模板示例，那个例子展示了阴影体的用法）。

DirectX 9.0 的游戏中应尽快实现的一项特征是具备从单一像素 shader 到多重渲染目标

[1] 译注：镶嵌（tessellate）操作由图形卡的镶嵌器（tessellator）完成，硬件镶嵌器是一种对多边形进行细分的功能部件，镶嵌器使用高阶表面几何体来使游戏中原始简单的模型变得更为光滑和细致。

（multiple render target）的渲染能力（这称作 G-buffer）。然后就可以在另一种通道中混合最多 4 个渲染目标。DirectX 9.0 中有一项与多重渲染目标类似、但不如其实用的特征就是多重元素纹理（multi-element texture）。像素 shader 在一个多重元素纹理中储存至多 4 个纹理，然后可以从那儿读取它们。另外还新增了位移贴图（displacement map），它是一种新的高阶表面类型，不过在写这本书的时候，似乎还只有 Matrox Parhelia 和 ATI RADEON 9700 PRO 在硬件上能支持它。

DirectX 9.0 中两项让人盼望已久的特征是对最高 24 位颜色精度的支持和根据监视器的 gamma 支持来改变 gamma 值的能力。gamma 支持使用了一个非常简单的过滤器，它在数据从一个表面渲染到屏幕上之前起作用，因此所谓的“gamma 控制”并不会改变表面本身的内容。

DirectX 9.0 SDK 提供了一个自 DirectX 6 开始就沿用至今的程序设计框架，本书的所有例子都使用这个框架。从 DirectX 8.0 开始，该框架被称为通用文件（common file），可以在“C:\DXSDK9\Samples\C++\Common”目录中找到这些文件。

通用文件使你可以直接访问你所需要的东西，同时封装了设置 Direct3D 的细节，这对于你的学习很有帮助。因为你可以集中精力于本质的内容，同时还能看到所有最低层的东西。该框架提供了一个通用的平台，让你可以实现自己的专用功能。作为一个初学者，可以通过这些快捷的、经过良好测试的框架代码来避免许多基本的错误，从而使你能更专注于学习实质的内容。中级或专业的程序员也可以把该框架作为一个不错的测试平台，而专业的程序员还可通过阅读 DirectX Graphics 通用文件源代码编写出适合自己的框架，以此更好地适应他们的需要。

1.5　小结

本章介绍了 DirectX API 的这段漫长的发展历程，由最初的 DirectX 2 发展到第一个真正“功德圆满”的 DirectX 8。尽管在 DirectX 8 与 DirectX 9 之间，就 API 接口而言变化不是很大，但在几个新的 shader 版本中则显露出了当今显卡的 shader 能力。DirectX 的将来会朝着具有一套更为稳定的 API 和不断进化的 shader 能力的方向发展，这意味着 API 的使用者不再需要每年重新学习一种新的 API。因此学习 DirectX 9 将会是一项有效的长期投资。

即使系统中存在一个可用的 HAL，通常游戏也会根据 HAL 所能提供的特性来开启或关闭相应的特性。这个过程运作如下：

（1）游戏检查 HAL 的能力（capability）；

（2）游戏适当地开启或关闭某些特性；

（3）若某些特性被关闭，便会以一种低效的方法来实现，可能是模拟某个特性。这通常会导致图像质量降低。

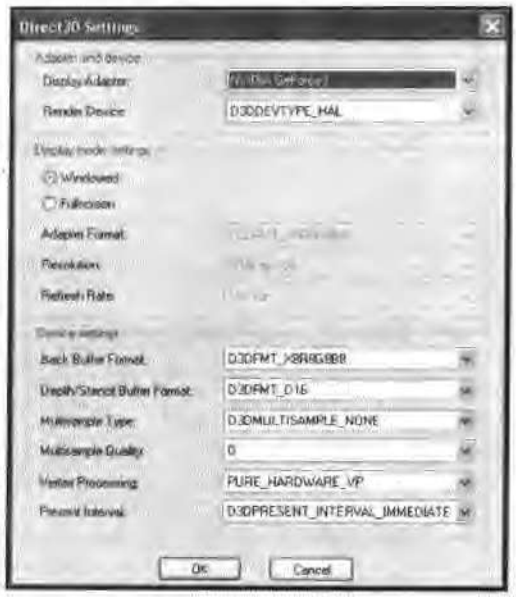


图 2.1　Direct3D Settings 对话框中的设备选择

以前，有一个 HEL（Hardware Emulation Layer，硬件模拟层）可以在软件中模拟某些特性。但是，自从 DirectX 8.0 以后就不再提供 HEL 了。如果想让游戏在老式硬件上运行的话，那只有使用 DirectX 7.0 及其 HEL（称为"RGB 设备"）了。

就像其"前辈"一样，DirectX 9 提供了一个使用由软件开发者编写的 HEL 的接口。这类 HEL 现在被称为"插件式软件设备"。然而据我所知，迄今为止还没有谁去实现过这类 HEL。由此可见，DirectX 8.0/8.1/9.0 的游戏都严格依赖于硬件所提供的功能。

图示 2.2 展示了 Direct3D API、HAI 和 GDI（Graphics Device Interface，图形设备接口）API 之间的关系。

正如你所见，Direct3D 使用 HAL 通过 DDI（Device Driver Interface，设备驱动接口）来访问图形硬件。HAL 是主要的设备类型，它支持硬件加速的光栅化和硬/软件顶点处理。如果计算机的显示适配器支持 Direct3D，HAL 便存在于该计算机中。请注意，显卡驱动程序必须至少支持 DirectX 7 版本的 DDI 接口才能运行 DirectX 9.0 应用程序。要检查 DDI 的版本，可运行"DirectX Diagnostic Tool（DirectX 诊断工具）"并检阅保存下来的文本文件。可通过点

击“DirectX Properties（DirectX 属性）”对话框中的“DxDiag”按钮来启动此工具（见图 2.3）。

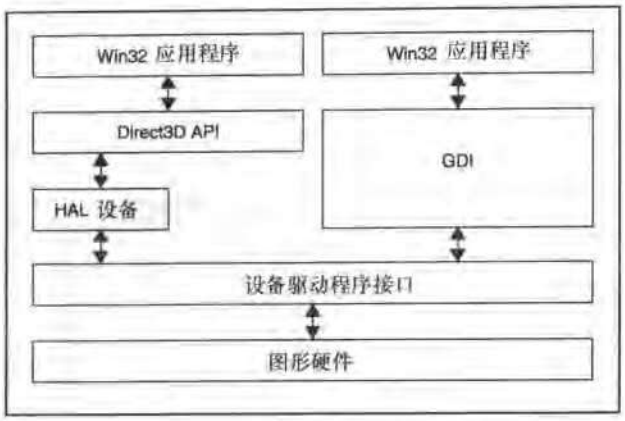


图 2.2　DDI、HAL 和 GDI 之间的关系

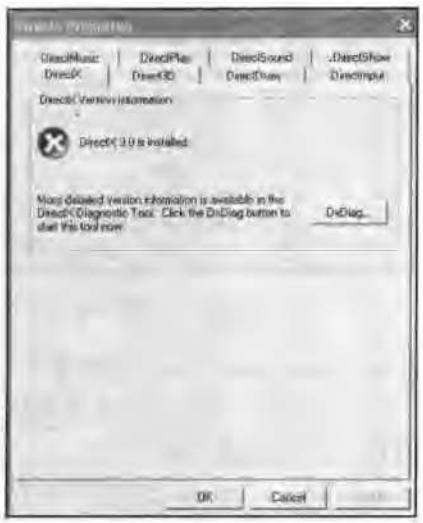


图 2.3　DirectX 属性对话框包含了 DxDiag 按钮

因此，一款显卡若要运行 DirectX 9.0 的游戏就必须要有一个至少与 DirectX 7 兼容的驱动程序。

HAL 也能在软件中模拟部分或全部顶点处理。以下标志用于控制 HAL 和参考设备的顶点处理行为：

- D3DCREATE_SOFTWARE_VERTEXPROCESSING
- D3DCREATE_HARDWARE_VERTEXPROCESSING
- D3DCREATE_MIXED_VERTEXPROCESSING

另外，HAL 可以接通一个所谓的“纯设备”（pure device）。这种设备进行较少的状态检查，并且只有在不使用 Direct3D API 的某些特定函数时才能使用。它比“非纯的”硬件顶点处理设备稍快一点。DirectX 9.0 SDK 中的示例程序都会显示所选的顶点处理类型（见图 2.4）

图 2.4　HAL 展示了它的顶点处理方法

HAL 有 4 种不同的顶点处理方法。包括：

- SW VP（软件顶点处理）
- Mixed VP （混合顶点处理）
- HW VP（硬件顶点处理）
- Pure VP（纯顶点处理）

插件式软件设备和参考设备的行为必须与 HAL 设备相一致。为 HAL 设备编写的应用程序代码应该可以不加修改地应用到软件或参考设备上。事实上，参考设备才是标准的参考实现（reference implementation），软件设备和 HAL 需根据自身功能进行调整，以期与参考设备一致。

因此，必须检测每个设备的能力。参考光栅器支持所有的 Direct3D 9.0 特性，而插件式软件设备甚至连贴纹理（texturing）可能都不支持。

如果在用户的机器上没有可用的硬件加速器，尝试创建一个 HAL 设备将会失败。

2.2　插件式软件设备（Pluggable Software Device）

如果用户的计算机硬件没有提供运行一款游戏所需的全部 3D 操作，编写一个硬件模拟层（现称为“插件式软件设备”）也是完全可能的。现今，软件/游戏厂商必须自己来开发这个硬件模拟设备，而在以前的 DirectX 运行时（run-time）环境中，微软用 RGB 设备来提供这种硬件模拟。软件设备由应用软件加载，并用 Direct3D 对象进行注册。注册一个插件式软件设备的函数是 IDirect3D9::RegisterSoftwareDevice。DirectX 9 DDK（Device Driver Kit，设备驱动工具包）提供了更多关于如何编写一个插件式软件设备的信息。

2.3 参考光栅器（Reference Rasterizer）

只有安装了 SDK 才会提供参考设备。当一个应用程序在一台未安装 SDK 的机器上请求参考设备时，会返回一个 NULL（空）参考设备。NULL 参考设备什么也不做，所有渲染操作的结果都将呈现为一个黑屏。

参考光栅器支持所有 Direct3D 特性，该设备为精确性而非速度做了优化，应该只将其用于测试那些显卡不支持的特性。默认情况下，Direct3D 不会枚举到它。Direct3D SDK 安装程序会在注册表键"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Direct3D\Driver"中设置一个名为 EnumReference 的 DWORD 非零值[①]。另外，硬件设备、软件设备和参考设备都无法渲染 8 位的渲染目标表面（render-target surface）。

2.4 设备控制

在"DirectX Properties"（DirectX 属性）对话框中可配置所有的设备控制，这个对话框位于计算机的控制面板中（见图 2.5）。

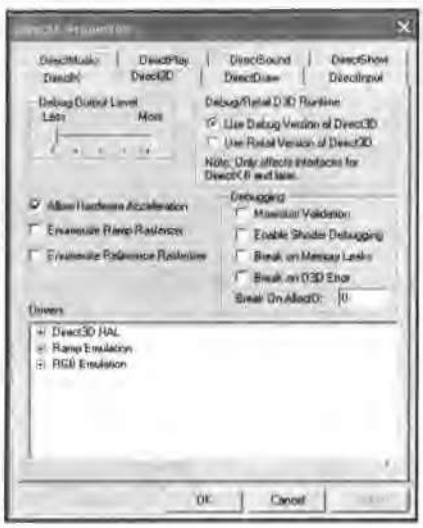


图 2.5 在 DirectX 属性对话框中配置所有设备控制

注意，对话框中的"Enable Shader Debugging"单选框以后会比较有用。另外，"Debug/Retail D3D Runtime"区域可让你在调试（debug）和零售（retail）版本中切换选择，这对在调试版

译注①：也可通过运行"控制面板->DirectX（属性）"来设置是否可枚举到参考光栅器，该对话框 Direct3D 标签页中的"Enum Reference Rasterizer"单选框就是用于控制该键值的设置。

本和零售版本间来回切换以测试零售版的性能将会很有用。

有了 HAL/插件式软件驱动/参考驱动这一系列的方法，微软的 Direct3D 便可保证在缺少特定 3D 硬件的环境中，仍可以使用同一套 API[1]。那很棒，不是吗？接下来，本章开始时提及的另外两个要求又是如何做到的呢：Direct3D 如何处理版本变化？学起来容易吗？是否因其已被精心规划过了，因此对程序员而言就是透明的？这些便是 DirectX SDK 的 COM 接口设计可以帮得上忙的地方。

2.5　COM

如果你用过微软的其他 API，如果那些使用经历已经使你成为一个 COM 老手，那你可以跳过这一节。而剩下的人则必须投入一些精力来学习 COM。DirectX 从一开始就采用 COM 接口，通过学习使用 COM 的方法，你将会有很大的收获。

在 COM 术语中，一个软件组件（component）就是一大堆通过接口提供服务的代码，而接口则是一组相关的方法。一个 COM 组件通常是一个 DLL（Dynamic Link Library，动态链接库）文件，这个 DLL 文件可通过一种一致且既定的方式来访问。说完了……，它听上去不像是需要用上千页纸来阐述的问题，不是吗？[2]

COM 有许多优点。我想强调 3 点，那些是我认为对于像我们这样的 Direct3D 程序员而言比较重要的：

- COM 接口永远不会改变；
- COM 是独立于语言的（language-independent）；
- 只能访问一个 COM 组件的方法，而不是它的数据。

C++和许多其他语言都是基于源代码（source-based）的，因此，它们没有一个版本策略。COM 被设计为一种基于二进制（binary-based）的编程模型，并且 COM 接口永远无法通过任何方式来修改。

举一个例子来说明为什么需要 COM 吧。设想这样的情况，一个非 COM 的 DLL A 是由软件公司 C 开发的，另一个非 COM 的 DLL B 是由软件公司 D 开发的。只要两家公司都没有改变 DLL 的接口，那么 DLL A 和 B 便能互相兼容。但如果公司 D 改变了 DLL B 的接口，那么它就不能再与 DLL A 相兼容了[3]，除非公司 C 对 DLL A 的接口作出相应的变更。

这种情况不会发生在一个 COM DLL 身上。当 COM 对象被更改，一个新命名的全新接口将会添加到 DLL 中。因此，每个 COM DLL 都必须支持自它发布以来所有的接口。若上述

译注 1：作者的意思应该是，若当前运行环境中无硬件加速设备，即 HAL 不可用时，可考虑自己编写插件式软件设备，再退一步，可以使用参考设备，无论使用哪一种设备，操作它们的 API 都是一样的。

译注 2：虽不需要上千页来阐述，但这么几行的叙述也稍微少了点，关于 COM 更详细的描述，译者推荐 Dale Rogerson 所著的《COM 技术内幕——微软组件对象模型》。

译注 3：因为 A 在被开发出来时只知道旧 B 的接口，对于新 B 的接口无从得知。

情况中的两家公司都使用 COM DLL，对 DLL B 接口的一个更改就会导致一个新的接口附加到 B 中。由于 B 有了两个接口，所以 DLL A 仍能够连接上旧的接口。[1]

这便是 DirectX 实现其接口的方法。Direct3D 9.0 或 DirectX Graphics 支持以下接口：

- IDirect3D
- IDirect3D2
- IDirect3D3
- IDirect3D7
- IDirect3D8
- IDirect3D9

表 2.1 显示了这些接口对应的 DirectX 版本。

表 2.1　　DirectX 版本及接口

DirectX 版本	接口
DirectX 2	IDirect3D
DirectX 3	IDirect3D2
DirectX 6	IDirect3D3
DirectX 7	IDirect3D7
DirectX 8	IDirect3D8
DirectX 9	IDirect3D9

此书编写时最新版本的 DirectX 支持从 IDirect3D 到 IDirect3D9 的接口。这意味着用 IDirect3D2 的老游戏依然可以在当前硬件上运行，因为最新版本的 DirectX 仍支持此接口。换句话说，若一家公司开发了一款与 Windows NT4.0 兼容的游戏，而 Windows NT 4.0 只支持 DirectX 3.0 SDK 的 IDirect3D2 接口，那这家公司将只能使用 IDirect3D2 接口来完成所有的工作。自从 DirectX 3.0 开始，每个版本的 Direct3D DLL 中都实现了 IDirect3D2 接口，并且会继续在将来的版本中实现。上述的游戏就可以在所有具有 DirectX 3.0 以上版本运行时的平台上运行了。

COM 是独立于语言的。无论你偏爱的语言是 Visual Basic、Pascal 还是其他语言，你都可以用偏爱的编译器程序包来访问 COM 对象。Delphi 用户尤其喜爱这个特点，但自从 DirectX 7.0 问世以来，Visual Basic 的倡导者们也已在所有 DirectX SDK 中提供了一个他们自己的良好实现。当将游戏中的某些部分如世界编辑器（world editor）用 Delphi 编写，而其他部分（或许是游戏世界中不同的地方）用 Visual C++编译器编写，独立于语言的作用便体现出来了。

译注[1]：原文此段是以 COM DLL A 更改为例，为结合上一段的例子继续说明，译者将此段 A 和 B 对调，不影响读者理解，特此申明。

COM 只能通过方法来访问，而不能直接访问数据对象。从面向对象的角度看，这是一个很好的设计。正如你在下一章中将要看到的，不能直接去调用一个方法，而是需要通过一个名为 v-table 或 VTBL 的虚函数表来进行两次间接调用。这些 v-table 同时也成为一个独立于语言接口的关键。有了这些优点，COM 可助你做到语言无关，并保证能访问过去和现在版本的 Direct3D 接口。现在就试试吧！

2.6　小结

这一章介绍了 HAL 和 COM 背后的概念。HAL 是显卡的主要设备驱动程序。当 HAL 不支持某一特性时，就会引发一种低效机制，给用户一种类似的视觉体验。另一种选择是，游戏也可以在一个由游戏厂商编写的插件式软件设备上运行。

COM 提供了一种版本策略，保证所有使用早期 DirectX 版本开发的游戏都能在将来的 DirectX 版本中运行。它还帮助你封装了数据，并对不同语言提供一个一致的接口。

第 3 章

编 程 约 定

这一章要介绍整本书中都会用到的编程约定。在本章中你将会学到：

- 如何访问 COM 对象；
- 在 DirectX SDK 及本书中使用的命名约定（匈牙利表示法）；
- 实用的 Direct3D 应用程序调试技巧；
- 如何使用及为何使用返回码。

3.1 访问 COM 对象

当调用 COM 组件中的一个方法时，必须使用 v-table，在决定使用 C 还是 C++开发代码前还需考虑一些问题。我将以一个例子作为开始，假设你想要在世界级 3D 游戏的一个场景中设置一个光源。若使用 C 语言，你需要使用以下代码：

```
hr = m_pd3dDevice->lpVtbl->SetLight(m_pd3dDevice, 0, &light);
```

为了调用 COM 接口，方法的第 1 个参数传递 this 指针（称为 lpdd），并使用 this 指针作为一个指向接口 v-table 的指针（在此称为 lpVtbl）来引用接口的方法（见图 3.1）。

虽然宏（macro）可以使 C 程序员的工作变得简单些（例如，在 ddraw.h 中），但 C++却可更进一步简化 COM 对象的使用。这是因为 COM 对象与 C++对象是二进制兼容的，编译器可以使用相同的方法处理 COM 接口和 C++抽象类。一个从纯虚基类（一个类中只要有等于 0 的虚方法[1]）派生出的 C++类实例和一个 COM 接口具有相同的内存布局。因此，在 C++中，lpVtbl 指针是隐式引用的，this 参数也是隐式传递的。这样，前面描述的 C 调用在 C++中看上去就是：

译注[1]：由于在 C++中用符号 "= 0" 表示纯虚函数，因此这里说 "等于 0 的虚方法"。

```
hr = m_pd3dDevie->SetLight(0, &light);
```

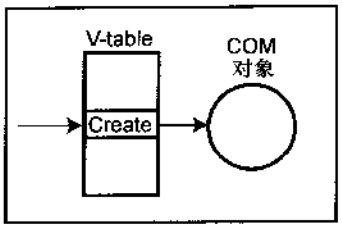


图 3.1　v-table

通常，在 C++中大多数 DirectX 调用都是如下的形式：

```
lpinterface->methodname
```

正如本书引言中所述，为了编译 DirectX 程序，必须包含一些导入程序库：

- d3dx9.lib
- d3dxof.lib
- d3d9.lib
- winmm.lib
- dxguid.lib

这些库中的大部分都是为开发环境的链接器提供接口名的导入库[①]（import library）。

现在已知所谓 COM 对象就是一系列的接口（简单的方法指针）集合和一系列比较特殊的 v-table 的集合，接下来要看一个如何使用 COM 的示例。另外，应该注意以下 3 点：

- Direct3D 运行时 COM 对象和 DLL 必须由 Windows 注册和加载。DirectX 安装程序会为你做这件事情；
- 必须在 Windows 项目中包含先前提及的库，才能正确链接到所调用的包装方法（欲了解更多信息，可参阅本书引言中“设置 Visual C++ 6.0”和“设置 Visual C++ .NET”两节）；
- 必须在源文件和 IDE（如 Visual C++）的包含路径条目表格中都包含正确的头文件，这样编译器才可以看到 DirectX Graphics 的头信息、原型和数据类型（欲了解更多信息，可参阅本书引言中“设置 Visual C++ 6.0”和“设置 Visual C++ .NET”两节）。

IDirect3D9 接口指针的数据类型为：

```
LPDIRECT3D9 g_pd3d = NULL;
```

要创建一个 IDirect3D9 的 COM 对象并从中获取一个接口指针，只需使用 Direct3DCreate9()方法即可：

```
m_pD3D = Drect3DCreate9( D3D_SDK_VERSION );
```

译注①：导入库是用于标识动态链接库导出信息的二进制程序库，DirectX 的绝大部分方法都在动态链接库中实现，而导入库中仅有一些短小的转发程序，用于把应用程序的调用转移到动态链接库中，这个转发程序就是下文提到的包装方法。

传给 Direct3DCreate9()惟一的一个参数应该是 D3D_SDK_VERSION。这是告知 Direct3D，你已使用了正确的头文件。当头文件或其他变化需要应用程序重新生成时，这个值会增加。若版本不匹配，Direct3DCreate9()则会失败。

获取的接口指针便可用于访问 IDirect3D9 接口。可用它来调用接口中的方法，例如：

```
// Create the device
hr = m_pD3D->CreateDevice( m_d3dSettings.AdapterOrdinal(),
                           pDeviceInfo->DevType, m_hWndFocus,
                           behaviorFlags,
                           &m_d3dpp, &m_pd3dDevice );
```

对于 COM 就说这些，你会在整本书中都用到 COM。如果对那些奇怪而神秘的参数名感到困惑，继续读下去就会得到一个解释。

3.2 命名约定

我使用一个简单的变量标记方案，该方案称为“匈牙利表示法（Hungarian notation）”，源于微软，微软用其开发它自己的项目。这个名字来自于它的发明者 Charles Simonyi，他是一个匈牙利人、一个富有传奇色彩的微软程序员。这种表示法有助于使变量具有正确的格式，使别人能更容易地阅读你的代码，不过它同时也会让那些从没有见过匈牙利表示法的人感到困惑。

匈牙利表示法用变量类型作为它们的前缀。表 3.1 展示了我所使用的前缀和它们对应的类型。你可能还看到过别的前缀，不过这张表显示的是那些比较常用的前缀。

变量的命名约定以变量类型作为名字前缀，而方法的命名约定则相对好读一点且只是说明了方法的用途。在所有的方法中，方名字[1]的首字母是大写的，并且不允许使用下划线。

```
HRESULT ConfirmDevice( D3DCAPS9* pCaps, DWORD dwBehavior, D3DFORMAT adapterFormat,
                       D3DFORMAT backBufferFormat )

HRESULT EnumerateDevices( D3DAdapterInfo* pAdapterInfo, CArrayList* pAdapterFormatList )
```

如你所见，函数的各参数都遵循与普通变量一样的命名约定：参数 pCaps 指向一个 D3DCAPS9 结构，ConfirmDevice()的第 3 个和第 4 个参数使用相同的类型来描述表面格式，由于 Direct3D 提供的类型名称全为大写，而 EnumerateDevices()的两个参数没有全部大写，由此可知它们不是由 Direct3D 提供的类型。

类型和常量由大写字母开头，可以在名字中使用下划线，例如：

```
#define D3DPRESEBT_BACK_BUFFERS_MAX     3L
```

译注[1]：即组成方法名的几个单词。

必须在所有的 C++类前缀加一个大写字母 C，将每一个子名字的第一个字母大写。例如：

```
class CD3DMesh
{
    public:
};
```

表 3.1　　命名约定中的前缀和类型

前缀	类型	例子
w	WORD，字	wSpeed
dw	DWORD，双字	dwHitList
f	FLOAT，单精度浮点数	fSpeed
d	DOUBLE，双精度浮点数	dDirection
p	指针	pArray
v	D3DVECTOR3，Direct3D 三维向量	vLight
l	LONG，长整数	lPitch
s	字符串	sQuestion
sz	以 0 结尾的字符串	szQuestion
h	句柄	hResult
I	COM 接口	IDirect3D
m_p	类成员指针	m_pFloorVertices
m_f	类成员浮点数	m_fStartTimeKey
c	常数	cText
b	BOOL，布尔	bCheck

CD3Dmesh 类用于处理.X 文件的加载，它在通用文件 d3dfile.h 中实现。说到这里，为何不尝试一下呢？让我们来看一个典型的 Direct3D 应用程序类：

```
class CMyD3DApplication : public CD3DApplication
{
    CD3DFont* m_pFont;
    CUSTOMVERTEX m_QuadVertices[4];
    LPDIRECT3DTEXTURE9 m_pCustomNormalMap;
    LPDIRECT3DTEXTURE9 m_pFileBasedNormalMap;
    D3DXVECTOR3 m_vLight;
    BOOL m_bUseFileBasedTexture;
    BOOL m_bShowNormalMap;
    HRESULT CreateFileBasedNormalMap();
```

```
        HRESULT CreateCustomNormalMap();
        HRESULT ConfirmDevice( D3DCAPS9*, DWORD, D3DFORMAT );
        LRESULT MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam );
    protected:
        HRESULT OneTimeSceneInit();
        HRESULT InitDeviceObjects();
        HRESULT RestoreDeviceObjects();
        HRESULT InvalidateDeviceObjects();
        HRESULT DeleteDeviceObjects();
        HRESULT Render();
        HRESULT FrameMove();
        HRESULT FinalCleanup();
    public:
        CMyD3DApplication();
    };
```

在这段代码的开头有一个字体类的类成员指针，接着的成员是一个使用自定义顶点结构构成的四边形顶点数组。然后，两个类成员指针指向 Direct3D 纹理对象。一个光照向量保存在 D3DXVECTOR3 中。m_bUseFileBasedTexture 表示是否使用了一个基于文件的纹理，m_bShowNormalMap 表示是否显示普通的贴图。正如你从这个类里所看到的（你也可以从通用文件目录中选取一个其他文件来看看），DirectX SDK 的所有文件都使用匈牙利表示法。

3.3 调试 DirectX

调试 DirectX 应用程序是一项非常具挑战性的事。以下是一些帮助你起步的提示：

- 在*.cpp 文件的开头、包含 windows.h 之前，定义（define）STRICT 符号。这将导致更加严格的类型检查，从而帮助你找到错误；
- 使用 DirectX SDK 中的 DXDiag 实用程序来报告你的错误，但必须确认你已清楚地知道系统上有些什么东西；
- DirectX 控制面板允许你将调试输出等级（debug output level）设置为 0～5 之间的任意值。要访问控制面板，选择“开始—>设置—>控制面板—>DirectX—>Direct3D”。在同一个标签页中，你还可以在调试版本和零售版本的运行时之间来回切换；
- D3DX 库是一个静态库。为了帮助调试，D3DX 还有一个专门用作调试的动态库。为了使用这个动态库，需要链接 d3dx9d.lib，它是一个对应于 D3DX9D.DLL 的导入库；
- 使用一个内存管理器可减少内存分配和释放问题。Visual C++提供了一个称作 CRT 的内存管理器，在互联网上你还可以找到更适合于游戏开发的内存管理器，只需在 http://www.flipcode.com 或 http://www.gamedev.net 上搜索一下。另外，BoundsChecker（http://www.compuware.com）是一个商业解决方案；
- 可以使用 Visual C++ .NET 的一个插件来调试顶点 shader 和像素 shader。我将在本书介绍 shader 的章节里更详尽地介绍这个调试器；

■ 通常只使用 DirectX 默认的调试版符号文件（symbol file），但是如果你想使用类似于 Intel VTune 的工具来测算应用程序的性能，那么就必须从微软的网站下载 DirectX 发布版的符号文件；

■ OutputDebugString()函数是一个很有用的调试辅助函数，它可以在程序运行期间将调试信息输出到 IDE 的调试窗口中；

■ 对于较大型的应用程序，在程序初始化期间，使用简单的日志文件类来记录问题是很有帮助的。你可以从互联网上找到一组非常有用的日志文件类。第 12 章“使用*.md3 文件”中的一个示例就使用了一个这样的日志文件类；

■ 为了检查是否发生了纹理对齐或纹理过滤方面的问题，使用一个调试用的纹理是一种常见的做法。通常你可以尝试给该纹理着以不同的颜色或使用一些文本来查看它是如何对齐的。图 3.2 给出了一个例子；

■ 只有在使用多个监视器的系统中或远程调试的情况下，Visual C++ GUI 调试器才可以调试全屏独占应用程序。现今使用多个监视器的系统是很常见的；至于远程调试，在 DirectX SDK 帮助文件中也有介绍。若不具备多监视器或远程调试的能力，那么可以像通用文件中实现的框架那样，把应用程序构建为窗口模式。

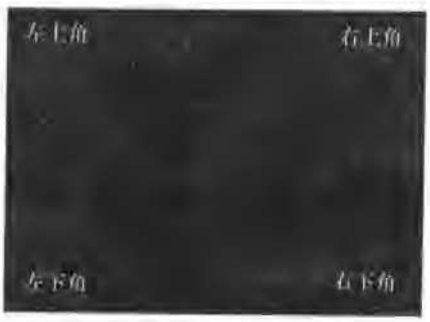


图 3.2　使用纹理来测试布局

3.4　返回码

返回码是当方法完成后所返回的值。它表示一个调用是否成功，如果可能的话，它还会指出调用失败的原因。通过使用 DirectX 9.0 SDK 提供的宏 SUCCEEDED()和 FAILED()可以简化检查返回码的工作。它们以一个 HRESULT 作为参数，返回参数是表示一个成功代码还是失败代码。由于并不是所有方法都会在成功后返回 S_OK，所以不要试图将 HRESULT 与 S_OK 进行比较判断。比如，若没有什么需要做的，一个函数就可能会返回 S_FALSE 来表示它什么都没做。

每一个 DirectX 方法都会返回一个 HRESULT，因此需要组织一下你的代码，以使其可以检查并处理所有可能的错误。这项工作一开始可能会让人感到很沮丧，不过这很快就会成为

一种习惯，由此得到的回报就是获得了一个更经得起考验、更强健的程序。

3.5 小结

本章向你介绍了一些 Direct3D 程序设计的一般知识，这里提到的规则和约定将帮助你以一种一致的方式来创建示例程序。这些规则文档在游戏开发团队中是很常见的，这也许会成为建立一个你自己的游戏开发团队的起点。

第 4 章

3D 基础、Gouraud 着色、纹理映射基础

这一章将向你简要地介绍一些非常重要的 3D 概念，并解释了 Gouraud 着色和 Flat 着色之间的区别。另外，还将给出一个如何将纹理映射到物体的大致概念。在本章中你将会学到，如何：

- 使用顶点和朝向；
- 使用面和多边形；
- 使用 Gouraud 着色；
- 进行基础的纹理映射。

4.1 3D 基础

默认情况下，Direct3D 引擎使用左手坐标系。每一根正轴（x、y、z）都远离观看者而去。图 4.1 展示的世界坐标系可以让这个概念更加清楚一点。

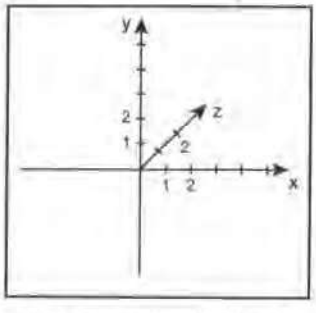


图 4.1 Direct3D 坐标系

y 轴的正方向朝上，x 轴的正方向朝右，z 轴的正方向朝屏幕里面、也就是远离观看者的方向。OpenGL 坐标系的 z 轴正方向则朝着屏幕外面，这种坐标系称为右手坐标系。Direct3DX 实用程序库中有一些内置函数可以帮助你将使用 OpenGL 的右手坐标系的一个应用程序移植到 Direct3D 的左手坐标系。正如你将会在第 12 章“使用*.md3 文件”中看到的，*Quake3* 和它的.md3 文件格式使用的是它们自己的坐标系，这个坐标系的 z 轴正方向朝上，y 轴的正方向朝着屏幕外面。然而，大多数游戏使用的坐标系与 OpenGL 或 Direct3D 所用的坐标系相同。此外，由于我们使用的是浮点数坐标值，所以即使使用很小的数值也不会影响其精确度。接下

来讨论顶点（vertex），让我们来看一个正方形，如图 4.2 所示。

正方形有 4 个顶点，v1 到 v4。顶点定义了一条或多条线的交点，当然在这里我们所关心的线条都是笔直的线。为什么要使用坐标系和顶点呢？原因就是 Direct3D 需要一种方法来将图元（primitive）放置到场景里去，你也需要一种测量距离和空间的方法。现在把正方形放置到坐标系中（见图 4.3）。

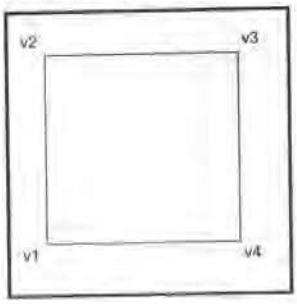


图 4.2　一个正方形

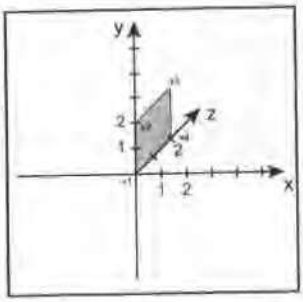


图 4.3　一个在 Direct3D 坐标系中的正方形

要在 3D 空间中定义一个顶点，就必须定义 3 个坐标值：x、y、z。坐标系的原点是(0, 0, 0)。正方形的顶点坐标可描述为：

```
v1 (0, 0, 0)
v2 (0, 2, 0)
v3 (0, 2, 2)
v4 (0, 0, 2)
```

这样就可以了。你可以用一个 3D 空间中的向量（vector）来定义顶点的位置，该向量从原点到顶点。这就是游戏程序员使用的所谓“自由向量”（或称“零向量”）。它的起点假设为原点，只需要描述终点（见图 4.4）。

向量是3D游戏中最重要的概念之一，它是描述方向和大小的数学实体(比如用于表示速度)。向量通常用一个粗体字母表示，例如 **a**。因此，你可以说向量 **v** = (1, 0, 1)。第 1 个值表示在 x 轴方向上的单位，第 2 个值表示在 y 轴方向上的单位，第 3 个值表示在 z 轴方向上的单位。

向量不仅可以用于定义顶点的位置，还可用于定义一个方向。例如，向量 vUp (0, 1, 0) 定义了朝上的方向（见图 4.5）。要注意，当将向量用于定义方向或顶点时，它的实际长度并

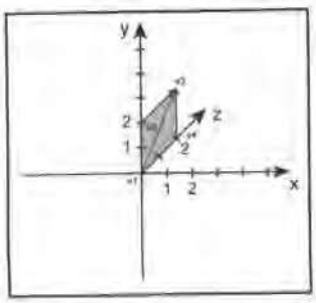


图 4.4　用一个自由向量（或“零向量”）描述正方形的第 3 个顶点

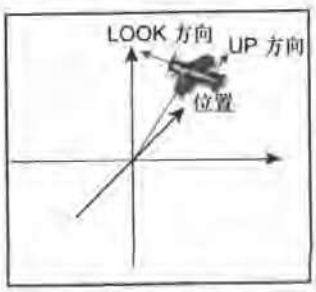


图 4.5　飞机的原点

不重要。但在某些时候向量的长度则是很重要的，例如用于定义法向量或光向量的情况，你会在本书稍后接触到这类向量。

正如你所预期的，Direct3D 为定义向量提供了两个默认的结构：在 d3d9types.h 中默认版本的向量结构 D3DVECTOR 和在 Direct3DX（Direct3D 实用程序库）的 d3dx9math.h 中增强版本的向量结构 D3DXVECTOR3，它们可处理用到向量的所有数学例程。向量 v3 可以定义为：

```
D3DVECTOR v3(0, 2, 2);          // or ...
D3DXVECTOR3 v3(0, 2, 2);        // used by the utility library
```

4.2　理解顶点

自由向量或零向量通常用于描述顶点的位置。不过，组成一个顶点的数据远不止位置这一项。若顶点有颜色，还需要一个用于保存颜色的变量；若它有法线，则需要有一个保存顶点法线的向量；若用到了纹理，顶点还应有相应变量来保存纹理坐标。由此一个顶点结构可能如下所示：

```
struct Vertex
{
    D3DVECTOR3 vPosition;
    DWORD dwDiffuse;            // diffuse vertex color
    D3DVECTOR3 vNormal;         // vertex normal
    FLOAT u, v;                 // texture coordinates of vertex
}
```

我将在下一章中介绍“顶点法线”和“纹理坐标”这两个术语所表示的含义。需要认识到的是，顶点通常是一个结构，包含的信息远不止位置信息。Direct3D 的运行时是以这种方式来接受数据的：所有顶点都储存在一个顶点缓冲器中，顶点缓冲器允许应用程序分配和访问通常只有驱动程序才可访问到存储器。我将在第 5 章“基础”中讨论关于顶点缓冲器的更多细节。

4.3　使用朝向

在虚拟世界中测量和组织空间对每个游戏来说都是很重要的，但需要一种定义物体朝向的方法。

描述物体的朝向至少需要 4 个向量——LOOK、UP、RIGHT、POSITION。POSITION 向量定义了物体的位置；LOOK 向量定义了物体所指的方向；RIGHT 向量定义物体的右侧指向；UP 向量仅在物体会绕着 LOOK 向量旋转时才是必需的，UP 向量表示了哪个方向对于物体来说是“上”或“下”。

可将物体绕着向量作旋转。若要改变飞机的俯仰度（pitch），则需将其绕着 RIGHT 向量旋转；若要横滚（roll）飞机，则需将其绕着 LOOK 向量旋转；若要改变飞机所面对的方向（yaw；航向、偏航），则需将其绕着 UP 向量旋转；若要使物体沿着 LOOK 向量的方向移动，则需改变 POSITION 向量。

4.4 理解面（face）

任何 3D 物体都是由面构成的，如图 4.6 所示。

为了创建一个看上去光滑的物体，需要成百个甚至上千个面（或表面），如图 4.7 所示。

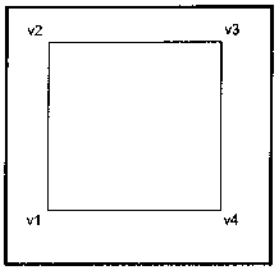


图 4.6　一个正方形的面

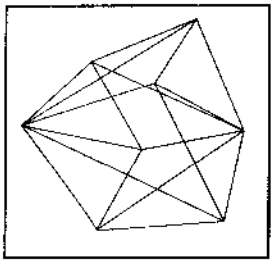


图 4.7　一个由三角形构成的立方体

所有模型和 3D 世界都可被分解为点、线和三角形。Direct3D 只能渲染这些图元，若为了减少面的数量而使用 4 个或更多顶点的多边形来设计形状的话，Direct3D 会将每个多边形面分割为一些三角形。由此为了获得更好的性能，你应该使每个物体都易于被分割为三角形。因此，通常应使用三角形来构建物体。一个由若干三角形构成的物体称为“网格（mesh）”，如图 4.7 所示的立方体。

图 4.8 展示了一个更复杂的网格，它由 9 个面构成，9 个面来自于 10 个三角形。

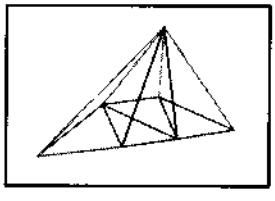


图 4.8　更复杂物体的面

最后需要注意的一点是，Direct3D 会剔除背面。这种背面剔除（Backface Culling）特性可删除所有背对着观看者或照相机的三角形，即那些在默认情况下按逆时针次序的顶点所构成的三角形。平均而言，在任何特定时刻，游戏世界中的一半三角形是背对着照相机的，因此利用该特性有助于缩短渲染的时间。根据要渲染的对象不同，有时可以用 D3DCULL_NONE 标志来关闭背面剔除。为了进行背面剔除，需要计算一个三角形两条边的叉积，叉积结果是一个与平面（由三角形的那两条边构成）垂直的向量。该向量就是平面的面法线，它的方向决定了这个三角形是面朝照相机的还是背向照相机的。由于 Direct3D API 总是以一致的顶点次序来计算叉积，所以可知一个三角形的顶点是以顺时针次序给出的还是以逆时针次序给出的。你可在附录 C“数学入门”读到更多关于叉积的内容。

4.5 理解多边形（polygon）

多边形可以在 2D 或 3D 空间中定义。现在，我们所说的多边形都是指至少由 3 个顶点定义的 n 边封闭形状。最简单的多边形就是三角形。由于三角形的所有顶点都保证共面，因此 Direct3D

用三角形来构成绝大多数多边形。而渲染那些由不共面的顶点所构成的多边形是很低效的。

4.6　理解法线（normal）

法线是垂直于面的向量，可用于定义一个面的朝向或可见侧（见图 4.9）。

一个顶点法线通常垂直于一个面，并且位于顶点之上，具有一个单位的长度。另一类法线是面法线，它通常垂直于面，并且位于面的中央。由于 Direct3D 会自动地计算面法线，所以你无需为它们操心。顶点法线对光照计算很重要，你将会在第 9 章"用高级 Shader 语言进行 Shader 编程"中看到这些内容。另外，面法线和顶点法线对着色模式（shading mode）而言也很重要。例如，面法线经常被用于 Flat 着色，而顶点法线则被用于 Gouraud 着色。

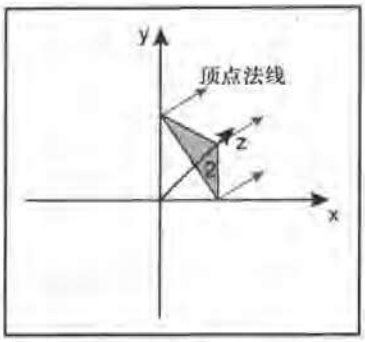


图 4.9　一个三角形的顶点法线

4.7　理解法线和 Gouraud 着色

顶点法线向量被用于 Gouraud 着色模式（自 Direct3D 6.0 以来的默认着色模式），控制光照及获得某些纹理效果。那么什么是着色模式呢？着色是进行光照计算并根据计算结果确定像素颜色的过程。这些像素颜色稍后将用于在屏幕上表示物体（更专业地说，着色是光栅器对像素进行上色的方法）。Flat 着色照亮每个多边形或面，Gouraud 着色照亮了每个顶点（见图 4.10）。

图 4.10　以线框模型、Flat 着色模型、Gouraud 着色模型来呈现的茶壶

如图 4.10 中所见，用户通过 Flat 着色可以清楚地察觉出茶壶的各个面，而通过使用 Gouraud 模型却可看到一个圆滑的茶壶。这是如何做到的呢？让我们用图 4.9 中的三角形来举一个简单的例子。由于所有的顶点法线指向同一方向，所以顶点之间的面上各处都受到了相同的光照。在使用 Gouraud 着色时，该三角形将呈现为一个平面三角形。若使用 Flat 着色，也会得到同样的效果。因为 Flat 着色是对每一个面进行着色，也就意味着将使用位于三角形中央的面法线来计算光照。这与前面 Gouraud 着色时用顶点法线计算所得到

的结果非常相近[1]。现在使顶点法线不垂直于面，如图 4.11 所示。

使用 Flat 着色，没有任何变化，因为三角形中央的面法线仍垂直于面。而使用 Gouraud 着色，面的边角看上去就圆滑了。这是因为顶点法线的方向随着顶点的不同而不同，若几何体是一个大致圆形的物体或具有某些圆形的部分，则 Gouraud 着色将使几何体看上去很圆滑，而事实上这只是人为创造出的效果。

在 Direct3D 中 Gouraud 着色模式是默认模式，你无需为设置它做任何额外的事情。因此，没有代码。

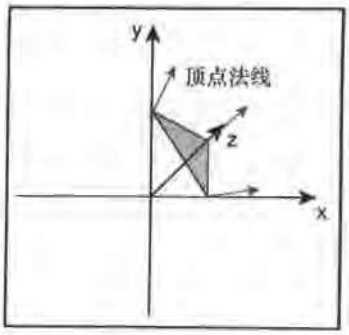


图 4.11　不垂直于三角形的顶点法线

4.8　纹理映射（Texture-Mapping）基础

在 20 世纪 80 年代和 90 年代初的 3D 游戏中，物体的外表基本都是那种发亮的塑胶质感的样子。这些实心色彩的物体看上去比较乏味和无趣，它们缺乏几类细节信息，而这些细节信息能给 3D 物体带来真实的视觉复杂度（如子弹孔、不规则表面、生锈的部分、裂缝、指纹、钉在钢板上的螺丝钉等等）。而这些痕迹都通过纹理来产生。通过纹理，你可以用一张图像替代几何体来提供复杂的视觉细节。以基本观点来看，纹理只是一张由像素色彩组成的简单位图，可以借助于 Adobe Photoshop 或类似程序来创作这些位图。不过在当前上下文中，纹理指的是一种可映射到一个或多个面上的特殊图像（见图 4.12）。

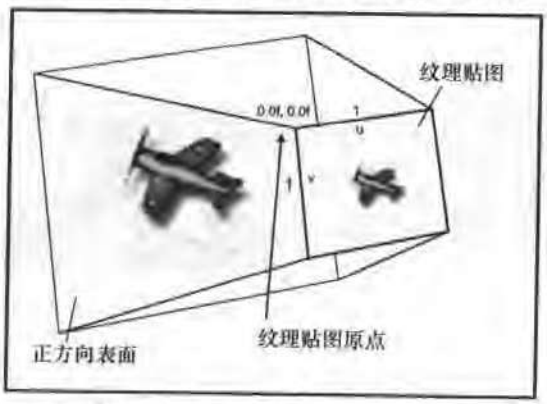


图 4.12　将一张纹理映射到一个正方形上

在图 4.12 中，可以看到纹理贴图是如何被映射到正方形上的。由于正方形比纹理贴图大，所以贴图必须进行缩放。为了将图素（texel，texture element（纹理元素）的缩写）映射到图

译注[1]：因为图 4.10 所示三角形的顶点法线和面法线朝向相同。

元上去，Direct3D 要求在纹理中对所有图素都使用一个统一的地址范围。因此，它使用一种具有通用性的编址方案，其中所有图素地址都在区间[0.0, 1.0]中。纹理坐标空间中，u 表示水平方向，v 表示垂直方向。纹理的右下角是(1.0f, 1.0f)，左上角是(0.0f, 0.0f)，注意在这里没有涉及到纹理的实际尺寸，即使纹理的宽度要比高度大也是这样编址。若赋予的纹理坐标超出该区间，则可以创造出某些特殊的贴纹理的效果。你将在第 7 章"纹理映射基础"中读到更多关于这个概念的内容。

需将纹理坐标分配给一个网格的各顶点。因此，纹理坐标可作为顶点的一个分量被直接包含在顶点结构中，如"理解顶点"一节中所示的顶点结构。

图 4.12 中正方形的纵横比（高度与宽度的比值）与纹理相同。在这种情况中，纹理坐标(0.0, 0.0)、(1.0, 0.0)、(1.0, 1.0)、(0.0, 1.0)被分配到图元的各顶点，这将导致 Direct3D 缩放纹理以覆盖整个正方形。分配纹理坐标可能如下所示：

```
cvVertices [0] = {0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f};    // x, y, z, rhw, tu, tv
cvVertices [1] = {1.0f, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f};
cvVertices [2] = {1.0f, 1.0f, 0.5f, 1.0f, 1.0f, 1.0f};
cvVertices [3] = {0.0f, 1.0f, 0.5f, 1.0f, 0.0f, 1.0f};
```

这是一个典型的顶点结构，它由位置向量、RHW（Reciprocal of Homogeneous W，齐次 W 的倒数）值和纹理的 tu、tv 坐标组成（我将在第 5 章中更加详细地解释 RHW）。现在我想着重解释一下最后两个变量，即一对纹理坐标。值 1.0f 会将纹理进行拉伸至覆盖整个物体。如果想将纹理应用到只有它一半宽的矩形上，就需要决定如何应用上去。可以将整个纹理应用到矩形上；不过这要求先改变纹理的纵横比，否则纹理将被挤压以适合该矩形。也可以只将纹理的左半部分或右半部分应用到矩形上；为这种选择，需将纹理坐标(0.0, 0.0)、(0.5, 0.0)、(0.5, 1.0)、(0.0, 1.0)分配给各顶点。

另外，还可以通过使用纹理坐标将纹理的一部分分配给一个图元。如果要将纹理映射到一个三角形上，则可能会如图 4.13 所示那样来映射。

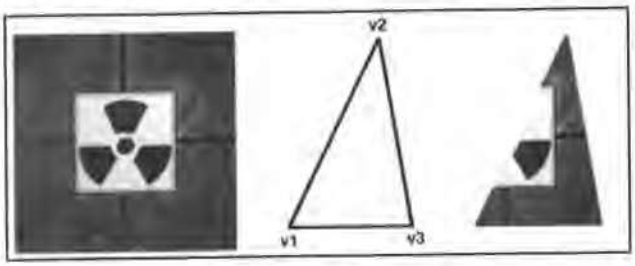


图 4.13　将一个纹理的某部分映射到一个三角形上

以上只是很简单的一部分，现在来看一个更高级的纹理映射。研究一下这张 *Quake 2* 中.md2 文件的纹理，如图 4.14 所示。

图 4.15 展示了.md2 模型所表示的整个武士。

图 4.14　一个.md2 模型的纹理

图 4.15　一个贴有纹理的.md2 模型

可以看到，纹理的不同部分被映射到了该模型的不同多边形上。你可以使用一个专门的程序来绘制纹理，并在模型上设置纹理坐标。MilkShape 3D 就是一个用于制作这类模型的很好工具，你可以在“http://www.swissquake.ch/chumbalum-soft”上找到它。

4.9　小结

我希望你能通过以上描述对纹理映射有一个大致的印象。现在，准备进入下一章了，让我们来真正接触一下 DirectX。

第 5 章

基　　础

这一章将使你体验到第一次使用DirectX 9.0编程的感觉。本章将介绍一个通用文件框架，并说明如何构造一个顶点结构、如何填充顶点缓冲器以及如何显示存储在顶点缓冲器中的几何图形。此外，还会通过将一张纹理映射到一个简单的四边形上来介绍纹理映射。我将先从介绍特征开始，然后再转到如何使用本章示例程序的话题上。

所有的示例程序都拥有以下特征：

- 它们既可在窗口模式中运行，也可在全屏模式中运行。可以通过按 Alt+Enter 在两种模式之间切换；
- 按 F2 将弹出一个"Direct3D Settings"对话框，在其中可改变许多与设备相关的特征；
- 在进行一个动画时，所有示例程序都提供了一种逐帧播放模式，允许用户按回车键和空格键来逐步播放一个动画；
- 按 Esc 键，可关闭示例程序。

5.1　编译示例

我要先介绍一下编译第一个示例 basic 的几个必要步骤，这可以作为你的一个热身。basic 显示了一个类似彩虹般颜色的光谱（见图 5.1）。

所有需要的文件都位于 CD-ROM 的"\chapter 5\basic"目录下。你需要把 basic 目录复制到储存 DirectX SDK 示例的默认 DirectX 目录下。（在我的计算机中，这个路径是"C:\DXSDK\Samples\C++\Direct3D"。）复制完后，basic 目录的路径应该是"C:\DXSDK\Samples\C++\Direct3D\Basic"。

该目录中应包括以下这些文件（见图 5.2）：

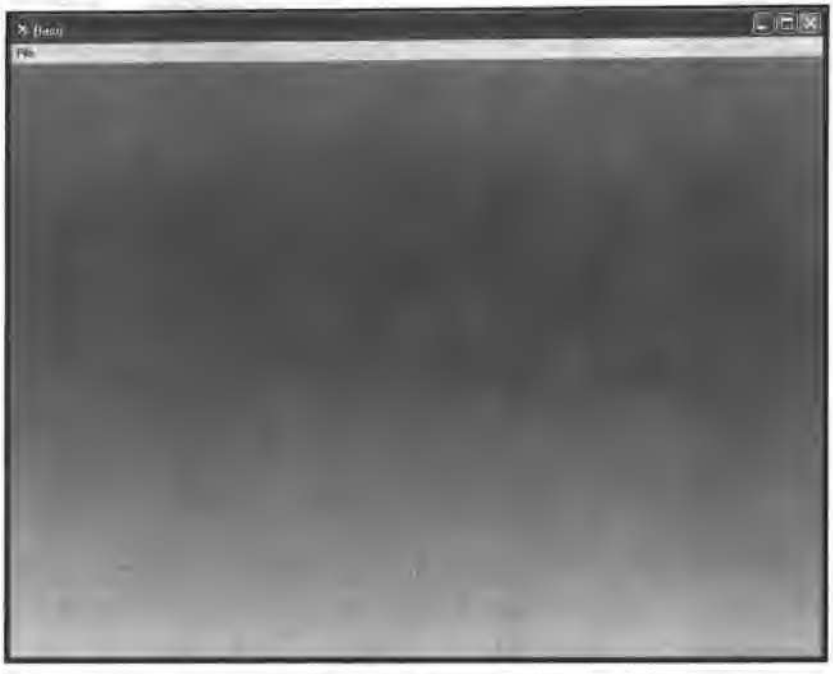


图 5.1　顶点颜色

- **basic.cpp**，主文件；
- **winmain.rc**，资源文件（菜单、加速键、图标信息等等）；
- **resource.h**，winmain.rc 的头文件；
- **basic.dsw**，Visual C/C++ 6.0 所需的“workspace（工作区）”文件；
- **basic.dsp**，Visual C/C++ 6.0 所需的“project（项目）”文件；
- **basic.sln**，Visual C++ .NET 所需的“solution（解决方案）”文件；
- **basic.vcproj**，Visual C++ .NET 所需的“project”文件；
- **readme.txt**，一份示例的说明。

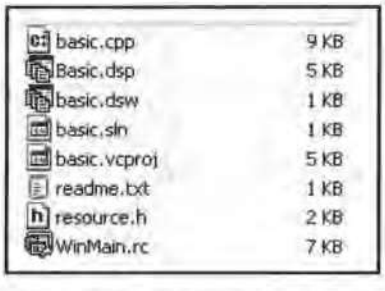


图 5.2　第一个示例程序所需的源文件

Visual C/C++ 6.0 使用 basic.dsw 和 basic.dsp 两个文件来记录各种路径以及你对应用程序的各种选项。主文件是“workspace”文件。在 VisualC/C++ 6.0 中，你可以通过选择“File—>Open Workspace”来打开其名称对应的文件。另外，你也可点击*.dsw 文件来打开一个 workspace。

Visual C++ .NET 则使用 basic.sln 和 basic.vcproj 两个文件来记录各种路径以及你对应用程序的各种选项。主文件是“solution”文件。在 Visual C++ .NET 中，你可以通过选择“File—>Open Solution”来打开其名称对应的文件。同样，你也可点击*.sln 文件来打开一个 solution。

要成功地编译这个示例，还必须将其与以下库文件相链接：

- d3dx9dt.lib
- d3dxof.lib
- d3d9.lib
- winmm.lib
- dxguid.lib
- comctl32.lib

正如第 4 章所述，这些库文件大部分是 COM 的包装程序。Direct3DX 静态链接库的发布版本名称为 d3dx9.lib。system32 目录下还有一个名为 d3dx9dt.dll 的动态链接库版本，可用于创建调试版本，只需将示例链接到 d3dx9dt.libCOM 包装程序就可以使用它了。

本书所有示例程序都使用了 DirectX 9 SDK 提供的一个通用文件框架。接下去几页将会对这个框架作一个介绍。

5.2 DirectX Graphics 通用体系结构

自 DirectX 6.0 SDK beta 版问世以来，通用文件（common file）就为许多游戏编程社团所接受了。通用文件框架（framework）可以帮助开发者提高开发速度，因为：

- 通常它可帮助你避免许多在 Direct3D 编程中所面临的“如何做”的问题，由此可使你的注意力更多地集中在真正的内容上；
- 它是一个普遍通用的且经过严格测试的基础起点，可大大减少你花在调试上的时间；
- DirectX SDK 中的所有 Direct3D 示例都使用了这个架构，因此学习如何使用它的难度不是很大；
- 它的窗口模式更有利于调试程序；
- 你自己开发的产品代码是可以基于这些通用文件的，所以掌握如何使用这些通用文件会是一种更高效的开发途径。

目录“C:\DXSDK\Samples\C++\Common\Src”中的 17 个*.cpp 文件列出了通用文件的一个高层视图。这些文件封装了大部分编写一个 Direct3D 应用程序时所需的基础功能。其中最重要的文件是 d3dapp.cpp，它包含了 CD3DApplication 类，并提供了 7 个可让你改写的函数。

- virtual HRESULT OneTimeSceneInit() { return S_OK; }
- virtual HRESULT InitDeviceObjects() { return S_OK; }
- virtual HRESULT RestoreDeviceObjects() { return S_OK; }
- virtual HRESULT DeleteDeviceObjects() { return S_OK; }
- virtual HRESULT Render() { return S_OK; }
- virtual HRESULT FrameMove(FLOAT) { return S_OK; }

■ virtual HRESULT FinalCleanup() { return S_OK; }

在基于该框架代码来创建一个应用程序时，你所要做的一切就是创建一个新项目，然后在主原文件中给出这些函数新的实现即可。这种方式也体现在所有 DirectX3D SDK 的 Direct3D 示例中。

这 7 个函数被称为通用文件框架的“公有接口（public interface）”。（见图 5.3）

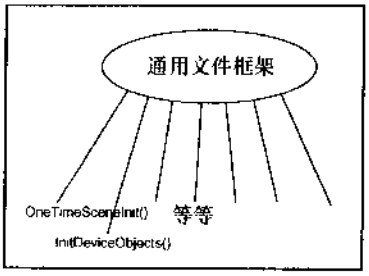


图 5.3 通用文件框架的公有接口

5.3 Basic 示例

通用文件框架中最重要的部分就是这个名为 CD3DApplication 的应用程序基类。

```
class CMyD3DApplication : public CD3DApplication
{
    LPDIRECT3DVERTEXBUFFER9 m_pVB; // Buffer to hold vertices
    DWORD m_dwSizeofVertices;
protected:
    HRESULT OneTimeSceneInit();
    HRESULT InitDeviceObjects();
    HRESULT RestoreDeviceObjects();
    HRESULT InvalidateDeviceObjects();
    HRESULT DeleteDeviceObjects();
    HRESULT FinalCleanup();
    HRESULT Render();
    HRESULT FrameMove();
    HRESULT ConfirmDevice (D3DCAPS9* pCaps,
                           DWORD dwBehavior,
                           D3DFORMAT adapterFormat,
    D3DFORMAT backBufferFormat );
public:
    CMyD3DApplication();
};
```

前两个变量分别保存了顶点缓冲器（vectex buffer）和所有顶点的大小。在接下来的所有示例中，我都将使用一项 Direct3D 的顶级特征来储存顶点——顶点缓冲器。通常，驱动程序从一个 Direct3D 缓冲器中复制数据，数据则保存在应用程序自己的缓冲中。而如果通过顶点缓冲器的话，那么数据在一开始就被复制到了驱动程序的缓冲中。也就是说，数据是存储在显卡可直接访问到的存储器中的，如此便可避免原先必须要从 Direct3D 应用程序缓冲复制数据到驱动程序缓冲这个步骤。这是一个很大的性能提升，可节省下大量的 CPU 周期。换句话说就是，管理一个顶点缓冲器就意味着允许应用程序分配和访问那些通常只能被驱动程序访问的存储器。通过 Lock()和 Unlock()函数来完成这样的操作。

尤其对于那些支持在硬件中进行变换和光照的普通显卡来说，顶点缓冲器能给予极大的性能提升。并且，在硬件中进行变换和光照也将会是所有即将面世的显卡应支持的一项基本

特征。因此有必要在一开始的 basic 示例中就使用顶点缓冲器。

现在让我们回到上述的那个应用程序类。在顶点缓冲器变量后还有许多方法。其中 ConfirmDevice()方法是最先被执行的，它用于检查显卡的能力，本书及 SDK 中的每个示例都使用到了它。你可以在其中指定对显卡的能力要求，若一些必需的能力不被支持的话，框架便会切换到使用参考光栅器，或切换为顶点软件处理。另外，此函数还可为一些不被支持的特征实现一种“回退”机制[1]。

以下代码可用于检查显卡是否支持 1.1 版的顶点 shader。

```
HRESULT CMyD3DApplication::ConfirmDevice( D3DCAPS9* pCaps, DWORD dwBehavior,
                                          D3DFORMAT adapterFormat,
                                          D3DFORMAT backBufferFormat )
{
    // check hardware support for vertex shaders
    if( (dwBehavior & D3DCREATE_HARDWARE_VERTEXPROCESSING ) ||
       (dwBehavior & D3DCREATE_MIXED_VERTEXPROCESSING ) )
    {
        // if there is no support choose software vertex shaders
        if( pCaps->VertexShaderVersion < D3DVS_VERSION(1,1) )
            return E_FAIL;
    }
    return S_OK;
}
```

若显卡不支持硬件或混合顶点处理，那么硬件也同样不会支持可编程的顶点 shader。那样的话，框架将会切换为软件顶点处理，并且顶点 shader 也将在一个软件模拟的模式中执行。

在调用了 ConfirmDevice()之后，框架将会执行以下函数来启动一个应用程序：

（1）ConfirmDevice()

（2）OneTimeSceneInit()

（3）InitDeviceObjects()

（4）RestoreDeviceObjects()

在应用程序运行期间，会循环地执行以下函数：

- FrameMove()
- Render()

以上两个函数是每个框架应用程序的主循环。另外，在程序运行期间，用户可能会改变窗口的大小。在这种情况下，框架将会调用：

译注[1]：

1）即尝试以另一种更低效的方法来运行，详见第 2 章“硬件抽象层”一节；

2）跟踪、阅读通用文件的源代码，以了解 CD3DEnumeration::BuildVertexProcessingTypeList()、CD3DApplication::ConfirmDeviceHelper()、ConfirmDevice()三者之间的调用关系，并理解 ConfirmDevice()在其中所“扮演的角色”以及它的返回值对框架后续执行的影响；

3）需要注意的是参考光栅器支持所有 Direct3D 特征，所以当检查到它的时候，应用程序对其各项能力检测都会成功，因此 Direct3D 应用程序始终可以在参考光栅器上运行，但会相当慢。

- InvalidateDeviceObjects()
- RestoreDeviceObjects()

如果用户通过按 F2 或点击“File—>Change Device”来更改设备（HAL 或 REF）的话，框架会调用：

- InvalidateDeviceObjects()
- DeleteDeviceObjects()
- InitDeviceObjects()
- RestoreDeviceObjects()

如果用户退出应用程序，框架会调用：

- InvalidateDeviceObjects()
- DeleteDeviceObjects()
- FinalCleanup()

这里有一些相互匹配的函数对。InvalidateDeviceObjects()销毁 RestoreDeviceObjects()所创建的东西，DeleteDeviceObjects()销毁 InitDeviceObjects()所创建的东西，FinalCleanup()函数销毁 OneTimeSceneInit()所创建的东西。这样设计的目的是为了赋予每对函数一个明确且相对独立的任务。在游戏生命周期的开始和结束时会调用 OneTimeSceneInit()/FinalCleanup()函数对。这两个函数分别用于加载和删除那些与设备无关的数据，几何数据便是典型的这种数据。每个应用程序在执行周期内需要的所有东西都是在 OneTimeSceneInit()调用中进行一次性永久初始化的。你可以在其中加载几何数据、计算相应的值等。基本上，任何一次性的资源分配都应在这里完成。

InitDeviceObjects()/DeleteDeviceObjects()函数对所处理的目标是那些与设备相关的数据，正如其名称所暗示的那样。如果设备发生改变，并且已加载的数据也要随之改变的话，那么就应该在这里进行重新加载。接下来的 Basic2、Basic3 等示例都将在这两个函数中对它们的顶点缓冲器、索引缓冲器、纹理进行加载、重新创建和销毁。

InvalidateDeviceObjects()/RestoreDeviceObjects()函数对用于对窗口大小的改变作出反应。例如，你可在此编写设置投影矩阵（projection matrix）的代码。另外，接下来的几个示例还会在 RestoreDeviceObjects()中设置大多数的渲染状态（render state）。还需注意一下 InvalidateDeviceObjects()的调用时机，它是在用户已改变了窗口大小或已更改了设备后被调用的，它必须对这些情况作出响应。而 RestoreDeviceObjects()的调用时机则是：应用程序开始时、或窗口大小已被改变时、或设备已被更改时[①]。由此可见，在游戏开始时是不会在 RestoreDeviceObjects()之前调用 InvalidateDeviceObjects()的。而在所有其他情况中，这两个方法都是成对调用的。

Render()方法的作用显而易见，它作为 3D 渲染的入口点在渲染每一帧时被调用。在其中可以设置渲染状态、清空视口以及渲染一个场景。在一个动画程序中，FrameMove()方法用

译注①：后两个时机与 InvalidateDeviceObjects()相同，会紧接着 InvalidateDeviceObjects()调用 RestoreDeviceObjects()。

于容纳所有动画相关的代码，包括更新矩阵、更新纹理坐标、更新对象坐标以及其他一些根据时间而变化的行为。最后一个框架函数 FinalCleanup()，可用于销毁为几何数据分配的内存、删除文件对象等。

为了使你对通用文件中每个方法的内部运作有一个更深刻的理解，我将从头至尾介绍一下 basic.cpp。

5.3.1 ConfirmDevice()、OneTimeSceneInit()和 InitDeviceObjects()函数

由于这个示例不需要任何特定的设备能力，所以 ConfirmDevice()函数为空。这个示例仅使用 4 个顶点来显示一个具有颜色的窗口客户区。为了便于处理，我将在 RestoreDeviceObjects()方法中，随同顶点缓冲器创建一起加载这些顶点。因此，OneTimeSceneInit()函数也为空，然而通常它用于加载场景中的模型几何数据。此外，在这个示例中不需要初始化任何与设备相关的东西，诸如字体类或纹理，所以 InitDeviceObjects()也为空。

```
HRESULT CMyD3DApplication::ConfirmDevice( D3DCAPS9* pCaps, DWORD dwBehavior,
                                          D3DFORMAT adapterFormat,
                                          D3DFORMAT backBufferFormat )
{
    return S_OK;
}

HRESULT CMyD3DApplication::OneTimeSceneInit()
{
    return S_OK;
}

HRESULT CMyD3DApplication::InitDeviceObjects()
{
    return S_OK;
}
```

请注意，所有的这些函数都返回一个 HRESULT 类型的值 S_OK，该值表示一切运行正常。

5.3.2 RestoreDeviceObjects()方法

当应用程序启动时，在 InitDeviceObjects()之后调用了 RestoreDeviceObjects()方法。而当用户在应用程序运行过程中调整了窗口的大小时，则会先调用 InvalidateDeviceObjects()，然后再调用 RestoreDeviceObjects()。第 1 个示例中大多数真正有意义的内容都在这里。

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    // fill the vertex buffer with the data for a quad
    VERTEX Vertices[] =
```

```
    {
        { 0.0f, 0.0f, 0.5f, 1.0f, 0x00ff0000, }, // x, y, z, rhw, color
        {(float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 0x00ff0000, },
        {(float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f,
            1.0f, 0x0000ff00, },
        {0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0x0000ffff, },
    };

    m_dwSizeofVertices = sizeof(Vertices);

    // sixth parameter is new in DX9
    if( FAILED( m_pd3dDevice->CreateVertexBuffer(m_dwSizeofVertices,
                                                 D3DUSAGE_WRITEONLY,
                                                 FVF,
                                                 D3DPOOL_MANAGED, &m_pVB, NULL ) ) )
                return E_FAIL;

    VERTEX* pVertices;
    // DX9: the third parameter has changed from BYTE** to VOID**
    if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (VOID**)&pVertices, 0 ) ) )
        return E_FAIL;
    memcpy( pVertices, Vertices, m_dwSizeofVertices);
    m_pVB->Unlock();

    return S_OK;
}
```

顶点结构说明了储存在顶点缓冲器中的数据的布局。

```
// A structure for our custom vertex type
struct VERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex
    DWORD color;        // The vertex color
};

// Fixed-Function Vertex structure
// In DX8: #define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
const DWORD FVF = (D3DFVF_XYZRHW | D3DFVF_DIFFUSE);
```

在 RestoreDeviceObjects()中，使用位置数据、一个 RHW 值和一个颜色值来填充了 4 个结构（见图 5.4）。

```
// fill the vertex buffer with the data for a quad
VERTEX Vertices[] =
{
    { 0.0f, 0.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 0xffff0000, },
    { (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f,
      1.0f, 0xff00ff00, },
    { 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0xff00ffff, },
};
```

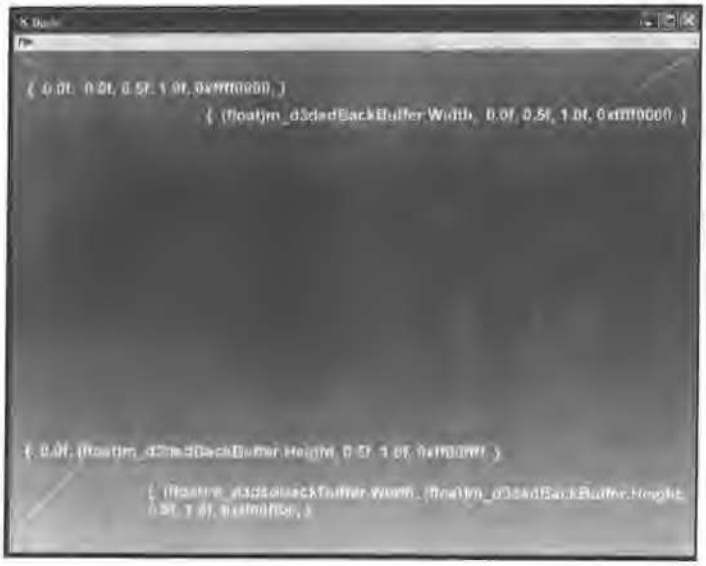


图 5.4　basic 示例的顶点分布

所有顶点都是在屏幕空间（screen space）中给出的，正如你所见，以上代码使用了后备缓冲的宽度和长度来指定顶点位置。术语“屏幕空间”指的是一个参考框架（frame of reference），该参考框架的坐标直接与后备缓冲中的 2D 位置相对应。[①]

由于不需要以任何方式来变换顶点，所有就不需要有另一个诸如世界坐标系或模型坐标系之类的参考框架了。这是处理这些 Direct3D 中顶点最有效的方法。

这里没有做任何世界空间、观察空间、投影空间的变换。所以，第 1 个顶点（即顶点数组的第 1 个元素）位于窗口客户区的左上角，顶点的 x、y 坐标为(0.0f, 0.0f)；顶点数组的第 2 个元素描述了第 2 个顶点，位于右上角；第 3 个元素描述了第 3 个顶点，位于右下角；数组的第 4 个顶点元素描述了第 4 个顶点，位于左下角。每个顶点都有一个 z 值为 0.5f。改变这个值不会引起任何变化，因为这个示例不使用 z 缓冲器（z-buffer），如 CMyD3DApplication 构造函数中所示：

```
m_d3dEnumeration.AppUsesDepthBuffer = FALSE
```

在上述顶点数组中每行的第 4 项称为“RHW 值”。它告知 Direct3D 流水线顶点已经变换到了齐次空间（homogenous space），在这个示例中会这样告知 Direct3D 是因为顶点已经在屏幕空间中了。（在第 6 章“动画初步”中将会介绍更多关于这方面的内容。）最后一项是颜色值。从 0x 的使用中可以看到这里使用了十六进制的数值。由于 4 个颜色值（ARGB）都是

译注①：这里所谓的“屏幕空间”就是指基于后备缓冲的高度和宽度所形成的一个 2D 区域，其中后备缓冲的左上角为原点(0, 0)。这里是将 3D 空间中的点直接对应到 2D 空间中去。在指定 3D 顶点坐标时使用 2D 的“屏幕空间”坐标，主要是为了避开坐标变换，见下文。

通过一组十六进制值来表示的，所以非常便于书写和理解。第 1 对 ff 表示 alpha，第 2 对代表红色，依此类推。因此，左上角和右上角的顶点都是红色的。大多数绘画程序，如 Adobe Photoshop 或 Jasc Paint Shop Pro，都提供了一个颜色选择对话框来告诉你 RGB 颜色所对应的十六进制值（见图 5.5）。

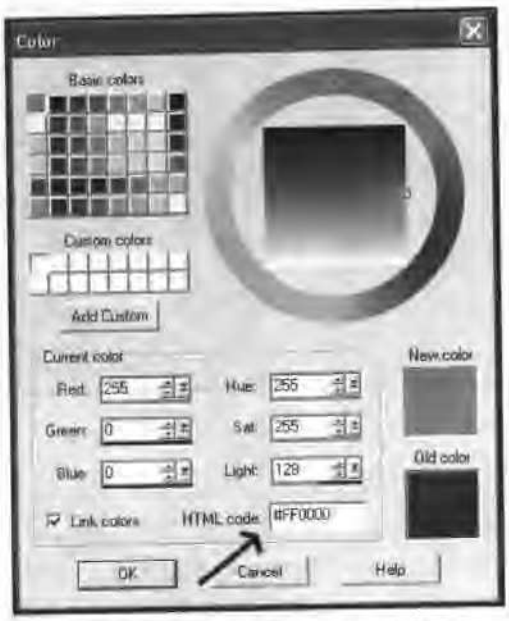


图 5.5　Jasc Paint Shop Pro 中的颜色拾取器

可通过以下代码来创建顶点缓冲器，然后便可使用顶点缓冲器来保存这些顶点值了。

```
if( FAILED( m_pd3dDevice->CreateVertexBuffer(
                                        // size of vertex
                                        m_dwSizeofVertices,
                                        // buffer usage
                                        D3DUSAGE_WRITEONLY,
                                        // description of vertex buffer
                                        FVF,
                                        // memory pool
                                        D3DPOOL_MANAGED,
                                        // pointer to vertex buffer
                                        &m_pVB, NULL ) ) )
    return E_FAIL;
```

创建顶点缓冲器时所指定类型参数的正确与否，很大程度上影响着应用程序的性能。CreateVertexBuffer()函数就是用于创建顶点缓冲器的。m_dwSizeofVertices 保存着所有顶点的大小，FVF 保存着顶点描述，m_pVB 保存着返回顶点缓冲器接口的指针。顶点大小的含义显而易见，但有必要解释一下顶点描述。这个参数是一个用于描述当前图元集合（set of primitive）中顶点格式的标志。换句话说，它告诉了 Direct3D 一些关于顶点结构的情况，而

我们就是使用顶点结构来保存顶点信息的。顶点标志定义在 basic.cpp 文件起始处，定义语句如下：

```
// Fixed-Function Vertex structure
// In DX8: #define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
const DWORD FVF = (D3DFVF_XYZRHW | D3DFVF_DIFFUSE);
```

这些标志告诉 Direct3D 引擎，数据在顶点缓冲器中是如何组织的。如这段源代码的注释所示，在 DirectX 8 升级到 DirectX 9 的过程中，标志的语法发生了一些变化。这些变化很容易应对，现在是用一个 const DWORD 变量来代替#define 保存顶点标志。d3d9types.h 头文件中声明了这些标志，这些标志可用以明确描述一个顶点格式。另外在这个文件中还提供了一些辅助宏，它们给出了一些标志的常用组合形式。每个绘制函数（如 DrawPrimitive*()系列）或 IDirect3DDevice9 的 CreateVertexBuffer()函数都需要一个这样的标志组合，并用它们来确定顶点缓冲器是如何构成及如何渲染一个图元的。这些标志主要是告诉系统，应用程序使用到了哪些顶点成分，诸如位置、法线、颜色、纹理坐标个数等。换句话说，一个特定的顶点格式标志的存在与否都向系统传达了这样一条信息——内存中有哪些顶点成分字段，以及哪些是你忽略不用的。一个应用程序若想节省存储开销，或者尽可能地降低渲染一个 3D 世界所需的处理带宽，那么就应该只使用那些必须使用的顶点成分。

附注

只有在使用固定功能（fixed-function）的流水线时，才需要以这种方式来定义顶点标志。一个使用顶点 shader 的程序是不需要使用这些标志的，因为顶点 shader 的顶点声明有一个等价的功能。我将在后面的一个示例程序中解释这个概念。

D3DFVF_XYZRHW 标志告知系统你的游戏引擎已对顶点作过了变换，Direct3D 不需要再对它们作变换。也就是说，Direct3D 就不会再用世界矩阵、观察矩阵或投射矩阵来变换顶点了。在这个示例程序中，顶点必须在屏幕空间中传递。

D3DFVF_XYZ 标志不同于 D3DFVF_XYZRHW，它告知系统你的游戏没有变换过顶点，要求 Direct3D 来做顶点变换的工作。

D3DFVF_DIFFUSE 表示顶点格式中包含了一个漫反射（diffuse）颜色成分。使用 D3DFVF_SPECULAR 表示顶点格式中包含一个镜面反射（specular）颜色成分。

附注

当使用顶点 shader 代替固定功能的流水线时，便完全不是这么一回事了。在使用顶点 shader 时，顶点必须在 HCLIP 空间中传递，也就意味着在−1.0f～1.0f 的范围内。在第 6 章中可看到这样的示例。

当使用固定功能的流水线时，顶点元素和顶点输入寄存器的映射关系是固定的。因此特定的顶点成分（如位置或法线）必然会被放到顶点输入存储器的特定寄存器中去。所以，顶

点结构中的各个成分必须要以一种特定的次序来放置，如下所示：

（1）变换过的/未变换的顶点位置，浮点数 x、y、z 坐标；

（2）RHW，齐次 w 坐标的倒数（仅适用于已变换过的顶点）；

（3）混合（Blending）权重值，为 1～5 之间的浮点数；

（4）顶点法线，浮点数 x、y、z 构成的法线坐标；

（5）顶点的点大小[1]；

（6）漫反射颜色，漫反射的 RGBA 值；

（7）镜面反射颜色，镜面反射的 RGBA 值；

（8）纹理坐标集合 1～8，浮点数 u 和 v 构成的坐标值。

例如，你的顶点结构可以是这样的：

```
typedef struct
{
    FLOAT x,y,z;      // position
    DWORD diffuse;    // diffuse color
    FLOAT u,v;        // one pair of texture coordinates
} MYVERTEX;
```

你必须按如下方法来定义这个顶点结构的标志：

```
const DWORD D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1);
```

另一个示例是：

```
struct LineVertex
{
   FLOAT x, y, z; // position
};
const DWORD LINE_VERTEX = ( D3DFVF_XYZ );
```

以下是一个更复杂的示例：

```
typedef struct SObjVertex
{
   FLOAT x, y, z;              // position
   FLOAT nx, ny, nz;           // normal
   DWORD diffuse;              // diffuse color
   DWORD specular;             // specular color
   FLOAT tu, tv;               // first pair of texture coordinates
   FLOAT tu2, tv2, tw2;        // second pair of texture coordinates
   FLOAT tu3, tv3;             // third pair of texture coordinates
   FLOAT tu4, tv4;             // fourth pair of texture coordinates
} SObjVertex;
...
const DWORD gSObjVertexFVF = (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_SPECULAR |
                              D3DFVF_NORMAL | D3DFVF_TEX4 |
                              D3DFVF_TEXCOORDSIZE2(0) | D3DFVF_TEXCOORDSIZE3(1) |
```

译注[1]：每个顶点在摄像机空间或设备空间中的大小，详见 SDK 文档。

```
D3DFVF_TEXCOORDSIZE2(2) | D3DFVF_TEXCOORDSIZE2(3));
```

D3DFVF_TEX4 标志指示 Direct3D 引擎，这里将会使用到 4 个顶点坐标。D3DFVF_TEXCOORDSIZE*标志指定了每组纹理坐标的值的个数（2 或 3）和纹理坐标的次序。比如，第 2 组纹理坐标由 3 个浮点值分量组成。这种结构对于立方体纹理或体纹理十分有用。

另一个很重要的概念是 Direct3D 的资源管理。在 Direct3D 8.0 中引入了一个资源管理器，专门负责对纹理和几何数据实施统一的管理。它可用于管理 mipmap、体贴图、立方体贴图、顶点缓冲器和索引缓冲器。

在调用 CreateVertexBuffer()方法时，可传入一个 D3DPOOL_MANAGED 标志来指定使用这个资源管理器。这个标志定义了用于保存顶点缓冲器的存储器类型。现在你或许想要问：为什么要使用不同的存储器类型呢？原因是这样的，在显卡中有存储器（比如 128-MB DDRAM II）、在系统内存中也有存储器（比如 1024-MB RAM），Direct3D 在处理应用程序的存储器请求时会按不同的任务类型使用正确的存储器，但你必须提示它你想用资源来完成什么任务。另外有一点要说明的是，资源管理器只能管理那些使用了 D3DPOOL_MANAGE 内存池标志的资源。

在 Direct3D 中，选择一种管理资源的方法（或定义用于保存资源的存储器类型）称为“选择一个池（choosing a pool）”。在 DirectX 9 中有 4 种不同类型的池。

```
typedef enum _D3DPOOL {
    D3DPOOL_DEFAULT = 0,        // AGP, graphics card memory
    D3DPOOL_MANAGED = 1,        // AGP, graphics card memory && copy in sys memory
    D3DPOOL_SYSTEMMEM = 2,      // system memory, re-creatable
    D3DPOOL_SCRATCH = 3,        // system memory, not re-creatable
    D3DPOOL_FORCE_DWORD = 0x7fffffff
} D3DPOOL;
```

如果指定一个临时池（scratch pool），资源会存储在系统内存中，并且当设备丢失时，将无法重新创建它。这种设备丢失的情况可能会发生在窗口模式和全屏模式之间切换时，或在这两种模式中改变分辨率时。Direct3D 设备无法访问这些资源，它们也不能被设置为纹理或渲染目标。然而，它们总是可以被成功地创建出来、被锁定或被复制。DirectX 9 引入这种内存池的原因是由于它从 API 中删除了 CreateImageSurface()函数。通常，D3DPOOL_SCRATCH 是连同 CreateOffscreenPlainSurface()一起使用的，后者将返回一个与之前通过 CreateImageSurface()来创建的、具有相同特征的表面。因此，它主要用于创建图像表面。

如果指定一个系统池（system pool），资源会储存在系统内存中。这种池的优势在于设备丢失后，资源不需要重新创建。另外，可以在池中锁定和更新资源。因此，对于那些经常需要通过软件来处理的数据，推荐使用这种池。

当指定了一个托管池（managed pool）时，资源会储存在显卡或 AGP 存储器中，并且会留有一份拷贝在系统内存中。较之系统池，这类池的优势在于资源可位于显卡或 AGP 存储器中，而且只有在系统内存中的资源数据发生变化时才需要进行一次复制，将数据更新到 AGP/

显卡存储器中。

当指定了一个默认池（default pool）时，资源通常储存在显卡或 AGP 存储器中。它们对于渲染设备而言是最易于访问的存储器类型。在顶点缓冲器和索引缓冲器经常需要更新的情况下，就必须使用这种池。因为系统池和托管池所做的“系统内存到 AGP/显卡存储器的复制”是非常慢的。相对于托管池而言，你应当优先考虑将所有资源分配为默认池资源。如果不这么做的话，将会导致一定程度的性能损失。默认池的缺点在于它必须由应用程序自己来管理。

附注

AGP（Accelerated Graphics Port，加速图形端口）是一种基于 PCI 的接口，它是专门为 3D 图形应用程序的需求而设计的。32 位 AGP 通道直接将图形控制器连接到主存储器。尽管通道只以 66 MHz 的频率运行，但它支持在时钟周期的上升沿和下降沿时刻进行数据传送，所以这使它实际能达到 133 MHz 的高效运行速度。

每一种池都有其自身的功能和用途。或许你会认为默认池可以更有效地更新动态的顶点缓冲器和索引缓冲器。的确，默认池对于那些不需要被应用程序修改的纹理来说是一个很好的选择。另一方面，经常需要被修改的纹理可通过临时池、系统池或托管池来访问。

如果性能不是很重要的话，使用托管池永远是一种安全的选择。这是将存储器映射到设备中最容易且最好的方法。资源会在必要时被自动地复制到设备可访问的存储器中。托管资源（managed resource）是由系统内存支持的，并且当设备丢失时，不需要重新创建。根据我的经验：除非你知道更好的方法，否则就应使用托管池。它可以在任何类型的驱动程序上运行。并且对于具有一体化内存架构（Unified Memory Architecture）的主板，你必须要使用它。因为在这些主板上，显存与系统内存（或 RAM）位于同一个硬件单元上。

这也就是为什么我在刚才所讨论的 CreateVertexBuffer()函数中为顶点缓冲器选择托管池的原因。这个函数的第 5 个参数返回一个指向顶点缓冲器的指针。在 DirectX 9 中，新近引入了第 6 个保留参数，应将其设置为 NULL。也就是说，它在当前还没有什么作用，但可能会在下一个 DirectX 版本中使用到。

在创建了顶点缓冲器并使用 Lock()锁定后，便可以用 memcpy()来填充顶点缓冲器了。

```
VERTEX* pVertices;
// DX9: the third parameter has changed from BYTE** to VOID**
if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (VOID**)&pVertices, 0 ) ) )
            return E_FAIL;
memcpy( pVertices, Vertices, m_dwSizeofVertices);
m_pVB->Unlock();
```

锁定一个资源意味着允许 CPU 访问其存储内容。除了允许处理器访问外，其他任何涉及资源的操作都会在锁定期间被串行化（serialize）。对一个资源同一时刻只能锁定一次，那怕

第二次锁定是一个非重叠的区域也不行。并且当表面未解锁前，禁止在此表面进行任何加速器操作。Lock()函数的语法为：

```
HRESULT Lock(
    UINT OffsetToLock,      // offset into the vertex buffer
    UINT SizeToLock,        // == vertices * SizeOfVertex
    VOID ** ppbData,        // pointer to memory buffer
    DWORD Flags             // hints on the usage of the vertex buffer
);
```

第 1 个参数让程序员可为顶点缓冲器的锁定操作提供一个起始偏移量。比如，与第 2 个参数一起使用时，便可让你将几个模型放入一个顶点缓冲器中，并仅锁定某个特定模型所在的区域。

第 2 个大小参数应指定为，顶点的数量乘以一个顶点的大小。第 3 个参数返回一个指向内存缓冲区的指针。最后一个参数让程序员对如何使用顶点缓冲器给出以下提示：

- D3DLOCK_DISCARD，应用程序将覆盖顶点缓冲器中的所有位置；
- D3DLOCK_READONLY，应用程序不会写入顶点缓冲器；
- D3DLOCK_NOOVERWRITE，顶点缓冲器中的数据不会被覆盖；
- D3DLOCK_NOSYSLOCK，允许在 Lock()期间改变显示模式；
- D3DLOCK_NO_DIRTY_UPDATE，对锁定的顶点缓冲器不添加 dirty 区域[①]。

D3DLOCK_DISCARD 与 D3DLOCK_NOOVERWRITE 标志仅对那些会被有规律更新（指定了 D3DUSAGE_DYNAMIC 标志）的缓冲器才有效，而这里的情况并非如此。D3DLOCK_READONLY 标志只有在应用程序不会向缓冲器写入时才有用。它使资源以非原生（non-native）格式存储，省去了在解锁时再次压缩的步骤。程序可能无法提供一个有用的标志，所以在最后一个参数中提供 0 也是允许的，这提示驱动程序让它来决定该如何对顶点缓冲器的访问进行优化。

以顶点数据填充顶点缓冲器后，必须调用 m_pVB->Unlock()来解除锁定。这是设备相关对象初始化和恢复的最后一个步骤。直到现在为止，游戏已通过 ConfirmDevice()、OneTimeSceneInit()、InitDeviceObjects()、RestoreDeviceObjects()经历了完整的初始化过程。接下来进入渲染循环，该循环主要由 FrameMove()和 Render()的函数调用组成。

5.3.3　FrameMove()函数

由于场景不需要动画，所以在 FrameMove()中无需包括任何代码。在接下来的几个示例中，很多事情都在这里发生。

```
HRESULT CMyD3DApplication::FrameMove()
```

译注：默认情况下锁定操作会给资源添加一个标志，以表示锁定区域已被修改过了。从而对于托管池来说，可能就需要将内存中的资源数据更新到 AGP/显卡存储器中。其中涉及到的这个锁定区域就称为“dirty 区域”。

```
{
    return S_OK;
}
```

5.3.4 Render()函数

在每一帧渲染时都会调用 Render()方法，并且它还是 3D 渲染的入口点。在其中可以设置渲染状态、清除渲染目标以及渲染一个场景。

```
HRESULT CMyD3DApplication::Render()
{
    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        // In DX8: m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
        // Passing an FVF to IDirect3DDevice9::SetFVF specifies a legacy FVF with stream 0.
        m_pd3dDevice->SetFVF(FVF ); // new in DX9
        // DX8: only three parameters. In DX9 the third parameter is new
        m_pd3dDevice->SetStreamSource( 0, m_pVB, 0, sizeof(VERTEX) );
        m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
        // End the scene.
        m_pd3dDevice->EndScene();
    }
    return S_OK;
}
```

Render()调用了 BeginScene()/EndScene()函数对，前者在开始渲染之前调用，后者在渲染之后调用。BeginScene()会使系统检查其内部数据结构和渲染表面的可用性与有效性。它还会设置一个内部标志来标识一个场景正在处理过程中。当一个场景不处于处理状态时，尝试调用任何渲染方法都将失败，并返回 D3DERR_SCENE_NOT_IN_SCENE。当渲染完成后，必须调用 EndScene()，它将清除那个标识场景处于处理过程的内部标志，刷新缓存数据以及确认渲染表面是否完好。

附注

你应避免在一个应用程序中多次使用 BeginScene()/EndScene()函数对，因为像 PowerVR Kyro 和 Kyro II （http://www.powervr.com）这样的场景捕获（scene-capture）图形芯片对于多次调用 BeginScene()/EndScene()函数对的情况是无法运行的。

Direct3D 可以从一条或多条数据流中装配每个顶点，这些顶点被用于 Direct3D 流水线的不同处理环节。例如，第 1 条数据流可以保存所有位置和法线，第 2 条保存所有颜色值，第 3 条保存所有纹理坐标。SetStreamSource()函数就是用于支持这样的功能。这个方法的声明为：

```
HRESULT SetStreamSource(
    UINT StreamNumber,
```

```
    IDirect3DVertexBuffer9 *pStreamData,
    UINT OffsetInBytes,
    UINT Stride
);
```

第 1 个参数定义了一条数据流。根据底层硬件的不同，最多可以有 16 条不同的数据流。DirectX 7 兼容的硬件只支持一条数据流，但一些较新的硬件可以支持最多 16 条数据流。

第 2 个参数提供了该流的数据。通常，它是一个指向顶点缓冲器的指针，在此便是 m_pVB。

当你使用一个固定功能的流水线时（就如本例这种情况），顶点流的跨距必须与 FVF 计算出来的顶点尺寸相符合。在这个示例中，这意味着必须在第 4 个参数中提供 VERTEX 结构的大小。因此，使用固定功能的流水线时，不能将一个顶点缓冲器用于几个流。

DirectX 9 SDK 中的 Fur 演示就是一个很好的例子，它说明如何通过顶点 shader 来使用流。下面的伪代码说明了这个演示是如何使用流的。

```
All passes: stream 0: Position, Normal, Diffuse vertex color
if (finpass)
    fin pass stream 1: texture coordinates for fins
if (shellpass)
    shells pass stream 2: texture coordinates for shells
```

在每条程序执行通路中，SetStreamSource()都设置了 0 号流用于保存位置、法线、漫反射颜色值。当增加了另一条执行通路（finpass）时，便通过 SetStreamSource()来设置了 1 号流，从而选择 fin（毛翅、毛刺）的纹理坐标。另外也可以在另一条执行通路（shellpass）中用 SetStreamSource()来设置 2 号流，以选择 shell（外壳）的纹理坐标。fin 和 shell 执行通路的打开或关闭可使整个系统具有伸缩性。例如，当一个具有皮毛的对象位于非常远的距离时，显示其皮毛是没有意义的，因为在这么远的距离上根本看不到它。

附注

通过顶点声明使用顶点 shader 时，跨距应大于或等于由顶点声明计算出来的大小。这才有可能将一个顶点缓冲器用于几条流。

请注意 DirectX 9 中新版本的 SetStreamSource()添加了设置一个偏移量的功能，类似于用来锁定顶点缓冲器的 Lock()函数。这使得将多个游戏模型的数据置于一个顶点缓冲器中变得更加容易。

DirectX 9 中的另一个变化是引入了 SetFVF()函数，它仅用于固定功能的流水线。如果使用了一个顶点 shader，则必须用 SetVertexShader()来代替。在这里没有使用到顶点 shader，所以应使用 SetFVF()函数。该函数的惟一参数就是顶点格式。

```
m_pd3dDevice->SetFVF( FVF ); // new in DX9
```

在 Render()中最重要的函数调用是：

```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

自从 7.0 版本之后，DrawPrimitive()有了一些变化。它从当前数据输入流的集合中渲染指定类型的非索引几何图元。在本例中，仅有一条由 SetStreamSource()声明的数据流。DrawPrimitive()的语法是：

```
HRESULT DrawPrimitive(
    D3DPRIMITIVETYPE PrimitiveType,
    UINT StartVertex,
    UINT PrimitiveCount
);
```

第 1 个参数定义了要使用的图元类型。共有 6 种不同的图元类型：D3DPT_POINTLIST、D3DPT_LINELIST、D3DPT_LINESTRIP、D3DPT_TRIANGLELIST、D3DPT_TRIANGLESTRIP、D3DPT_TRIANGLEFAN。

```
typedef enum _D3DPRIMITIVETYPE {
D3DPT_POINTLIST = 1,
D3DPT_LINELIST = 2,
D3DPT_LINESTRIP = 3,
D3DPT_TRIANGLELIST = 4,
D3DPT_TRIANGLESTRIP = 5,
D3DPT_TRIANGLEFAN = 6
D3DPT_FORCE_DWORD = 0x7fffffff,
} D3DPRIMITIVETYPE;
```

本例使用了三角扇形（D3DPT_TRIANGLEFAN），其中所有的三角形共用同一个顶点（见图 5.6）。

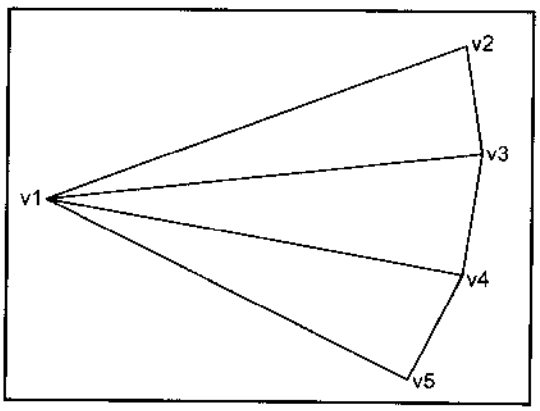


图 5.6　一个由 3 个三角形组成的扇形（三角扇形）

其中我们只使用了 2 个三角形，因此不太容易找到它们所共用的顶点。所以乍看之下，很难察觉出我们使用的是一个三角扇形（见图 5.7）。

DrawPrimitive()的第 2 个参数让用户可为顶点选择一个起始索引。这点与在 Lock()、SetStreamSource()中的起始索引概念一致。最后一个参数应设为图元的个数。表 5.1 可帮助你

在非索引的顶点情况中计算图元个数。

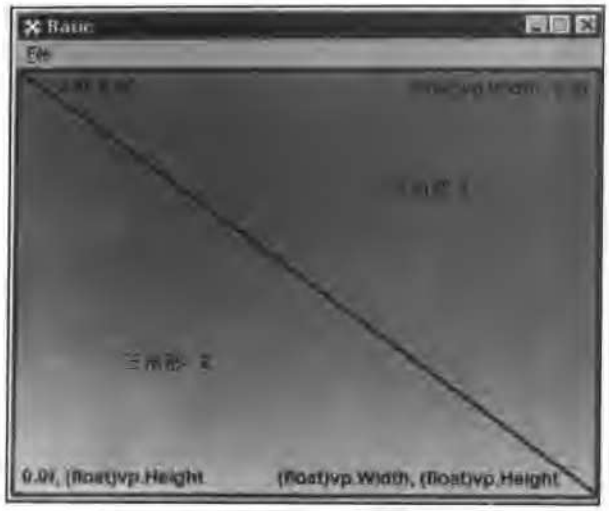


图 5.7　示例程序中由 2 个三角形组成的扇形

表 5.1　　图元个数（以下 n 表示顶点个数）

D3D 图元类型	图元个数
D3DPT_POINTLIST	PrimitiveCount = n
D3DPT_LINELIST	PrimitiveCount = n/2
D3DPT_LINESTRIP	PrimitiveCount = n-1
D3DPT_TRIANGLELIST	PrimitiveCount = n/3
D3DPT_TRIANGLESTRIP	PrimitiveCount = n-2
D3DPT_TRIANGLEFAN	PrimitiveCount = n-2

在本例中，顶点个数是 4，所以图元个数为 2。再举一个例子，通过 4 个三角形所组成的扇形来构建一个四边形的话，则需要有 6 个顶点。

我已解释完了 DrawPrimitive()函数的最后一个参数，我们就此结束对这个主渲染循环（由 FrameMove()与 Render()组成）的介绍。

当用户更改了设备、或更改了分辨率、或退出应用程序时，框架将会离开这个渲染循环。随后将会执行 InvalidateDeviceObjects()和 DeleteDeviceObjects()函数中的清理代码。下一节将介绍这些代码。

5.3.5　InvalidateDeviceObjects()函数

当游戏关闭、或用户改变窗口大小、或改变设备，都将调用 InvalidateDeviceObjects()。

另外，在应用程序的启动阶段不会调用 InvalidateDeviceObjects()。

在本例中，当用户改变窗口大小时，会调用 InvalidateDeviceObjects()函数，随后再调用 RestoreDeviceObjects()来重新创建 InvalidateDeviceObjects()所销毁的东西。对顶点缓冲器来说始终是这样来处理的（例如，当用户每次改变窗口大小时）：在 InvalidateDeviceObjects()函数中释放它，然后再在 RestoreDeviceObjects()函数中重新创建。

```
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    SAFE_RELEASE( m_pVB );
    return S_OK;
}
```

SAFE_RELEASE 宏在 dxutil.h 中定义，如下：

```
#define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
```

Release()是每个 DirectX COM 对象都会实现的一个标准释放函数。它会释放对象所占用的全部系统资源。

5.3.6 DeleteDeviceObjects()函数

DeleteDeviceObjects()是与 InitDeviceObjects()相对应的函数。因此，它用于删除 InitDeviceObjects()函数所创建的东西。在本例中，由于 InitDeviceObjects()没有初始化任何与设备相关的东西。所以在这里，你无需删除任何东西。

```
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    return S_OK;
}
```

5.3.7 FinalCleanup()函数

框架会调用的最后一个函数是 FinalCleanup()。这是最后一次机会，对已分配的内存进行释放或对已占用的资源进行销毁。

```
HRESULT CMyD3DApplication::FinalCleanup()
{
    return S_OK;
}
```

由于在 FinalCleanup()的对应函数 OneTimeSceneInit()中没有占用什么资源（如几何数据或纹理），所以在这里也没有什么东西需要释放。

你已经了解了第一个 Direct3D 示例，恭喜！

5.4 Basic2 示例

与前一个示例相比，这个示例演示了以下这些额外的特征：

- 如何将纹理应用到对象上；
- 如何使用框架提供的字体类。

通过框架提供的字体类支持，这个示例可以显示出关于底层硬件及其性能方面的一些有用数据（见图 5.8）。

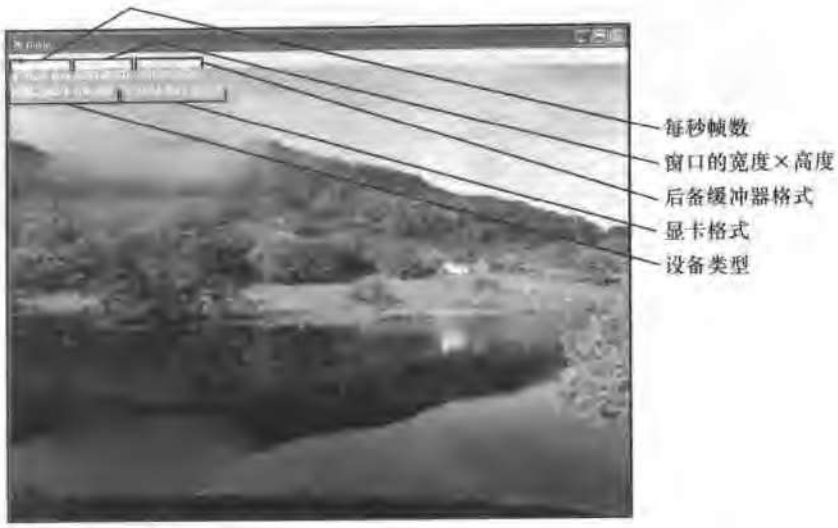


图 5.8 Basic2 示例

帧速率计数器表示这个示例一秒钟可以显示多少帧。在帧速率计数器的右边显示了窗口尺寸。在本例中，窗口大小为 800×600 像素。你可以通过调整窗口的大小来改变这个值。窗口尺寸的右边是后备缓冲器（back buffer）的格式。这可以帮助你识别一些由于纹理或资源不兼容于后备缓冲器格式而产生的问题。

第二行显示这个程序所选择的 HAL 类型。我在第 2 章中曾介绍过各种不同类型的 HAL。如果选择了参考光栅器，则该值为 REF。第二行最后一条信息是编写 HAL 的显卡厂商名称，我这里显示出“NVIDIA”的字样。虽然我的显卡是 Leadtek（丽台）出品的，但我使用的却是 NVIDIA 的参考显卡驱动程序。

除了这些通过字体类显示出来的文字信息外，窗口客户区中还显示了一张纹理。我使用了 DirectX 9.0 SDK 的 media 文件夹中的一张纹理。可以看出，在本例中这张纹理看上去有比较明显的像素粒子（见图 5.9）。不过，你会在第 7 章“纹理映射基础”中学习到如何过滤纹理，从而可以有效地解决这个问题。

图 5.9 纹理映射

为了在一个框架示例中使用一张 2D 的纹理贴图，必须：

- 在顶点结构中使用 2 个额外的纹理坐标变量，并将它们一起填充到顶点缓冲器中；
- 在 InitDeviceObjects()函数中，使用 D3DUtil_CreateTexture()创建纹理；
- 在 Render()函数中，根据相应的状态使用 SetTexture()设置纹理；
- 在 DeleteDeviceObjects()中，使用 SAFE_RELEASE()宏释放纹理。

为了记住这些步骤，可以想象一下一张纹理的生命周期。你必须先创建一张纹理，然后使用它（设置纹理），最后为了释放它所占用的内存而删除它（释放纹理）。纹理的格式以及对纹理可进行的操作都依赖于显卡的能力，它是一种与设备相关的资源。因此，必须要在一个与具体设备相关的上下文中创建它。而 InitDeviceObjects()/ DeleteDeviceObjects()函数便是这样一个理想的地方。

使用 d3dfont.cpp 中的字体类比使用纹理简单得多。为了使用一种字体来显示文本，可按以下步骤操作：

（1）在应用程序类的构造函数中，创建字体类；

（2）在应用程序类的 InitDeviceObjects()函数中，调用字体类的 InitDeviceObjects()函数；

（3）在应用程序类的 RestoreDeviceObjects()函数中，调用字体类的 RestoreDeviceObjects()函数；

（4）在应用程序类的 Render()函数（应用程序类中渲染循环的一部分）中，调用字体类的 DrawText()函数；

（5）在应用程序类的 InvalidateDeviceObjects()函数中，调用字体类的 InvalidateDeviceObjects()函数；

（6）在应用程序类的 DeleteDeviceObjects()函数中，调用字体类的 DeleteDeviceObjects()

函数；

（7）在应用程序类的析构函数中，删除字体类。

正如你所见，d3dfont.cpp 中字体类所使用的函数名与框架所使用的函数名相同。因此，你在使用字体类时惟一要做的事情就是在各个框架函数中调用字体类的相应同名函数即可。

现在让我们来看一下如何在 Basic2 示例程序中完成所有这些事情。由于这个简单的示例程序不要求任何特殊的显卡能力，所以 Basic2 的 ConfirmDevice()函数为空。另外，由于这里要加载的几何体的数据量与前一个示例差不多，所以 OneTimeSceneInit()也为空。因而，在这里我就不再提及这些函数了。

在应用程序类的构造函数中，出现了第一段新的代码。为了创建字体类，必须在应用程序类的构造函数中调用字体类的构造函数。这里使用的字体是 12 号粗体 Arial。

```
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle                    = _T("Basic");
    m_d3dEnumeration.AppUsesDepthBuffer = FALSE;
    m_dwCreationWidth                   = 800; // Width used to create window
    m_dwCreationHeight                  = 600;
    m_pVB = NULL;
    m_pFont = new CD3DFont( _T("Arial"), 12, D3DFONT_BOLD );
}
```

在前一个示例中，我没有涉及到应用程序类的构造函数，那是为了尽量减少你要接受新信息的数量。然而，你现在必须要了解它，因为这个构造函数初始化了一些应用程序类的默认值。类构造函数定义了应用程序窗口的宽度和高度，并将窗口标题设置为字符串“Basic”。将那个布尔值设置为 FALSE 表示不会用到深度缓冲器，接着是将顶点缓冲器初始化为 NULL。另外，通过 new 操作符为字体类对象分配了必要的内存。你可以尝试修改一下这里的某些值，看看会有效果吧，比如，你可以将窗口的宽度和高度设为 640×480，或者修改一下窗口的标题、字体大小、字体类型等。

现在已经初始化好了应用程序类，接下去让我们来看一下本例中第一个与框架相关的函数 —— InitDeviceObjects()函数。

5.4.1　InitDeviceObjects()函数

InitDeviceObjects()函数用于初始化与设备相关的对象。当应用程序启动时或用户按 F2 改变设备时，将调用这个函数。为了使用字体类对象，还必须在这里调用字体类的函数 InitDeviceObjects()。

```
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    m_pFont->InitDeviceObjects( m_pd3dDevice );
```

```
    // Load the texture for the background image
    if( FAILED( D3DUtil_CreateTexture( m_pd3dDevice, _T("Lake.bmp"),
                                       &m_pBackgroundTexture) ) )
        return D3DAPPERR_MEDIANOTFOUND;

    return S_OK;
}
```

其中调用了 D3DUtil_CreateTexture()实用（utility）函数来创建背景纹理。它的第 1 个参数接受一个指向 Direct3D 设备的指针；第 2 个参数是纹理文件名；最后一个参数返回一个 LPDIRECT3DTEXTURE9 类型的指针，指向纹理对象。这个函数会为你完成所有复杂的初始化步骤，并会根据你提供的文件名来搜索一张相应的纹理，非常方便、好用。在附录 D"使用 D3DXCreateTextureFromFileEx()创建纹理"中，还将介绍一个功能更强大的纹理创建函数。

到目前为止，你已经为字体类对象分配了内存、创建了纹理并提供了一个句柄。

5.4.2 RestoreDeviceObjects()函数

RestoreDeviceObjects()函数与第一个示例相比并没有太大改变。当用户改变窗口大小或改变了设备时会调用这个函数。现在这个函数调用了字体类的函数 RestoreDeviceObjects()，以此来响应这类用户的交互操作。

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    // we don't need any transformation -> position will be delivered in screen coordinates

    m_pFont->RestoreDeviceObjects();

    // fill the vertex buffer with the new data
    // Initialize to render a quad
    VERTEX Vertices[] =
    {
        { 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv
        { (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f, },
        { (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height,
                  0.5f, 1.0f, 1.0f,1.0f, },
        { 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0.0f,1.0f, },
    };

    m_dwSizeofVertices = sizeof(Vertices);

    // sixth parameter is new in DX9
    // Create a square for rendering the background
    if( FAILED( m_pd3dDevice->CreateVertexBuffer(m_dwSizeofVertices,
                                                 D3DUSAGE_WRITEONLY, FVF,
                                                 D3DPOOL_MANAGED, &m_pVB, NULL ) ) )
```

```
        return E_FAIL;

    VERTEX* pVertices;
    // the third parameter changed from BYTE** to VOID** in DX9
    if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (VOID**)&pVertices, 0 ) ) )
        return E_FAIL;
    memcpy( pVertices, Vertices, m_dwSizeofVertices);
    m_pVB->Unlock();

    return S_OK;
}
```

在本例中必须改变顶点结构以处理纹理映射。

```
// A structure for our custom vertex type
struct VERTEX
{
    FLOAT x, y, z, rhw;      // The transformed position for the vertex
    FLOAT tu, tv;            // texture coordinates
};
```

现在每个顶点用两个纹理坐标替代了原先的颜色值，以此来布设（lay out）纹理映射。

```
{ 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv
{ (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f, },
{ (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 1.0f,1.0f, },
{ 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0.0f,1.0f, },
```

每个顶点的最后两个值分别是 tu 和 tv，它们对应于纹理坐标 u 和 v。为了将图素映射到图元上，Direct3D 要求所有图素都使用一个统一的地址范围。因此它使用了一种具有通用性的编址方案，其中所有图素地址都在区间[0.0, 1.0]中。

为了将纹理的一角"固定"到左上角的顶点上，可将 u 值和 v 值都设为 0.0f；而把 u 值和 v 值都设为 1.0f，则可将纹理的右下角"固定"到右下角的顶点上。纹理地址空间与屏幕空间一样，都是以左上角为原点，如图 5.10 所示。

顶点标志也需反映出顶点结构的不同布局。

```
// Fixed-Function Vertex structure
const DWORD FVF = (D3DFVF_XYZRHW | D3DFVF_TEX1);
```

通过以上的 FVF 定义，Direct3D 引擎会期望每个顶点结构具有 4 个用于描述顶点位置的浮点数和一对用于保存纹理坐标的浮点数。

与前例相比，RestoreDeviceObjects()函数的其他部分没有什么变化。顶点缓冲器以相同的方式来创建、锁定和填充。

图 5.10　纹理布局

你可以跳过这里的框架函数 FrameMove()，因为不需要为任何东西产生动画，所以它是空的。继续向前，来看一下应用程序主循环中的 Render()函数。

5.4.3　Render()函数

在这个示例中，Render()函数较之前例有两处新变化。首先它调用了字体类的绘制代码，其次将纹理映射到 4 个顶点上。

```
// Begin the scene
if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
{
    // Set up texture stage states for the background
    m_pd3dDevice->SetTexture( 0, m_pBackgroundTexture );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );

    // Passing an FVF to IDirect3DDevice9::SetFVF specifies a legacy FVF with stream 0.
    m_pd3dDevice->SetFVF(FVF );
    m_pd3dDevice->SetStreamSource( 0, m_pVB, 0, sizeof(VERTEX) );
    m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );

    // Output statistics
    m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );
    m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );

    // End the scene.
    m_pd3dDevice->EndScene();
}
```

与前例相比，本例中多了两段新的代码。第一段代码通过一个主函数 SetTexture()设置了

一张纹理，第二段代码使用 DrawText()来定位和设置将显示在窗口客户区中的文本。我先来介绍前一段纹理映射代码。

IDirect3DDevice9::SetTexture()方法将一张纹理指派给设备中的某个特定阶段。Direct3D 提供 8 个纹理阶段，这意味着可以为一次 DrawPrimitive()调用同时设置最多 8 张纹理。（然而，你也可以多次调用 DrawPrimitive()，这称为“多次渲染（multi-pass rendering）”。每一次就是一个 DrawPrimitive()调用。）在 SetTexture()的第 1 个参数中给出一个希望使用的纹理阶段编号。通常，纹理阶段从 0 开始编号，每个纹理阶段的编号依次递增，另外可为每个纹理阶段指派一张纹理。在这里将背景纹理指派给了 0 号纹理阶段。

当一个应用程序选择了一张纹理作为当前纹理时，它就指示 Direct3D 设备在下一次改变当前纹理之前，一直将该纹理应用到所有被渲染的图元上去。如果 3D 场景中的每个图元都有其自己的纹理，则必须在绘制每个图元之前设置相应的纹理，也就是说应在调用 DrawPrimitive()函数之前进行纹理设置。

在设置完纹理后，必须调用 SetTextureStageState()来设置适当的纹理阶段操作，本例中的设置代码如下：

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_SELECTARG1);
```

```
HRESULT SetTextureStageState(
    DWORD Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    DWORD Value
);
```

SetTextureStageState()可用于在不同的材质（material）之间对颜色和 alpha 作混合（blending）。可用它来指定 Direct3D 纹理引擎或像素引擎的操作和参数。首先应选择想要使用的纹理，在第 1 个参数中给出持有该纹理接口的纹理阶段号。第 2 个参数是希望设置的纹理状态类型。有两种主要的类型：D3DTSS_COLORx 系列状态和 D3DTSS_ALPHAx 系列状态，两者分别用于控制一个 RGB 向量流和一个 alpha 标量流在通过像素处理流水线中可并行处理部分（parallel segments of the pixel pipeline）的操作。在这个示例中，第 2 个 SetTextureStageState()通过 D3DTSS_COLOROP 来选择一个颜色操作。D3DTOP_SELECTARG1 参数告知纹理单元使用第 1 个参数（argument）作为输出。第 1 个 SetTextureStageState()调用指出了第 1 个参数是通过 SetTexture()设置的纹理颜色。在本书的第二部分，你将会深入了解到多重纹理（multi-texturing）和多次纹理（multipass texturing）。

在设置纹理后，将以与前例相同的方式来调用 SetFVF()、SetStreamSource()、DrawPrimitive()。

在调用完这些函数后，必须调用字体类的绘制函数 DrawText()。这是一个非常便于使用的函数，它能够完成显示一段文本所需的全部事情。

```
m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );
m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );
```

前两个参数决定显示文本的位置，其中坐标值以左上角为原点（就像为顶点提供的纹理映射值和屏幕空间值那样）。下一个参数是文本颜色。本例将 alpha 通道、红色通道和绿色通道设为 255、蓝色通道设为 0。若降低 alpha 值，字符会变成透明的。

在两次调用中最后一个参数分别为 m_strFrameStats 和 m_strDeviceStats，它们是定义在 d3dapp.cpp 中的两个数组。m_strFrameStats 表示每秒的帧数、窗口的宽度和高度、背景缓冲器的格式；m_strDeviceStats 表示设备的类型和描述（例如，hw vp 和设备的名称，前者表示硬件顶点处理）。你可以翻阅一下我在本节开始处对这些数组内容的解释。

5.4.4 InvalidateDeviceObjects()函数

InvalidateDeviceObjects()调用了字体类的同名函数。它还释放了 RestoreDeviceObjects()创建的顶点缓冲器。

```
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    m_pFont->InvalidateDeviceObjects();
    SAFE_RELEASE( m_pVB );
    return S_OK;
}
```

5.4.5 DeleteDeviceObjects()函数

DeleteDeviceObjects()是对应于 InitDeviceObjects()的函数，它用于删除先前在 InitDeviceObjects()中创建的对象。

```
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    m_pFont->DeleteDeviceObjects();
    SAFE_RELEASE( m_pBackgroundTexture );
    return S_OK;
}
```

5.4.6 FinalCleanup()函数

和往常一样，结束时会调用 FinalCleanup()。由于启动时在 OneTimeSceneInit()中没有分配任何资源，所以这里什么事情也不用做。

```
HRESULT CMyD3DApplication::FinalCleanup()
{
    return S_OK;
}
```

在 OneTimeSceneInit()中没有占用任何东西，所以在此不需要做任何事情。

现在你应该对如何在 Direct3D 中使用纹理映射、如何使用 d3dfont.cpp 提供的字体类有一个大致的印象了吧。Basic3 示例将介绍使用一种新的图元类型来显示几何体——三角条形（triangle strip）。

5.5　Basic3 示例

与前例相比，Basic3 示例只有一处小变化。它使用三角条形作为 DrawPrimitive()中的渲染图元，代替了原先的三角扇形。在一些新型的显卡上，这种图元更适合于 cache。因此，它可能会有更好的执行性能。不过对于本例这样只使用了 4 个顶点的程序而言，你不会感觉到有什么速度上的差异。较之前例，这里只有两处变化。一处是在 RestoreDeviceObjects()中，以另一种次序设置了顶点缓冲器中的顶点值。

```
{ 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, },
{ (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.0f,0.0f, },
{ 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0.0f, 1.0f, }, // x, y, z, rhw, tu, tv
{ (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 1.0f,1.0f,},
```

这个示例给出的顶点就像佐罗在他的敌人前额上留下的 Z 印记那样。首先给出左上角的顶点，然后是右上角的顶点，接着分别是左下角和右下角的顶点。图 5.11 展示了这个 Z 形状。

图 5.11　使用三角条形图元时给出的顶点次序

前一个示例为一个三角扇形使用了以下顶点次序：

```
{ 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv
{ (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f, },
{(float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 1.0f,1.0f, },
{ 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0.0f,1.0f, },
```

这里的顶点次序围绕着一个四边形，依次为：左上角、右上角、右下角、左下角。这种“围绕着一个角旋转”的给出次序是描述三角扇形图元的典型方法。例如，若对一个同样的四边形使用 6 个顶点的话，则顶点给出的次序应依次为：上端中间、右上角、右下角，下端中间、左下角、左上角；但是同样的情况对于三角条形而言，则次序应依次为：左下角、左上角、上端中间、下端中间、右上角、右下角。注意，在描述三角条形图元的顶点中，任意 3 个连续的顶点都构成一个三角形。

在当前的这些示例中都只使用了 4 个顶点，所以这还可以通过手工方式来设置。而在当今的实际应用中，一个完整的 3D 模型通常是由数千个顶点组成的。这样的模型都是需要借助于 3D 建模软件（如 discreet 公司的 3ds max、Caligari 公司的 trueSpace）来构造的。将这些建模软件生成的顶点从三角扇形重新排列为三角条形是一项很有价值的任务，反之亦然。而该项任务通常可以借助于图元程序库来完成。在 NVIDIA 的网站（http://developer.nvidia.com）上就有一个这样的库，它可以将一个模型的所有顶点重新排序为三角条形。

对前例源代码所做的第二处更改是 Render()函数中的 DrawPrimitive()，在调用 DrawPrimitive()时将图元类型指定为三角条形。

```
HRESULT CMyD3DApplication::Render()
{
    …
    m_pd3dDevice->DrawPrimitive (D3DPT_TRIANGLESTRIP, 0, 2);
    …
    return S_OK;
}
```

关于本例就是这些新内容。下一个示例将介绍索引缓冲器（index buffer）背后的思想，并会使用一种称为“索引化的三角形列表（indexed triangle list）”的新图元类型。

5.6 Basic4 示例

Basic4 示例展示了以下两项 Direct3D 特征：

- 索引缓冲器（index buffer）；
- 一种图元类型——索引化的三角形列表。

和前面一样，我将注意力集中在那些新加的特征上。对于当前大多数的显卡来说，使用索引

缓冲器和索引化的图元会比使用非索引化的图元带来更大的性能提升。

为了以其最简单的形式来使用索引缓冲器，你必须完成以下3个步骤：

（1）在创建顶点缓冲器的同一位置，创建并填充索引缓冲器（在本例中就是RestoreDeviceObjects()）；

（2）在渲染循环中，使用SetIndices()设置索引，并采用索引化的图元（如索引化的三角形列表）来显示几何体；

（3）在释放顶点缓冲器的同一位置，释放索引缓冲器（在此本例中就是InvalidateDeviceObjects()）。

为了使用一个索引缓冲器，必须先声明一个索引缓冲器对象。可将其作为应用程序类的一个成员，并在应用程序类的构造函数中把它初始化为NULL。

```
LPDIRECT3DINDEXBUFFER9 m_pIB;
DWORD m_dwSizeofIndices;
...
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle = _T("Basic");
    m_d3dEnumeration.AppUsesDepthBuffer = FALSE;
    m_dwCreationWidth  = 800; // Width used to create window
    m_dwCreationHeight = 600;
    m_pVB = NULL;
    m_pIB = NULL;
    m_pFont = new CD3DFont( _T("Arial"), 12, D3DFONT_BOLD );
}
```

在RestoreDeviceObjects()函数中，调用CreateIndexBuffer()创建索引缓冲器。

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    // we don't need any transformation -> position will be delivered in screen
    coordinates

    m_pFont->RestoreDeviceObjects();

    // fill the vertex buffer with the new data
    // Initialize to render a quad
    VERTEX Vertices[] =
    {
         { 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, }, // x, y, z, rhw, tu, tv
         { (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.0f, 0.0f, },
         { (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f,
           1.0f, 1.0f,1.0f, },
         { 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0.0f,1.0f, },
    };

    m_dwSizeofVertices = sizeof(Vertices);

    // sixth parameter is new in DX9
```

```
    // Create a square for rendering the background
    if( FAILED( m_pd3dDevice->CreateVertexBuffer( m_dwSizeofVertices,
                                                  D3DUSAGE_WRITEONLY, FVF,
                                                  D3DPOOL_MANAGED, &m_pVB, NULL ) ) )
        return E_FAIL;

    VERTEX* pVertices;
    // the third parameter changed from BYTE** to VOID** in DX9
    if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (VOID**)&pVertices, 0 ) ) )
        return E_FAIL;
    memcpy( pVertices, Vertices, m_dwSizeofVertices);
    m_pVB->Unlock();

    // Initialize the Index buffer
    WORD wIndices[] = {0, 1, 2, 0, 2, 3};
    m_wSizeofIndices = sizeof(wIndices);

    // Create the index buffer
    // six parameter new in DX9
    if( FAILED( m_pd3dDevice->CreateIndexBuffer( m_wSizeofIndices,
                                                 0, D3DFMT_INDEX16,
                                                 D3DPOOL_MANAGED, &m_pIB, NULL ) ) )
      return E_FAIL;

    VOID* pIndices;
    // DX9 third parameter changed from BYTE** to VOID **
    if( FAILED( m_pIB->Lock( 0, m_wSizeofIndices, (VOID**)&pIndices, 0 ) ) )
        return E_FAIL;
    memcpy( pIndices, wIndices, sizeof(wIndices) );
    m_pIB->Unlock();

    return S_OK;
}
```

本例使用的顶点次序与第一个示例相同，但却有一个本质上的区别。一个非索引化的三角形列表由 2 个孤立的三角形组成，这 2 个三角形不需要以任何形式来互相连接（见图 5.12）。

图 5.12 显示出 2 个三角形之间没有任何形式的连接。在一个三角形列表中，每个三角形都是网格中一个独立的多边形。再考虑一下这样 2 个三角形，它们的 2 个顶点互相重叠（见图 5.13）。

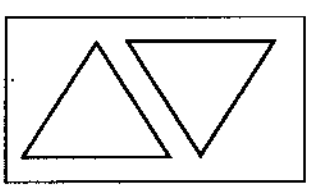


图 5.12　三角形列表中的 2 个三角形

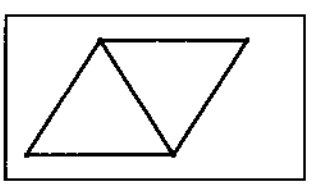


图 5.13　2 个邻接的三角形

如果使用非索引化的三角形列表来组成一个完整模型，那么其中会存在许多重叠的顶点。这是因为它的各个三角形之间没有缝隙，彼此都是紧密相连的。这就需要做许多重复的

工作，从而导致性能效率降低。另外，由于计算机算术精度的问题，可能会无法精确地计算出顶点位置，所以在经过顶点处理后，三角形之间可能会出现一些缝隙。

以上问题的解决方法就是使用一个索引缓冲器。在顶点缓冲器中只需提供 4 个顶点，每 2 个三角形共用其中的 2 个顶点。可见索引缓冲器可以帮助你重复使用顶点。

在使用过程中，每个顶点具有一个从 0 到 3 的数字。该数字对应于顶点缓冲器中顶点的次序（依次为左上角、右上角、右下角、左下角）。因此，左上角的顶点为 0，右上角的顶点为 1，右下角的顶点为 2，左下角的顶点为 3（见图 5.14）。

图 5.14　被编号的顶点

对于两个邻接三角形的重叠边来说，一般只需向 Direct3D 渲染引擎提供 2 个顶点就可以了。通过给每个顶点编号，就可以在索引缓冲器中给出相同的顶点号来重复使用顶点了。索引缓冲器的内容为：

```
WORD wIndices[] = {0, 1, 2, 0, 2, 3};
```

如你所见，每个顶点都有一个它自己编号，在本例中编号范围为 0～3。第 1 个三角形用到了顶点 0、1、2，第 2 个三角形用到了顶点 0、2、3。这 2 个三角形都用到了顶点 0 和顶点 2。因此，所谓的编号只是顶点缓冲器中的一个整数偏移量。这就告知了 DrawIndexed Primitive()，使用顶点 0、1、2 来绘制第 1 个三角形，使用顶点 0、2、3 来绘制第 2 个三角形。

那么现在一个基本的问题就是：你应该使用哪种图元类型？在许多应用程序中，很多多边形都共用了某些顶点。所以，只要减少顶点变换时重复的顶点，以及减少经由总线传到渲染设备时重复的顶点，那么应用程序的性能就能获得极大提升。非索引化的三角形列表不共

享顶点，因此它不是最佳方法。而三角条形或三角扇形则要求多边形之间必须存在着特定的连接关系。如果几何体模型数据结构可以自然地通过三角条形或三角扇形来构建的话，那么这两种图元便是最佳的选择，因为它们可以最大化地减少传输到驱动程序的数据量。

另外，如果几何体数据可以通过索引被重用的话，那么就可以使用索引缓冲器来改善现在程序的性能。如果每个顶点被索引多于 2 次的话，则会有更可观的性能提升。在某些显卡上，索引化的三角条形的处理速度会比索引化的三角形列表稍微快一点。但是在另一些显卡上，两者的处理速度却没有什么差别。无论如何，你都应该在不同的情况中对各种图元类型及其处理速度进行充分的实测，由此才能得到一个最佳方案。

有几种限制因素会导致性能下降。归纳一下，主要有三类可能的限制：CPU 限制、顶点吞吐量限制和填充率限制。CPU 限制和顶点吞吐量限制从字面上就已经说明了其含义。而填充率限制指的是由应用程序造成的一个性能瓶颈，这个瓶颈位于 Direct3D 流水线的尾部与光栅化阶段之间（我将在第 9 章“使用高级 Shader 语言进行 Shader 编程”中介绍 Direct3D 流水线）。应该通过对各方面进行正确的实测来找出限制所在，然而这是一项艰巨的工作，并且需要一些经验。不过无论如何，只要使用了正确的图元类型就会有利于减轻顶点吞吐量限制方面的负担。但这却不会对另外两类限制因素起到什么帮助作用。

可调用 CreateIndexBuffer()来创建索引缓冲器。

```
HRESULT CreateIndexBuffer( // size of index buffer in bytes
                           UINT Length,
                           // usage flags
                           DWORD Usage,
                           // index buffer format
                           D3DFORMAT Format,
                           // memory pool
                           D3DPOOL Pool,
                           // pointer to index buffer
                           IDirect3DIndexBuffer9** ppIndexBuffer,
                           // reserved for future DX versions
                           HANDLE* pHandle)
```

这个函数的第 1 个参数为索引缓冲器的长度，第 2 个参数为各种使用提示的组合，第 3 个参数为所需的格式。第 2 个参数用于告知驱动程序索引缓冲器的用法，这里的标志与 CreateVertexBuffer()函数中可选择的标志差不多。由于本例没有什么特殊的使用要求，所以将该参数指定为 0。格式参数可让用户在两种不同的索引缓冲器格式之间进行选择：16 位（D3DFMT_INDEX16）和 32 位（D3DFMT_INDEX32）。32 位的索引缓冲器格式可以储存更多的索引，因此对于一个具有极多顶点的模型而言，这种格式便有能力储存下该模型的全部索引。但是，这种格式在某些显卡上运行速度要比 16 位格式慢。

第 4 个参数要求指定希望选用的池类型。记住，DirectX 9.0 中存在 4 种不同类型的池，并且有一条经验法则是——在大多数情况下，选择托管池总是正确的。因此在这里就指定了一个托管池。

第 5 个参数返回一个指向索引缓冲器的指针，可以通过该指针来访问索引。做法与顶点

缓冲器类似，通过 Lock()函数锁定索引缓冲器后，使用 memcpy()填充 Lock()返回的缓冲区。

```
VOID* pIndices;
// DX9 third parameter changed from BYTE** to VOID **
if( FAILED( m_pIB->Lock( 0, m_wSizeofIndices, (VOID**)&pIndices, 0 ) ) )
    return E_FAIL;
memcpy( pIndices, wIndices, sizeof(wIndices) );
m_pIB->Unlock();
```

第 6 个参数是 DirectX 9.0 中的一个新参数，保留用于将来的功能。现在应将其设为 NULL。

Lock()和 Unlock()函数的参数与顶点缓冲器中的 Lock()和 Unlock()类似。惟一的区别就是第 2 个参数应指定为欲锁定的索引大小，而不是顶点大小。

为了使用索引缓冲器，必须在 Render()函数中设置它。

```
HRESULT CMyD3DApplication::Render()
{
    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        // Set up texture stage states for the background
        m_pd3dDevice->SetTexture( 0, m_pBackgroundTexture );
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );

        // Passing an FVF to IDirect3DDevice9::SetFVF specifies a legacy FVF with stream 0.
        m_pd3dDevice->SetFVF(FVF );
        m_pd3dDevice->SetStreamSource( 0, m_pVB, 0, sizeof(VERTEX) );
        m_pd3dDevice->SetIndices( m_pIB ); // new in DX9 only takes one parameter
        // second parameter new in DX9
        m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                    0,
                                    0,
                                    4, // number of vertices
                                    0,
                                    2); // number of primitives

        // Output statistics
        m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );
        m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );

        // End the scene.
        m_pd3dDevice->EndScene();
    }

    return S_OK;
}
```

以上代码通过 SetIndices()来设置索引缓冲器，该函数只接受一个参数，即一个索引缓冲器指针。由于在 RestoreDeviceObjects()函数中创建了索引缓冲器，所以必须在 InvalidateDeviceObjects()函数中释放它。

```
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    m_pFont->InvalidateDeviceObjects();
    SAFE_RELEASE( m_pVB );
    SAFE_RELEASE( m_pIB );
    return S_OK;
}
```

第 4 个示例的其余部分与第 4 个示例中的都一样。因此，你可以马上接下去看第 5 个示例。

5.7 Basic5 示例

Basic5 与前例仅有一点不同，Basic5 使用了索引化的三角条形。因此只需改动 3 处：

- 顶点的次序（三角条形）；
- 索引的次序（佐罗的 Z）；
- DrawIndexedPrimitive()中的第 1 个参数（图元类型）。

下面的代码以一个三角条形的次序来设置顶点。这段代码位于示例的 RestoreDeviceObjects()函数中。

```
...
{ 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, },
{ (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.0f,0.0f, },
{ 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0.0f, 1.0f, }, // x, y, z, rhw, tu, tv
{(float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 1.0f,1.0f, },
...
```

这个次序与 Basic3 示例中所用的次序相同，回忆一下，Basic3 使用的是非索引化的三角条形。另外，还需要改变索引的次序以构成 Z 的形状，就像这样：

```
WORD wIndices[] = {0, 1, 2, 3, 2, 0};
```

索引编号 0、1、2、3 分别对应于顶点缓冲器中的第 1、第 2、第 3、第 4 个顶点。最后两个索引值(2 和 0)将渲染器带回到起点[1]。准备好正确的顶点和索引后，DrawIndexPrimitive()函数必须要以 D3DPT_TRIANGLESTRIP 标志作为其绘制的图元类型。

5.8 小结

完成了对一系列 basic 示例的介绍。准备好继续前进了吗？在下一章中，你将学习到动画方面的基本知识。听起来很有趣，不是吗？！

译注[1]：这不是必须的，也就是说，这个索引数组 wIndices 其实只要有前 4 个元素就可以了。

第 6 章

动 画 初 步

回忆一下，在上一章中所有示例都是借助于 DirectX SDK 的通用文件构建的，其中重载了 CD3DApplication 的方法。

```
HRESULT OneTimeSceneInit()      { return S_OK; }
HRESULT InitDeviceObjects()     { return S_OK; }
HRESULT RestoreDeviceObjects()  { return S_OK; }
HRESULT DeleteDeviceObjects()   { return S_OK; }
HRESULT Render()                { return S_OK; }
HRESULT FrameMove( FLOAT )      { return S_OK; }
HRESULT FinalCleanup()          { return S_OK; }
```

你还学习了上述每个方法各自的任务。本章将引领你进入动画的世界，并在其中创建一个虚拟照相机系统。你将学习到如何：

- 使用变换（transformation）；
- 在 3D 世界里移动物体；
- 移动照相机（camera）；
- 组合多个变换；
- 使用投影（projection）变换；
- 使用视口；
- 使用深度缓冲器（depth buffer）。

6.1 理解变换与视口

如何通过变换来实现真实世界中物体运动的效果呢？为了更好地理解这个问题，可以设想一下这样的情景。你拿着摄像机站在卧室里给你的女朋友拍摄录像，起初她在卧室里所站

立的位置及朝向与你手持的摄相机一致。当她开始走动时，你便随着她的身影转动手中的摄像机。为了获得更鲜明的图像，你可能会试着改变镜头的属性，或者为了更好地记录下她的灿烂笑容，你或许还会放大或缩小场景。要实时地处理这些情况是相当困难的事情，但这也正是你要在实时图形编程中所要试图去处理的事情。

在计算机游戏中你需要通过变换（transformation）来模拟所有这些运动。通过世界变换（world transformation）和观察变换（view transformation）来分别改变女朋友和照相机的位置与朝向。通过投影变换（projection transformation）来放大与缩小场景，或改变相机镜头的属性，从而获得更鲜明的图像。

附注

另外还有一种变换称为“模型变换（model transformation）”，在本章中我没有涉及到这种变换。在现实生活中，当你的女朋友开始跳舞时，她的位置和朝向并没有发生变化，但四肢和身体却发生了相对的移动，此时就需要用到这种变换。

整个场景发生在三维空间中，所以最后需要将三维场景映射到二维的胶片上，视口（viewport）就是这种用于接受映射的二维“胶片”。

6.1.1 世界变换

在 3D 世界中，我们使用世界变换来移动一个物体。图 6.1 显示了两个大小相同但位置和朝向不同的立方体。

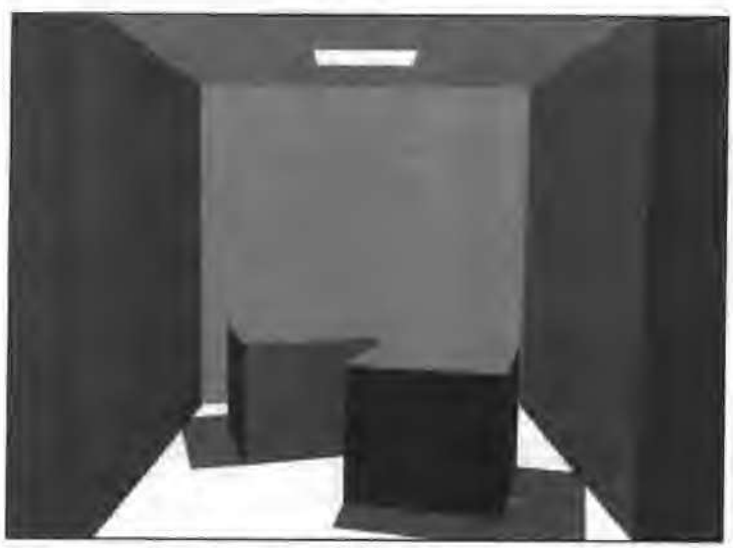


图 6.1 两个位置和朝向不同的立方体

图 6.2 显示了将其中一个立方体移动到正确位置上所需的变换。

以下步骤详述了将这个立方体移动到正确位置上需要经历的所有变换。

（1）立方体的中心原点位于世界坐标系的原点；

（2）旋转立方体；

（3）将立方体移动到新位置。

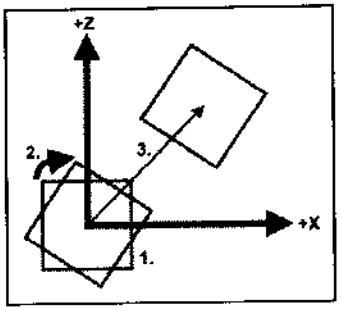


图 6.2　一个立方体的世界变换

第一步中的初始位置来自于顶点缓冲器中的数据，顶点缓冲器及其数据在前一章中已介绍过了。对于当前这种静态物体而言，它的数据是永远不会改变的。因此，立方体总是以一种默认的位置和朝向作为初始状态。在第二步中旋转了立方体，在第三步中将立方体移动到新的位置。这一过程在每帧中都重复进行，因此在每次显示一个场景时，所有物体都会被正确地摆放到它们各自的位置和朝向上。

这些步骤的次序非常重要。如果第二步与第三步次序对调的话，物体则会以一种意想不到的方式来移动，如果先移动立方体，再旋转，那它将会围绕世界坐标系的中心旋转，而不是围绕其自身几何体时中心或重心旋转。如图 6.3 所示。

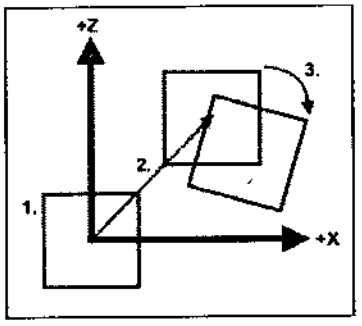


图 6.3　第二、三步交换后，立方体的世界变换

使用一个世界变换矩阵或短世界矩阵（short world matrix）来变换立方体的各个顶点是一种非常通用的做法（请阅读附录 C“数学入门”，在其中还可以学习更多关于矩阵的知识）。使用矩阵的一个好处便是你可以通过一个顶点缓冲器和几个不同的变换矩阵方便地绘制出同一物体的几个实例。那个显示图 6.2 和图 6.3 中立方体的应用程序只使用了一套几何体数据，也就是说只使用了一个顶点缓冲器。这样还节省了内存和带宽，因为发送到显卡上的几何数据就只有一个模型。

正如我先前所说的，变换出现的次序是很重要的。也就是说矩阵乘法不满足交换率，即表示为[a] * [b] <> [b] * [a]。变换的次序取决于想要达到的效果，在本章的第二个示例程序中将讨论这个问题。一个 4×4 的世界矩阵包含了 4 个向量，分别用于存储一个物体的朝向和位置。

```
ux uy uz 0
vx vy vz 0
wx wy wz 0
tx ty tz 1
```

向量 u、v、w 描述了相对于世界坐标系而言物体的朝向向量，向量 t 描述了相对于世界

坐标系而言物体的位置向量。图 6.4 展示了这 4 个向量所描述的一个立方体的朝向和位置。

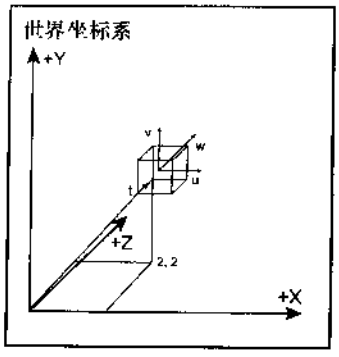


图 6.4　世界坐标系中的一个正方体

将立方体的每个顶点都乘以这个矩阵便可以获得正确的朝向和位置。为了描述图 6.4 中的立方体位置，必须将立方体的每个顶点都乘以以下这个矩阵。

```
1, 0, 0, 0
0, 1, 0, 0
0, 0, 1, 0
2, 2, 2, 1
```

这个矩阵不会改变立方体的朝向。它只是将立方体的原点平移到了(2, 2, 2)。可以这样来访问一个 4×4 的矩阵：

```
D3DMATRIX mat;
mat._11 = 1.0f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 = 1.0f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 2.0f; mat._42 = 2.0f; mat._43 = 2.0f; mat._44 = 1.0f;
```

为了访问第 3 行第 1 列的元素，可使用 mat._31；为了修改立方体在 x 轴上的位置，可修改矩阵元素 mat._41。

D3DXMatrixTranslation()函数可用于构建一个平移矩阵。平移最适合于描述位置的线性变化。该变换可以用一个 delta 向量(tx, ty, tz)来表示。其中，tx（或通常为 dx）表示物体在其 x 轴上的位置变化；ty 或 dy 表示物体在其 y 轴上的位置变化；tz 或 dz 表示物体在其 z 轴上的位置变化。虽然我们没有 D3DXMatrixTranslation()的源代码，但它应该是这样的：

```
inline VOID D3DXMatrixTranslation(D3DXMATRIX* m, FLOAT tx, FLOAT ty, FLOAT tz)
{
    D3DXMatrixIdentity(&m);
    m._41 = tx; m._42 = ty; m._43 = tz;
}
=
1  0  0  0
0  1  0  0
0  0  1  0
tx ty tz 1
```

Direct3D 还提供了用于构建旋转矩阵的函数。旋转是指围绕一根轴作环行运动。增量角用于描述物体从当前朝向开始所要旋转的度数。为了将物体绕 z 轴旋转 1 度，只需指示物体绕着它的 z 轴旋 1 度即可，而不用管物体的当前朝向以及它是如何达到这一朝向的。这就是实际应用中的操作方法。

D3DXMatrixRotationY()用于将物体绕着 y 轴旋转，其中 fRads 表示所要旋转的度数。这个方法的源代码应该是这样的：

```
VOID D3DXMatrixRotationY(D3DXMATRIX* mat, FLOAT fRads)
{
    D3DXMatrixIdentity(mat);
    mat._11 =  cosf( fRads );
    mat._13 = -sinf( fRads );
    mat._31 =  sinf( fRads );
    mat._33 =  cosf( fRads );
}
=
cos fRads     0    -sin fRads    0
0             1     0            0
sin fRads     0    cos fRads     0
0             0     0            1
```

如果想要将物体绕着 x 轴旋转，则需要使用 D3DXMatrixRotationX()，其中 fRads 表示所要旋转的度数

```
VOID D3DXMatrixRotationX(D3DXMATRIX* mat, FLOAT fRads)
{
    D3DXMatrixIdentity(mat);
    mat._22 =  cosf( fRads );
    mat._23 =  sinf( fRads );
    mat._32 = -sinf( fRads );
    mat._33 =  cosf( fRads );
}
=
1  0             0            0
0  cos fRads sin fRads        0
0  -sin fRads cos fRads       0
0  0             0            1
```

D3DXMatrixRotationZ()将物体绕着 z 轴旋转，其中 fRads 表示所要旋转的度数。

```
VOID D3DXMatrixRotationZ(D3DXMATRIX* mat, FLOAT fRads)
{
    D3DXMatrixIdentity(mat);
    mat._11 =  cosf( fRads );
    mat._12 =  sinf( fRads );
    mat._21 = -sinf( fRads );
    mat._22 =  cosf( fRads );
}
=
cos fRads     sin fRads     0 0
-sin fRads    cos fRads     0 0
0             0             1 0
0             0             0 1
```

另一个函数 D3DXMatrixMultiply()对变换来说也十分有用。D3DXMatrixMultiply()的原型为 D3DXMATRIX* D3DXMatrixMultiply (D3DXMATRIX* pOut, CONST D3DXMATRIX* pM1, CONST D3DMATRIX* pM2)。也就是做了(*pOut)=(*pM1)*(*pM2)。该函数的返回值与 pOut 参数中所返回的值相同。这样便可以将 D3DXMatrixMultiply()函数直接作为另一个函数的参数来使用。矩阵乘法是一种一个矩阵被另一个矩阵变换的运算。矩阵乘法即为矩阵中各

行与各列乘积的和。

```
a b c d      A B C D
e f g h   *  E F G H =
i j k l      I J K L
m n o p      M N O P
mat._11 = a*A+b*E+c*I+d*M
mat._12 = a*B+b*F+c*J+d*N
mat._13 = a*C+b*G+c*K+d*O
mat._14 = a*D+b*H+c*L+d*P
mat._21 = e*A+f*E+g*I+h*M
mat._22 = e*B+f*F+g*J+h*N etc.
```

一个计算矩阵乘法的函数可以是这样的：

```
D3DXMATRIX* D3DXMatrixMultiply (D3DXMATRIX* pOut,
                                CONST D3DXMATRIX* pM1,
                                CONST D3DMATRIX* pM2)
{
    FLOAT pM[16];
    ZeroMemory( pM, sizeof(D3DXMATRIX) );
    for( WORD i=0; i<4; i++ )
       for( WORD j=0; j<4; j++ )
          for( WORD k=0; k<4; k++ )
             pM[4*i+j] += pM1[4*i+k] * pM2[4*k+j];
    memcpy( pOut, pM, sizeof(D3DXMATRIX) );
    return (pOut);
}
```

根据不同的处理器该函数还可以作出许多高度的优化。以下这种相乘矩阵的做法更棒：

```
D3DXMATRIX matRotX, matRotY;
matWorld = matTrans * matRotX * matRotY;
```

这种做法用到了 C++特征中的操作符重载，你可在附录 B“C++入门”中看到更多关于该特征的介绍。

另外一个很有用的矩阵函数是 D3DXMatrixScaling()。该函数接受 x、y、z 3 个比例因数，以及一个用于表示缩放的矩阵。这个函数的 C 语言源代码应该是这样的：

```
D3DXMATRIX* D3DXMatrixScaling(D3DXMATRIX *pOut, FLOAT sx, FLOAT sy,FLOAT sz)
{
    pOut->_11 = sx; pOut->_22 = sy; pOut->_33 = sz; pOut -> 44=1;
    return pOut;
}
=
sx      0       0       0
0       sy      0       0
0       0       sz      0
0       0       0       1
```

可通过这个函数来改变几何体的外形。例如，要在一个场景中放置几个不同大小的立方体，只需按不同的比例系数对同一个立方体的高度进行缩放，然后将它们置于场景中即可。

通过这种方法，你只需使用一个顶点缓冲器其中也只要存放一个立方体的几何数据，然后重复地使用它变换出不同的立方体。

使用矩阵乘法的一个优点就是执行缩放、旋转、平移所用的时间都是相同的，因此一个专用的变换引擎的执行性能是可预期且稳定的。这种特性使得游戏开发者可在执行性能与质量的取舍考量中作出明智的选择。

到目前为止，你已经了解了如何通过矩阵来做顶点变换，但你或许还不知道如何在 Direct3D 中设置变换矩阵。以下这行代码为 Direct3D 执行顶点变换提供了世界矩阵：

```
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matWorld );
```

D3DTS_WORLD 标志用于设置设备中与世界变换相关的矩阵。D3DTS_VIEW 和 D3DTS_PROJECTION 标志分别用于设置观察矩阵和投影矩阵，稍后我将介绍这些概念。

1. 1steps 示例

1steps.exe 示例程序位于 CD-ROM 的\1steps 目录下，它说明了 Direct3D 是如何执行世界变换的。该示例将一个四边形绕着它的 y 轴旋转（见图 6.5）。

图 6.5　一个绕着自身 y 轴旋转的四边形

前一章中曾演示过一个类似的贴有纹理的矩形。而现在我们借助于 Direct3D 使这个矩形动了起来。为了通过 Direct3D 引擎实现这种动画效果，必须修改一些代码。第一步涉及到 Direct3D 变换引擎，首先移除顶点标志中的 RHW 标志。

```
// A structure for our custom vertex type
struct VERTEX
{
    FLOAT x, y, z; // The untransformed position for the vertex
    FLOAT tu, tv;
};

// Fixed-Function Vertex structure
const DWORD FVF = (D3DFVF_XYZ | D3DFVF_TEX1);
```

去除了 RHW 值后，顶点缓冲器中的内容如下：

```
VERTEX Vertices[] =
{
    { -1.0f,-1.0f, 0.0f, 1.0f, 1.0f, }, // x, y, z, tu, tv
    {  1.0f,-1.0f, 0.0f, 0.0f, 1.0f, },
    {  1.0f, 1.0f, 0.0f, 0.0f, 0.0f, },
    { -1.0f, 1.0f, 0.0f, 1.0f, 0.0f, },
};
```

注意，x 和 y 位置值不是在屏幕空间中指定的。这里给出的位置与前章示例一样，表示整个后备缓冲器的尺寸。在默认情况下，Direct3D 变换单元将窗口客户区的中心作为世界原点。因此，四边形左下角顶点为(−1.0f, −1.0f)、右下角顶点为(1.0f, −1.0f)、右上角顶点为(1.0f, 1.0f)、左上角顶点为(−1.0f, 1.0f)。索引缓冲器以如下次序设置这些顶点：

```
WORD wIndices[]={3, 2, 0, 2, 1, 0};
```

这些索引值表示：左上角、右上角、左下角、右上角、右下角、左下角。第一个三角形位于左上角，第二个三角形位于右下角。

主要的变换操作都是在框架函数 FrameMove()中完成的。

```
HRESULT CMyD3DApplication::FrameMove()
{
    // rotates the object about the y-axis
    D3DXMATRIX matRotY;
    D3DXMatrixRotationY( &matRotY, m_fTime * 1.5f);

    D3DXMATRIX matTrans;
    D3DXMatrixTranslation(&matTrans, 0.0f, 0.0f, 0.0f);

    m_pd3dDevice->SetTransform( D3DTS_WORLD, &(matRotY * matTrans));
    return S_OK;
}
```

实用函数 D3DXMatrixRotationY()构建出一个绕 y 轴旋转的矩阵。这里使用了 Direct3D 实用程序库提供的矩阵类型。该库提供的所有类型名称都是以 D3DX*开头的。另外，框架提供了 m_fTime 变量，在应用程序的任何地方都可以使用它。它表示应用程序从启动到现在经历的时间，起始值为 0.0f。将该时间变量乘以 1.5f，意在稍微加快点旋转的速度。接下去，D3DXMatrixTranslation()设置 matTrans 矩阵的成员._41、._42、._43。为了组合旋转矩阵和平

移矩阵，可以将它们相乘，在这里是将旋转矩阵乘以平移矩阵。最后，将矩阵的乘积提供给 SetTransform()函数。如果硬件支持顶点变换，则该函数将世界矩阵传递给硬件的变换单元；否则，Direct3D 必须执行软件顶点处理。

Direct3D 引擎会完成余下的工作。在这里我没有提及观察矩阵、投影矩阵和深度缓冲器的使用。因为稍后我将在“观察变换”和“投影变换”两节中专门介绍它们，并且在本章最后介绍深度缓冲器背后的概念以及如何使用深度缓冲器。

如果你想扩展一下 1steps 示例，你可以尝试将以下源代码复制到 FrameMove()中。那样的话，四边形便会同时绕着 x 轴和 y 轴旋转了。

```
HRESULT CMyD3DApplication::FrameMove()
{
    D3DXMATRIX matRotY, matRotX;
    D3DXMatrixRotationY( &matRotY, m_fTime * 1.5f);
    D3DXMatrixRotationX( &matRotX, m_fTime * 1.5f);

    D3DXMATRIX matTrans;
    D3DXMatrixTranslation(&matTrans, 0.0f, 0.0f, 0.0f);
    m_pd3dDevice->SetTransform( D3DTS_WORLD, &(m_RotY * m_RotX * matTrans));

    return S_OK;
}
```

正如你所见，通过将两个旋转矩阵相乘，便可以把这种旋转合二为一。

附注

这个示例还显示出背面剔除（back-face culling）的效果。默认情况下，Direct3D 引擎不会处理四边形的背面，因为这样能使它更有效率地工作。例如，对于一个立方体而言，你是看不到其背面的，这便是进行背面剔除的意义。有一些 SetRenderState()函数的标志可改变背面剔除的行为。例如，以下这行代码可关闭背面剔除：

```
d3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
```

2. 1steps2 示例

1steps2.exe 示例程序位于 CD-ROM 的\1steps2 目录下。它将告诉你变换次序的重要性（见图 6.6）。

该示例使用 4 个视口来显示 4 个不同的场景，在这些场景中对四边形分别应用了不同的变换矩阵。这个程序的灵感来自于 Kelly Dempski 的 *Real-Time Rendering Tricks and Techniques in DirectX*（Premier Press, 2002）中的一个示例程序。

程序用到的所有矩阵都是在 Render()函数中创建的。通常，我会在 FrameMove()函数中创建它们。但是，现在为了能够更清楚地整个过程，我将它们放到了 Render()函数中。这个程序使用了 4 个不同的矩阵。

图 6.6　变换的次序

```
D3DXMATRIX RotationMatrix1;
D3DXMATRIX RotationMatrix2;
D3DXMATRIX TranslationMatrix;
D3DXMATRIX ScalingMatrix;
```

第一个和第二个旋转矩阵（RotationMatrix1 和 RotationMatrix2）围绕 z 轴旋转。

```
D3DXMatrixRotationZ(&RotationMatrix1, m_fTime * 1.5f);
D3DXMatrixRotationZ(&RotationMatrix2, -m_fTime * 1.5f);
```

第一个旋转矩阵是逆时针旋转的，第二个是顺时针旋转的。TranslationMatrix 是通过 D3DXMatrixTranslation()函数创建的，用于将旋转的四边形向右移动 3 个单位。

```
D3DXMatrixTranslation(&TranslationMatrix, 3.0f, 0.0f, 0.0f);
```

另外，还用到了一个名缩放矩阵 Scaling Matrix。

```
D3DXMatrixScaling(&ScalingMatrix, 1.0f, 0.5f, 1.0f);
```

这个矩阵将高度缩小一半。

至此我已介绍了 1steps2 示例程序中会用到的每种变换。接下来的部分将告诉你，每个视口如何使用这些变换矩阵来产生不同的动画。

左上视口

在左上视口中只用到了第一个旋转矩阵 RotationMatrix1，其中四边形在视口中央绕着 z 轴旋转。由于只使用了一个矩阵，因此不用考虑矩阵乘法次序的问题。

左下视口

左下视口中展示了一个绕着其自身 z 轴旋转的四边形。在这里除了作旋转之外，还通过 TranslationMatrix 矩阵将四边形向右平移了 3 个单元。平移矩阵在乘法次序中位于旋转矩阵之后。

右上视口

右上视口展示了一个绕着视口中心 z 轴旋转的四边形。并且，还在 y 轴方向上对它的高

度进行了缩放。

与左下视口相比，由于这里对调了旋转矩阵和平移矩阵在乘法中的次序，所以四边形在距离原点 3 个单位之外绕着 z 轴旋转，并它还绕着其自身的 z 轴在旋转。在平移矩阵和旋转矩阵之后是缩放矩阵，缩放操作是在世界坐标系的 y 轴方向上进行的，因此看上去这个四边形好像是被压扁了一样。

右下视口

右下视口展示了一种稍微复杂点的情况，其中四边形在距离视口中心 3 个单位之外绕着视口中心旋转，但它不绕着自身的 z 轴旋转。另外，还将它的高度缩小了一半。这里的缩放和右上视口不一样，原因在于缩放矩阵是乘法次序中的第一个矩阵。另外还通过一个反向旋转矩阵（RotationMatrix2）阻止了四边形本身被旋转。由于 RotationMatrix2 旋转的方向正好与 RotationMatrix1 相反，所以这两次旋转相互抵消了。这个过程就好比是一个行星围绕着另一个行星的轨道进行旋转一样。

请注意，在使用先前提到的三个旋转函数来旋转三维空间中的物体时，可能会导致丢失一维，这便是著名的“万向节锁（gimbal lock）”现象。不过，这些函数对于现今大多数游戏来说还是非常适用，因为在大多数游戏中只需要用到“2.5 维”就足够了。对于“万向节锁”的一种解决方法是通过向量来实现三维空间中物体的旋转，另一种方法是使用四元数。

3. 1steps3 示例

1steps3.exe 示例示范了一种借助于向量来实现旋转的技术，这种技术同样可以做到将物体绕着所有三根轴作旋转。图 6.7 显示了两个简单的模型，你可以任意地旋转和移动它们。这是一个很基础的、不具有任何物理特性或光照的太空飞行模拟器。

图 6.7　两艘太空船

按下 1 键可以调出一个小巧的帮助系统。旋转第一艘飞船的按键分别为：Q/E 键用于偏航（yaw）、W/S 键用于俯仰（pitch）、A/D 键用于横滚（roll）、F/V 键用于移动。要旋转第二个太空船，对应旋转的按键为 U/O、I/K、J/L 键，移动按键为 H/N 键。

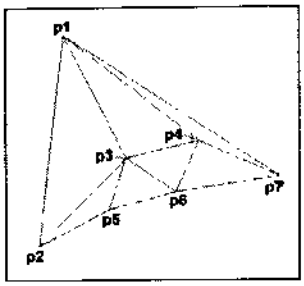


图 6.8　太空船的线框视图

后续所有的示例程序都将使用这个简单的几向模型。由于接下来的示例都不使用光照，因此需要通过这种具有方向性物体来识别它的朝向。图 6.8 显示了一幅太空船的线框视图。

整个模型是在 OneTimeSceneInit()函数中创建。

```
// Points which make up an object geometry
D3DXVECTOR3 p1 = D3DXVECTOR3( 0.00f, 0.00f, 0.50f );
D3DXVECTOR3 p2 = D3DXVECTOR3( 0.50f, 0.00f,-0.50f );
D3DXVECTOR3 p3 = D3DXVECTOR3( 0.15f, 0.15f,-0.35f );
D3DXVECTOR3 p4 = D3DXVECTOR3(-0.15f, 0.15f,-0.35f );
D3DXVECTOR3 p5 = D3DXVECTOR3( 0.15f,-0.15f,-0.35f );
D3DXVECTOR3 p6 = D3DXVECTOR3(-0.15f,-0.15f,-0.35f );
D3DXVECTOR3 p7 = D3DXVECTOR3(-0.50f, 0.00f,-0.50f );

// vertices for the top
m_pvObjectVertices[0].p = p1;
m_pvObjectVertices[0].tu = 0.0f;
m_pvObjectVertices[0].tv = 0.5f;
m_pvObjectVertices[1].p = p2;
m_pvObjectVertices[1].tu = 0.5f;
m_pvObjectVertices[1].tv = 1.0f;
m_pvObjectVertices[2].p = p3;
m_pvObjectVertices[2].tu = 0.425f;
m_pvObjectVertices[2].tv = 0.575f;
m_pvObjectVertices[3].p = p4;
m_pvObjectVertices[3].tu = 0.425f;
m_pvObjectVertices[3].tv = 0.425f;
m_pvObjectVertices[4].p = p7;
m_pvObjectVertices[4].tu = 0.5f;
m_pvObjectVertices[4].tv = 0.0f;

// vertices for the bottom
...
// vertices for the rear
...

// Vertex indices for the object
m_pwObjectIndices[ 0] = 0; m_pwObjectIndices[ 1] = 1; m_pwObjectIndices[ 2] = 2;
m_pwObjectIndices[ 3] = 0; m_pwObjectIndices[ 4] = 2; m_pwObjectIndices[ 5] = 3;
m_pwObjectIndices[ 6] = 0; m_pwObjectIndices[ 7] = 3; m_pwObjectIndices[ 8] = 4;
m_pwObjectIndices[ 9] = 5; m_pwObjectIndices[10] = 7; m_pwObjectIndices[11] = 6;
m_pwObjectIndices[12] = 5; m_pwObjectIndices[13] = 8; m_pwObjectIndices[14] = 7;
m_pwObjectIndices[15] = 5; m_pwObjectIndices[16] = 9; m_pwObjectIndices[17] = 8;
m_pwObjectIndices[18] = 10; m_pwObjectIndices[19] = 15; m_pwObjectIndices[20] = 11;
```

```
m_pwObjectIndices[21] = 11; m_pwObjectIndices[22] = 15; m_pwObjectIndices[23] = 12;
m_pwObjectIndices[24] = 12; m_pwObjectIndices[25] = 15; m_pwObjectIndices[26] = 14;
m_pwObjectIndices[27] = 12; m_pwObjectIndices[28] = 14; m_pwObjectIndices[29] = 13;
```

m_pvObjectVertices[0] 与 m_pvObjectVertices[5] 位于点 1 上、m_pvObjectVertices[1] 与 m_pvObjectVertices[6]位于点 2 上、m_pvObjectVertices[3]与 m_pvObjectVertices[11]位于点 3 上，依此类推。每一个点都声明为 D3DXVECTOR3 类型的向量。m_pvObjectVertices 的后两个赋值的变量是纹理坐标。大多数纹理（如位图）都是由颜色值构成的二维数组。每个颜色值称为“纹理元素（texture element）”或“图素（texel）”。每个图素有一个图素坐标也就是一个在纹理中具有惟一性的地址。Direct3D 程序用 u、v 值来指定图素坐标，这就像 2D 笛卡尔坐标系那样用 x、y 来指定坐标。其中所谓的“地址”则可以理解为是一个行列号。Direct3D 在所有纹理中对所有图素都使用一个统一的地址范围，也就是说它使用了一套通用的编址方案，所有图素地址都在[0.0, 1.0]范围内。以下代码设置了索引：

```
// Vertex indices for the object
m_pwObjectIndices[ 0] = 0; m_pwObjectIndices[ 1] = 1; m_pwObjectIndices[ 2] = 2;
m_pwObjectIndices[ 3] = 0; m_pwObjectIndices[ 4] = 2; m_pwObjectIndices[ 5] = 3;
m_pwObjectIndices[ 6] = 0; m_pwObjectIndices[ 7] = 3; m_pwObjectIndices[ 8] = 4;
m_pwObjectIndices[ 9] = 5; m_pwObjectIndices[10] = 7; m_pwObjectIndices[11] = 6;
m_pwObjectIndices[12] = 5; m_pwObjectIndices[13] = 8; m_pwObjectIndices[14] = 7;
m_pwObjectIndices[15] = 5; m_pwObjectIndices[16] = 9; m_pwObjectIndices[17] = 8;
m_pwObjectIndices[18] = 10; m_pwObjectIndices[19] = 15; m_pwObjectIndices[20] = 11;
m_pwObjectIndices[21] = 11; m_pwObjectIndices[22] = 15; m_pwObjectIndices[23] = 12;
m_pwObjectIndices[24] = 12; m_pwObjectIndices[25] = 15; m_pwObjectIndices[26] = 14;
m_pwObjectIndices[27] = 12; m_pwObjectIndices[28] = 14; m_pwObjectIndices[29] = 13;
```

这段代码为 DrawIndexedPrimitive()中指定的 D3DPT_TRIANGLELIST 图元类型设置了索引。正如你在前一章所学到的，Direct3D 允许你使用两种方法来定义多边形，定义多边形的每个顶点，或者定义一系列对顶点列表的索引。第二种方法通常较快且较灵活，因为它使物体的许多多边形可共享顶点数据。

本例使用的物体由 7 个点组成，这 7 个点被用于 16 个顶点的初始化（飞行器顶部 5 个、底部 5 个、后方 6 个）。因此，大约平均要索引两个以上顶点。

这两个物体是借助于 m_pObjects 结构来定义的，m_pObjects 结构只保存了一个矩阵变量。要实现碰撞检测或其他的高级功能，则需要在此保存更多的信息。

```
struct Object
{
    D3DXMATRIX matLocal;
};
```

矩阵中的 3 个向量表示了模型的朝向，分别称为 Look、Up、Right 向量。图 6.9 显示了附着在模型上的这 3 个向量。

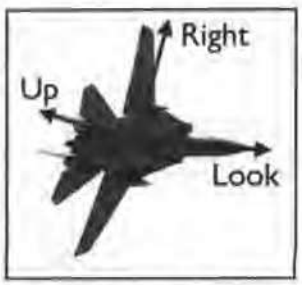


图 6.9　Look、Up、Right 向量

在图 6.9 中可以看到飞机的 Look、Up、Right 向量。该图示有些过于简化了，严格意义上这 3 个向量都应该起始于物体的重心位置。

Look 向量描述了物体面对的方向，它是物体局部坐标系的 z 轴。为了将物体面向屏幕内部，需将 Look 向量设为 D3DVECTOR(0, 0, 1)。

仅用 Look 向量不足以描述物体的朝向。比如，物体若上下颠倒着放置，Look 向量则无法反映出这种朝向上的变化。这就需要 Up 向量的帮助了，它指向相对于物体来说垂直的“上”方向，就像物体自身的 y 轴一样。因此在通常情况下将 Up 向量定义为 D3DVECTOR (0, 1, 0)。如果物体被颠倒过来了，则 Up 向量应为 D3DVECTOR(0, -1, 0)。

这些向量在矩阵中存储次序是：第三行是 vLook 向量、第二行是 vUp 向量、第一行是 vRight 向量。以下矩阵的这 3 个向量都以默认值填充。

```
// yellow object
D3DXMatrixIdentity(&m_pObjects[0].matLocal);
m_pObjects[0].matLocal._41 = -1.0f; // Position

// red object
D3DXMatrixIdentity(&m_pObjects[1].matLocal);
m_pObjects[1].matLocal._41 = 1.0f;
```

D3DXMatrixIdentity()将矩阵设为单位矩阵，因此 vLook 向量为(0.0f, 0.0f, 1.0f)、vUp 向量是(0.0f, 1.0f, 0.0f)、vRight 向量为(1.0f, 0.0f, 0.0f)。接着为物体选择了一个位置，分别设置了两个矩阵中第四行向量的第一个元素。矩阵的第 4 行用于描述物体的位置。

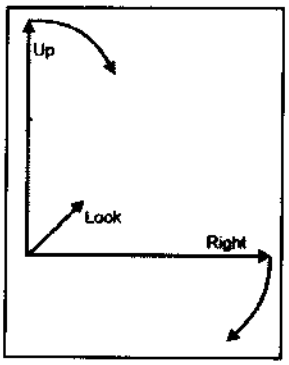


图 6.10　Up 和 Right 向量绕着 Look 向量旋转

旋转一个物体实质上就是将 Look、Up、Right 向量中的两个绕着另一个作旋转。比如，要横滚物体，就需要将 Up 和 Right 向量绕着 Look 向量旋转，图示 6.10 可以帮助你更形象地认识这个旋转。

要使物体产生俯仰，必须将 Up 和 Look 向量绕着 Right 向量旋转。要使物体产生偏航，必须将 Look 和 Right 向量绕着 Up 向量旋转。D3DXVec3 TransformCoord()可用于完成这种向量旋转的计算，该函数通过一个给定的矩阵来变换一个 3D 向量。以下代码具体说明了实现这种旋转的核心内容。

```
// rotate the LOOK & RIGHT Vectors about the UP Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matYaw);
D3DXVec3TransformCoord(&vRight, &vRight, &matYaw);

// rotate the LOOK & UP Vectors about the RIGHT Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matPitch);
D3DXVec3TransformCoord(&vUp, &vUp, &matPitch);

// rotate the RIGHT & UP Vectors about the LOOK Vector
```

```
D3DXVec3TransformCoord(&vRight, &vRight, &matRoll);
D3DXVec3TransformCoord(&vUp, &vUp, &matRoll);
```

每对向量分别借助于偏航、俯仰、横滚矩阵来旋转，而这些矩阵是借助于D3DXMatrixRotationAxis()函数产生的，该函数构建了一个可绕着任意轴旋转的矩阵。这些矩阵是在向量旋转之前构建的。

```
// Matrices for pitch, yaw and roll
D3DXMatrixRotationAxis(&matPitch, &vRight, fPitch );
D3DXMatrixRotationAxis(&matYaw, &vUp, fYaw );
D3DXMatrixRotationAxis(&matRoll, &vLook, fRoll);
```

在旋转向量之前，必须重新归一化所有向量。在旋转计算过程中，由于计算机对浮点数的处理存在精度问题，所以会带来少许的累加错误。在经过几次旋转后，这些舍入错误将使 3 个向量不再相互垂直（见图 6.11）。

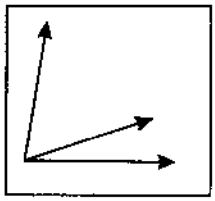


图 6.11　不垂直的向量

以下代码用于重新归一化向量：

```
// base vector regeneration
D3DXVec3Normalize(&vLook, &vLook);
D3DXVec3Cross(&vRight, &vUp, &vLook); // Cross Product of the UP and LOOK Vector
D3DXVec3Normalize(&vRight, &vRight);
D3DXVec3Cross(&vUp, &vLook, &vRight); // Cross Product of the RIGHT and LOOK Vector
D3DXVec3Normalize(&vUp, &vUp);
```

在使用这些向量创建上述旋转矩阵前，必须总是先对它们进行重新归一化。重新归一化这些向量是一项很耗时的计算任务。上述源代码通过以下步骤来使向量相互垂直。

（1）归一化 Look；

（2）计算 Look 与 Up 的叉积以得到 Right；

（3）归一化 Right；

（4）计算 Look 与 Right 的叉积以得到 Up；

（5）归一化 Up。

归一化一个向量意味着将其长度变为一个单位。你需要一个单位长的向量来简化旋转计算。可通过以下方法来计算一个向量的长度：

$\|A\| = \text{sqrt}(x^2 + y^2 + z^2)$

这就是勾股定理的平方根。在数学中一个向量的大小通过两条竖线括住一个大写字母来表示，如||A||。虽然没有 D3DXVec3Normalize()的源代码，但它应该是以下这样的：

```
D3DXVECTOR3* D3DXVec3Normalize(D3DXVECTOR3* v1, const D3DXVECTOR3* v2)
{
    *v1 = *v2/(float) sqrt(v2->x * v2->x + v2->y * v2->y + v2->z * v2->z);
    return (v1);
}
```

这段代码以向量的长度来除向量，向量的长度由勾股定理的平方根计算而得。

为了获得相互垂直的向量，就需要计算两个向量的叉积，从而获得与这两个向量垂直的第 3 个向量（见图 6.12）。比如，在背面剔除中，就使用叉积来确定多边形面对的方向，它使用多边形的两条边来生成一条法线。因此，你也可用这种方法来生成任意表面的法线，只要所使用的两个向量都位于表面所在的平面中即可。

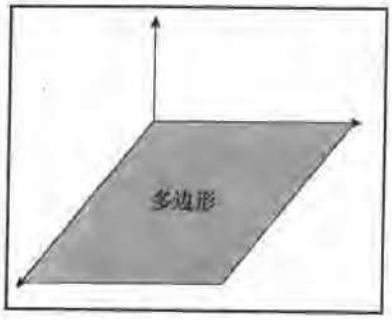


图 6.12　叉积

叉积的计算公式写为 U×V。D3DXVec3Cross()的源代码是不对外公开的，但它应该是这样的：

```
D3DXVECTOR3* D3DXVec3Cross (
    D3DXVECTOR3* pOut,
    const D3DXVECTOR3* pV1,
    const D3DXVECTOR3* pV2)
{
    pOut->x = pV1->y * pV2->z - pV1->z * pV2->y;
    pOut->y = pV1->z * pV2->x - pV1->x * pV2->z;
    pOut->z = pV1->x * pV2->y - pV1->y * pV2->x;

    return pOut;
}
```

正如你在前段代码中所看到的，Look 向量通过叉积重建了与它垂直的 Right 和 Up 向量（在附录 C 中你会学到更多关于叉积的知识）。因此，Look 向量称为重新归一化过程中的“导向（leading）”向量。

这些向量储存在一个矩阵中，每个向量都可单独地用于旋转一个物体。这些向量按以下次序储存：

```
m_pObjects[0].matLocal._11 = vRight.x;
m_pObjects[0].matLocal._12 = vRight.y;
m_pObjects[0].matLocal._13 = vRight.z;
m_pObjects[0].matLocal._21 = vUp.x;
m_pObjects[0].matLocal._22 = vUp.y;
m_pObjects[0].matLocal._23 = vUp.z;
m_pObjects[0].matLocal._31 = vLook.x;
m_pObjects[0].matLocal._32 = vLook.y;
m_pObjects[0].matLocal._33 = vLook.z;
m_pObjects[0].matLocal._41 = vPos.x;
m_pObjects[0].matLocal._42 = vPos.y;
m_pObjects[0].matLocal._43 = vPos.z;
```

第 1 行储存了 Right 向量，第 2 行储存了 Up 向量，第 3 行储存了 Look 向量，第 4 行储存了位置。

以下代码显示了旋转和定位一个物体的全部内容：

```
D3DXVECTOR3 vRight, vUp, vLook, vPos;
FLOAT fRoll, fPitch, fYaw;

FLOAT fspeed = fOneStep;
```

```
fRoll = fPitch = fYaw = 0.0f;

if (m_bKey['D']) fRoll -= fOneStep;
if (m_bKey['A']) fRoll += fOneStep;
if (m_bKey['S']) fPitch -= fOneStep;
if (m_bKey['W']) fPitch += fOneStep;
if (m_bKey['Q']) fYaw -= fOneStep;
if (m_bKey['E']) fYaw += fOneStep;

vRight.x = m_pObjects[0].matLocal._11;
vRight.y = m_pObjects[0].matLocal._12;
vRight.z = m_pObjects[0].matLocal._13;
vUp.x = m_pObjects[0].matLocal._21;
vUp.y = m_pObjects[0].matLocal._22;
vUp.z = m_pObjects[0].matLocal._23;
vLook.x = m_pObjects[0].matLocal._31;
vLook.y = m_pObjects[0].matLocal._32;
vLook.z = m_pObjects[0].matLocal._33;
vPos.x = m_pObjects[0].matLocal._41;
vPos.y = m_pObjects[0].matLocal._42;
vPos.z = m_pObjects[0].matLocal._43;

// base vector regeneration
D3DXVec3Normalize(&vLook, &vLook);
D3DXVec3Cross(&vRight, &vUp, &vLook); // Cross Product of the UP and LOOK Vector
D3DXVec3Normalize(&vRight, &vRight);
D3DXVec3Cross(&vUp, &vLook, &vRight); // Cross Product of the RIGHT and LOOK Vector
D3DXVec3Normalize(&vUp, &vUp);

// Matrices for pitch, yaw and roll
D3DXMATRIX matPitch, matYaw, matRoll;
D3DXMatrixRotationAxis(&matPitch, &vRight, fPitch );
D3DXMatrixRotationAxis(&matYaw, &vUp, fYaw );
D3DXMatrixRotationAxis(&matRoll, &vLook, fRoll);

// rotate the LOOK & RIGHT Vectors about the UP Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matYaw);
D3DXVec3TransformCoord(&vRight, &vRight, &matYaw);

// rotate the LOOK & UP Vectors about the RIGHT Vector
D3DXVec3TransformCoord(&vLook, &vLook, &matPitch);
D3DXVec3TransformCoord(&vUp, &vUp, &matPitch);

// rotate the RIGHT & UP Vectors about the LOOK Vector
D3DXVec3TransformCoord(&vRight, &vRight, &matRoll);
D3DXVec3TransformCoord(&vUp, &vUp, &matRoll);

// move forward
if (m_bKey['R'])
{
    vPos.x+=fspeed*vLook.x;
    vPos.y+=fspeed*vLook.y;
    vPos.z+=fspeed*vLook.z;
}

// move back
```

```
if (m_bKey['F']) // Key END
{
    vPos.x-=fspeed*vLook.x;
    vPos.y-=fspeed*vLook.y;
    vPos.z-=fspeed*vLook.z;
}

m_pObjects[0].matLocal._11 = vRight.x;
m_pObjects[0].matLocal._12 = vRight.y;
m_pObjects[0].matLocal._13 = vRight.z;
m_pObjects[0].matLocal._21 = vUp.x;
m_pObjects[0].matLocal._22 = vUp.y;
m_pObjects[0].matLocal._23 = vUp.z;
m_pObjects[0].matLocal._31 = vLook.x;
m_pObjects[0].matLocal._32 = vLook.y;
m_pObjects[0].matLocal._33 = vLook.z;
m_pObjects[0].matLocal._41 = vPos.x;
m_pObjects[0].matLocal._42 = vPos.y;
m_pObjects[0].matLocal._43 = vPos.z;
```

这段代码展示了如何从矩阵中获取各个向量，然后对它们进行重新归一化、旋转。此外，还将位置数据加上速度系数与Look向量的乘积，以使物体可以沿着Look向量所指的方向移动。

目前，你已经看到了第一种在3D中旋转物体的方法。1steps4示例将会介绍另一种使用四元数来作旋转的方法，它比当前的方法更有效率。

4. 1steps4示例

1steps4.exe示例展示了另一种更高效的旋转物体的方法。你可以使用与前例相同的按键来操作物体。整个实现代码如下。

```
FLOAT fRoll, fPitch, fYaw;
fRoll = fPitch = fYaw = 0.0f;

if (m_bKey['D']) fRoll -= fOneStep;
if (m_bKey['A']) fRoll += fOneStep;
if (m_bKey['S']) fPitch -= fOneStep;
if (m_bKey['W']) fPitch += fOneStep;
if (m_bKey['Q']) fYaw -= fOneStep;
if (m_bKey['E']) fYaw += fOneStep;

// Update rotation matrix
D3DXQUATERNION qR;
D3DXMATRIX matRot;
D3DXQuaternionRotationYawPitchRoll (&qR, fYaw, fPitch, fRoll);
D3DXMatrixRotationQuaternion (&matRot, &qR);
D3DXMatrixMultiply (&m_pObjects[0].matLocal, &matRot, &m_pObjects[0].matLocal);

// Update position
D3DXVECTOR3 vLook; // Look vector
vLook.x = m_pObjects[0].matLocal._31; // extract Look vector
vLook.y = m_pObjects[0].matLocal._32;
vLook.z = m_pObjects[0].matLocal._33;
```

```
// move forward
if (m_bKey['R'])
{
    m_pObjects[0].matLocal._41 += fOneStep * vLook.x;
    m_pObjects[0].matLocal._42 += fOneStep * vLook.y;
    m_pObjects[0].matLocal._43 += fOneStep * vLook.z;
}

// move back
if (m_bKey['F'])
{
    m_pObjects[0].matLocal._41 -= fOneStep * vLook.x;
    m_pObjects[0].matLocal._42 -= fOneStep * vLook.y;
    m_pObjects[0].matLocal._43 -= fOneStep * vLook.z;
}
```

这真的是一段简洁方便的旋转物体代码。你甚至可以更进一步地减少代码的行数，但为了清晰地表述，我没有这么做。

早在 1843 年，William Rowan Hamilton 爵士就已经发明了作为复数扩展的四元数(quaternion)，但是直至 1985 年才有一个叫 Ken Shoemake 的人在 SIGGRAPH[①]（http://www.siggraph. org）把四元数引入计算机图形处理领域。

附注

SIGGRAPH 的官方任务陈述为：我们的任务是促进有关计算机图形与交互技术的信息的产生和传播。

四元数将三维空间中旋转的概念扩展到四维空间，这对于表示和处理 3D 中点的旋转很有用。四元数还可用于实现：

- 骨骼动画（skeletal animation）；
- 反向动力学动画（inverse cinematics）；
- 3D 物理学。

在游戏中可以使用四元数来取代旋转矩阵，它可于描述 3D 空间中绕任意轴的任意旋转。那么有什么理由要用四元数来代替旋转矩阵呢？

- 四元数占用的空间比矩阵少；
- 一些操作在视觉上会显得更为平滑，比如可以在两个四元数之间做插值运算。

让我们先从一个基本的问题开始讨论。什么是四元数？

对于这个问题我更偏向于回答：一个四元数是一个向量加上一次旋转。但这绝不是一种数学方式的描述，不过这个简单的描述却可帮助我在脑海中想象它（见图 6.13）。

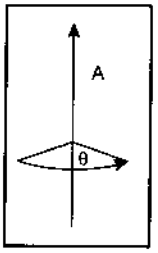


图 6.13　一个四元数表示一个绕 A 轴的旋转

四元数中包含有一个标量和一个 3D 向量。其中，标量通常用 w 表示，

① 译注：Special Interest Group on Computer Graphic，美国计算机协会的计算机图形专业组。

而向量既可用单一实体 v 来表示，也可以用它的 3 个分量 x、y、z 来表示。以下给出了这两种表示法的示例说明。

```
[w v]
[w (x, y, z)]
```

使用 Direct3D 的四元数类型时，可以在四元数变量的名称后面添加.x、.y、.z、.w 来分别访问它的 w、x、y、z 分量。

向量 v 定义为旋转轴。为了简化计算，v 应为一个单位向量。v 的方向定义了"正"旋转[3]的方向，标量 p 是绕 v 轴旋转的数量。如此，一组(p, v)就是定义了一次旋转，即绕 v 轴旋转 p 弧度。不幸的是，p 和 v 并不是简单地直接储存于四元数的 4 个数字中。以下等式显示了 p 和 v 是如何存储在一个四元数中的，同时使用了两种四元数的表示形式。

```
q = [cos(p/2) sin(p/2)v]
q = [cos(p/2) (sin(p/2)x, sin(p/2)y, sin(p/2)z)]
```

在本例中偏航、俯仰、横滚都被转换成了一个四元数，也就是说将欧拉角（Euler angle）转换成了一个四元数。在进行转换时需要有一点技巧，运算的次序必须正确。通过将欧拉角转换成 3 个独立的四元数，从而便可将任意向量作为旋转坐标轴。然后将这 3 个四元数相乘后，即可得到一个最终组合后的四元数。

3 个欧拉角（偏航、俯仰、横滚）可分别构成 3 个独立的四元数。

```
Qx = [ cos(yaw/2), (sin(yaw/2), 0, 0)]
Qy = [ cos(pitch/2), (0, sin(pitch/2), 0)]
Qz = [ cos(roll/2), (0, 0, sin(roll/2))]
```

通过乘法 Qx * Qy * Qz，便可获得一个最终的四元数。以下函数用于将偏航、俯仰、横滚转换成一个四元数：

```
D3DXQUATERNION * D3DXQuaternionRotationYawPitchRoll (
    D3DXQUATERNION * pOut,
    FLOAT yaw,
    FLOAT pitch,
    FLOAT roll)
{
    FLOAT sinY, cosY, sinP, cosP, sinR, cosR;

    sinY = sin(0.5f * yaw);
    cosY = cos(0.5f * yaw);

    sinP = sin(0.5f * pitch);
    cosP = cos(0.5f * pitch);

    sinR = sin(0.5f * roll);
    cosR = cos(0.5f * roll);
```

译注[3]：绕向量 v 的旋转符合右手旋转法则，右手旋转法则是指用右手握住旋转轴向量，并使大拇指指向与该向量相同的方向，那么其余四指弯曲后所形成的环形方向即为绕该向量旋转的正方向，正数角度的旋转都将按正方向进行。

```
    pOut->x = cosR * sinP * cosY + sinR * cosP * sinY;
    pOut->y = cosR * cosP * sinY - sinR * sinP * cosY;
    pOut->z = sinR * cosP * cosY - cosR * sinP * sinY;
    pOut->w = cosR * cosP * cosY + sinR * sinP * sinY;

    return pOut;
}
```

3 个四元数在底下的 4 行中相乘，最终结果储存在 pOut 中。

由于 Direct3D 变换引擎需要用矩阵来执行旋转，因此必须将四元数转换为矩阵格式。可通过以下代码来完成这种转换：

```
D3DXMATRIX * D3DXMatrixRotationQuaternion(
    D3DXMATRIX * pOut,
    CONST D3DXQUATERNION *pQ)
{
    D3DMath_MatrixFromQuaternion (&pOut, *pQ.x, *pQ.y, *pQ.z, *pQ.w);
return pOut;
}

// From d3dmath.cpp:
VOID D3DMath_MatrixFromQuaternion( D3DMATRIX& mat, FLOAT x, FLOAT y, FLOAT z,
                                   FLOAT w )
{
    FLOAT xx = x*x; FLOAT yy = y*y; FLOAT zz = z*z;
    FLOAT xy = x*y; FLOAT xz = x*z; FLOAT yz = y*z;
    FLOAT wx = w*x; FLOAT wy = w*y; FLOAT wz = w*z;

    mat._11 = 1 - 2 * ( yy + zz );
    mat._12 = 2 * ( xy - wz );
    mat._13 = 2 * ( xz + wy );

    mat._21 = 2 * ( xy + wz );
    mat._22 = 1 - 2 * ( xx + zz );
    mat._23 = 2 * ( yz - wx );

    mat._31 = 2 * ( xz - wy );
    mat._32 = 2 * ( yz + wx );
    mat._33 = 1 - 2 * ( xx + yy );

    mat._14 = mat._24 = mat._34 = 0.0f;
    mat._41 = mat._42 = mat._43 = 0.0f;
    mat._44 = 1.0f;
}
```

请注意以上前两个函数的源代码是无法获取的，在此所展示的仅是一种示意而已。

关于世界变换，你已学习了：

- 使用矩阵来储存旋转以及其他变换；
- 认识到了矩阵乘法中次序的重要性；
- 在 3D 中使用 3 个朝向向量旋转物体；

■ 在3D中使用一个由偏航、俯仰、横滚构建的四元数来旋转物体。

另外，我会在本章后面的“使用视口”一节中介绍如何创建并使用视口。

就以上这些示例而言，还有一个明显有待改进的地方，那就是实现一种基于时间的物理模拟。不过这个话题有点超出了本书的讨论范围，但是你也可以考虑一下，使用这种物理模拟来将用户的输入解释为物体的移动速度。既然已知飞船的几个相互垂直的基向量（Look、Up、Right），那么实现这种模拟就只是一个简单的一维物理问题了。

```
Position += deltaTime * InputSpeed * forward
```

利用这点便可实现阻尼（damping）效果。那看上去会非常棒，当你的飞船作减速或旋转时会有一些动力制约。你还可以给不同的飞船赋予不同的速度，从而让用户可以通过飞船的速度感觉到飞船的大小。一艘较大的飞船行动迟缓，而一艘较小的飞船则灵活轻巧。

回顾了世界变换之后，接下去继续学习观察变换。下一节中主要的挑战就是构建一个照相机类。

6.1.2 观察变换

观察变换通常用于描述一个观察者在场景中的位置和朝向。许多人通过照相机看场景的例子来描述这种变换。观察变换可通过一个观察变换矩阵或一个短观察矩阵来完成。这种矩阵在概念上不同于世界矩阵，但机制是一样的，这两种类型的变换矩阵都以相同的方式构建。世界矩阵和观察矩阵的主要区别在于3个朝向向量的储存方式。世界矩阵以行的次序存储它们，而观察矩阵则以列的次序储存它们。

使用观察矩阵来构建一个立体的照相机模型后，便允许玩家通过一系列不同的观察点来看世界。本节概述了两种在游戏中比较有用的照相机模型。

■ 一种是由Direct3D为游戏提供的高效的照相机模型。在这个模型中，照相机不能绕所有的轴作三维旋转。不过，该模型适用于大多数情况；

■ 另一种是经过改进的照相机模型，它允许你在3D中任意地旋转照相机，适用于飞行模拟器或太空射击类型的游戏。

在Direct3D中第一种照相机模型由D3DXMatrixLookAtLH()函数提供支持。该函数接受位置、Look向量、Up向量及一个用于表示照相机位置和朝向的矩阵。不幸的是，该函数的源代码是无法获取的。但它的内容应该大致如下。

```
#define MX_EPS5 1e-5f
D3DXMATRIX * D3DXMatrixLookAtLH ( D3DXMATRIX * pOut, const D3DXVECTOR3 * pEye,
                                  const D3DXVECTOR3 * pAt,
                                  const D3DXVECTOR3 * pUp)
{
    // the three orientation vectors of the camera
    D3DXVECTOR3 uAxis, vAxis, nAxis;

    // create look vector and normalize it
```

```
    nAxis = (*pAt) - (*pEye);
    if (D3DXVec3Normalize(&nAxis, &nAxis) == NULL) return NULL;

    // create up vector
    vAxis = (*pUp) - nAxis * D3DXVec3Dot(pUp, &nAxis);

    if (D3DXVec3Length(&vAxis) < MX_EPS5)
    {
        vAxis.x = - nAxis.y * nAxis.x;
        vAxis.y = 1.0f - nAxis.y * nAxis.y;
        vAxis.z = - nAxis.y * nAxis.z;

        if (D3DXVec3Length(&vAxis) < MX_EPS5)
        {
            vAxis.x = - nAxis.z * nAxis.x;
            vAxis.y = - nAxis.z * nAxis.y;
            vAxis.z = 1.0f - nAxis.z * nAxis.z;
            if (D3DXVec3Length(&vAxis) < MX_EPS5) return NULL;
        }
    }

    // normalize up vector
    D3DXVec3Normalize(&vAxis, &vAxis);

    // create right vector
    D3DXVec3Cross(&uAxis, &vAxis, &nAxis);

    pOut->_11 = uAxis.x; pOut->_12 = vAxis.x; pOut->_13 = nAxis.x; pOut->_14 = 0.0f;
    pOut->_21 = uAxis.y; pOut->_22 = vAxis.y; pOut->_23 = nAxis.y; pOut->_24 = 0.0f;
    pOut->_31 = uAxis.z; pOut->_32 = vAxis.z; pOut->_33 = nAxis.z; pOut->_34 = 0.0f;
    pOut->_41 = -D3DXVec3Dot(pEye, &uAxis);
    pOut->_42 = -D3DXVec3Dot(pEye, &vAxis);
    pOut->_43 = -D3DXVec3Dot(pEye, &nAxis);
    pOut->_44 = 1.0f;

    return pOut;
}
```

这个函数的内部创建了 3 个向量。它使用照相机的位置以及它所注视的位置来创建一个名为 nAxis 的 Look 向量。Up 向量为 vAxis，它由参数提供，并作了伸缩和归一化。Right 向量由 Look 向量和 Up 向量的叉积计算而得。一个照相机使用这些朝向向量的方式类似于在 3D 中定位物体时所用的方式（见图 6.14）。

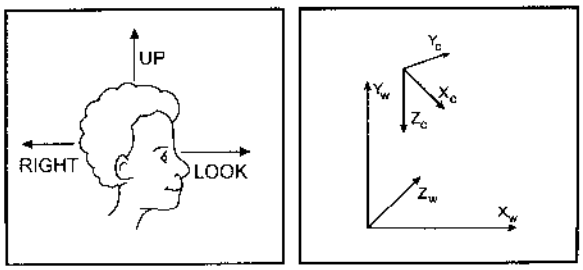


图 6.14　Up、Right、Look 向量

Look 向量描述了照相机面对的方向，它是照相机局部坐标系的 z 轴，在图 6.14 中为 Z_c。为了将照相机的方向设置为面对着屏幕内部，则需将 Look 向量设为 D3DVECTOR (0, 0, 1)。

仅用 Look 向量不足以描述照相机的朝向。照相机可能会被颠倒过来放置，如此 Look 向量则无法反映出这种朝向上的变换。这就需要借助于 Up 向量了，Up 向量指向相对于照相机来说垂直的“上”方向，就像照相机自身的 y 轴一样，在图 6.14 中它被标注为 Y_c。因此，当照相机正常摆放时，应将 Up 向量定义为 D3DVECTOR (0, 1, 0)；如果将照相机颠倒过来放置，则应将 Up 向量定义为 D3DVECTOR(0, −1, 0)。Look 向量和 Up 向量的叉积生成了 Right 向量，在图 6.14 中将其标注为 X_c。

为了更形象地理解照相机绕着它的一根轴作旋转的概念，请试想一位坐在一架 F-22 座舱中的飞行员。如果这个飞行员在 F-22 中踩下了向左或向右的踏脚板，则 Look 向量和 Right 向量就会绕着 Up 向量或 y 轴进行旋转（偏航效果）。如果他向左或向右推动飞行摇杆，则 Up 向量和 Right 向量就会绕着 Look 向量或 z 轴旋转（横滚效果）。如果他向前或向后推动飞行摇杆，则 Look 向量和 Up 向量就会绕着 Right 向量或 x 轴旋转（俯仰效果）。

D3DXMatrixLookAtLH()会返回一个观察矩阵。你可以看到上述代码是如何将向量填充到列中的。第一列的前 3 个值储存了 Up 向量，第二列的前 3 个值储存了 Right 向量，第三列的前 3 个值储存了 Look 向量。

D3DXMatrixLookAtLH()对于构建一个跟随式照相机（follow-up camera）是很有帮助的。比如，Look 向量指向照相机希望跟随的物体，而照相机的位置向量则可位于该个物体的周围或其前后。完成这个功能的伪代码如下：

```
static FLOAT tic = -200.0f * rand();
tic += 0.01f;

vLook = m_pObjects[0].vLoc; // position of object that should be in the focus of the camera
vPos.z = vLook.z - 5.0f;// position of the camera
vPos.x = vLook.x ;
vPos.y = vLook.y + (FLOAT)(1.0f + 0.3f * sin(tic * 0.33f)); // slightly up and down
   move: more lively

D3DXMATRIX view;
D3DXMatrixLookAtLH( &view, &vPos, // pEye = Position
                    &vLook,    // pAt
                    &vUp );    // pUp
```

这段代码是为简单应用设计的。比如，它对于头罩式跟踪显示（head-mounted tracking display）或者跟随式照相机来说就很不错，但它无法帮助你构建一个用于太空射击游戏或飞行模拟器中的照相机。因此，这类游戏更多地会使用第二种照相机模型。第二种照相机模型有两种实现方法，一种是将照相机绕着向量旋转，另一种是使用四元数将照相机绕着任意轴旋转。我将先介绍第一种绕着其自身的轴旋转的照相机。

1. 1steps5 示例

先前我介绍了在 3D 中借助于 Up、Right、Look 向量来旋转物体的方法，旋转一个照相机时也可以使用类型的方法（参阅图 6.14）。在图 6.14，所示的照相机技术只使用了 3 个向量中的 2 个（Look 和 Up）来跟踪照相机的朝向，而这项技术用到了所有 3 个向量。将照相机绕着 Look 向量旋转表示横滚它；将照相机绕着 Right 向量旋转可改变它的俯仰度；而将照相机绕着 Up 向量旋转则可使它偏航。

本例将这些向量分别存储在 4 个向量对象中，而非像原来那样存储在矩阵中，本例还将照相机朝向物体。控制物体的方法与 1steps3 示例相同。如果你已经了解了前例中那段将物体绕着一个向量作旋转的源代码，那么以下这段负责照相机旋转的源代码对你来说应该是相当熟悉的。

```
static D3DXVECTOR3 vCameraLook=D3DXVECTOR3(0.0f,0.0f,1.0);    // look vector
static D3DXVECTOR3 vCameraUp=D3DXVECTOR3(0.0f,1.0f,0.0f);     // up vector
static D3DXVECTOR3 vCameraRight=D3DXVECTOR3(1.0f,0.0f,0.0f);  // right vector
static D3DXVECTOR3 vCameraPos=D3DXVECTOR3(0.0f,0.0f,-5.0f);   // position vector

fPitch = fYaw = fRoll = 0.0f;
fspeed = fOneStep * 1.5f;

if (m_bKey[VK_UP])     fPitch=-1.0f * m_fElapsedTime;      // fPitch
if (m_bKey[VK_DOWN])   fPitch=+1.0f * m_fElapsedTime;
if (m_bKey[VK_LEFT])   fYaw=+1.0f * m_fElapsedTime;        // fYaw
if (m_bKey[VK_RIGHT])  fYaw=-1.0f * m_fElapsedTime;
if (m_bKey['C'])       fRoll=-1.0f * m_fElapsedTime;       // fRoll
if (m_bKey['X'])       fRoll=+1.0f * m_fElapsedTime;

// camera forward
if (m_bKey[VK_HOME]) // Key HOME
{
    vCameraPos.x+=fspeed*vCameraLook.x;
    vCameraPos.y+=fspeed*vCameraLook.y;
    vCameraPos.z+=fspeed*vCameraLook.z;
}

// camera back
if (m_bKey[VK_END]) // Key END
{
    vCameraPos.x-=fspeed*vCameraLook.x;
    vCameraPos.y-=fspeed*vCameraLook.y;
    vCameraPos.z-=fspeed*vCameraLook.z;
}

// base vector regeneration
D3DXVec3Normalize(&vCameraLook, &vCameraLook);
D3DXVec3Cross(&vCameraRight, &vCameraUp, &vCameraLook); // Cross Product of the UP and
   LOOK Vector
D3DXVec3Normalize(&vCameraRight, &vCameraRight);
D3DXVec3Cross(&vCameraUp, &vCameraLook, &vCameraRight); // Cross Product of the RIGHT
   and LOOK Vector
D3DXVec3Normalize(&vCameraUp, &vCameraUp);
```

```
// Matrices for pitch, yaw and roll
D3DXMatrixRotationAxis(&matPitch, &vCameraRight, fPitch );
D3DXMatrixRotationAxis(&matYaw, &vCameraUp, fYaw );
D3DXMatrixRotationAxis(&matRoll, &vCameraLook, fRoll);

// rotate the LOOK & RIGHT Vectors about the UP Vector
D3DXVec3TransformCoord(&vCameraLook, &vCameraLook, &matYaw);
D3DXVec3TransformCoord(&vCameraRight, &vCameraRight, &matYaw);

// rotate the LOOK & UP Vectors about the RIGHT Vector
D3DXVec3TransformCoord(&vCameraLook, &vCameraLook, &matPitch);
D3DXVec3TransformCoord(&vCameraUp, &vCameraUp, &matPitch);

// rotate the RIGHT & UP Vectors about the LOOK Vector
D3DXVec3TransformCoord(&vCameraRight, &vCameraRight, &matRoll);
D3DXVec3TransformCoord(&vCameraUp, &vCameraUp, &matRoll);

D3DXMATRIX view;
D3DXMatrixIdentity( &view );
view._11 = vCameraRight.x;
view._12 = vCameraUp.x;
view._13 = vCameraLook.x;
view._21 = vCameraRight.y;
view._22 = vCameraUp.y;
view._23 = vCameraLook.y;
view._31 = vCameraRight.z;
view._32 = vCameraUp.z;
view._33 = vCameraLook.z;
view._41 = - D3DXVec3Dot( &vCameraPos, &vCameraRight );
view._42 = - D3DXVec3Dot( &vCameraPos, &vCameraUp );
view._43 = - D3DXVec3Dot( &vCameraPos, &vCameraLook );

m_pd3dDevice->SetTransform(D3DTS_VIEW, &view);
```

在这段源代码的起始处可以看到 3 个朝向向量和位置向量是如何被初始化的。本例使用 4 个向量来取代矩阵，保存朝向数据和位置数据，其实这并没有多大的差别。接下去，重新归一化了向量、创建了旋转矩阵、并将两个向量绕着旋转向量进行旋转。这些操作与 1steps3.exe 示例所做的事情是相同的。然而惟一不同的是朝向向量在观察矩阵中的储存方式。朝向向量是以列的次序储存的，而非前例中以行的次序。另外，矩阵中的位置数据是通过将照相机位置与 3 个朝向向量做点积计算而得的。为了将照相机的位置移到 3 个朝向向量的原点，需要将位置向量乘以 3 个朝向向量并取反。

现在你已经知道了在 3D 中如何中将一个照相机绕所有的 3 根轴旋转，那是通过将两个向量绕另一个向量作旋转来做到的。接下去，准备学习另一种更先进的做法，在以下这个示例中，将通过一个四元数来做到同样的事情。

2. 1steps6 示例

或许你曾经玩过古墓丽影系列中的某个版本（有谁会忘记那个带枪的女孩（Lara Croft）

呢？该游戏是最早使用四元数旋转来模拟所有照相机运动的商业游戏之一。许多第三人称游戏都通过四元数来使用虚拟照相机，这些游戏通常将一个虚拟照相机置于玩家角色的身后或身旁一定距离的位置上。目前，大多数这类照相机都是用四元数来定向的。

使用一个四元数来旋转一个照相机的方法与前一节中使用四元数来旋转一个 3D 物体的方法相类似。本示例以 1steps4 为基础，将一个四元数用于旋转照相机。另外，照相机可分别向上、下、左、右滑动。以下是主要的源代码片断：

```
fRoll = fPitch = fYaw = 0.0f;
D3DXVECTOR3 vPos(0.0f, 0.0f, 0.0f);
static D3DXMATRIX matView = D3DXMATRIX( 1.0f, 0.0f, 0.0f, 0.0f,
                                        0.0f, 1.0f, 0.0f, 0.0f,
                                        0.0f, 0.0f, 1.0f, 0.0f,
                                        0.0f, 0.0f,-5.0f, 1.0f);

// Process keyboard input
if (m_bKey[VK_UP])      fPitch += fOneStep; // Pitch Up
if (m_bKey[VK_DOWN])    fPitch -= fOneStep; // Pitch Down
if (m_bKey[VK_LEFT])    fYaw += fOneStep;   // Turn Left
if (m_bKey[VK_RIGHT])   fYaw -= fOneStep;   // Turn Right
if (m_bKey['C'])        fRoll += fOneStep;  // Rotate Left
if (m_bKey['X'])        fRoll -= fOneStep;  // Rotate Right
if (m_bKey[VK_HOME])    vPos.z += fOneStep; // Move Forward
if (m_bKey[VK_END])     vPos.z -= fOneStep; // Move Backward
if (m_bKey[VK_NUMPAD4]) vPos.x -= fOneStep; // Slide Left
if (m_bKey[VK_NUMPAD6]) vPos.x += fOneStep; // Slide Right
if (m_bKey[VK_NUMPAD8]) vPos.y += fOneStep; // Slide Down
if (m_bKey[VK_NUMPAD2]) vPos.y -= fOneStep; // Slide Up

// Update position and view matrices
D3DXMATRIX matR, matTemp;
D3DXQuaternionRotationYawPitchRoll (&qR, fYaw, fPitch, fRoll);
D3DXMatrixRotationQuaternion (&matR, &qR);
D3DXMatrixMultiply (&matView, &matR, &matView);
D3DXMatrixTranslation (&matTemp, vPos.x, vPos.y, vPos.z);
D3DXMatrixMultiply (&matView, &matTemp, &matView);
D3DXMatrixInverse (&matTemp, NULL, &matView);

m_pd3dDevice->SetTransform(D3DTS_VIEW, &matTemp );
```

这次使用 matView 矩阵来储存照相机的朝向。正如你已经知道的，矩阵中第三列存储了照相机的 Look 向量，第二列存储了 Up 向量，第一列存储了 Right 向量。另外，留意一下照相机的位置数据，它现在保存在 vPos 变量中。上述代码完成了以下几步工作：

（1）根据偏航、俯仰、横滚构建一个四元数，其中包含了相关的旋转信息；

（2）从四元数创建一个旋转矩阵；

（3）将这个旋转矩阵乘以观察矩阵；

（4）依据 vPos 中位置的变化来平移观察矩阵；

（5）获取以上结果矩阵的逆矩阵；

（6）将此矩阵提供给 Direct3D 引擎。

前 3 个步骤已在世界变换一节中介绍过了。这 3 步的结果是得到一个包含旋转信息的矩阵。第 4 步是设置照相机的位置。在第 5 步中，使用了一个 Direct3D 实用程序库的函数来求逆矩阵。为什么需要求一个逆矩阵呢？使用一个四元数来旋转照相机时，由于照相机会被平移到任意点上，然后绕着该点作旋转。如果不求出旋转矩阵的逆矩阵，则绕任意点所作的旋转将不是线性的。现在的问题就是如何来求一个旋转矩阵的逆矩阵？

通过相反的旋转角度即可获得一个旋转矩阵的逆矩阵。绕原点作一次 A 角度的旋转，可通过一次-A 角度的旋转来抵消。如果一个矩阵具有一个逆矩阵，则该逆矩阵可书写为 A^{-1}，并可得 $AA^{-1}=A^{-1}A=I$。

矩阵 A 乘以其逆矩阵的结果是一个单位矩阵 I。一个旋转矩阵的逆矩阵可用于将一个经过旋转的点映射回旋转前的位置。[1]

这一节介绍了朝向向量在观察矩阵中的储存方法，观察矩阵则通过 SetTransform()来设置。此外，还介绍了一种旋转照相机的技术，其中借助于 3 个相互垂直的向量来定位照相机。该技术的缺点在于每次使用这 3 个向量之前都必须重新归一化它们。倘若使用四元数来控制照相机的旋转，就不会这么低效了。尽管四元数很难理解，但现今看来，这已成为了一种普遍的照相机控制技术。

对于先前所讨论的照相机系统仍存在改进的余地。在 CD-ROM 的\1steps7 目录下有一个 1steps7.exe 示例，它实现了一个简单的基于时间的物理系统。在其中你可以感觉到飞船在旋转或移动时的速度。

6.1.3 投影变换

透视投影变换将照相机的观察截头体（viewing frustum；类似于削去顶部的金字塔形状，它定义了照相机可以看到的内容）转换成一个立方体空间，如图 6.15 所示（通过一个立方体几何形状来作剪裁是非常容易）。

物体靠近照相机时会变大，远离照相机时会变小。在这种变换中，原来平行的线通常在经过投影变换后将不再平行。该变换将透视法应用到一个 3D 场景中。它将 3D 几何体投影为一种可在 2D 显示器上显示的形式。近平面和远平面定义了可视物体的距离，在最终的渲染结果中将不会包含在此距离范围之外的物体。视域（field-of-view；FOV）角定义了视图的宽度和高度（见图 6.16）。

译注[1]：这段话作者未尽其言，让我来进一步揭示一下观察变换矩阵的本质。试想照相机处于世界坐标系原点，Look 为 Z 轴正方向、Up 为 Y 轴正方向、Right 为 X 轴正方向；如果你旋转照相机并移动到一个观察位置，那么这其中便进行了旋转和平移变换，这些变换矩阵的乘积矩阵 TVIEW 即为置位这个照相机所需的组合变换矩阵。那么对于这个照相机而言观察变换矩阵是什么呢？很简单，就是 TVIEW 的逆矩阵，即反方向的平移和旋转。所以所谓“观察变换”其实就是将原来所有基于世界坐标系的顶点变换到照相机坐标系中去。此乃简要译注，对于有兴趣进一步了解详情的读者，建议参阅计算机图形学方面的书籍。

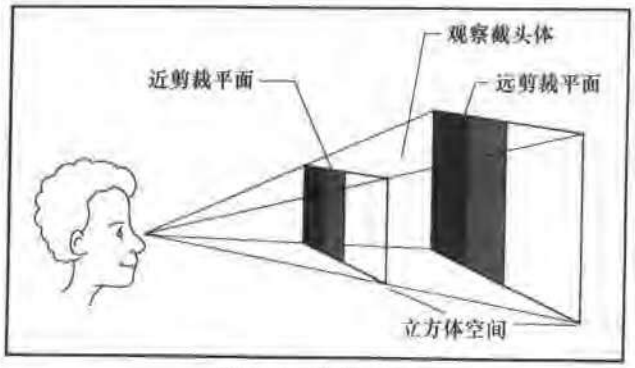


图 6.15　投影空间

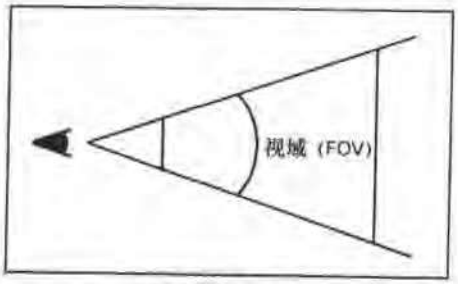


图 6.16　视域

比如，变焦镜头具有一个较小的视域，这就意味着风景中某块较小的部分将会占据图像中某块较大的区域，这会使得物体最终变得更大。D3DX 库中包含了十几种创建投影矩阵的函数。其中有一些是针对于右手坐标系的版本，这些函数可用于将一个现有基于 OpenGL 的引擎移植到 Direct3D 上。在本章的示例中，我仅使用了其中的一个 D3DXMatrixPerspectiveFovLH()函数。

```
D3DXMATRIX matProj;
FLOAT fAspect = m_d3dsdBackBuffer.Width / (FLOAT)m_d3dsdBackBuffer.Height;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, fAspect, 1.0f, 500.0f );
m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

它的语法是：

```
D3DXMATRIX * D3DXMatrixPerspectiveFovLH(
    D3DXMATRIX *pOut,
    FLOAT fovY,
    FLOAT Aspect,
    FLOAT zn,
    FLOAT zf
);
```

第一个参数返回投影矩阵。第二个参数接受 y 轴方向的视域角，以弧度表示。第三个参数给出高宽比。提供后备缓冲器的高度与宽度是很有用的，因为如果用户改变了窗口的大小，

那么就可以通过改变高宽比来保证仍能正确地显示出物体的大小。最后两个参数用于指定近/远剪裁平面，必须分别为每个平面指定一个 z 值。比如，在物体间穿梭飞行时，便可将近剪裁平面设为 1.0f，远剪裁平面设为 500.0f。投影矩阵还可用于处理其他一些复杂问题，诸如避免深度缓冲器的视觉错误（artifact）、渲染高分辨率的屏幕截图、从渲染中排斥多余的几何体，不过这些概念已超出了本书的讨论范围。在大多数情况中，前面所示的那样来设置一个投影矩阵就已经足够了。

6.1.4 使用视口

在完成了以上 3 个变换（世界变换、观察变换、投影变换）步骤之后，设备还需要决定如何将图形数据最终映射到每个像素上。这可以通过定义一个视口来帮助设备完成这项操作。在默认情况下，一个窗口应用程序中视口的大小被定义为该窗口客户区的大小，而一个全屏应用程序中视口的大小则定义为屏幕的分辨率。因此，通常不需要显式地设置视口大小。不过 Direct3D 也允许你将窗口中的某一个部分区域指定为视口。这对于在同一个窗口中渲染多个视图是很有用的，本章先前的 1steps2.exe 示例就是这样一种程序。在图 6.17 中，通过两条白线来界分 4 个视口。

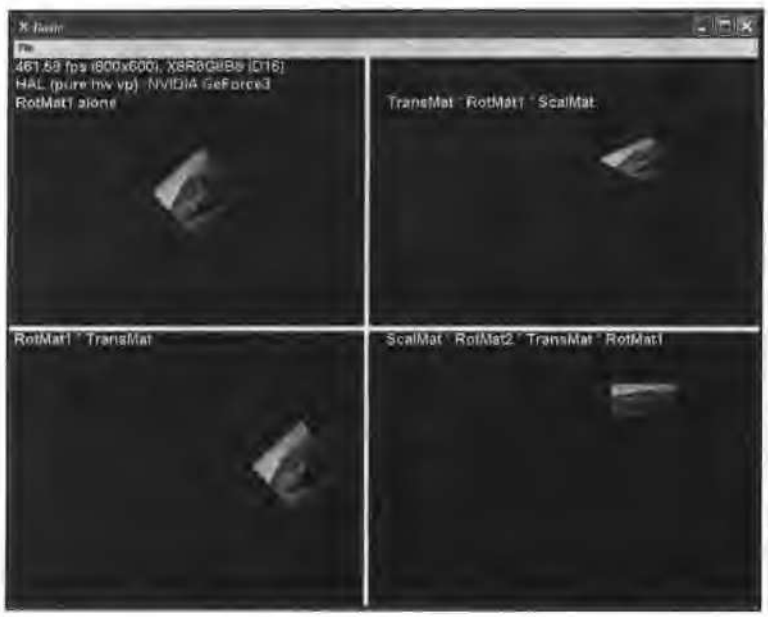


图 6.17 4 个视口

通过 D3DVIEWPORT9 结构来定义一个视口。

```
typedef struct _D3DVIEWPORT9 {
    DWORD X;
```

```
    DWORD Y;
    DWORD Width;
    DWORD Height;
    float MinZ;
    float MaxZ;
} D3DVIEWPORT9;
```

图 6.18 视口矩形

X、Y、Width、Height 值对应于视口矩形（见图 6.18）。

这个结构定义了视口在窗口或屏幕中的矩形位置，另外还有两个成员用于指定渲染哪个深度范围内的场景。Direct3D 规定 x 值的范围从-1.0f 到 1.0f，y 值的范围从 1.0f 到-1.0f。这两个值一起表示了哪个深度范围内的场景将被渲染出来，或者说不会被剪裁掉（这项改变是在 DirectX 8 中引入的；而在 DirectX 7 中，这些成员被用于剪裁）。通常将 MinZ 和 MaxZ 设为 0.0f 和 1.0f，这样可以渲染深度缓冲器中整个深度范围内的值。将它们设为一些其他值则可以渲染出某些特殊效果的画面，比如，将这两个值都设为 0.0f，则强制系统在前景中进行渲染。在设置一个新视口前，可以先调用 GetViewport()保存旧视口，这样在渲染完成后使可重置视口。

HRESULT GetViewport(D3DVIEWPORT9 *pViewport);

有了一个旧视口的副本后，就可以用 SetViewport()函数来设置一个新视口了，该函数与 GetViewport()的语法基本相同。

```
HRESULT SetViewport(CONST D3DVIEWPORT9 *pViewport);
```

这个函数定义了窗口中的一个区域，随后绘图操作的结果都将显示在该矩形区域内。如果一个新视口定义在设备边界之外，那么 SetViewport()则会失败。请注意，视口矩形的高宽比应与投影矩阵所定义的高宽比相匹配，否则物体看上去会像是被挤压过一样。

1steps2 示例的源代码介绍了如何使用 4 个视口。这些视口在应用程序类中声明。

```
D3DVIEWPORT9 m_RViewport;
D3DVIEWPORT9 m_SRTViewport;
D3DVIEWPORT9 m_TRViewport;
D3DVIEWPORT9 m_RTRSViewport;
```

第 1 个视口表示的视图显示了一个经由旋转矩阵变换的四边形，其名字中的大写字母 R 即为此意。第 2 个视口表示的视图显示了一个经由“缩放—旋转—平移”矩阵变换的四边形，其名字中的大写字母 SRT 即为此意。第 3 个视口表示的视图显示了一个经由“平移—旋转”矩阵变换的四边形，其名字中的大写字母 TR 即为此意。第 4 个视口表示的视图显示了一个经由“旋转—平移—旋转—缩放”矩阵变换的四边形，其名字中的大写字母 RTRS 即为此意。4 个视口的大小都是在 RestoreDeviceObjects()中设定的，这表示每次改变窗口大小时都将重新计算这些视口的大小。

```
// Setup the sub-viewports
D3DVIEWPORT9 MainViewport;
```

```
m_pd3dDevice->GetViewport(&MainViewport);

// Each viewport fills a quarter of the window
m_RViewport.Width =
m_SRTViewport.Width =
m_TRViewport.Width =
m_RTRSViewport.Width = MainViewport.Width / 2;
m_RViewport.Height =
m_SRTViewport.Height =
m_TRViewport.Height =
m_RTRSViewport.Height = MainViewport.Height / 2;

m_RViewport.Y = m_SRTViewport.Y = 0;
m_RViewport.X = m_TRViewport.X = 0;

m_TRViewport.Y = m_RTRSViewport.Y = MainViewport.Height / 2;
m_SRTViewport.X = m_RTRSViewport.X = MainViewport.Width / 2;

// Set the full Z range for each viewport
m_RViewport.MinZ =
m_SRTViewport.MinZ =
m_TRViewport.MinZ =
m_RTRSViewport.MinZ = 0.0f;
m_RViewport.MaxZ =
m_SRTViewport.MaxZ =
m_TRViewport.MaxZ =
m_RTRSViewport.MaxZ = 1.0f;
```

在这段代码中，首先获取当前视口的信息。该视口应保存着窗口客户区的大小。然后，将客户区矩形分成 4 个相同大小的矩形，4 个视口就由这 4 个矩形的值来给出。请注意，必须使用大写的变量名来访问.X 和.Y 值。

在 Render()中每设置完一个视口后，就调用 DrawPrimitiveIndex()函数将四边形绘制到该视口中。

```
// Draw the rotated data
m_pd3dDevice->SetViewport(&m_RViewport);
m_matWorld = RotationMatrix1;
m_pd3dDevice->SetTransform(D3DTS_WORLD, &m_matWorld);
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                    0,
                                    0,
                                    4, // number of vertices
                                    0,
                                    2); // number of primitives
```

通过第 5 章的介绍，你应该已对 DrawIndexedPrimitive()函数比较熟悉了。在绘制完各个四边形后，接着绘制所有文字，显示显卡的信息和不同视口的标注信息。这些信息必须显示在最初那个横跨整个窗口客户区的视口中，所以需要再次调用 SetViewport()来设置该视口。

```
// Set the viewport back to the full window. This ensures
// that any other processing (such as Clear or the DrawText functions) takes place
// over the full window.
```

```
m_pd3dDevice->SetViewport(&MainViewport);

// Output statistics
m_pFont->DrawText( 2, 0, D3DCOLOR_ARGB(255,255,255,0), m_strFrameStats );
m_pFont->DrawText( 2, 20, D3DCOLOR_ARGB(255,255,255,0), m_strDeviceStats );
m_pFont->DrawText( 2, 40, D3DCOLOR_ARGB(255,255,255,255), _T("RotMat1 alone") );
m_pFont->DrawText( m_SRTViewport.X, m_SRTViewport.Y, D3DCOLOR_ARGB(255,255,255,255),
                   _T("TransMat * RotMat1 * ScalMat") );
m_pFont->DrawText( m_TRViewport.X, m_TRViewport.Y, D3DCOLOR_ARGB(255,255,255,255),
                   _T("RotMat1 * TransMat") );
m_pFont->DrawText( m_RTRSViewport.X, m_RTRSViewport.Y, D3DCOLOR_ARGB(255,255,255,255),
                   _T("ScalMat * RotMat2 * TransMat * RotMat1") );
```

以上代码中，我使用视口尺寸将文字定位到各视口的边缘位置上。由于视口不是 COM 对象，所以不需要在 DeleteDeviceObjects()或 FinalCleanup()函数中释放它。

用几句话来概括一下视口的用法：通过 GetViewport()可获取视口数据，以视口的大小和提供给深度缓冲器的 MinX、MinY 值来填充一个视口结构。应在 DrawPrimitive*()命令之前用 SetViewport()来设置视口。在完成绘制后，应恢复原来的视口，以便在一次渲染中清理整个渲染目标（render target），以及通过 Direct3D 框架提供的字体类来绘制文字。

最后让我们来看一个非常重要的特征。该特征遍及于本章所有的示例，但至今尚未提及，那就是“深度缓冲”。

6.1.5 深度缓冲（Depth Buffering）

覆绘（overdrawing）是所有游戏开发者在绘制 3D 场景时都会碰到的一个主要问题。考虑这样一个 3D 场景，你正通过一扇窗口向一间房间内张望。通过窗口你可以看到房间里的某些墙壁和物体，而另一些是看不到的。

渲染这种场景最有效的方法就是仅渲染那些可以被观察者看到的像素。如果渲染了那些看不到像素（比如，房间里那些通过窗口无法看到的物体），那么这种多余的操作称为“覆绘”。这是因为窗口嵌在墙壁中，当绘制墙壁的一个像素时会覆盖房内物体的像素。也就是说，离照相机较近的像素会阻挡离照相机较远的像素。

深度复杂度（depth complexity）是对一场景中覆绘数量的一种度量值。它表示在所有渲染的像素中最终可见像素的比例。例如，如果一个场景的深度复杂度为 3，则表示被渲染的像素数量是屏幕上实际可见像素数量的 3 倍。这也就意味着，在某种特定的帧速率情况下，你需要以 3 倍于无覆绘的填充率来显示一个场景。另外，根据场景内容复杂程度的不同，深度复杂度可能从 1 到超过 10 不等，但通常是 2 或 3 左右。覆绘是使 3D 游戏变得低效率的一个主要原因。

Direct3D 支持一种所谓“深度缓冲器”的东西，它在防止覆绘中扮演了重要的角色。这个缓冲器储存着显示器中每个像素的深度信息。在显示虚拟世界之前，应当清除深度缓冲器中

的每个像素，将它们设置为最远可能的深度值。在光栅化时，深度缓冲算法会获取当前多边形所涉及的每个像素的深度。如果一个像素比先前储存在深度缓冲器中的像素更接近于照相机，则较近的像素将被显示出来，并且这个新的深度值也将覆盖深度缓冲器中原先的内容。每次绘制多边形的像素时都将执行一遍这个过程。

深度缓冲器是由Direct3D框架创建的，只需将枚举类的一个成员变量指定为TRUE即可。枚举类用于选择合适的设备解决方案，比如是否需要深度/模板（stencil）缓冲器的支持。

```
m_d3dEnumeration.AppUsesDepthBuffer = TRUE;
```

为了在每一帧的渲染中都使用深度缓冲器，就必须在 Render()函数的每一帧渲染前先清理深度缓冲器。这可通过以下代码来完成，以下代码同时还清理了渲染目标。

```
// Clear the render target | z-buffer
m_pd3dDevice->Clear( 0L, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                     D3DCOLOR_XRGB(0,0,128), 1.0f, 0L );
```

根据文档叙述，Clear()方法用于清理视口。而我的理解是，它用于清理由视口所界定的渲染目标。因为视口就是直接对应于渲染目标的。这个函数还可以清理模板缓冲器。它的声明如下：

```
HRESULT Clear(
    DWORD Count,
    const D3DRECT *pRects,
    DWORD Flags,
    D3DCOLOR Color,
    float Z,
    DWORD Stencil
);
```

这个方法在 pRects 中接受一个或多个矩形。矩形的数量储存在 Count 中。Flags 变量表示所需清理的存储器类型，可以清理 Z 缓冲器、渲染目标和模板缓冲器。在 Z 变量中指定准备设置的 z 值，在 Stencil 变量中指定准备设置的模板值。

这就是启用深度缓冲器所需的全部操作。不幸的是，有时在一个游戏引擎中会存在着一些深度缓冲器的视觉错误。在理想情况下，深度缓冲器与颜色缓冲器具有相同的大小和形状，颜色缓冲器[1]即存储了稍后将要绘制到屏幕上的内容。就当今的硬件而言，深度缓冲器的每个像素通常都是 16 位或 24 位的。而参考光栅器甚至可以支持 32 位的深度缓冲器格式。在一个透视观察环境中，世界空间的距离值与深度缓冲器中的值（称为"z 值"）不是成比例对应的（见图 6.19）。也就是说任意两个相邻的深度值之间的距离并不是相等的，其中较小的深度值间的距离比较相近些。

译注[1]：也就是前述的"渲染目标"、"后备缓冲器"，在本书很多处都用到了这几种称谓，在本书上下文中，它们多是指同一样东西。

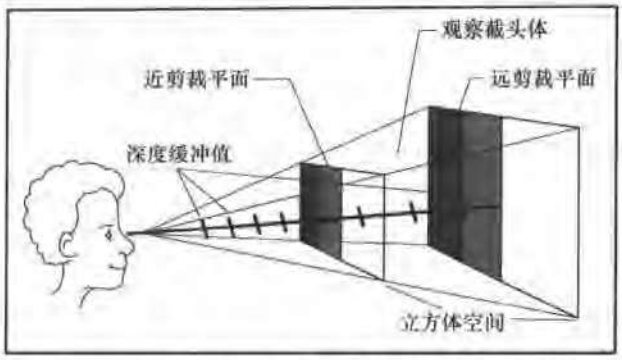


图 6.19　深度缓冲器中的值

如果丢失了深度缓冲的精度，那么就有极有可能会发生渲染错误。而深度精度取决于深度缓冲器的位数。比如，一个深度缓冲器若只有 16 位的深度信息，那么就很有可能会发生精度丢失的问题。不过，这些问题通常只会发生在 Z 缓冲器中。很显然，解决该问题的方法便是选择更高精度的深度缓冲器格式，比如 24 位。[1]

Direct3D 还提供了另一种称为“W 缓冲器”的解决方法。W 缓冲器减少了 Z 缓冲器在处理远距离物体时所遇到的问题，并且无论对于多远或多近的物体都具有稳定的执行性能。以前的 DirectX 7.0 SDK 提供了一个非常具有启发性的示例来说明 Z 缓冲器与 W 缓冲器在实际应用中的区别。它显示了在 16 位颜色模式中深度缓冲器丢失精度时所引发的渲染错误。如果你想要试一下那个示例程序，请先选择一个 16 位颜色模式，这样才能看到这种错误（见图 6.20）。

图 6.20　深度缓冲器的视觉错误

译注[1]：在经过变换和剪裁后，场景中所有物体的 z 值都被固定在一定范围内，这样便可使用固定精度的 Z 缓冲器。由于大规模复杂场景中各对象的尺寸可能相差有 6 个数量级以上，即超过 100 万倍，所以深度信息必须具有比任意两个对象的 z 值更高的精度。此外，在采用透视变换时，由于 z 值被压缩也会引起精度问题。一般采用 24 位存储深度信息就足够了，可以表示 2^{24} 种不同的深度状态；在指定使用 24 位的 Z 缓冲器前应先检查显卡是否支持这种格式。

当深度缓冲器正常工作时，你可以看到绿色正方形上的一小段文字。在绿色正方形后面是一个蓝色正方形。当 Z 缓冲器处理远距离数据时，便会丢失精度，此时蓝色正方形会覆盖这段文字。

通过 SetRenderState()函数可以切换为使用 W 缓冲器。

```
m_pd3dDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_USEW ); // w-buffer
```

使用以下代码可以检查是否支持 W 缓冲器：

```
if( d3dCaps.RasterCaps & D3DPRASTERCAPS_WBUFFER)
    m_bDeviceSupportsW = TRUE;
else
    m_bDeviceSupportsW = FALSE;
```

一个基于 W 算法的深度缓冲器与 Z 缓冲算法相比，它可以更均匀地分布于近剪裁面与远剪裁面之间。这样做的主要好处就是解决了近/远剪裁面间的距离比例问题。这便使得应用程序在仍能获得相对精确的深度缓冲的前提下，支持最大的深度范围。然而，基于 W 算法的深度缓冲器也并不是完美的，有时候对于较近的物体会显露出表面被隐藏的视觉错误。W 缓冲器方法的另一个缺点就它不如 Z 缓冲那样广泛地为硬件所支持。所以为了保险起见，还是请使用 Z 缓冲器。

6.2 附加资源

这里有篇简单有效的 COM 指南：http://www.gamedev.net/reference/articles/ article1299.asp。

我在这里找到过几个不错的关于照相机朝向问题的页面：http://crystal.sourceforge.net。

3 篇关于四元数极好的文章：

- Shoemake Ken 的“Euler Angle Conversion”，在 Paul S. Heckbert 编辑的图书 *Graphics Gems* 中 222～229 页（Toronto:Academic Press Professional）；
- Void Sobeit 的“Quaternion Powers”，位于 http://www.gamedev.net/reference/articles/article1095.asp；
- Downs Laura 的“Using Quaternions to Represent Rotation”，位于 http.cs.berkeley.edu/~laura/cs184/quat/quaternion.html。

这里有一些不错的关于数学题材的页面：

- http://www.sisweb.com/math/tables.htm；
- http://www.swin.edu.au/astronomy/pbourke。

一个中级的 DirectX/OpenGL/Glide/Genesis3D 论坛，位于 Witchlord 网站中：http://www.angelcode.com，或许也会对你有所帮助。

如果你还不知道这个对于游戏程序员而言的最佳网站，那请访问

http://www.gamasutra.com。

如果你想更进一步地看些读物，我推荐 *Real-Time Rendering Tricks and Techniques in DirectX*（Premier Press, 2002）。

6.3　小结

本章接连几个示例程序向你介绍了如何使用世界变换、观察变换和投影变换。还向你介绍了视口与深度缓冲器。在下一章中，你将会学习到纹理映射和纹理过滤。准备好了吗？让我们继续前进吧！

6.4　第一部分提问

问：插件式软件设备（pluggable software device）的用途是什么？

答：如果在 HAL 中不支持一项特征，则可以让插件式设备取得控制并模拟硬件。DirectX 9 SDK 不提供这种设备。

问：使用参考光栅器（reference rasterizer）的目的是什么？

答：为了测试那些你的显卡不支持的特征。

问：对于游戏编程员来说，最重要的 COM 特征是什么？

答：
- COM 接口永远不会改变；
- 独立于语言；
- 只能访问方法，而不能访问数据。

问：什么是 v-table？

答：一个虚函数表，其中引用了在一个 COM 对象中实现的各个方法。你必须通过 v-table 来访问 COM 的方法。

问：如何在 3D 空间中定义一个顶点的位置？

答：指定一个从原点到该顶点所在位置的向量。

问：如何在 3D 空间中定义一个物体的朝向？

答：使用 Look、Up、Right 向量。

问：为什么必须要以顺时针次序来指定三角面的顶点？

答：Direct3D 为了获得更快的渲染速度通常不绘制物体的背面。默认情况下，它只绘制物体中以顺时针顶点次序定义的那些面。这称为“背面剔除（back-face culling）”。因此，建议以正确的次序来定义顶点。

问：如何将纹理映射到一个面上的？

答：Direct3D 对一张纹理中所有的图素（texel）使用一段统一的地址范围。其左上角为

(0.0f, 0.0f)，右下角为(1.0f, 1.0f)。

问：什么是通用文件（common file）？

答：它们封装了开始编写一个 Direct3D 游戏所需的大部分基础功能。并且这些代码都经历过仔细地检验和测试。

问：OneTimeSceneInit()/InitDeviceObjects()方法的用途是什么？

答：OneTimeSceneInit()用于完成加载一个场景所需的所有几何数据，它只会被调用一次。InitDeviceObjects()初始化所有对应于每个设备的对象，比如将纹理数据加载到一个设备表面、设置变换矩阵、构建顶点缓冲器、设置渲染状态等。它是在应用程序启动时或设备变更时被调用的。

问：DeleteDeviceObjects()与 FinalCleanUp()之间的有什么区别？

答：DeleteDeviceObjects()是在应用程序退出时或设备变更时被调用。FinalCleanup()是框架退出前在应用程序中最后调用的一个函数。可将它用于释放已分配的内存（如纹理）、删除文件对象等。

问：D3DUtil_CreateTexture()可以加载什么纹理？

答：*.bmp、*.jpg、*.png、*.ppm、*.dds、*.tga 格式的位图文件。

问：什么是视口？

答：为了指示 Direct3D 如何剪裁截头体（frustum），你需要给出视口的尺寸。该尺寸表示了如何将透视投影的观察截头体剪裁为立方体形状。

问：BeginScene()/EndScene 这对函数有什么用？

答：BeginScene()促使系统检查其内部数据结构、检查渲染表面的可用性和有效性，并设立内部标志以标识当前已处于一个场景的渲染过程。EndScene()清除表示处于场景渲染过程的内部标志，刷新高速缓存数据，并确认渲染表面是否正常。

问：设置一张纹理意味着什么？

答：一次 SetTexture()调用就是指示 Direct3D 设备将该纹理应用于后续所有的图元，直到设置了另一个纹理或关闭了该纹理阶段（texture stage）为止。

问：什么是三角条形？

答：一系列相连的三角形。

问：顶点标志的作用是什么？

答：它告诉了 Direct3D 关于储存顶点信息的顶点结构的情况。某一种特定顶点格式标志的存在与否都传递给系统相应的信息，即你用到了哪些顶点成分字段，它存储在内存中，以及你没有用到哪些成分。仅使用必要的顶点成分，可使你的应用程序节省内存开销，并且有利于最小化渲染 3D 场景所需的处理带宽。

问：必须要使用 Direct3D 变换和光照流水线吗？

答：不是必须的，你可以自主选择是否跳过它。如果想跳过它，只需使用适当的顶点标志（_RHW）和适当的顶点结构即可。

问：世界变换、观察变换和投影变换的用途是什么？

答：世界变换将一个物体从模型空间变换到世界空间，世界空间是对一个3D场景的绝对引用框架（absolute frame of reference）。观察变换将3D场景从世界空间变换到照相机空间，该变换将照相机放置到世界空间的原点，并使其朝着z轴的正方向。投影变换好比是控制照相机的内部参数，比如为照相机选择一个合适的镜头参数。

问：在游戏编程中，使用矩阵有什么好处？

答：一个使用矩阵的变换引擎的执行性能是可预期且稳定的。这种特性使得游戏开发者可在执行性能与质量的取舍考量中作出明智的选择。

问：D3DXMatrixLookAtLH()提供的默认照相机控制技术有什么问题？

答：这个函数是为简单应用而设计的。例如，它对于头罩式跟踪显示（head-mounted tracking display）或跟随式照相机来说是很棒的，但它无法帮助你构建一个用于太空射击游戏或飞行模拟器的照相机。

问：请简要描述一下如何使用四元素来旋转照相机。

答：通过偏航、俯仰和横滚（即欧拉角的三个参数）来构建一个四元素，然后把它转换成一个矩阵，最后求该矩阵的逆矩阵。

问：为什么需要使用深度缓冲器？

答：Z缓冲器或W缓冲器用于减少覆绘所带来的问题。这些缓冲器保存着场景中每个像素的深度信息。如果一个像素比先前储存在深度缓冲器中的像素更接近于照相机，则将显示最近的那个像素，并且该像素的深度值也将被存入深度缓冲器覆盖原先的深度值。

问：使用起来最快的图元是什么？

答：这取决于你使用的图元数据。在大多数情况下，索引图元类型，诸如索引化的三角形列表或索引化的三角条形，应该是最快的图元类型。但如果物体中只有1个或2个顶点会共享一个索引，则应考虑使用非索引图元。在特定的情况下，找出最快图元的最佳方法就是评测使用不同图元时的执行性能。

问：计算两个向量的叉积可用来干什么？

答：叉积的结果是形成第三个向量，该向量垂直于由前两个向量所构成的平面。它可用于确定多边形所面对的方向，还可用于为一个由两个相交向量所确定的平面导出平面方程式。

的功能只适用于设置多纹理单元的状态。由于许多已安装的显卡尚不支持更高级的像素 shader，故本章及第 8 章"使用多重纹理"还将介绍多纹理单元的使用。而像素 shader 可完成与 Direct3D 流水线中的多纹理单元相同的任务，并且在将来它很有可能会取代多纹理单元的功能。

图 7.1　示例程序

表 7.1　　纹理映射选项

选项	含义	按键
Dark Mapping	黑暗映射	1
Animated Dark Mapping	黑色映射动画	2
Blending with Material Diffuse Color	与材质漫反射颜色相混合	3
Darkmap with Material Diffuse Color	混合了材质漫反射颜色的黑暗贴图	4
Glow Mapping	发光映射	5
Detail Mapping	细节映射	6
Modulate Alpha	调制 alpha	7
Blending with Frame Buffer	通过帧缓冲器混合	8
Trilinear Filtering	三次线性过滤	F5
Anisotropic Filtering	各向异性过滤	F6
Clamp Addressing Mode	夹持寻址模式	F9
Mirror Once	一次镜像	F11
Border Color	边框颜色	F12
Exit Menu	退出菜单	F1

SetSamplerState()用于设置采样器状态。其中还包括一个设置镶嵌器单元（tessellator unit；这是显卡中的一个功能单元，用于处理高阶（higher-order）表面，诸如 Bezier patches、N-patches 等[1]）的状态，用于采样置换贴图（displacement map）。这些状态现在都改为以 D3DSAMP_作为前缀，这样便可以在从 DirectX 8.x 移植过来时进行编译时刻的错误检测。这些状态包括：

- D3DSAMP_ADDRESSU、D3DSAMP_ADDRESSV、D3DSAMP_ADDRESSW
- D3DSAMP_BORDERCOLOR
- D3DSAMP_MAGFILTER、D3DSAMP_MINFILTER、D3DSAMP_MIPFILTER
- D3DSAMP_MIPMAPLODBIAS
- D3DSAMP_MAXMIPLEVEL
- D3DSAMP_MAXANISOTROPY

SetSamplerState()可用于指定纹理的过滤方式、贴夹方式（tiling/clamping）、MIPLOD 方式等等。有两项与固定功能的多纹理单元相关的设备能力参数：MaxTextureBlendStages 和 MaxSimultaneousTextures，分别表示允许应用程序使用的纹理混合阶段（texture-blending stage）和纹理的最大数量。比如在 GeForce3 中，可以使用最多 8 个纹理混合阶段，以及可同时设置最多 4 张纹理；在 RADEON 9700 PRO 中，也可以使用最多 8 个纹理混合阶段，但可同时设置最多 8 张纹理。MaxTextureBlendStages 表示驱动程序认为存在几个 D3DTSS_或 D3DSAMP_混合阶段，而 MaxSimultaneousTexture 表示你可以在同一时刻使用几张不同的纹理。这两者可不是一回事，使用额外的混合阶段而不使用额外的纹理是允许的。例如，在 DirectX 7 硬件上实现 Dot3 product 光照时就使用了 3 个混合阶段和 2 张纹理[2]。

SetSamplerState()的语法是：

```
HRESULT SetSamplerState(
    DWORD Sampler,
    D3DSAMPLERSTATETYPE Type,
    DWORD Value);
```

第一个参数是纹理阶段号，这对应于你在 SetTexture()方法中所使用的数字。第二个参数是以 D3DSAMP_为前缀的采样器状态类型之一。最后一个参数是一个对应于采样器状态的值。例如，以下代码为纹理阶段 0 打开各向异性过滤（我稍后将在“纹理过滤和抗锯齿”一节中告知更多关于这个主题的细节）。

```
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MINFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MIPFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAXANISOTROPY, 8);
```

SetTextureStageState()函数为使用固定功能的多纹理单元提供了附加的操作功能。但它不

译注[1]：详见第 1 章“Direct3D/DirectX 的历史”中“Direct3DX 实用程序库”一节的译注。

译注[2]：Dot3 product 可大致理解为对两个 RGB 颜色参数进行三元数的点积运算，它可作为一种实现近似于凹凸映射效果的主要方法，详见 DirectX SDK 中一个叫“DotProduct3”的示例程序和相关文档。

能与像素 shader 一起使用。这个函数的语法是：

```
HRESULT SetTextureStageState(
    DWORD Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    DWORD Value);
```

正如你所见，它的语法与 SetSamplerState()函数的语法非常相似。第一个参数是纹理阶段号。这对应于你在 SetTexture()方法中所使用的数字。第二个参数是以 D3DTSS_为前缀的阶段状态类型之一。最后一个参数是一个对应于阶段状态的值。例如，以下代码将纹理颜色不作任何修改地映射到物体上。

```
m_pd3dDevice->SetTexture( 0, m_pBackgroundTexture );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
```

SetTextureStageState()方法对于处理以下内容很有用：

- 不同的纹理坐标；
- 颜色操作；
- alpha 操作；
- 凹凸映射/环境映射（bump mapping/environment mapping）操作。

请注意，这些操作只对 DirectX 9 的固定功能的多纹理单元有效，不能将它们与像素 shader 连用。接下去，让我们从研究纹理坐标入手。

7.2 使用纹理坐标

一张纹理采用两个坐标(u, v)来将它的各个颜色值关联到一个多边形区域中。通常的做法就是将每对坐标(u=列、v=行)所对应的颜色值储存在一个两维数组中，每个颜色值都有一个惟一的地址。该地址称为“纹理坐标”，用于在纹理自身的坐标空间中表示一个位置。该数组中的一个元素通常称为一个“图素（texel，texture element 纹理元素的缩写）”。

为了通过指定纹理坐标来访问纹理中的每个图素，Direct3D 需要对纹理中的所有图素都使用一个统一的地址范围。Direct3D 采用了一种一般化的编址方案，其中纹理地址由[0.0, 1.0]区间内的坐标值组成。这样在处理纹理坐标时，便无须关心纹理贴图的实际尺寸了。在将一张砖墙纹理映射到一个矩形多边形上时，只要使用纹理坐标(0.0f, 0,0f)、(1.0f, 0.0f)、(1.0f, 1.0f)、(0.0f, 1.0f)即可，这将致使 Direct3D 缩放该纹理以覆盖到整个多边形区域。或者，也可以仅将纹理的左半部分或右半部分应用到多边形上，如果选择应用纹理的左半边，则纹理坐标可分别为(0.0f, 0.0f)、(0.5f, 0.0f)、(0.5f, 1.0f)、(0.0f, 1.0f)。不过这样做时，你还须留意一下纹理和多边形它们各自的纵横比（高度与宽度的比值）是否相近。另外，也可以通过指定纹理坐标定义出一张纹理中的某些区域，例如(0.1f, 0.1)、(0.5f, 0.4f)、(0.3f, 0.8f)、(0.2f, 0.6f)。在现

```
...
VERTEX Vertices[] =
{
    // x, y, z, rhw, tu, tv, tu2, tv2
    { 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.5f,},
    ...
};
```

你可以通过这种方式来使用最多 8 组纹理坐标（D3DFVF_TEX1～D3DFVF_TEX8）。D3DTSS_TEXCOORDINDEX 则用于选用其中某组特定的纹理坐标。

```
m_pd3dDevice->SetTexture(0, m_pWallTexture);
m_pd3dDevice->SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, 0);
...
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 1);
...
```

第一张纹理使用了顶点的第一组纹理坐标，第二张纹理使用了第二组纹理坐标。

也可以重复使用一组纹理坐标。在上例中，可以对两个纹理阶段选用相同的纹理坐标。

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0);
...
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 0);
...
```

第二个纹理阶段使用了与第一个纹理阶段相同的纹理坐标。因此，只须为每个顶点在顶点缓冲器中存储一组纹理坐标即可。这种重用纹理坐标的方法可以减少 Direct3D 引擎发往显卡的数据量。

7.3 使用纹理寻址模式（texture-addressing mode）

通过将顶点的纹理坐标设置为 0.0f～1.0f 之间的值，以此来选择一张纹理贴图的图素是一套简单易懂的规则。但是如果给出的纹理坐标超出了这个范围又会如何呢？在 0.0～1.0 范围以外的值将如何改变纹理的外观取决于所选择的纹理寻址模式。有几种不同的寻址模式：包装（wrap）、镜像（mirror），夹持（clamp）、边框颜色（border color）、一次镜像（mirroronce）。应仔细地检查显卡的能力以判断显卡支持哪些寻址模式。通常，显卡会支持若干种寻址模式，但不太会支持所有的寻址模式。在本章附带的示例程序中，通过背景纹理展示了不同的纹理寻址模式。

7.3.1 包装纹理寻址模式

包装纹理寻址模式在纹理坐标的每个整数相接处重复纹理，这是 Direct3D 所使用的默认

模式。尝试将以下这段源代码插入示例程序中：

```
VERTEX Vertices[] =
{
    // x, y, z, rhw, tu, tv
    { 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, 0.0f, 3.0f, },
    { 0.0f, 0.0f, 0.5f, 1.0f, 0.0f, 0.0f, },
    { (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f,
          3.0f, 3.0f, },
    { (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 3.0f, 0.0f, },
};
```

另外，还需使用以下代码片断来打开包装寻址模式：

```
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP);
```

背景纹理将在 u 和 v 方向上分别被应用 3 次，看上去应如图 7.3 所示。

图 7.3　包装纹理寻址模式

如果选择使用以下的纹理坐标，则纹理将以另一种方式贴上去（见图 7.4）。

```
VERTEX Vertices[] =
{
    // x, y, z, rhw, tu, tv
    { 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, -1.5f, 3.0f, },
    { 0.0f, 0.0f, 0.5f, 1.0f, -1.5f, -3.0f, },
    { (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f,
          1.0f, 1.5f, 3.0f, },
    { (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.5f, -3.0f, },
};
```

设置镜像寻址模式需要通过 D3DTADDRESS_MIRROR 参数来调用 SetSamplerState() 方法。

```
m_pd3dDevice->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_MIRROR);
m_pd3dDevice->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_MIRROR);
```

以下结构可用于演示镜像纹理寻址模式：

```
VERTEX Vertices[] =
{
    // x, y, z, rhw, tu, tv
    { 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, -0.5f, 1.5f, },
    { 0.0f, 0.0f, 0.5f, 1.0f, -0.5f, -0.5f, },
    { (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height,
          0.5f, 1.0f, 1.5f,1.5f, },
    { (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.5f,-0.5f, },
};
```

7.3.3　夹持纹理寻址模式

夹持纹理寻址模式只将纹理应用到多边形上一次，然后对于超出部分涂上纹理边缘像素的颜色。这里使用了与前例相同的顶点数据，应用这种寻址模式的结果看上去应如图 7.6 所示。

图 7.6　夹持寻址模式

纹理周边的像素颜色分别延伸到了图元的四周。应按如下所示来调用 SetSamplerState() 方法：

```
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP);
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);
```

如果想要在一个网格上将多张纹理缝合起来，由于默认的包装模式在纹理的边上作了双线性插值，这会导致来自于纹理相对边的视觉错误（artifact），而夹持模式则适用于解决这类问题。不过它也不能完全解决这个问题，要彻底解决，必须修改纹理坐标。

7.3.4 边框颜色纹理寻址模式

当设置了边框颜色纹理寻址模式，则纹理周围会出现一个你所选定颜色的边框。可用以下代码片断来设置边框颜色模式：

```
m_pd3dDevice->SetSamplerState (0, D3DSAMP_ADDRESSU, D3DTADDRESS_BORDER);
m_pd3dDevice->SetSamplerState (0, D3DSAMP_ADDRESSV, D3DTADDRESS_BORDER);
m_pd3dDevice->SetSamplerState (0, D3DSAMP_BORDERCOLOR, 0x00aaaaaa);
```

顶点设置如下：

```
VERTEX Vertices[] =
{
    // x, y, z, rhw, tu, tv
    { 0.0f, (float)m_d3dsdBackBuffer.Height, 0.5f, 1.0f, -0.5f, 1.5f, },
    { 0.0f, 0.0f, 0.5f, 1.0f, -0.5f, -0.5f, },
    { (float)m_d3dsdBackBuffer.Width, (float)m_d3dsdBackBuffer.Height, 0.5f,
          1.0f, 1.5f,1.5f, },
    { (float)m_d3dsdBackBuffer.Width, 0.0f, 0.5f, 1.0f, 1.5f,-0.5f, },
};
```

通过这种纹理模式来使用一张相对较小的纹理，纹理的四周会有一个颜色边框，如此便可以在墙壁上伪造出一张巨大的纹理。

7.3.5 一次镜像纹理寻址模式

DirectX 8.0 引入了一种新的纹理寻址模式，它类似于镜像模式与夹持模式的组合。在一次镜像纹理寻址模式中，纹理在-1.0～1.0 范围内作镜像，而在该范围之外作夹持。这个模式很酷哦。

```
m_pd3dDevice->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_MIRRORONCE);
m_pd3dDevice->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_MIRRORONCE);
```

7.4 纹理包装（Texture Wrapping）

纹理包装与包装纹理寻址模式可不是一回事情。后者用于决定如何映射[0.0, 1.0]边界之外的图素坐标，而前者用于决定如何在纹理坐标之间作插值计算。纹理包装影响着 Direct3D

对应用了纹理的多边形进行光栅化的方式，以及使用纹理坐标（为每个顶点指定的）的方式。Direct3D 通常将纹理视作一个 2D 平面，假设 0.0 与 1.0 重合，Direct3D 将在纹理范围内找出点 a 到点 b 的最短路径，并在其中插入新的图素。如果点 a 表示的(u, v)位置为(0.6, 1.0)、点 b 为(0.1, 0.1)，则插值线将如图 7.7 所示。

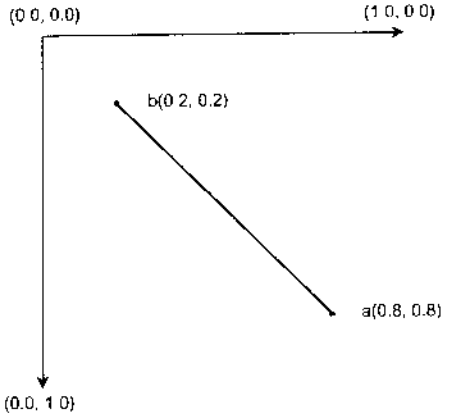


图 7.7　默认的纹理包装

a 与 b 之间的最短路线一般要穿越纹理的中间部分。如果启用了 v 方向上的纹理包装，则光栅器将在 v 方向的最短路径上进行插值处理（见图 7.8）。另外，在 u 方向的纹理包装应如图 7.9 所示。

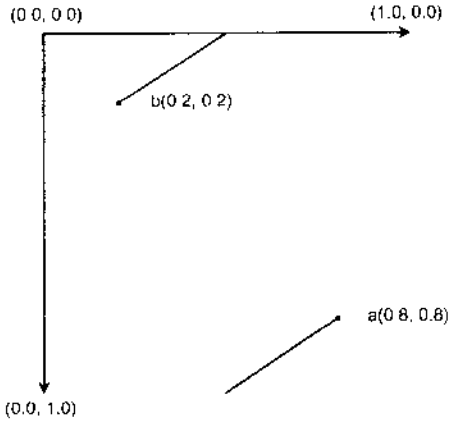


图 7.8　在 v 方向上的纹理包装

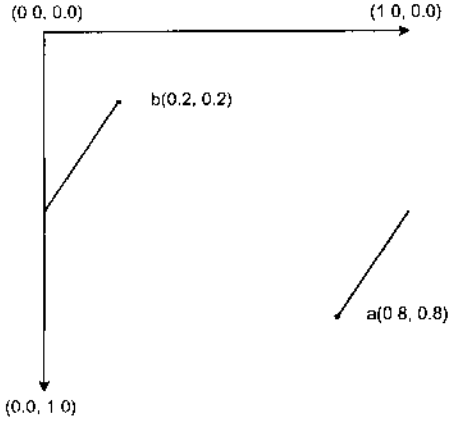


图 7.9　在 u 方向上的纹理包装

要为流水线中的每个纹理阶段启用纹理包装，必须通过 D3DRS_WRAPx 来设置渲染状态，其中 x 为要设置的纹理阶段号（从 0 到 7）。要启用某个方向上的纹理包装，则需包含表示 u 方向的 D3DWRAP_U 或表示 v 方向的 D3DWRAP_V。

你可以想象一下，在某个方向上启用纹理包装时，Direct3D 将纹理视作围绕着一个圆柱

体而包装起来的（见图 7.10）。

可以看到对纹理中最短路径的插值计算不再是穿越纹理中部了，而是穿越了纹理的边框，即 0.0 或 1.0 所在的位置。在 v 方向上的包装也是同样的原理，只是变成了一个横放过来的圆柱。

同时在 u 和 v 两个方向上的包装则稍微有点复杂。在这种情况中，可以将纹理想像成一个环形（torus）。例如，为了将一张纹理包裹到一个球体上时，就需要同时在两个方向上使用包装。可通过调用以下函数来启用第一个纹理阶段在 u 和 v 方向上的纹理包装：

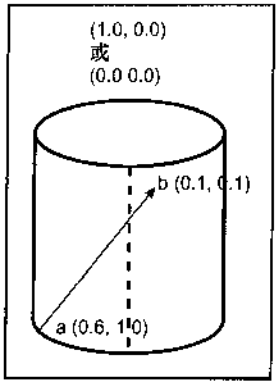


图 7.10　想象一个圆柱体，在纹理的 u 方向上进行纹理包装

```
m_pd3dDevice->SetRenderState( D3DRS_WRAP0, D3DWRAP_U |
    D3DWRAP_V );
```

在启用纹理包装后将使[0.0, 1.0]区间之外的纹理坐标成为无效值，所以纹理包装不能与各种纹理寻址模式共用。

7.5　纹理过滤和抗锯齿（Anti-Aliasing）

试想一个逐渐远离观察者的棋盘。在远离到一定程序后，棋格的投影便会变得比屏幕上的一个像素还要小。如果黑色和白色像素都投射到了屏幕的同一个像素上，显然这个屏幕像素应呈现为某种灰色。而默认情况下纹理映射程序是不会考虑这些条件的，最终绘制到屏幕上的像素颜色不是黑色就是白色，仅取决于绘制的先后次序。这将导致图像看上去很不自然，还会产生一些视觉错误。

另一个例子是所谓的“阶梯效果[1]”。造成这种效果的原因是因为流行的 3D 图形 API 都有一些默认的简单光栅化算法，而这些算法所绘制的边线都是由一些不同的小块像素组成的（见图 7.11）。

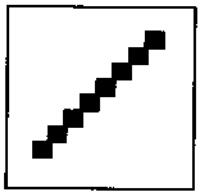


图 7.11　阶梯效果

这两个示例所表现出来的图像失真问题都是由光栅图形的本质所导致。为了克服这些缺点，Direct3D 提供了 3 种纹理过滤方法和所谓的“mipmap”（MIP 是拉丁语“multum in parvo[2]”的缩写，或可解释为“many things in a small place[3]”的缩写。继续读下去会看到一个更完整的解释）。3 种纹理过滤方法是：

- 最近点采样；

译注[1]：或俗称“锯齿”。

译注[2]：小中见大、小型而内容丰富。

译注[3]：许多东西集中在一个较小的区域内。

- 线性纹理过滤；
- 各向异性（anisotropic）纹理过滤。

过滤是指通过给定的 uv 坐标从纹理贴图中获取图素的一种方法。在先前讨论的棋盘例子中，应考虑两种不同的情况：放大（magnification）和缩小（minification）。

举例来说，将一张 64×64 的纹理映射到一个 400×400 像素的多边形时，便会发生放大，这会导致“阶梯效果”。而缩小则发生在相反的情况中，例如将一张 64×64 的纹理映射到一个 10×10 像素的多边形时。缩小会导致“像素抖动”，看上去会让人感觉有点晕眩。

大多数较新的硬件都支持 4 种不同的过滤，以此来减少放大和缩小所造成的视觉错误。这 4 种过滤是：点采样、双线性过滤（或在 Direct3D 术语中称为“线性过滤”）、三线性过滤（或在 Direct3D 术语中称为“线性过滤＋mipmap”）、各向异性过滤。

7.5.1　Mipmap

mipmap 由一系列纹理组成，其中每张纹理是一张分辨率逐渐降低的图像（见图 7.12）。

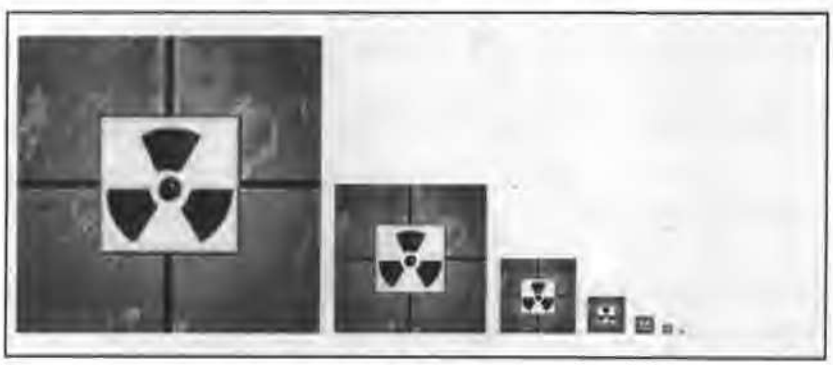


图 7.12　mipmap（一系列分辨率逐渐降低的纹理）

mipmap 序列中每一级的高度和宽度都是前一级高度和宽度的一半。这些层级既可以是正方形也可以是长方形。

如果提供给 CreateTexture()方法正确的参数，就可以自动地产生和使用 mipmap 了。CreateTexture()用到了 D3DXCreateTextureFromFileEx()方法，后者将在附录 D“使用 D3DXCreateTextureFromFileEx()创建纹理”中详细解释。

使用 mipmap 可以确保无论是靠近还是远离纹理，纹理都可保持它们原有的真实性和质量，从而减少了放大和缩小所带来的负面效果。另外还有两个理由促使我们去使用 mipmap。其一，能将一张 256×256 的纹理过滤到一个 2×2 的多边形上，是不是很棒，不过这也将需要较长的时间；其二，mipmap 是一种可有效降低纹理内存流量的方法。

Direct3D 会根据各种使用纹理的情况在 mipmap 中自动地挑选出适合的 mip 层级，它将挑选出一个图素与像素的比值最接近于 1 的 mip 层级。例如，如果一个多边形在屏幕上大约

是 128×128 像素，则 Direct3D 会使用 128×128 的 mip 层级；如果多边形在屏幕上是 95×95 像素，则 Direct3D 会选择 64×64 的 mip 层级。

7.5.2 最近点采样

最近点采样是一种最简单的过滤器。它将纹理坐标对齐到最接近的整数，再将那个位于整数坐标上的纹理像素作为最终的颜色。这对于处理那些与屏幕上多边形图像大小差不多的纹理来说是一种快速、高效的方法。PlayStation 1 控制台以及第一代的 3D 游戏，诸如 Descent 和 Quake，都采用这种过滤方式。

Direct3D 会将区间[0.0, 1.0]内的浮点纹理坐标映射到区间为[-0.5, n-0.5]的整数纹理空间中，其中 n 是一张特定尺寸的纹理上图素的个数。由此得到的结果再四舍五入为一个最接近的整数，不过这可能会在处理图素边界时发生一些错误。系统可能会选择这个或那个采样图素，例如，在计算边界像素时，计算结果就有可能会从原来的一个图素，到第二次计算时突然变为另一个图素。如此反复，便会造成令人不快的视觉错误，尤其是在将一张较小的纹理映射到一个较大的多边形上时（放大），这种视觉错误发生得就更为频繁。以前 Quake 就存在这样一些点采样的视觉错误，Quake 是根据距离来选择一个 mipmap，并对选择的结果应用点采样。

由于现今多数显卡都已为线性过滤作了优化，因此应用程序应尽可能避免使用最近点采样。

7.5.3 线性纹理过滤

Direct3D 中的线性纹理过滤使用的是双线性纹理过滤，它计算相对于采样点距离最近的 4 个图素的平均值（见图 7.13）。好，现在我来一步步地介绍这个过程。双线性过滤首先计算出一个图素地址，该地址通常不是一个整数。接着，求得该地址最接近的整数地址，并找到这个整数地址所在的纹理像素。然后，计算出最接近于这个采样点的上、下、左、右图素的加权平均值。

图 7.13 双线性过滤（http://www.reactorcritical.com 授权）

可通过以下代码来开启线性纹理过滤：

```
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR );
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR );
```

也可以将 D3DTSS_MIPFILTER 作为 SetSamplerState 的第二个参数，这表示在两个 mipmap 级之间也使用纹理过滤器，默认值为 D3DTEXF_NONE。如果给出了 D3DTEXF_LINEAR，即表示使用三线性纹理过滤，或称“mipmap 过滤”。

```
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR );
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR );
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR );
```

三线性纹理过滤看起来比双线性过滤好很多，尤其是在一个场景中穿行时（见图 7.14）。它是如何运作的呢？对于每个像素，三线性过滤会先选择两张最接近的 mipmap，将它们双线性过滤为两张理想大小的 mipmap，然后根据理想的 mip 级组合这两张过滤后的 mipmap 中的对应像素。例如，理想的 mip 级是 3.4，那么组合的过程就是将第 3 张 mipmap 中 0.6 倍（1 减 0.4）的对应像素加上第 4 张 mipmap 中 0.4 倍的对应像素。

在老式显卡中，需要检查能力标志位以判断显卡是否支持三线性过滤。而有些硬件可能支持在一个纹理阶段中作三线性过滤。

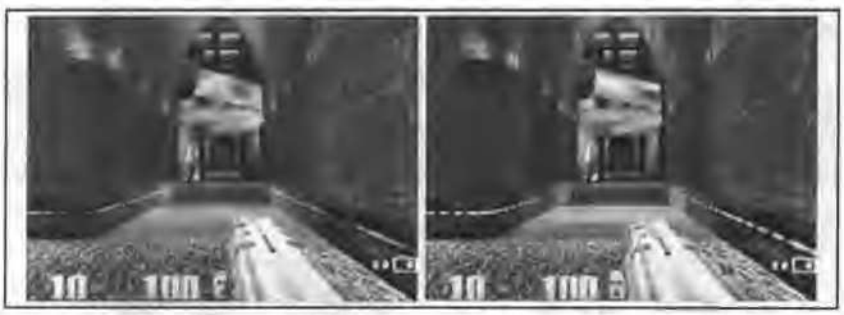


图 7.14　三线性/双线性过滤（http://www.reactorcritical.com 授权）

7.5.4　各向异性过滤

双线性过滤和三线性过滤的一个缺点就是它们都使用正方形采样区域来采样图素。如果纹理是侧着朝向你的话，则会发生一种称为“各向异性”的失真效果。

Direct3D 根据一个屏幕像素的伸张度（elongation：长度除以宽度）来测量一个纹理像素的各向异性，该屏幕像素将被反向映射回纹理空间中。当一个屏幕像素在反向映射后被伸张了，则各向异性过滤将采样更多的图素以减少由标准线性过滤所造成的模糊。就各向异性过滤来说，来自于纹理贴图的图素数量应在 16～32 个之间，若使用更多的图素则会要求一个相当大的存储器带宽，而在传统的渲染系统中几乎不可能有这样的配置，除非使用一个昂贵的

存储器架构。基于片面（tile-based）的渲染（在 PowerVR 卡中使用）可节省大量的带宽，所以可以通过它来实现各向异性过滤。

各向异性渲染对于那些与屏幕不平行的多边形上的纹理贴图，提供了更佳的深度细节和精确显示，在视觉上是一个很大的改进。根据像素样本或 tap 的数目不同，存在几种不同级别的过滤质量。64tap 的各向异性过滤提供了比 8tap 的各向异性过滤更好的质量，但它也相对更慢。各向异性过滤是 3D 加速卡中最新实现的过滤类型。图 7.15 显示了点采样、双线性过滤和三线性过滤之间的区别。

图 7.15　点采样/双线性过滤/三线性过滤（http://www.reactorcritical.com 授权）

可以按如下方式调用 SetSamplerState()方法来使用各向异性过滤：

```
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MINFILTER, D3DTEXF_ANISOTROPIC);
m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAXANISOTROPY, 16);
```

ATI Radeon 支持的最大各向异性等级为 16。GeForce DDR 卡支持的最大各向异性等级为 2。可以在 DirectX Caps Viewer 中可以查看最大各向异性等级，也可以在 ConfirmDevice() 方法中检查相应的标志来获取该值。在示例程序中按 F5 和 F6 键可以看到不同的过滤效果（见图 7.16）。

图 7.16　三线性过滤/各向异性过滤

通过观察墙壁纹理中央的迹象可以察觉出三种过滤方式之间的区别，各向异性过滤更为锐化。老式显卡所支持的各向异性过滤质量的差别非常大，检查最终实际的视觉效果并在某些显卡上自动关闭各向异性过滤是有很意义的。

7.5.5　抗锯齿（Anti-Aliasing）

抗锯齿是一项可以立即改变所有PC游戏视觉品质的技术，它是计算机3D图形中的"Holy Grail[1]"。尤其是3dfx极大促进了全屏抗锯齿的应用和发展。如今它已是DirectX Graphics中的一项默认特征，是诸多支持的多采样类型之一。FSAA（Full Screen Anti-Aliasing，全屏抗锯齿）通过调整图像中每条斜线周围的亮度来隐藏其上的锯齿效果，它沿着这些边缘产生局部模糊的效果，并淡化了原本阶梯层级似的外观。那么它是怎么做到的呢？

全屏多采样对每个像素进行多次采样，这些不同的样本被混合后输出到屏幕。这便为抗锯齿或其他效果提供了改进的图像质量。

还有一种更高级的多采样方法是在光栅化阶段完成的，因此它仅影响三角形和三角形组，而不影响直线。它能提高多边形边缘的分辨率，以及深度测试和模板测试的分辨率。这种多采样称为"可屏蔽多采样（maskable multisampling）"，不过只有较新的硬件才支持这种采样。检查一下能力标志位或"Direct3D Settings"对话框（见图 7.17），如果只可选择D3DMULTISAMPLE_NONMASKABLE 标志，则表示不支持这种多采样。然而，对于D3DMULTISAMPLE_NONMASKABLE 标志而言，它允许用户在"Multisample Quality"字段中选择一个普通多采样的质量等级。

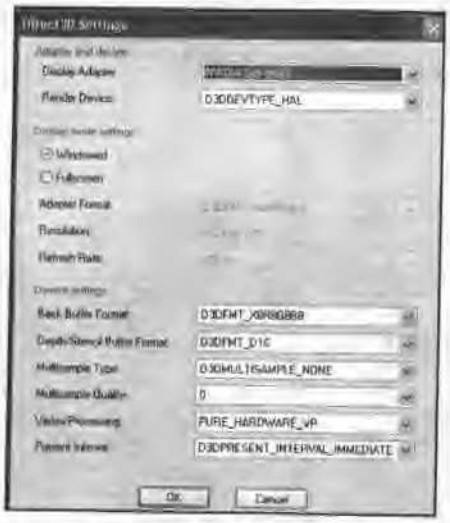


图 7.17　开启多采样

译注[1]：圣杯，传说耶稣在最后晚餐中所用的杯子，意指长期以来梦寐以求的东西。

开启多采样后就可以提供一个更高质量的视觉外观（见图 7.18），最终结果将更为平滑、更为逼真。

图 7.18　开启多采样的结果和未开启多采样的结果

多采样可用于实现新一代游戏中的某些特殊效果。例如，指定只有某些样本才受特定渲染通道的影响、允许运动模糊的模拟（simulation of motion blur）、景深聚焦效果（depth-of-field focus effect）、反射模糊（reflection blur）等等。

7.5.6　alpha 混合（blending）

使用 alpha 混合可将图元颜色与先前储存在帧缓冲器中像素的颜色组合起来（见图 7.19），这就是平时所说的与帧缓冲器相混合。这种形式的 alpha 混合可用于模拟半透明的物体、组合两副图像、实现一些特殊效果（如力场、火焰、离子束、光照映射）。

图 7.19　alpha 混合

alpha 混合的最大优点就是它允许你作多次渲染（multipass rendering；意味着以不同的技术多次对同一几何体进行渲染），并允许将这些渲染通道混合在一起。我将在第 8 章中对这一概念作更多的讨论。

alpha 混合是像素流水线中最后一个步骤。在 alpha 混合阶段，通过混合系数将新像素与当前已在帧缓冲器中的像素进行组合。可以将两个像素相加、相乘或使用 alpha 成分线性地组合它们，等等。

执行 alpha 混合就是组合两个颜色：源颜色（source color）和目的颜色（destination color）。源颜色是准备绘制到帧缓冲器中的像素，目的颜色是已存在于帧缓冲器中的像素。alpha 混合使用一个公式来控制源物体与目的物体之间比例。

```
FinalColor = SourcePixelColor * SourceBlendFactor + DestPixelColor * DestBlendFactor
```

SourcePixelColor 来自于某个将被渲染到当前像素位置上的图元；DestPixelColor 则来自于帧缓冲器中的当前像素位置。

通过改变 SourceBlendFactor（D3DRS_SRCBLEND）和 DestBlendFactor（D3DRS_DESTBLEND）标志可以产生各种期望的效果。FinalColor 是混合后最终写入帧缓冲器的颜色。

现在来看一些例子。如果想使 alpha 混合不起作用，只是绘制像素而不考虑帧缓冲器中已存在的内容。则可以这么做：

```
FinalColor = SourcePixelColor * 1.0 + DestPixelColor * 0.0
```

其中 DestBlendFactor 为 0.0，所以方程式可化简为：

```
FinalColor = SourcePixelColor
```

相应的源代码如下所示：

```
m_pd3dDevice->SetRenderState (D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
```

如果想要将源颜色与目的颜色相乘，则需把 SourceBlendFactor 设为 0.0、DestBlendFactor 设为 SourcePixelColor。

```
FinalColor = SourcePixelColor * 0.0 + DestPixelColor * SourcePixelColor
```

相应的源代码如下所示：

```
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR);
```

在示例程序中我使用以下代码来产生透明的立方体（按 F1 键后再按 8 来观看）：

```
if (m_bTex8 == TRUE)
{
```

```
        m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
        m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
        m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCCOLOR);
        m_pd3dDevice->SetTexture( 0, m_pWallTexture);
    }
    // render primitives
    if (m_bTex8 == TRUE)
    {
        // switch off alpha blending
        m_pd3dDevice->SetRenderState (D3DRS_ALPHABLENDENABLE, FALSE);
    }
```

上述代码段中的函数调用使 Direct3D 在源颜色（将被渲染到当前位置的图元颜色）和目的颜色（已在帧缓冲器中当前位置上的颜色）之间执行了一个线性混合。这便形成了一种类似于彩色玻璃般的效果。透过源物体可看到目的物体上的某些颜色，而其余的颜色则像是被源物体吸收了。方程式如下所示：

```
FinalColor = SourcePixelColor * SourceColor + DestPixelColor * InverseSourceColor
```

其中 InverseSourceColor 为：

$$(1 - SourcePixelColor_{red},\ 1 - SourcePixelColor_{green},\ 1 - SourcePixelColor_{blue},\ 1-SourcePixelColor_{alpha})$$

另外还有许多 alpha 混合系数，在 DirectX 9.0 SDK 文档中有相关的解释，只要搜索“blending”即可。

alpha 混合要求进行不少额外的数学运算和存储器访问，因此有必要在用到 alpha 混合时才打开它，不用时就关闭它。在示例程序调用了 Render()之后，就马上将 ALPHABLENDENABLE 设为 FALSE 了。

惟一需要担心的就是显卡是否支持 alpha 混合。可以通过检查特定的能力标志位来确定这一点。

```
if( (0 == ( pCaps->DestBlendCaps & D3DPBLENDCAPS_INVSRCCOLOR)) &&
    (0 == ( pCaps->SrcBlendCaps & D3DPBLENDCAPS_SRCCOLOR)) &&
    ...
```

D3DPBLENDCAPS_*标志用于检查显卡是否支持各种 D3DBLEND_*标志。D3DCAPS9 结构的成员 DestBlendCaps 或 SrcBlendCaps 可用于检查设备是否具有支持各项 D3DRS_DESTBLEND/D3DRS_SRCBLEND 参数的能力。

7.6 小结

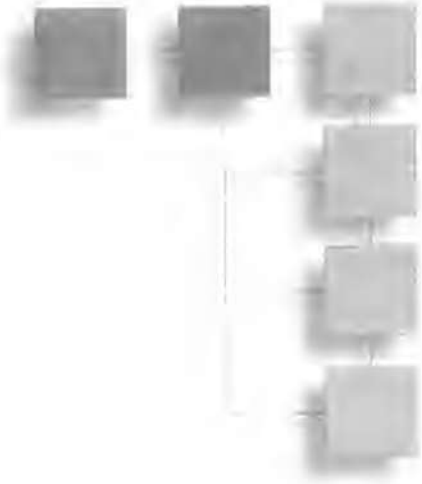
本章介绍了纹理映射的基础知识。你学习到了如何使用纹理坐标，以及它们是如何随着不同的纹理寻址模式（诸如包装纹理寻址模式、镜像纹理寻址模式）影响物体的视觉外观。讨论纹理时最重要的一个话题就是纹理过滤。不同的纹理过滤方法对视觉外观有着不同的效

果，同时也影响着应用程序的执行性能。根据显卡的不同，有时或许需要牺牲纹理的质量来换取速度，从而得到一个较为满意的执行性能。

本章最后介绍的 alpha 混合是一种用于混合两张纹理（一张作为源纹理，另一张作为目的纹理）的有效方法。

迄今为止，我还只是将注意力集中在使用一张纹理上，下一章将告诉你如何在同一时间巧妙地处理多张纹理。

第 8 章



使用多重纹理

这一章将介绍通过多纹理单元（multitexturing unit）可进行的颜色操作和 alpha 操作。在支持像素 shader 的 DirectX 8 出现之前，程序员只能通过 Direct3D 接口来使用多纹理单元，从而进行像素颜色方面的处理。在当前，众多老式显卡仍在使用传统的多纹理单元，就在我写此书时，它依然占据了游戏市场的主流。

在本章中，你将学习到如何：

- 使用黑暗映射（dark mapping）；
- 将纹理与材质的漫反射颜色相混合；
- 使用一张混合了材质漫反射颜色的黑暗贴图。

8.1 多次渲染（Multipass Rendering）

在 DirectX 6.0 引入多纹理单元时，其中最有趣的一项特征便是它可以在同一时间将多张纹理映射到同一个多边形上。以下这段伪代码逐步地展现了这个过程。

```
// texture #1
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
m_pd3dDevice->SetStreamSource( 0, m_pCubeVB, 0, sizeof(CUBEVERTEX) );
m_pd3dDevice->SetFVF( FVF_CUBEVERTEX );
m_pd3dDevice->SetIndices( m_pCubeIB, 0 );
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 24, 0, 36/3);
// texture #2
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
m_pd3dDevice->SetTexture(0, m_pDetailTexture);
```

```
m_pd3dDevice->SetStreamSource( 0, m_pCubeVB, 0, sizeof(CUBEVERTEX) );
m_pd3dDevice->SetFVF( FVF_CUBEVERTEX );
m_pd3dDevice->SetIndices( m_pCubeIB );
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 24, 0, 36/3);
```

因为使用了多次来渲染不同的纹理，故称为“多次渲染[1]”。Brian Hook 在他的 Course 29（收录于 SIGGRAPH '98）中提到过 *Quake 3* 共使用了 10 次渲染（见表 8.1）。

表 8.1　　Quake3 中使用的 10 次渲染

次数	描述
（1~4）	累积凹凸贴图（Accumulate bump map）
5	漫反射光照（Diffuse lighting）
6	基本纹理（Base texture），具有镜面反射成分
（7）	镜面光照（Specular lighting）
（8）	放射性光照（Emissive lighting）
（9）	体积/大气效果（Volumetric/atmospheric effect）
（10）	屏幕显示（Screen flash）

在当时，只有最快的机器才支持使用 10 次来渲染单独一帧，并且仍能保持合理的帧速率。如果图形加速卡不能保证合理的帧速率，则可以适当地去除去括号中的那几次渲染。

显然，一个渲染器采用的渲染次数越多，那么它的整体性能也就会越低。为了减少渲染次数，一些图形加速卡开始支持多重纹理（multitexturing），也就是在一次渲染中可以访问到两张或更多的纹理。以下代码给出了一个多重纹理的伪代码表示。

```
// texture #1
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);

// texture #2
m_pd3dDevice->SetTexture(1, m_pDetailTexture);
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADDSIGNED);
m_pd3dDevice->SetStreamSource( 0, m_pCubeVB, 0, sizeof(CUBEVERTEX) );
m_pd3dDevice->SetFVF( FVF_CUBEVERTEX );
m_pd3dDevice->SetIndices( m_pCubeIB );
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 24, 0, 36/3);
```

在此只调用了一次渲染函数（单次渲染），就将两张纹理一次性地映射到一个多边形上。

---

[1]：或称“多道渲染”。

如果第二张纹理用到了对第一张纹理操作所产生的结果，则称为“纹理混合”。最多可以将 8 张纹理混合到一起，这称为“纹理混合的级联（texture-blending cascade）”（见图 8.1）。

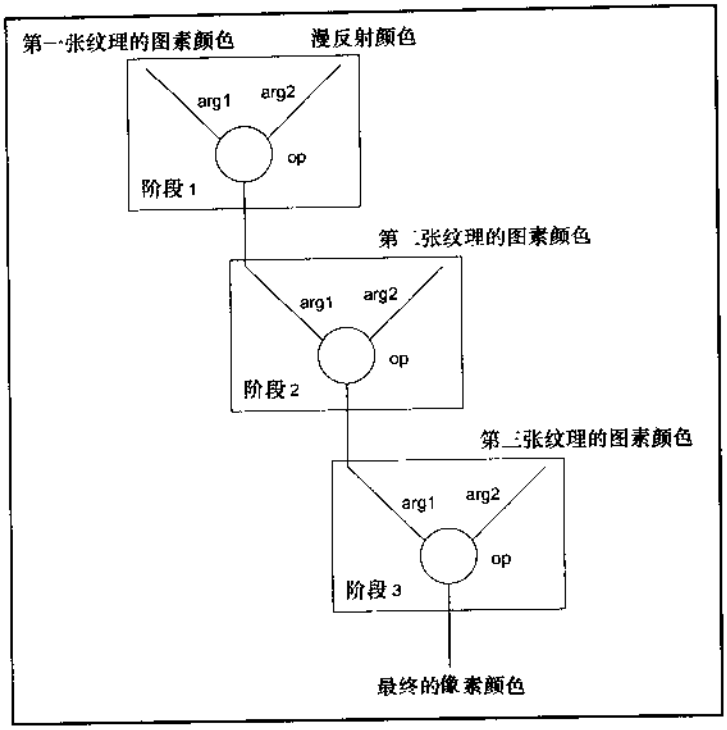


图 8.1　纹理混合的级联

在 Direct3D 6.0 之前，流水线中的某些阶段会计算出图素的颜色，然后将这些颜色与顶点间内插的图元颜色相混合。而自从 Direct3D 6.0 之后，便可以级联最多 8 个纹理操作单元了。你可以决定是将多张纹理分多次应用到一个普通图元上（多次渲染），还是在一次渲染中完成对多张纹理的应用（多重纹理），不过后者需要显卡支持才行。每个阶段的结果都会传递到下一个阶段，最后一个阶段的结果就将被光栅化到多边形上。

当前大多数 3D 硬件仅支持同时将两张纹理应用到一个普通的图元上，而较新的硬件则可同时处理最多 16 张纹理。

接下去让我们来研究一下 SetTextureStageState()所支持方法的各种颜色操作。

注意

老式的 3D 硬件基本上都不支持多重纹理，所以本示例无法在那些显卡上运行。（你会看到一个错误提示的消息框）

8.2　颜色操作

颜色操作是通过 D3DTSS_COLOROP 来设置的。设置颜色操作的语句大致如下所示：

```
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, op);
```

其中可用的 op 参数如下，这些值位于 D3DTEXTUREOP 枚举中。

- D3DTOP_DISABLE
- D3DTOP_SELECTARG1
- D3DTOP_SELECTARG2
- D3DTOP_MODULATE
- D3DTOP_MODULATE2X
- D3DTOP_MODULATE4X
- D3DTOP_ADD
- D3DTOP_ADDSIGNED
- D3DTOP_ADDSIGNED2X
- D3DTOP_SUBTRACT
- D3DTOP_BLENDDIFFUSEALPHA
- D3DTOP_BLENDTEXTUREALPHA
- D3DTOP_BLENDFACTORALPHA
- D3DTOP_BLENDTEXTUREALPHAPM
- D3DTOP_BLENDCURRENTALPHA
- D3DTOP_MODULATEALPHA_ADDCOLOR
- D3DTOP_MODULATEINVALPHA_ADDCOLOR
- D3DTOP_BUMPENVMAP
- D3DTOP_DOTPRODUCT3
- D3DTOP_MULTIPLYADD
- D3DTOP_LERP

每种颜色操作可接受最多 3 个颜色参数：D3DTSS_COLORARG0、D3DTSS_COLORARG1、D3DTSS_COLORARG2，可以将这些颜色参数设为以下任意值：

- D3DTA_DIFFUSE，使用漫反射颜色，这种颜色是在使用 Gouraud 着色时，对顶点成分中的漫反射颜色作插值计算而得的；
- D3DTA_SPECULAR，使用镜面反射颜色，这种颜色是在使用 Gouraud 着色时，对顶点成分中的镜面反射颜色作插值计算而得的；
- D3DTA_TFACTOR，使用纹理系数，可事先通过渲染状态 D3DRS_TEXTUREFACTOR

调用 SetRenderState()来设置的纹理系数；

- D3DTA_TEXTURE，使用本纹理阶段的纹理颜色；
- D3DTA_CURRENT，使用上一个混合阶段的结果；
- D3DTA_TEMP，使用一个用于读/写的临时寄存器颜色。

还可以将上述值与修饰标志 D3DTA_ALPHAREPLICATE 或 D3DTA_COMPLEMENT 其中之一作"位或"组合，前者用于复制 alpha 信息、后者用于对颜色参数取反。请继续往下读，让我们来研究一些实际应用中的例子。

8.2.1 黑暗映射（Dark Mapping）

绝大多数修饰表面的效果都是基于所谓"逐顶点（per-vertex）"方式计算的。也就是说参与实际计算的是三角形的每个顶点，而不是对每个像素进行渲染。有时这种技术会造成一些较为明显的视觉错误。试想这样一种情况，有一个很大的三角形，其表面近处有一个光源，当光源靠近该三角形的一个顶点时，就会看到这个三角形受光的效果。而当光源向三角形的中心靠近时，三角形的受光效果便会逐渐地消失。最坏的情况是当光源直接位于该三角形的中央时，整个三角形只有非常少的光亮，并且在它的中央会有一个亮点。由此可见，如果顶点未受到光照，则无法计算出正确的三角形面的颜色。为了产生基于像素光照的效果，最佳方法就是使用一张纹理贴图，该图的内容就是你所期望的光源类型照射在一张漆黑表面上的结果。

将两张纹理的颜色相乘称为"光照映射（light mapping）"（由于这种技术经常被用于使一张纹理变暗，故有时也称为"黑暗映射"）。为了设置黑暗映射，需在 Render()方法中添加如下代码：

```
// Set texture for the cube
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );

// Set the dark map
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_MODULATE );
```

更为详细的说明如下：

```
// first texture operation unit
// Associate texture with the first texture stage
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// use texture coordinate pair #1
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
```

```
// Set the first color argument to the texture associated with this stage
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// Use this texture stage's first color unmodified, as the output.
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );

// second texture operation unit
// Associate texture with the second texture stage
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
// use texture coordinate pair #1
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 0 );
// use the texture color from this texture stage.
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// Set the second color argument to the output of the last texture stage
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
// multiply result of stage 1 with result of stage 2
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_MODULATE );
```

这段代码将以下两张纹理组合成一张，如图 8.2 所示。

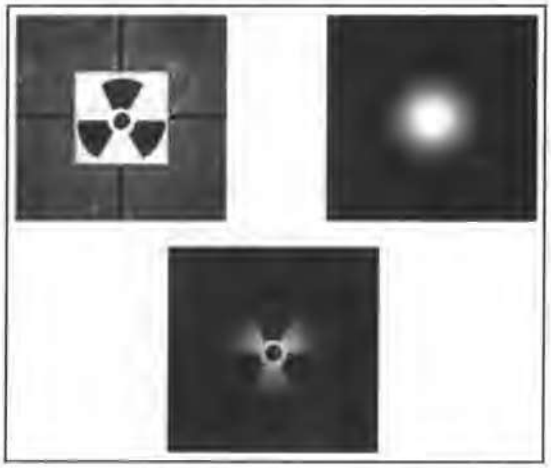


图 8.2 D3DTOP_MODULATE

这种类型的多重纹理之所以称为“黑暗映射”是因为最终结果中未受到光照的图素比原图中的图素更暗。许多 3D 射击游戏中都用到了这种技术，在 GLQuake1 中你就可以看到有这样的应用。

对于 RGB 颜色而言，渲染状态 D3DTSS_COLORARG1 与 D3DTSS_COLORARG2 用于控制颜色参数，D3DTSS_COLOROP 用于控制对这些参数所进行的操作。

第一个纹理操作单元将纹理 0 的数据传递到下一个阶段，其中控制参数 D3DTA_TEXTURE 表示本纹理阶段的颜色参数是纹理颜色。第二个纹理操作单元通过 Arg2 接收前一个阶段的结果，即纹理 0 的图素，并通过 Arg1 接收纹理 1 的图素，然后调制（D3DTOP_MODULATE）这两方面的图素。

有 3 种调制操作，如表 8.2 所示。

表 8.2　　调制操作

常量	描述	简单描述
D3DTOP_MODULATE	将参数的每个成分相乘	基础贴图 * 光照贴图
D3DTOP_MODULATE2X	将参数的每个成分相乘，并将乘积左移 1 位（等同于乘以 2 的效果）以使其变得更亮	基础贴图 * 光照贴图 << 1
D3DTOP_MODULATE4X	将参数的每个成分相乘，并将乘积左移 2 位（等同于乘以 4 的效果）以使其变得更亮	基础贴图 * 光照贴图 << 2

第一个纹理阶段（阶段 0）的默认操作是 D3DTOP_MODULATE；其他纹理阶段的默认操作是 D3DTOP_DISABLE。

SetTexture()函数用于将一张纹理赋予设备的某个纹理阶段。它的第一个参数必须是 0～7 范围内的数字，第二个参数为纹理接口指针。这个方法会增加第二个纹理接口参数的引用计数。因此，当不再需要使用某纹理时，应在相应纹理阶段中将其设为 NULL。如果你没有这么做，纹理接口就不会被释放，从而便会导致内存泄漏的问题。记住，自从 6.0 版本以后，Direct3D 就具有了可同时混合最多 8 张纹理的能力。

当应用程序选择了一张纹理作为当前纹理，也就是指示 Direct3D 将该纹理应用于此后所有将要渲染的图元，直到再次改变当前纹理为止。如果一个 3D 场景中的每个图元都有其各自不同的纹理，则必须在渲染每个图元之前先设置相应的纹理。

先前所列代码展示了多重纹理的做法。现在，让我们看一下如何通过多次渲染来达到同样的效果。

```
// Set texture for the cube
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTexture(0, m_pEnvTexture);
// draw polygon
```

其中 alpha 混合公式如下所示：

```
FinalColor = SourcePixelColor * 0.0 + DestPixelColor * SourcePixelColor
```

它模拟了多重纹理渲染代码中所使用的调制颜色的操作。

8.2.2　黑暗贴图动画

我通过 3 种调制操作创建了一个动画示例。

```
// animate darkmap
if (i < 40)
{
    m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
}
else if (i < 80)
{
    m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE2X);
}
else if (i < 120)
{
    m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE4X);
}
else if (i = 120)
{
    i = 0;
}
i++;
```

这里只是作为抛砖引玉，介绍了一种简单效果的实现。这个效果不是基于时间的，在前 40 帧中，SetTextureStage()使用了 D3DTOP_MODULATE，接下来的 40 帧中使用了 D3DTOP_MODULATE2X，最后 40 帧中使用了 D3DTOP_MODULATE4X。在后两个步骤中纹理变得更加明亮，直到显示完 120 个帧后又开始新的一轮。

8.2.3　混合纹理与材质漫反射颜色

有时很强的阳光会使物体上的颜色变得更亮。可以通过将纹理与材质的漫反射颜色相混合来模拟这种效果（见图 8.3）。

图 8.3　光照颜色加上纹理颜色

我是如何做到这种效果的呢？当一个白色材质反射一个定向光时，反射量越多则意味着纹理颜色的在最终显示结果中所占的成分越少。因此，那些被光直接照射到的立方体面会呈现为白色。可通过调用以下函数来设置材质：

```
D3DMATERIAL9 Material;
D3DUtil_InitMaterial( Material, 1.0f, 1.0f, 1.0f, 1.0f );
m_pd3dDevice->SetMaterial( &Material );
```

以下代码通过 SetLight()和 LightEnable(0, TRUE)来设置一个位于立方体上方的定向光。

```
HRESULT CMyD3DApplication::SetLights()
{
    if (m_bTex3 == TRUE || m_bTex4 == TRUE)
    {
        m_pd3dDevice->SetRenderState(D3DRS_AMBIENT, 0);
        D3DLIGHT9 light;
        D3DUtil_InitLight( light, D3DLIGHT_DIRECTIONAL, 0.0f, -5.0f, -5.0f );
        m_pd3dDevice->SetLight( 0, &light );
        m_pd3dDevice->LightEnable( 0, TRUE );
    }
    else if (m_bTex7 == TRUE)
    {
        m_pd3dDevice->LightEnable( 0, FALSE);
        // Set the ambient light.
        m_pd3dDevice->SetRenderState(D3DRS_AMBIENT, 0x00aaffaa);
    }
    return S_OK;
}
```

混合代码如下：

```
// Set texture for the cube
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_ADD);
```

更为详细的说明如下：

```
// Associate texture with the first texture stage
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// use the texture color from this texture stage.
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
// use the diffuse lighting information produced by gouraud shading
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
// add texture color and diffuse color
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_ADD);
```

正如你所见，这个效果只需要一次渲染，因此这里没有多重纹理的代码。对这段代码的简单描述便是“基础贴图 + 漫反射插值”。

在固定功能的流水线中，有三种获取漫反射颜色的途径：材质、漫反射顶点颜色、镜面

更为详细的说明如下：

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// use texture coordinate pair #1
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0);
// use as first argument the texture color of this texture
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
// use as the second argument the color of the material
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
// multiply both
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE);
```

如果没有光照，则纹理颜色会乘以 0，因此便看不到墙壁纹理了。所以在这种情况中，只能看到第二张纹理。

```
// Set darkmap
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0);
// use the texture color of this stage
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE);
// use the result of the previous texture stage = tex color X light color
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT);
// multiply the result of the previus texture with the color of this texture
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
```

如果第一张纹理是不可见的，则第一阶段的值 0 与第二阶段的颜色相乘的结果仍将为 0（黑色）。当没有光照射在立方体的某个面上时，该面的第一张纹理颜色会减弱为黑色。而当你处于一个真正漆黑的地方时，由于光照微弱，你是无法看到周围物体的颜色的，因此以上代码对于模拟这种情况很有用。这个过程可以简短地描述为“（基础贴图 * 光的颜色）* 黑暗贴图”。

使用多次渲染来实现同样的效果，如下所示：

```
// Set texture for the cube
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE);
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR);
m_pd3dDevice->SetTexture(0, m_pEnvTexture);
// draw polygon
```

这个过程中的 alpha 混合部分与我在黑暗映射示例中所使用的相同。

8.2.5 发光映射（Glow Mapping）

发光映射与黑暗映射正好相反。它对于创建那些具有独立于基础贴图的发光部分的物体很有用，比如发光二极管、按钮、建筑物内的灯光、太空船上的灯光等（见图 8.5）。

图 8.5　发光映射

发光映射应仅影响基础贴图上的发光区域，而不应影响到其余部分。因此需要对发光效果做加法，而不是做调制。

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);
// Set glow map
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD);
```

通过 D3DTOP_SELECTARG1 将第一个纹理阶段中未修改的颜色输出到下一个纹理阶段。然后使用 D3DTOP_ADD 加上第二个纹理阶段的颜色。使用多次渲染来实现同样的效果，如下所示：

```
// Set texture for the cube
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_SELECTARG1);
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
m_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
m_pd3dDevice->SetTexture(1, m_pEnvTexture);
// draw polygon
```

其中 alpha 混合公式如下所示：

```
FinalColor = SourcePixelColor * 1.0 + DestPixelColor * 1.0
```

该公式就是模拟了多重纹理代码中的 D3DTOP_ADD 功能。

8.2.6 细节映射（Detail Mapping）

在较新显卡上，你或许会打算使用一次额外的纹理渲染来完成细节映射（见图 8.6）。

图 8.6 细节映射

正如你所见，墙面看上去更粗糙了，就好像是一块粗糙的石灰泥。那么如何达到这种效果呢？试想这样的情景：你手持一把锯子在某个关卡中行走，突然怪物出现了。而在这个外星球上随处可以看到一种板条箱，所以你马上蹲伏到了其中一个板条箱后面。作为一位挥舞着锯子来拯救世界的英雄，正当你思索着自己的使命，你突然发觉眼前的这张箱面纹理看上去有点模糊。当你站在一步之遥的距离处看这张纹理还是挺正常的，可一旦你的鼻尖触及到那些在箱面上日久生锈的冰冷金属时，这张纹理就开始变得模糊了。

好了，回到真实世界吧。在上例所述的情况中，可以通过加上一张额外的细节纹理来锐化箱面的视觉外观。那么这张细节纹理起到了什么作用呢？

基础贴图（或是第一张纹理）的颜色未经修改便作为第二个纹理阶段中的第二个参数。然后通过 D3DTOP_ADDSIGNED 将灰色的细节纹理与基础贴图相加。这个操作本质上是做了一个加法，只是使用了有符号的颜色值(-127～128)来代替平时使用的无符号值(0～255)。在 ADDSIGNED 两张纹理时，-127 为黑色，128 为白色。它将参数的每个成分相加后，再加上-0.5 的偏移量，从而使有效值域变为-0.5～0.5。

在细节贴图中较亮的灰色图素将使基础贴图变得更亮，而较暗的灰色图素则会使基础贴图变暗。由此箱面将会呈现为一张粗糙的表面，从而使之看上去更为真实。可通过以下源代码片断来使用细节映射：

```
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
```

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);
// Set detail map
m_pd3dDevice->SetTexture(1, m_pDetailTexture);
m_pd3dDevice->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0 );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
m_pd3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADDSIGNED);
```

通过多次渲染也可以实现同样的效果，如下所示：

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);
m_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
m_pd3dDevice->SetTexture( 0, m_pWallTexture);
// draw polygon
// Set darkmap
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_ALPHABLENDENABLE, TRUE);
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_SRCBLEND, D3DBLEND_DESTCOLOR);
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_DESTBLEND, D3DBLEND_ SRCCOLOR);
m_pd3dDevice->SetTexture(1, m_pDetailTexture);
// draw polygon
```

其中 alpha 混合公式如下所示：

```
FinalColor = SourcePixelColor * DestPixelColor + DestPixelColor * SourcePixelColor
```

老式显卡是不支持 D3DTOP_ADDSIGNED 的，可以使用 MODULATE2X 来近似地代替它。MODULATE2X 的相应公式如下所示：

```
FinalColor = 2 * Arg1 * Arg2
FinalColor = Arg1 * Arg2 + Arg2 * Arg1
FinalColor = SourcePixelColor * DestPixelColor + DestPixelColor * SourcePixelColor
```

由于在老式显卡上进行 alpha 混合是非常耗时的计算，所以不要忘记在适当的时候打开或关闭 alpha 混合。

表 8.3 给出了一些我在源代码中未曾涉及的颜色操作，从及这些操作所对应的常量和描述。

纹理阶段操作会导致对所有颜色操作及以下内容做线性混合：

- 迭代的漫反射 alpha；
- 当前迭代的纹理 alpha；
- alpha 标量（用 D3DRS_TFACTOR 设置）；
- 预乘（pre-multiplied）的纹理 alpha；
- 前一个阶段计算出的 alpha。

在 ATI 以前的 RADEON 1.3 SDK 中，有一个示例通过 D3DTOP_BLENDCURRENTALPHA 和 D3DTOP_BLENDFACTORALPHA 参数来模拟了一面被霜覆盖着的玻璃。而对于其

他参数而言，可用于模拟真实世界中的哪些效果，我暂时还没有想到有什么合适的例子。

表 8.3　　其他颜色操作的常量与描述

常量	描述	公式
D3DTOP_BLENDDIFFUSEALPHA	使用由每个顶点计算而得的 alpha 插值来线性混合本纹理阶段	S_{RGBA} = Arg1 * (Alpha) + Arg2 * (1 – Alpha)
D3DTOP_BLENDTEXTUREALPHA	使用本阶段纹理中的 alpha 来线性混合本纹理阶段	S_{RGBA} = Arg1 * (Alpha) + Arg2 * (1 – Alpha)
D3DTOP_BLENDFACTORALPHA	使用通过 D3DRS_TEXTUREFACTOR 渲染状态设置的 alpha 标量来线性混合本纹理阶段	S_{RGBA} = Arg1 * (Alpha) + Arg2 * (1 – Alpha)
D3DTOP_BLENDTEXTUREALPHAPM	使用一个预乘 alpha 来线性混合本纹理阶段	S_{RGBA} = Arg1 + Arg2 * (1 – Alpha)
D3DTOP_BLENDCURRENTALPHA	使用来自上一纹理阶段的 alpha 来线性混合本纹理阶段	S_{RGBA} = Arg1 * (Alpha) + Arg2 * (1 – Alpha)

D3DTOP_MODULATEALPHA_ADDCOLOR 使用第一个参数的 alpha 来调制第二个参数的颜色，然后将结果加上第一个参数。

$$Result = Arg1_{RGB} + Arg1_{Alpha} * Arg2_{RGB}$$

D3DTOP_MODULATEINVALPHA_ADDCOLOR 则使用相反的 alpha 来进行计算。

$$Result = Arg1_{RGB} + (1 - Arg1_{Alpha}) * Arg2_{RGB}$$

以下两个参数只能在支持多重纹理且支持 3 个纹理阶段参数的显卡上使用。因此，它们被称为“三元纹理混合（triadic texture-blending members）”。

D3DTOP_MULTIPLYADD 完成了一个累积乘法（multiply-accumulate）操作。它取两个参数相乘，然后将乘积加上另一个输入/源参数，最后将和作为结果。

```
Result = Arg0 + Arg1 * Arg2
```

D3DTOP_LERP 根据第一个源参数指定的比例计算第二、第三个源参数的线性插值。

```
Result = (Arg0) * Arg1 + (1 - Arg0) * Arg2
```

接下来来看一下 alpha 操作。

8.3　alpha 操作

在介绍了颜色操作之后，我将专注于介绍大多数显卡都支持的 alpha 操作。设置 alpha 操作的语句大致如下所示：

```
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG1, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG2, arg );
m_pd3dDevice->SetTextureStageState( 1, D3DTSS_ALPHAOP, op );
```

其中 op 参数可以是：

- D3DTOP_DISABLE
- D3DTOP_SELECTARG1
- D3DTOP_SELECTARG2
- D3DTOP_MODULATE
- D3DTOP_MODULATE2X
- D3DTOP_MODULATE4X
- D3DTOP_ADD
- D3DTOP_ADDSIGNED
- D3DTOP_ADDSIONED2X
- D3DTOP_SUBTRACT
- D3DTOP_MULTIPLYADD
- D3DTOP_LERP

所有这些参数我都已在前一节“颜色操作”中解释过了，alpha 操作沿用了相同的概念。因此很容易采用相同的方法将它们运用于 alpha 操作。

每种 alpha 操作最多可使用 3 个参数（arg）：D3DTSS_ALPHAARG0、D3DTSS_ALPHAARG1、D3DTSS_ALPHAARG2。可以将这些 alpha 参数设为了以下任意值：

- D3DTA_DIFFUSE，使用漫反射颜色，这种颜色是在使用 Gouraud 着色时，对顶点成分中的漫反射颜色作插值计算而得的；
- D3DTA_SPECULAR，使用镜面反射颜色，这种颜色是在使用 Gouraud 着色时，对顶点成分中的镜面反射颜色作插值计算而得的；
- D3DTA_TFACTOR，使用纹组系数，可事先通过渲染状态 D3DRS_TEXTUREFACTOR 调用 SetRenderState()来所设置的纹理系数；
- D3DTA_TEXTURE，使用本纹理阶段的纹理颜色；
- D3DTA_CURRENT，使用上一个混合阶段的结果；
- D3DTA_TEMP，使用一个用于读/写的临时寄存器颜色。

还可以将上述这些值“位或”上 D3DTA_COMPLEMENT 修饰标志，以此来对 alpha 参数取反。在“颜色操作”那节中可以了解到所有这些参数。

Direct3D 在渲染一个场景时，它可以结合几种来源的颜色信息：顶点颜色、当前材质、纹理贴图、先前写入渲染目标的颜色，然后将其中一些颜色混合起来。同时你也可以使用一个叫作"alpha"的系数来指示 Direct 3D 该以怎么样的权重混合这些颜色。alpha 信息可以储存在顶点中、材质中、纹理贴图中：alpha 值为 0 表示完全透明，为 1 则表示不透明，其余 0～1 之间的值则表示不同程度的半透明。

如果要从一张纹理中获取 alpha，应将 D3DTA_TEXTURE 作为 alpha 参数；如果要使用来自于顶点颜色中的 alpha，应将 D3DTA_DIFFUSE 作为 alpha 参数，并确保 D3DRS_DIFFUSEMATERIALSOURCE = D3DMCS_COLOR1（这是默认值）；如果要使用来自于材质颜色中的 alpha，应将 D3DTA_DIFFUSE 作为 alpha 参数，并确保已设置了 D3DRS_DIFFUSEMATERIALSOURCE = D3DMCS_MATERIAL。

```
m_pd3dDevice->SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MATERIAL);
```

如果未用 SetRenderState()设置 D3DMCS_x 参数，则从默认来源获取漫反射颜色。下面这个 alpha 调制示例就是这种情况。

alpha 调制

将绿色的环境光与材质 alpha 相调制，便可以模拟出夜视镜观察的效果（见图 8.7）。

图 8.7　alpha 调制

```
// Set the ambient light.
D3DCOLOR d3dclrAmbientLightColor = D3DRGBA(0.0f,1.0f,0.0f,1.0f);
m_pd3dDevice->SetRenderState(D3DRENDERSTATE_AMBIENT, d3dclrAmbientLightColor);
```

可以使用以下代码来调制环境光与纹理颜色：

```
m_pd3dDevice->SetTexture( 0, D3DTextr_GetSurface("wall.bmp"));
```

幸好你可以编写一个算法来让计算机自动地考虑这个问题。那么这个算法的任务是什么呢？它需要跟踪可用的纹理内存总数，并了解哪些纹理是经常用到的而哪些是很少用到的。至少，这个纹理管理算法必须要能决定哪些现有的纹理资源可以通过一张纹理图像来重新加载，以及应该销毁哪些表面，然后由新的纹理资源取而代之。

Direct3D 有一个自动的纹理管理系统。其实在早先你已经用到过它了，当通过CreateTexture()创建一张纹理时，将 Pool 参数指定为 D3DPOOL_MANAGED 标志，就是向 Direct3D 请求了该系统的支持。在创建一张托管纹理时，不可以使用 D3DPOOL_DEFAULT 或 D3DPOOL_SYSTEMMEM 标志。

纹理管理器通过一个时间戳来跟踪纹理的使用情况，这个时间戳用于记录每张纹理最后被使用的时间。纹理管理器通过“最近最少使用”算法来决定哪些纹理应从视频内存中移除。若有两张纹理同时满足移除条件，则将根据纹理属性来作进一步地判断，这就好像是在比赛中为两个平分选手而增设的加时赛一样。如果它们具有相同的优先级，则移除最近最少使用的纹理；如果它们具有相同的时间戳，则移除低优先级的纹理。

通过为纹理表面调用 IDirect3DResource8::SetPriority()方法，可以给托管纹理赋予一个优先级。我的建议是：在你没有掌握更多知识之前，还是请使用 D3DPOOL_MANAGED。

8.6 附加资源

可以在 Gamasutra（http://www.gamasutra.com/features/programming/19981009/multitexturing_01.htm）上有几篇 Mitchell、Tatro、Bullard 撰写的文章，这些文章讨论了如何通过 DirectX 来使用多重纹理，这些是同类主题中最棒的几篇文章。他们还开发了一个工具用于可视化纹理操作，赶快试试看吧。NVIDIA 也有类似的可视化纹理操作的工具，位于 http://www.nvidia.com。另外，在 1997 年 8 月刊 *Game Developer* 杂志的第 12 页上有一篇很有趣的文章，名为“Multipass Rendering and the Magic of Alpha Rendering”，作者是 Brian Hook。最后，我还在找到了一些 ATI 以前的示例程序，也很有趣，它们位于 http://www.ati.com。

以下 Web 页面是两篇关于各向异性过滤的解释：

http://whatis.techtarget.com/definition/0,,sid9_gci870037,00.html

http://www.ping.be/powervr/Anisotropic.htm

8.7 小结

本章介绍了多纹理单元的使用方法，并介绍了如何编写一些常见的效果。由于普通硬件对多纹理单元的支持存在差异，所以非常有必要去仔细地检查各个能力标志位。

在未来几年里 DirectX 7 的显卡在市场上所占份额将会逐年减少。所以如果你的游戏不依

赖于 DirectX 7 的显卡，那么应使用更为灵活的像素 shader 来编写这些效果。

8.8　第二部分提问

问：为什么要使用纹理？

答：用来模拟一个粗糙的或不规则的表面，或者用来创建一种假象，感觉某物体好像是由许多多边形构成似的。

问：Direct3D 的编址方案是如何运作的？

答：它使用了一个一般化的编址方案，其中纹理地址由图素坐标构成，这些坐标映射在区间[0.0, 1.0]中。

问：对每张纹理可以使用多于一组的纹理坐标吗？

答：是的，可以对一张纹理使用最多 8 组纹理坐标，只需在顶点定义中声明它们，并在顶点结构中定义它们即可。

问：所谓的"纹理寻址模式"可用来做什么？

答：通过使用区间[0.0, 1.0]之外的纹理地址以及各种纹理寻址模式，可以产生一些特殊的效果。

问：默认纹理寻址模式是什么？其余的模式又是什么？

答：包装、镜像、夹持、边框颜色、一次镜像。

问：镜像纹理寻址模式有什么效果？

答：该模式在每个整数边界上对纹理作镜像处理，因此在区间[0.0, 1.0]之外的图素都被作了翻转。

问：夹持纹理寻址模式有什么效果？它有什么用？

答：它将纹理边缘上的像素颜色延伸到窗口的边界上。有时通过包装模式在纹理边缘作双线性插值会产生一些视觉错误，而夹持模式可以改善这个问题。

问：边框颜色寻址模式有什么效果？它的缺点又是什么？

答：该模式在纹理周围设有一个边框，边框颜色可由你来选择；通过这种模式可以伪造出一张贴在墙上的大纹理。缺点在于老式硬件可能不支持这种模式。

问：新引进的一次镜像纹理寻址模式有什么效果？

答：该模式在区间[−1.0, 1.0]内对纹理作镜像处理，该区间之外则作夹持处理。它对于体积光贴图（volumetric light map）很有用。

问：纹理包装是什么？它有什么用？

答：纹理包装向程序员提供了一种选项来决定如何在纹理坐标间作插值计算。举一个简单的例子：对于一张包裹在圆柱体上的纹理而言，采用默认的插值计算方法并不是很有效，通常需要穿越纹理的边界来计算插值。

问：纹理包装无法与什么共用？

答：纹理寻址模式。由于在区间[0.0, 1.0]之外的纹理坐标对于纹理寻址模式而言是有效的，但对纹理包装无效，因此只能二选其一。

问：什么是纹理过滤和纹理抗锯齿？

答：它们都是用于克服图形光栅化缺点的方法，可防止阶梯锯齿、以及放大和缩小时所产生的问题。

问：什么是 mipmap？

答：是指一系列的纹理，其中每个 mipmap 序列的高度和宽度都是前一级高度和宽度的一半。Direct3D 会根据当前绘制多边形的宽高自动选取一个合适的 mip 级纹理。

问：什么是线性纹理过滤？

答：它对距离采样点最近的 4 个图素求平均值。

问：什么是三线性过滤？

答：双线性过滤加上 mipmap。

问：什么是各向异性过滤？它有什么效果？

答：它不仅对最临近的 4 个图素求平均值，还会根据显卡的支持情况以及你所选择的质量对更多的图素求平均值。它也使用 mipmap。另外它还能提供更多的深度细节以及更为精确的纹理显示。

问：全屏抗锯齿是如何运作的？

答：它对每个像素进行多次采样，然后将不同的样本相互混合后输出到屏幕。由此便可实现一些有趣的特效，比如运动模糊、景深聚焦效果、反射模糊等等。

问：alpha 混合公式是什么？

答：FinalColor = SourcePixelColor * SourceBlendFactor + DestPixelColor * DestBlendFactor

问：alpha 混合是如何运作的？

答：它根据当前设置的混合系数，将当前位于像素流水线中的像素与已在帧缓冲器中的像素相混合。

问：什么是多次纹理？什么是多重纹理？

答：多次纹理是指分多次将多张纹理映射到同一个多边形上；多重纹理渲染则是指将多张纹理在一次渲染中映射到同一个多边形上。后者更快。

问：什么是纹理混合的级联？

答：这是指将一个纹理阶段的处理结果传递到下一个纹理阶段，最多可混合 8 个纹理阶段的处理结果。

问：为什么要使用黑暗/光照映射？

答：Direct3D 是基于“逐顶点”来计算光照。试想一个很大的三角形，在其表面近处有一个光源，只要光源靠近三角形的一个顶点时，就会看到这个三角形受光的效果。但若光源位于三角形的中央时，则只有少量的光照射在三角形上。黑暗/光照贴图可用于实现基于逐像素光照的效果。

问：黑暗/光照映射是如何运作的？

第 3 部分

DirectX 图形编程核心

- 第 9 章　使用高级 Shader 语言进行 Shader 编程
- 第 10 章　更多高级的 Shader 效果
- 第 11 章　使用文件
- 第 12 章　使用*.md3 文件

数据。

像素 shader 可在功能上取代传统的多纹理（multitexturing）阶段，它位于光栅化（rasterization）阶段之后。它从光栅器、顶点 shader、应用程序、纹理贴图几处获取数据。有一点务必记住，纹理贴图不仅可以存储颜色数据，还可用于存储所有对像素 shader 中的算法有用的数值信息。另外，像素 shader 的计算结果可以写入一个渲染目标，然后将该渲染目标设置为后续渲染的输入纹理，这种技术称为“渲染到渲染目标（rendering to the render target）”。像素 shader 通常输出一个 float4 类型的值。如果像素 shader 同时对几个渲染目标进行渲染，则最多可输出 4 个 float4 值，这种技术称为“多渲染目标（Multiple Render Targets）”或简称“MRT”。

9.3 通过 HLSL 实现常用光照公式

接下去的几个示例程序将实现一些常用的光照公式，从而演示如何使用 HLSL 来编写 shader，并进一步介绍顶点/像素 shader 的任务。

9.3.1 环境光照

在环境光照模型中，所有光线都均匀地从各个方向照射到物体上，这种光照模型在空间中是没有位置的，因此也将其称为“全局光照模型（global lighting model）”。（而“区域光照”则是一种更高级的全局光照模型。）

可以通过以下公式来描述环境光照的成分：

$$I = A_{intensity} * A_{color}$$

值 intensity 表示光照的强度，值 color 表示光的颜色。一个更复杂的包含环境光的光照公式如下：

$$I = A_{intensity} * A_{color} + Diffuse + Specular$$

其中 Diffuse 与 Specular 分别是漫反射和镜面反射光照公式的占位符，在接下去的示例中会看到这些公式。所有高级语言都支持以类似于数学表达式的方式来书写 shader 语句，比如乘法和加法。因此，在高级语言中可以这样书写环境成分：

```
float4 Acolor = {1.0, 0.15, 0.15, 1.0};
float Aintensity = 0.5;
return Aintensity * Acolor;
```

进一步地简化如下：

```
return float4 (0.5, 0.075, 0.075, 1.0);
```

在这里预先计算 intensity 与 color 的乘积是一种常见的做法。为了简洁易读起见，以下

示例程序就是这么做的。就算你不做这样的简化，HLSL 编译器也会在它的编译输出中计算出这两个常量值的运算结果。

以下源代码展示了一个顶点 shader 和一个用于显示环境光照的像素 shader：

```
float4x4 matWorldViewProj;

struct VS_OUTPUT
{
    float4 Pos: POSITION;
};

VS_OUTPUT VS( float4 Pos: POSITION )
{
    VS_OUTPUT Out = (VS_OUTPUT) 0;
    Out.Pos = mul(Pos, matWorldViewProj); // transform Position
    return Out;
}

float4 PS() : COLOR
{
    return float4(0.5, 0.075, 0.075, 1.0);
}
```

源代码起始处的 VS_OUTPUT 结构描述了顶点 shader 的输出值。顶点 shader 看上去像一个 C 函数，具有 VS_OUTPUT 类型的返回值和一个位于其名字（VS）后、圆括号内的输入参数 Pos。

顶点 shader 的输入值与输出值都用到了 POSITION 语义，它是通过前面的一个冒号（:）来标识的。语义帮助 HLSL 编译器为数据绑定正确的 shader 寄存器。顶点 shader 输入结构中的这句 POSITION 语义标识了该输入数据乃是一个位置数据。第二处语义则标识了顶点 shader 的返回值也是一个位置数据（顶点 shader 必须至少要输出一个位置数据），该位置数据将作为后续像素 shader 的输入值。还有一些其他关于顶点 shader 和像素 shader 的输入输出数据方面的语义，在接下来的示例程序中将会介绍其中的一部分。若想全面了解这些内容，请参阅 DirectX 9 SDK 文档。

顶点 shader 使用 matWorldViewProj 矩阵来变换顶点位置，matWorldViewProj 矩阵在此作为一个由程序提供的常量。最后，顶点 shader 输出位置值。

像素 shader 也遵循类 C 函数的语法。它的返回值总是 float4。由于这里使用语义 COLOR 标注了返回值，所以编译器会始终将该返回值作为颜色值来处理。与顶点 shader 相比，此处的像素 shader 没有显式的输入值，因为其名称（PS）后的括号内是空的。

顶点 shader 和像素 shader 的名字还是 shader 程序的入口点。因此，你必须要在编译器的命令行参数中提供它们的名字。

图 9.3 展示了使用环境光照模型的 Example 程序，这个程序位于 CD-ROM 的“Chapter9\Example”目录下。

你可能已经发现了一个问题：当茶壶以某种角度面对观察者时，就无法看清茶壶表面的

```
    float3 Light : TEXCOORD0;
    float3 Norm : TEXCOORD1;
};

VS_OUTPUT VS(float4 Pos : POSITION, float3 Normal : NORMAL)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    Out.Pos = mul(Pos, matWorldViewProj);          // transform Position
    Out.Light = normalize(vecLightDir);            // output light vector
    Out.Norm = normalize(mul(Normal, matWorld)); // transform Normal and normalize it
    return Out;
}

float4 PS(float3 Light: TEXCOORD0, float3 Norm : TEXCOORD1) : COLOR
{
    float4 diffuse = { 1.0f, 0.0f, 0.0f, 1.0f};
    float4 ambient = {0.1, 0.0, 0.0, 1.0};
    return ambient + diffuse * saturate(dot(Light, Norm));
}
```

与前例相比，这个顶点 shader 还多使用了一个顶点法线作为其输入数据。语义 NORMAL 指示编译器应如何将该数据绑定到顶点 shader 寄存器。“世界-观察-投影”矩阵、世界矩阵、光向量都是由应用程序提供的，分别通过常量 matWorldViewProj、matWorld、vecLightDir 传递给顶点 shader。

本例的顶点 shader VS 通过变量 Light 和 Norm 输出了向量 N 和 L。这两个向量都在顶点 shader 中通过内部函数 normalize()作了归一化。该函数返回归一化的向量 v = v/length(v)，如果 v 的长度为 0，则结果不确定。

Normal 向量的变换是通过乘以世界矩阵完成的。函数 mul(a, b)用于做 a 和 b 的矩阵乘法。如果 a 是一个向量，则把它当作一个行向量来看待；如果 b 是一个向量，则把它当作一个列向量来看待。a 的列数与 b 的行数必须相等，乘积结果为“a 的行数×b 的列数”。在这个示例中，mul()的第 1 个参数是一个位置向量（因此它被当作是一个行向量来看待），第 2 个参数是一个由 16 个浮点值（float4x4）组成的变换矩阵。图 9.6 展示了这个行向量和这个矩阵。

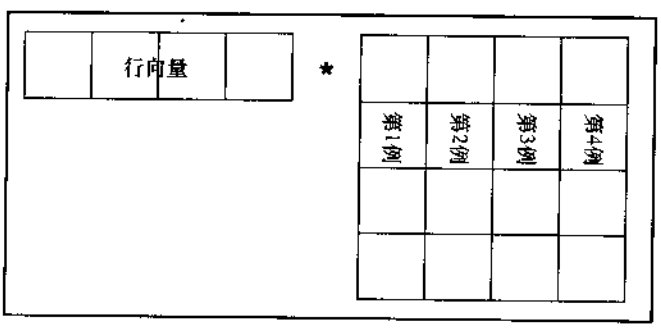


图 9.6　向量与矩阵的乘法

在像素 shader PS 中实现了整个光照公式，该公式位于 return 语句中，由一个环境成分和

多这方面的内容，这篇文章位于 http://www.gamasutra.com/features/20020801/beaudoin_01.htm。

像素 shader 的最后一行是 return 语句，其内容对应于先前展示的那个光照公式。

9.3.4　自身遮蔽阴影项（Self-Shadowing Term）[1]

通常将镜面反射光照模型和一个自阴影项一起使用，从而可以获得一种几何体自身遮蔽的阴影效果。如果光向量被几何体挡住了，该项便会将光的亮度设为 0。还可以将其用于线性地缩放光的亮度。这有助于在使用凹凸贴图时减少像素跳跃感（popping）。图 9.10 展示了两个例子，右边的使用了自阴影项，左边的没有使用。

图 9.10　自阴影项

有几种方法来计算该项。Example4 示例使用的是 S = saturate(4 * N.L)，这个示例位于 CD-ROM 的“Chapter9\Example4”目录下。（在 Ronald Frazier 所写的一篇文章“Advanced Real-Time Per-Pixel Lighting in OpenGL”中有更多关于自阴影项的内容，这篇文章位于 http://www.ronfrazier.net/apparition/index.asp?appmain=research/advanced_per_pixel_lighting.html。）这种实现重复利用了 N.L 的计算结果来进一步计算自阴影项。如此便引出了以下光照公式：

$$I = A + S * (D * N.L + (R.V)^n)$$

如果漫反射为 NULL，则自阴影项将使光照模型中的漫反射成分和镜面反射成分也减少为 NULL。漫反射成分还用于减弱镜面反射成分的强度。像素 shader 的代码如下：

```
float4 PS(float3 Light: TEXCOORD0, float3 Norm : TEXCOORD1, float3 View : TEXCOORD2) :
   COLOR
{
    float4 diffuse = { 1.0f, 0.0f, 0.0f, 1.0f};
    float4 ambient = {0.1, 0.0, 0.0, 1.0};

    float3 Normal = normalize(Norm);
    float3 LightDir = normalize(Light);
    float3 ViewDir = normalize(View);
    float4 diff = saturate(dot(Normal, LightDir)); // diffuse component

    // compute self-shadowing term
    float shadow = saturate(4* diff);

    float3 Reflect = normalize(2 * diff * Normal - LightDir); // R
```

[1]：为了使下面的写和阅读，译者将“自身遮蔽阴影（Self Shadowing）”简称为“自阴影”。

```
    float4 specular = pow(saturate(dot(Reflect, ViewDir)), 8); // R.V^n

    // I = ambient + shadow * (Dcolor * N.L + (R.V)n)
    return ambient + shadow * (diffuse * diff + specular);
}
```

9.3.5 凹凸映射（Bump Mapping）

凹凸映射可以产生一种表面好似具有高低起伏的几何体的假象（如凹槽、凸出部分）。图 9.11 显示了 Example5 程序，它位于 CD-ROM 的“Chapter9\Example5”目录下。这个程序使观察者产生了一种假象，好像在这个地球模型上有山的区域要比有水的区域更高些。

图 9.11 凹凸映射

凹凸贴图是一种图像文件（如*.dds 或*.tga），它与普通的颜色贴图相比，除了保存颜色值外还保存着法线，这些法线将取代顶点法线参与光照计算。在最常见的凹凸映射技术中，这些法线都保存在“纹理空间”或“切线空间（tangent space）”中。不过这会导致一个问题，由于通常是在模型空间或世界空间中处理光向量的，所以为了获得正确的结果，就必须将光向量变换到与凹凸贴图的法线相同的空间中去。这需要借助于一个纹理或切线空间坐标系来完成该变换（见图 9.12）。

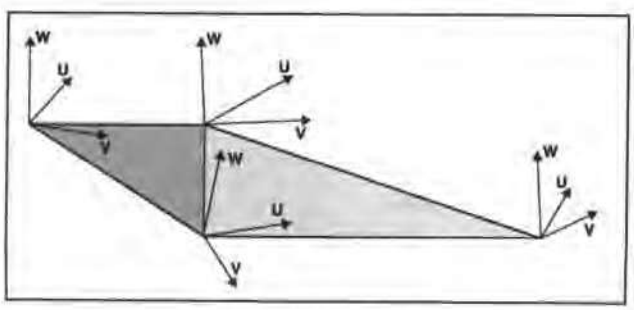


图 9.12 纹理空间坐标系

获取这种纹理空间坐标系最简单的方法就是使用 Direct3D 实用程序库 Direct3DX 提供的 D3DXComputeNormals()与 D3DXComputeTangent()函数。在 NVMeshMender 库中可以找到这些函数所涉及功能的源代码实现，NVMeshMender 库可以到 NVIDIA 网站（http://developer.nvidia.com）下载。计算纹理空间坐标系与计算顶点法线非常相似。举例来说，若 3 个三角形共享一个顶点，先计算面法线/面切线，然后将这 3 个三角形的向量相加以形成顶点法线/顶点切线。在当前示例程序的顶点 shader 中，对于纹理空间坐标系中的 UVW 向量分别采取：使用顶点法线代表 W 向量；借助于 D3DXComputeTangent()函数来计算 U 向量；使用 cross()函数计算 W 和 U 向量的叉积来获取 V 向量。

```
struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float2 Tex : TEXCOORD0;
    float3 Light : TEXCOORD1;
    float3 View : TEXCOORD2;
};

VS_OUTPUT VS(float4 Pos : POSITION, float2 Tex : TEXCOORD, float3 Normal : NORMAL, float3
             Tangent : TANGENT )
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    Out.Pos = mul(Pos, matWorldViewProj); // transform Position

    // compute the 3x3 transform matrix
    // to transform from world space to tangent space
    float3x3 worldToTangentSpace;
    worldToTangentSpace[0] = mul(Tangent, matWorld);
    worldToTangentSpace[1] = mul(cross(Tangent, Normal), matWorld);
    worldToTangentSpace[2] = mul(Normal, matWorld);

    Out.Tex = Tex.xy;

    Out.Light.xyz = mul(worldToTangentSpace, vecLightDir);    // L
    float3 PosWorld = normalize(mul(Pos, matWorld));
    float3 Viewer = vecEye - PosWorld;                        // V
    Out.View = mul(worldToTangentSpace, Viewer);

    return Out;
}

float4 PS(float2 Tex: TEXCOORD0, float3 Light : TEXCOORD1,
          float3 View : TEXCOORD2) : COLOR
{
    float4 color = tex2D(ColorMapSampler, Tex); // fetch color map
    float3 bumpNormal = 2 * (tex2D(BumpMapSampler, Tex) - 0.5); // bump map

    float3 LightDir = normalize(Light); // L
    float3 ViewDir = normalize(View);   // V

    float4 diff = saturate(dot(bumpNormal, LightDir)); // diffuse comp.
```

```
    float shadow = saturate(4 * diff);  // compute self-shadowing term

    float3 Reflect = normalize(2 * diff * bumpNormal - LightDir); // R

    // gloss map in color.w restricts spec reflection
    float4 spec = min(pow(saturate(dot(Reflect, ViewDir)), 3), color.w);

    return 0.2 * color + shadow * (color * diff + spec);
}
```

在顶点 shader 中创建了一个由 U、V、W（= N）构成的 3x3 矩阵，并通过它将 L 和 V 变换到纹理空间中去。

与前例相比，像素 shader 的主要区别是使用了一张颜色贴图和一张凹凸贴图，通过 tex2D() 来访问它们。tex2D()函数被声明为 tex2D(s, t)，其中 s 是采样器对象、t 是一个 2D 纹理坐标。请参考 DirectX 9 文档以了解更多其他的采样器函数（比如 texCUBE(s, t)，它用来获取一张立方体贴图）。

贯穿整个像素 shader 都是使用凹凸贴图中的法线来取代顶点中的法线，像素 shader 对从法线贴图中取出的值施以偏移（-0.5）和缩放（* 2.0）。施以这样的偏移是必须的，因为法线贴图是以无符号纹理格式存储的，其中每个值都是在[0, 1]区间内，而存在这样的限制主要是为了兼容老式硬件。因此，必须将这些法线还原到有符号区间中去。

与前例相比，这个像素 shader 还限制了镜面反射只能发生在地球模型的水域范围中。这是借助于 min()函数和一张光泽贴图（gloss map）来完成的，其中光泽贴图储存在颜色贴图的 alpha 值中。min()被定义为 min(a, b)，它的作用是在 a 和 b 中选出较少者。

在 return 语句中，环境光这一项被替换成了一个来自于颜色贴图的颜色值，并减弱了该颜色值的强度。这样做是为了保证，即使自阴影项减弱了漫反射和镜面反射成分，至少仍可以看到一个较黑暗的地球。

9.3.6 点光（Point Light）

最后一个示例是 Example6，它位于 CD-ROM 的“Chapter9\Example6”目录下。Example6 在前例基础上再加上一个逐像素计算的点光源。与一个定向光源的平行光线不同，点光源的光线是从点所在位置均匀地向各个方向散播（见图 9.13）。

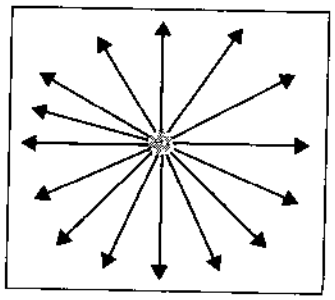


图 9.13 点光

在 NVIDIA 开发者网站上有一篇 Sim Dietrich 撰写的文章，其中提到了以下这个函数可以实现合理的衰减效果：

```
attenuation = 1 - d * d // d = distance
```

代码可表示为：

```
attenuation = 1 - (x * x + y * y + z * z)
```

Kelly Dempski 在他的 *Real-Time Rendering Tricks and Techniques in DirectX*（Premier Press，2002）中用距离的平方除以一个常量，来表示点光发生衰减效果的距离范围。

```
attenuation = 1 - ((x/r)² + (y/r)² + (z/r)²)
```

值 x/r、y/r、z/r 通过 TEXCOORDn 通道传递给像素 shader，并在像素 shader 中借助于 mul()函数将它们相乘。相关的源代码如下：

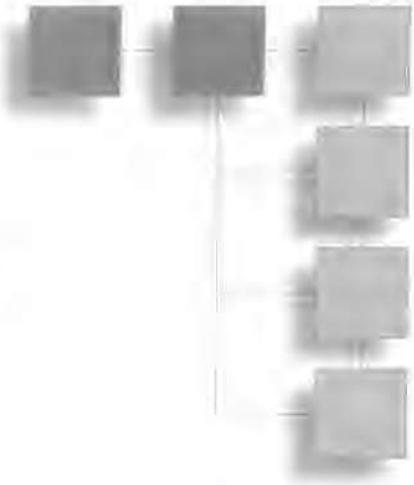
```
VS_OUTPUT VS(float4 Pos : POSITION, float2 Tex : TEXCOORD,
float3 Normal : NORMAL, float3 Tangent : TANGENT )
{
    ...
    float LightRange = 1.0;

    // point light
    Out.Att = Light * LightRange;
    ...
}

float4 PS(float2 Tex: TEXCOORD0, float3 Light : TEXCOORD1,
float3 View : TEXCOORD2, float3 Att : TEXCOORD3) : COLOR
{
    ...
    // attenuation
    float4 Attenuation = mul(Att, Att);

    // colormap * (self-shadow-term * diffuse + ambient) + self-shadow-term * // specular
    return (color * (shadow * diff + ambient) + shadow * spec) * (1 -Attenuation);
}
```

降低 LightRange 参数可增加光照的范围，而增加这个值则会使光照范围缩小。在像素 shader 中衰减值与自身相乘，因为这样做比使用指数二次方更有效率。像素 shader 的最后一行将衰减值乘以光照计算的结果，从而便可根据光源的距离来增加或减少光照的强度。

9.4　小结

本章介绍了 HLSL 的语法以及它的一些内部函数，其中展示了如何实现一些常用的光照公式。为了使你有能力可以开始编写自己的 shader，本章还引领你浏览了 6 个示例程序中的关键代码（都很简明易懂）。若期望更进一步地学习，我推荐你看一下 ATI 网站（http://www.ati.com）上 RenderMonkey 的示例，并在这些源代码中做进一步地研究。你也可以在我的网站（http://www.direct3d.info）上找到一个交互式的 RenderMonkey 教学指南。

第 10 章



更多高级的 Shader 效果

这一章将关注于一项称为“立方体环境映射（cube environment mapping）”的技术，它用于在物体的表面上反射其周围环境。当今大多数显卡都支持这项技术，并且它也将进一步地流行起来。理解这一技术背后的思想会为你打开许多扇通向高级游戏技术的大门，而这些技术都将在未来的几年中被广泛使用。

本章还将介绍一种阴影技术，在未来几年中这种阴影技术将会作为一项标准，用于创建场景中动态阴影的逼真描述。

在本章中，你将学到如何：

- 生成、访问和实现立方体贴图；
- 使用动态的折射/反射环境映射；
- 使用阴影和阴影体（shadow volume）。

10.1 使用立方体贴图

迄今为止在示例中尚未提及反射的概念。而若要创建出真实照片级效果的“虚拟环境”，则要求其中的表面能像它们在自然世界中那样反射光线。因此有时术语“反射映射（reflection mapping）”也就意味着“环境映射（environment mapping）”。

真实物体的表面在某种角度上对于光具有吸收、反射、传播三种不同的特性。为了再现这种自然物体的视觉丰富性，就必须实时地精确模拟出光在物体上的反射，并且没有令人不快的视觉错误。在光的物理行为建模中所涉及到的复杂性在于显式地跟踪遍及整个场景的次要（secondary）光线，一般有一些可选用的技术来模拟真实的反射。环境映射是一种高效的计算反射的技术，它在计算机合成的环境中做相关计算，并且不考虑那些无数的次要光线。

通过将一系列预先定义的图像塑造成包裹着物体的几何贴图形状，便可以在空间中为一个点定义出其位置上的反射。最常见的包裹着物体的预定义图像称为“立方体环境贴图”，或者简称为“立方体贴图”，所有新近的显卡都支持这种贴图。

10.1.1　生成立方体贴图

在概念上，立方体贴图就是以一个立方体的 6 个面来表现环境，这个立方体用于包裹场景中的一个或多个物体。生成立方体贴图的方法是，将照相机放置到需要实现反射的物体位置上，然后分别在 6 个方向上拍摄快照（正 x、负 x、正 y、负 y、正 z、负 z）。每张快照都应具有 90 度的视域（field-of-view）和正方形的高宽比。如此，将这些图像作为立方体贴图的 6 个面，然后将立方体的 6 个面紧密地缝合起来便可以创建出一张全方位的全景图（见图 10.1）。

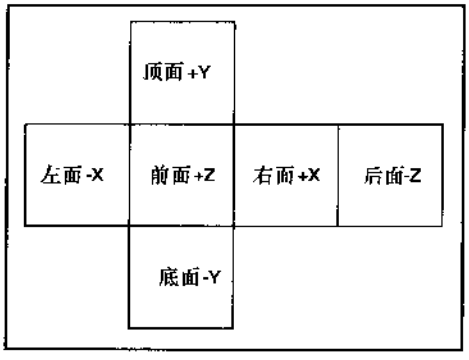


图 10.1　一张展开的立方体环境贴图，标识出了它的各个面

在本章前两个示例中，生成立方体贴图的过程是作为一个预处理步骤进行的；而后两个示例程序中，生成过程是由应用程序动态完成的。

10.1.2　访问立方体贴图

访问一张立方体贴图与创建一张立方体贴图非常相似。试着将立方体贴图的纹理坐标想象成一个 3D 向量，该向量由立方体的中心指向周围的面，那么该向量与立方体某个面的交点就是一个对应于该纹理坐标的图素。借助于一个入射线向量来创建一个访问立方体贴图的向量 R，该向量与眼睛向量（从顶点到眼睛的向量）相反。请注意图 10.2 中的反射向量与前一章的 Phong 方程式中所使用的向量并不一样。它是按以下公式计算的，其中 I 是一个从眼睛到顶点的向量（与眼睛向量相反）。

```
R = I - 2 * N * (I,N)
```

在许多网站和书籍中都可以找到关于这个 R 向量的数学依据，其中一个几何依据如图

10.3 所示。

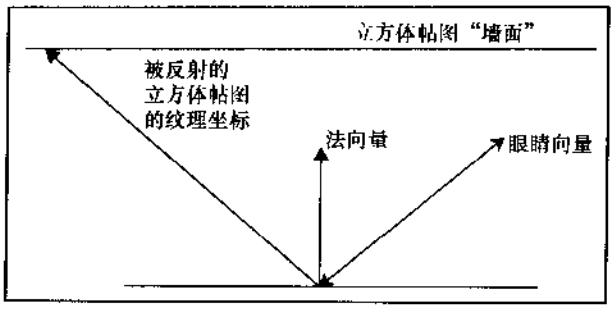


图 10.2　一条反射到立方体贴图上的射线描述

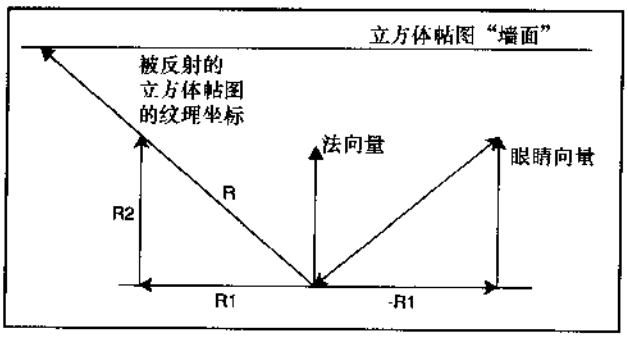


图 10.3　反射向量

```
R = R1 + R2
R1 = I - R2
```

可得：

```
R = I - 2 * R2
R2 = N * (I.N)
```

可得：

```
R = I - 2 * N * (I.N)
```

10.2　实现

第一个示例程序名为 CubeMap，位于 CD-ROM 的“Chapter10\CubeMap”目录下。该示例以一种最简单的形式演示了环境映射，它展示了一个由环境贴图营造出的具有铬一样外表的茶壶（见图 10.4）。

图 10.4　立方体环境映射

该示例是这种技术最直白的表现形式，然而它已经可以产生很漂亮的结果了。在这个示例中，顶点 shader 计算了入射向量和反射向量，并且将反射向量传递给像素 shader。像素 shader 从立方体环境贴图中取出值，并将其作为输出值使用。

```
float4x4 matWorldViewProj;
float4x4 matWorld;
float4x4 matView;
float4 vecEye;
texture CubeMap;

samplerCUBE CubeMapSampler = sampler_state
{
    Texture = <CubeMap>;
    MinFilter = Linear;
    MagFilter = Linear;
    AddressU = Wrap;
    AddressV = Wrap;
};

struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float3 Reflect: TEXCOORD4;
};

VS_OUTPUT VS(float4 Pos : POSITION, float2 Tex : TEXCOORD, float3 Normal : NORMAL)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    Out.Pos = mul(Pos, matWorldViewProj); // transform Position
    float3 Norm = normalize(mul(Normal, matWorld));

    // get a vector toward the camera/eye -> V
    float3 PosWorld = normalize(mul(Pos, matWorld));
    float3 Incident = normalize(PosWorld - vecEye);
```

```
    // Reflection Vector for cube map: R = I - 2 * N * (I.N)
    Out.Reflect = normalize(reflect(Incident, Norm));
    // Out.Reflect = normalize(Incident - 2 * Norm * dot(Incident, Norm));

    return Out;
}

float4 PS(float3 Ref : TEXCOORD4) : COLOR
{
    return texCUBE(CubeMapSampler, Ref);
}
```

与前一章中的示例一样，顶点位置通过应用程序提供的世界、观察和投影矩阵来变换，法线则通过世界矩阵来变换。入射线 I 由顶点位置减去眼睛位置计算而得。归一化的反射向量则借助于通用高级语言编译器提供的 reflect()函数计算而得，此函数的实现大致如下：

```
float3 reflect(float3 I, float3 N)
{
    return I - 2 * N * dot(I, N);
}
```

给定入射线方向 I 和表面法线 N，reflect()返回反射向量 R。在像素 shader 中，函数 texCUBE()从一个立方体贴图中采样一个图素，并将其作为输出，其中第 2 个参数为 3D 查找向量，这里将其指定为顶点 shader 中计算出的反射向量。

10.3 折射/反射环境映射

在学习了如何实现基本的环境映射后，现在我准备在前例基础上添加一种折射效果。在真实世界中，当光射线穿过一种介质到达另一种介质时，在这两种介质的相接处光射线会有稍许的弯曲，这种效果称为“折射”。最常见的折射例子就是当你站立在水边，看着水面下的一条鱼时，从鱼儿身上反射回来的光射线在离开水面时发生了弯曲，并且反射光还会随着水面形状的变化产生不同的弯曲。图 10.5 显示了这种由示例程序渲染出的近似效果。

CubeMap 2 示例程序位于 CD-ROM 的“Chapter10\CubeMap 2”目录下，它显示了一个具有部分反射和部分折射的茶壶。在这个示例中，折射效果近似于反射，它重用了反射方程式和一个经过缩放的法向量。你可以在 Kelly Dempski 的 *Real-Time Rendering Tricks and Techniques in DirectX*（Premier Press，2002）中读到更多关于该主题的内容。将法线缩短后会导致一个弯曲的折射向量，它看上去就像是眼睛向量的延长。

如果法线的长度缩短至 0，折射向量将与向量 I 相同。如果法线稍长一些，折射向量相对于表面就弯曲得更多。

以下像素 shader 读取了两次立方体贴图，第一次使用反射向量作为 3D 纹理坐标，第二次使用一个近似的折射向量作为 3D 纹理坐标。

图 10.5　折射/反射环境映射

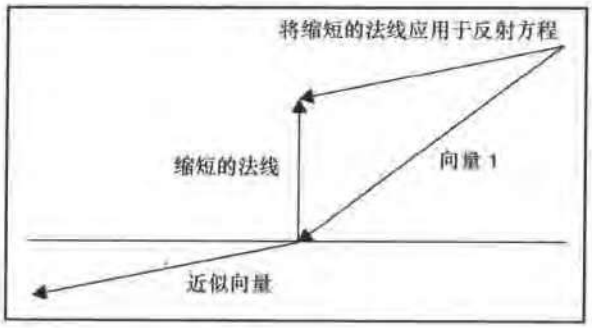


图 10.6　折射向量

```
struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float3 Reflect: TEXCOORD4;
    float3 Reflect2: TEXCOORD5;
};

VS_OUTPUT VS(float4 Pos : POSITION, float2 Tex : TEXCOORD, float3 Normal : NORMAL)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    Out.Pos = mul(Pos, matWorldViewProj); // transform Position
    float3 Norm = normalize(mul(Normal, matWorld));

    // get a vector toward the camera/eye -> V
    float3 PosWorld = normalize(mul(Pos, matWorld));
    float3 Incident = normalize(PosWorld - vecEye);

    // Reflection Vector for cube map: R = I - 2*N * (I.N)
    Out.Reflect = normalize(reflect(Incident, Norm));

    float3 ShortNorm = mul(Norm, 0.4);
```

```
    // Reflection Vector for cube map: R = I - 2*N * (I.N)
    Out.Reflect2 = normalize(reflect(Incident, ShortNorm));

    return Out;
}

float4 PS(float3 Ref : TEXCOORD4, float3 Ref2 : TEXCOORD5) : COLOR
{
    float4 tex1 = texCUBE(CubeMapSampler, Ref);
    float4 tex2 = texCUBE(CubeMapSampler, Ref2);

    return tex2 * 0.5 + tex1;
}
```

在顶点shader中，法线被缩短后储存到变量ShortNorm中。然后将这个变量传递给reflect()函数来计算折射向量。使用一个较大的常量值（接近1.0）会使结果看上去更接近于反射；而较小的值（接近0）则会使结果更接近于向量I，由此可产生一种缩放的效果。你可以尝试一下这些不同的值，看看各种不同的效果。

反射值和折射值随后被传到像素shader中，像素shader用它们分两次读取立方体贴图。然后，将通过折射向量读取到的立方体贴图的颜色值乘以0.5，以此来降低颜色的亮度。最后将这个值与通过反射向量读取到的颜色值相加。

10.3.1 动态折射/折射环境映射

以下示例（CubeMap 3，位于CD-ROM的“Chapter10\CubeMap 3”目录下）的主要改进在于立方体贴图的更新过程是动态的。也就是说，立方体贴图的内容每一帧都在变化。我使一架飞机围绕着茶壶飞行来突出这种动态更新的效果。茶壶上的反射是通过将立方体贴图投影到茶壶上而实现的。立方体贴图中的飞机和其他大部分内容的渲染代码都来自于DirectX 9 SDK的CubeMap示例（见图10.7）。

图10.7 动态立方体贴图

每帧都使用以下函数来动态地更新一张立方体贴图：

```
HRESULT CMyD3DApplication::RenderSceneIntoEnvMap()
{
    HRESULT hr;

    // Set the projection matrix for a field of view of 90 degrees
    D3DXMATRIXA16 matProj;
    D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI * 0.5f, 1.0f, 0.5f, 1000.0f );

    // Get the current view matrix, to concat it with the cubemap view vectors
    D3DXMATRIXA16 matViewDir( m_matView );
    matViewDir._41 = 0.0f; matViewDir._42 = 0.0f; matViewDir._43 = 0.0f;

    // Render the six cube faces into the environment map
    if( m_pCubeMap )
      hr = m_pRenderToEnvMap->BeginCube( m_pCubeMap );

    if(FAILED(hr))
      return hr;

    for( UINT i = 0; i < 6; i++ )
    {
       m_pRenderToEnvMap->Face( (D3DCUBEMAP_FACES) i, 0 );

       // Set the view transform for this cubemap surface
       D3DXMATRIXA16 matView;
       matView = D3DUtil_GetCubeMapViewMatrix( (D3DCUBEMAP_FACES) i );
       D3DXMatrixMultiply( &matView, &matViewDir, &matView );

       // Render the scene (except for the teapot)
       RenderScene( &matView, &matProj, FALSE );
    }

    m_pRenderToEnvMap->End( 0 );
    return S_OK;
}
```

源代码首先通过 D3DXMatrixPerspectiveFovLH()函数来设置投射矩阵，其中照相机具有 90 度的视域和正方形的高宽比。接下来两行代码将照相机定位到场景的中央。然后将照相机分别旋转至立方体贴图的 6 个方向上，并将场景分别渲染到立方体贴图的 6 个面上。函数 RenderScene()完成了这样的渲染，它不仅用于将场景渲染到立方体贴图上，还用于将场景渲染到输出窗口上。布尔变量 bRenderTeapot 用于防止将茶壶自身渲染到立方体贴图上。

```
HRESULT CMyD3DApplication::RenderScene( CONST D3DXMATRIXA16 *pView,
                CONST D3DXMATRIXA16 *pProject, BOOL bRenderTeapot )
{
    // Render the Skybox
    {
        D3DXMATRIXA16 matWorld;
        D3DXMatrixScaling( &matWorld, 10.0f, 10.0f, 10.0f );

        D3DXMATRIXA16 matView(*pView);
```

```
        matView._41 = matView._42 = matView._43 = 0.0f;

        m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
        m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
        m_pd3dDevice->SetTransform( D3DTS_PROJECTION, pProject );

        m_pd3dDevice->SetTexture(0, m_pCubeMap);
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
        m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR );
        m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR );
        if( (m_d3dCaps.TextureAddressCaps & D3DPTADDRESSCAPS_MIRROR)
              == D3DPTADDRESSCAPS_MIRROR )
        {
            m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_MIRROR );
            m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_MIRROR );
        }

        // Always pass Z-test, so we can avoid clearing color and depth buffers
        m_pd3dDevice->SetRenderState( D3DRS_ZFUNC, D3DCMP_ALWAYS );
        m_pSkyBox->Render( m_pd3dDevice );

        m_pd3dDevice->SetRenderState( D3DRS_ZFUNC, D3DCMP_LESSEQUAL );
    }

    // Render the Airplane
    {
        m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matAirplane );
        m_pd3dDevice->SetTransform( D3DTS_VIEW, pView );
        m_pd3dDevice->SetTransform( D3DTS_PROJECTION, pProject );

        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
        m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
        m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR );
        m_pd3dDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR );
        m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP );
        m_pd3dDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP );

        m_pAirplane->Render( m_pd3dDevice );

        m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matWorld);
    }

    // Render the environment-mapped ShinyTeapot
    if( bRenderTeapot )
    {
        //Rotate the object
        D3DXMatrixRotationY(&m_matWorld, (float)GetTickCount() / 1000.0f);

        D3DXMATRIX mWorldViewProj;
        D3DXMatrixMultiply(&mWorldViewProj,&m_matWorld, pView);
        D3DXMatrixMultiply(&mWorldViewProj, &mWorldViewProj, pProject);
        m_pEffect->SetMatrix( "matWorldViewProj", &mWorldViewProj );
        m_pEffect->SetMatrix( "matWorld", &m_matWorld);
        m_pEffect->SetMatrix( "matView", pView);
        m_pEffect->SetVector( "vecEye", &m_vEyePos);
```

```
        UINT nPasses;
        UINT iPass;
        if( m_pEffect != NULL )
        {
            D3DXHANDLE hTechnique = m_pEffect->GetTechniqueByName( "TShader" );
            m_pEffect->SetTechnique( hTechnique );

            m_pEffect->SetTexture("CubeMap", m_pCubeMap);
            m_pEffect->Begin( &nPasses, 0 );

            for( iPass = 0; iPass < nPasses; iPass ++ )
            {
                m_pEffect->Pass( iPass );
                m_pD3DXMesh->DrawSubset( 0 );
            }
            m_pEffect->End();
        }
    }
    return S_OK;
}
```

环境和飞机是通过固定功能的流水线渲染的。环境（在此指天空盒、飞机、茶壶）使用了它们各自的变换矩阵，立方体贴图在场景的中央进行渲染。飞机围绕着茶壶飞行，它是通过实用程序库文件 d3dfile.cpp 中的 Render()函数完成渲染的。此处茶壶的渲染是借助于一个效果（effect）文件中的顶点 shader 与像素 shader 完成的，这些 shader 就像本书中其他所有的 shader 示例一样。

当在 Render()函数中调用 RenderScene()时，将变量 bRenderTeapot 设为 TRUE；当在 RenderSceneIntoEnvMap()函数中调用 RenderScene()时，将变量 bRenderTeapot 设为 FALSE。

10.3.2　凹凸的动态折射/反射环境映射

CubeMap 4 示例将前一章凹凸映射示例中的功能与本章上一个展示动态反射/折射的示例结合了起来。CubeMap 4 示例位于 CD-ROM 的“Chapter10\CubeMap 4”目录下。将凹凸映射与动态更新的立方体贴图相结合引出了一段相当长的 shader。

```
struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float2 Tex : TEXCOORD0;
    float3 Light : TEXCOORD1;
    float3 View : TEXCOORD2;
    float3 Reflect: TEXCOORD3;
    float3 Reflect2: TEXCOORD4;
};

VS_OUTPUT VS(float4 Pos : POSITION, float2 Tex : TEXCOORD, float3 Normal : NORMAL,
             float3 Tangent : TANGENT )
```

```
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    Out.Pos = mul(Pos, matWorldViewProj); // transform Position

    // compute the 3x3 transform matrix
    // to transform from world space to tangent space
    float3x3 worldToTangentSpace;
    worldToTangentSpace[0] = mul(Tangent, matWorld);
    worldToTangentSpace[1] = mul(cross(Tangent, Normal), matWorld);
    float3 Norm = mul(Normal, matWorld);
    worldToTangentSpace[2] = Norm;

    Out.Tex = Tex.xy;

    // output light vector
    float3 Light = normalize(vecLightDir);
    Out.Light.xyz = mul(worldToTangentSpace, Light);

    // get a vector toward the camera/eye -> V
    float3 PosWorld = normalize(mul(Pos, matWorld));
    float3 Incident = normalize(PosWorld - vecEye);

    // view vector for bump map
    float3 Viewer2 = normalize(vecEye - PosWorld);
    Out.View = mul(worldToTangentSpace, Viewer2);

    // Reflection Vector for cube map: R = I - 2*N * (I.N)
    Out.Reflect = normalize(reflect(Incident, Norm));

    float3 ShortNorm = mul(Norm, 0.4);

    // Reflection Vector for cube map: R = I - 2*N * (I.N)
    Out.Reflect2 = normalize(reflect(Incident, ShortNorm));

    return Out;
}

float4 PS(float2 Tex: TEXCOORD0, float3 Light : TEXCOORD1,
          float3 View : TEXCOORD2, float3 Ref : TEXCOORD3,
          float3 Ref2 : TEXCOORD4) : COLOR
{
    float4 color = tex2D(ColorMapSampler, Tex); // fetch color map
    float4 bumpNormal = 2 * (tex2D(BumpMapSampler, Tex) - 0.5); // fetch bump map
    float4 tex1 = texCUBE(CubeMapSampler, Ref);
    float4 tex2 = texCUBE(CubeMapSampler, Ref2);

    // light vector
    float3 LightDir = normalize(Light);
    float3 ViewDir = normalize(View);

    // diffuse component
    float4 diff = saturate(dot(bumpNormal, LightDir));

    // compute self-shadowing term
    float shadow = saturate(4 * diff);

    // compute R vector
    float3 Reflect = normalize(2 * diff * bumpNormal - LightDir);
```

```
    // gloss map in color.w used to restrict specular reflection on non-glossy regions
    float4 spec = min(pow(saturate(dot(Reflect, ViewDir)), 15), color.w);

    return 0.2 * color + shadow * ((color * diff + spec) + ((tex2 * 0.5 + tex1) * 0.05));
}
```

在顶点 shader 的起始处，借助于 Tangent 向量、Normal 向量以及两者叉积的结果 Binormal 向量创建了一个纹理空间坐标系。然后将构造出来的 3×3 矩阵用于在切线空间中变换光向量 L 和观察者向量 V。另外，反射向量和折射向量是在世界空间中传给像素 shader 的。return 语句对你来说已经很熟悉了，最终结果就是将反射/折射示例的结果与凹凸映射示例的结果简单地相加起来。这个示例提供了几个可供调节的参数，你可以尝试使用不同的值来看看不同的效果。图 10.8 展示了该示例的屏幕截图。

图 10.8　凹凸的反射/折射立方体映射

你可以尝试修改一下这个示例，比如给予反射/折射部分更大的权重，或者只使用自阴影项来显示漫反射与镜面反射光照成分，而不使用反射/折射成分。

以下这些提示可以帮助你创建出更有效率的立方体环境贴图的实现：

- 对所有物体都使用同一张环境贴图；
- 尽可能使用静态的立方体贴图来代替动态的立方体贴图；
- 如果需要一张动态的立方体贴图，可以考虑每 5 帧生成一次；
- 不要在平坦的反射表面上使用立方体贴图；
- 基于距离来开关立方体映射。

10.4　使用阴影

光照可以在场景中营造出一种气氛，阴影则可以更进一步地烘托这种气氛，而且阴影还

能赋予物体更为逼真的视觉效果。阴影是创建逼真图像的重要元素，它能给予用户一种关于物体方位的视觉暗示。此外，阴影在游戏设计中也是一种非常重要的技法。例如，在游戏中玩家可能是扮演一个小偷，那么他为了隐藏自己，就有可能需要躲藏到阴影中去。

与在场景中创建和应用逼真、恰当的光照一样，渲染出令人信服的阴影也同样是一项极具挑战性的工作。虽然有几种截然不同的技术可用于创建阴影，但这些技术剖析阴影所使用的描述方法都是一样的。一个具有阴影的场景需要有一个用于逐顶点或逐像素计算的光源、一个抛投阴影的物体（称为“遮光板”）、以及一个被投射阴影的表面（称为“接受者”）。图 10.9 展示了阴影本身是由两个部分构成的：本影（umbra），这是阴影中央最黑暗的部分；半影（penumbra），这是由阴影的外部和边缘部分所构成的较柔和的一部分。

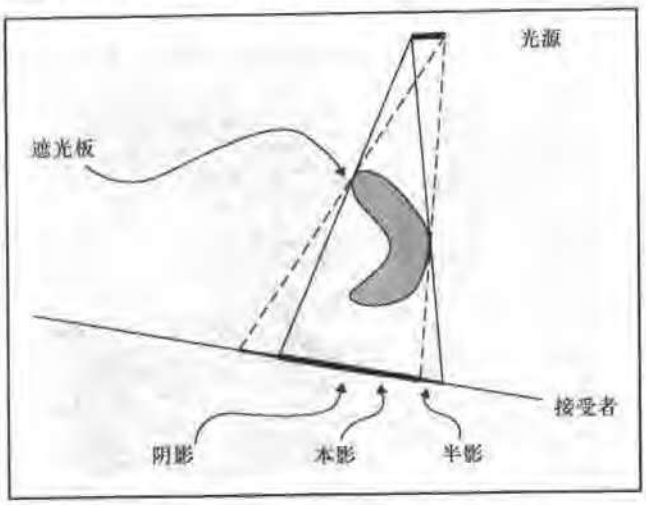


图 10.9　阴影术语：光源、遮光板、接受者、本影、半影

（授权自 Tomas Akenine-Möller/Eric Haines，*Real-Time Rendering,2nd edition* 授权）

半影造就了阴影中央最黑暗的部分与阴影周围较柔和的部分之间的渐变。若只有阴影中央最黑暗的那部分的话，阴影的周边会有一种突然终止的感觉，这样看起来会不太自然。然而，通过那部分较为柔和的阴影，即半影，它会从阴影中央的颜色（通常为黑色）逐渐转变为周边邻近像素的颜色，这样便可以产生更加逼真的效果。

在游戏中创建阴影有许多种方法。我将专注于介绍其中一种可在 DirectX 8 中使用的新方法，以及这种方法在 DirectX 9 显卡上的一种优化形式，即使用模板缓冲器（stencil buffer）的阴影体。

10.4.1　阴影体（Shadow Volume）

在接下来的两年里，阴影体应该会成为在游戏中创建阴影的首选技术。示例程序 Shadow Volume 位于 CD-ROM 的“Chapter10\Shadow Volume”目录下，它使用了在 DirectX 9 中引

入的双面模板缓冲器（two-sided stencil buffer），而在老式显卡上运行该示例时，它会退回到使用传统的单面模板缓冲器（见图 10.10）。

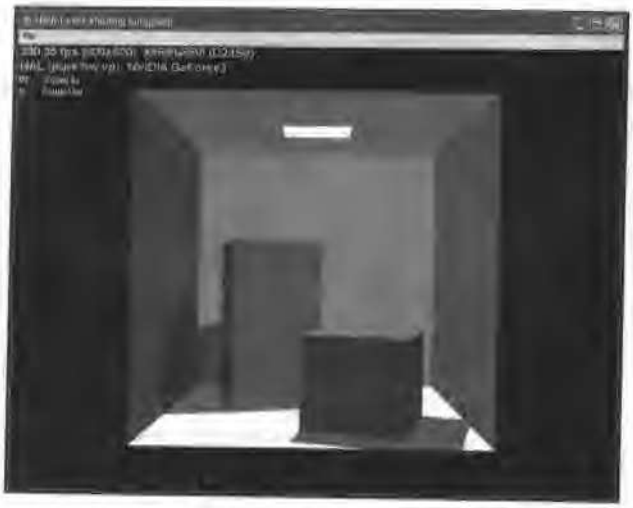


图 10.10　具有两个立方体的 Cornell 盒子，其中两个立方体都具有阴影

示例程序展示了一间放有两只盒子的房间。光源在天花板上，两只盒子都抛出了阴影。两只盒子的阴影各自占据了一个特定的空间体。所有在阴影体之外的东西都被照亮了，而所有在阴影体内的东西都是暗的。

图 10.11 是 ATI 的 RadeonShadowShader 示例，它展示了类似的场景，这个示例可以在 ATI 网站(http://www.ati.com/developer)上找到。其中，右图展示了来自于两个光源的两个阴影体。请留意一下光体（volume of the light）和阴影体的异同，右图中地面切断了所有的光线。

图 10.11　ATI 的 RadeonShadowShader 示例中的阴影体

实现阴影体的挑战之一便是要正确地切断光线。为了完成这项任务，需要从光源的角度来检测网格的轮廓边（silhouette edge）。轮廓边是指物体的一条边，这条边的特点是与它相连的多边形中只有一个多边形面对着光源，而其他的都背对光源。为了使顶点 shader 可以检测到这样的边，必须要先对源几何体进行预处理，如此才可能在任何方向上构造一个阴影体，

并且不需要临时创建新的几何体。

预处理步骤是用分裂四边形（degenerate quad）来替换物体中的共享边。图 10.12 描述了对一个立方体的预处理，为了简化讨论，这里仅显示了立方体前面的几个面。在预处理过程中，面的共享边被过滤掉了，在相同位置上取而代之的是分裂四边形。其中，每个分裂四边形对侧的两条边具有相同的位置值，但面法线不同。本章示例所使用的预处理技术也是 Imagire Takashi 在他的阴影体示例中所使用的，他的阴影体示例位于 http://www.t-pot.com。

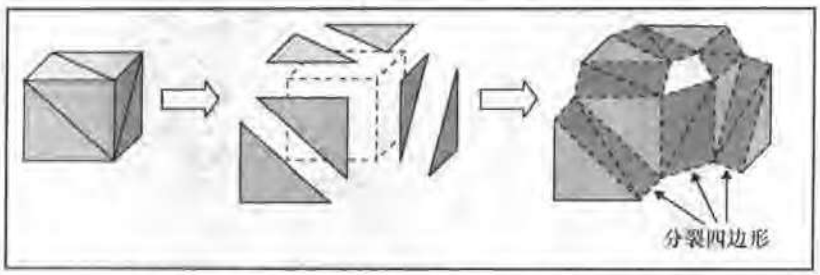


图 10.12　在一个平面上将共享边替换为分裂四边形

（Hun Yen Kwoon 授权图示）

我将从头至尾简要地介绍一下这个预处理算法：

（1）遍历源网格中所有的面；

（2）为每个面计算面法线；

（3）遍历每个面的 3 条边；

（4）将每条边插入到一张检验表中；

（5）如果一个面的 2 个顶点与其他面的 2 个顶点构成的是同一条边，则按以下 2 个步骤之一处理：

a．如果共享边的面法线不平行，则改变面的方向

或者

b．生成一个分裂四边形

首先，计算出所有面的面法线，以了解每个面的朝向。然后，搜索所有共享同一条边的顶点，即“如果一个面的 2 个顶点共用了另一个表面的 2 个顶点”，则插入一个分裂四边形。这个四边形将辅助顶点 shader 建立不依赖于光源位置的阴影体。

注意，这里所使用的预处理技术是不加区分地在每条共享边上插入分裂四边形。因此，为了减少多边形的总数，可以考虑检查一个共享边是否有机会成为一条轮廓边。如果一条共享边几乎没有可能成为一条轮廓边，那么就根本没有必要插入一个分裂四边形来替换它。有一种方法可用于判定一条边是否有可能成为轮廓的一部分，那就是测试共享边的面法线是否平行。如果两个面的法线几乎平行，则说明这条共享边位于一个平面上，因此它就几乎没有可能成为轮廓的一部分。

另外要注意，源网格应是一个封闭的网格。若要考虑缝隙或洞孔（比如由 T-junctions 形

成的）的话，则会使顶点 shader 中的轮廓判定算法变得非常复杂。并且，非封闭的网格还会使模板计算趋于不平衡。

在 Shadow Volume 示例程序的 CShadowVolume.cpp 文件中，Create()函数完成了这个预处理步骤。

```
HRESULT CShadowVolume::Create( LPDIRECT3DDEVICE9 pd3dDevice, LPD3DXMESH pSrcMesh )
{
    HRESULT ret = S_OK;
    SHADOW_VOLUME_VERTEX* pVertices;    // local copy of the vertex buffer
    WORD* pIndices;                     // local copy of the index buffer
    DWORD i, j, k, l, face;
    LPD3DXMESH pMesh;

    if( FAILED( pSrcMesh->CloneMeshFVF(D3DXMESH_SYSTEMMEM,
                                       D3DFVF_XYZ | D3DFVF_NORMAL,
                                       pd3dDevice, &pMesh ) ) )
      return E_FAIL;

    DWORD dwNumFaces = pMesh->GetNumFaces();

    // four times the number of faces and
    // three times the number of vertices.
    m_dwNumFaces = 4 * dwNumFaces;
    m_pVertices = new SHADOW_VOLUME_VERTEX[3*m_dwNumFaces];

    // lock buffers
    pMesh->LockVertexBuffer( 0L, (LPVOID*)&pVertices );
    pMesh->LockIndexBuffer ( 0L, (LPVOID*)&pIndices );

    // allocate space for normal
    D3DXVECTOR3 *vNormal = new D3DXVECTOR3[dwNumFaces];
    if(NULL==vNormal)
    {
        m_dwNumFaces = 0;
        ret = E_OUTOFMEMORY;
        goto end;
    }

    // compute face normals for each face
    for( i=0; i < dwNumFaces; i++ )
    {
        D3DXVECTOR3 v0 = pVertices[pIndices[3*i+0]].p;
        D3DXVECTOR3 v1 = pVertices[pIndices[3*i+1]].p;
        D3DXVECTOR3 v2 = pVertices[pIndices[3*i+2]].p;

        // calculate normal
D3DXVECTOR3 vCross1(v1-v0);
        D3DXVECTOR3 vCross2(v2-v1);
        D3DXVec3Cross( &vNormal[i], &vCross1, &vCross2 );

        // store faces
        m_pVertices[3*i+0].p = v0;
        m_pVertices[3*i+1].p = v1;
        m_pVertices[3*i+2].p = v2;
```

```
            m_pVertices[3*i+0].n = vNormal[i];
            m_pVertices[3*i+1].n = vNormal[i];
            m_pVertices[3*i+2].n = vNormal[i];
    }

        // step through the three edges of each face
        face = dwNumFaces;
        for( i = 0 ; i < dwNumFaces; i++ )
        {
            for( j = i + 1; j < dwNumFaces; j++ )
            {
                DWORD id[2][2]; // checklist
                DWORD cnt=0;

                for(k = 0; k < 3; k++)
                {
                    for(l = 0; l < 3; l++)
                    {
                        D3DXVECTOR3 dv;
                        D3DXVec3Subtract( &dv, &pVertices[pIndices[3*i+k]].p,
                                               &pVertices[pIndices[3*j+l]].p);

                        // if two vertices from two different faces form one edge
                        // put them into checklist and increment counter
                        if( D3DXVec3LengthSq( &dv ) < 0.001f )
                        {
                            // cnt counts until two …
                            id[cnt][0] = 3*i+k;
                            id[cnt][1] = 3*j+l;
                            cnt++;
                        }
                    }
                }

                // if two times two vertices from two faces share one edge …
                if(2 == cnt)
                {
                    // if one face has a different direction than the other face …
                    // alternative: check if the face normals are parallel,
                    // if not insert degenerate quad
                    // else exit if statement
                    if(id[1][0] - id[0][0]!=1)
                    {
                        // … adjust direction of face
                        DWORD tmp = id[0][0];
                        id[0][0] = id[1][0];
                        id[1][0] = tmp; // swap id[0][0] and id[1][0]
                        tmp = id[0][1];
                        id[0][1] = id[1][1];
                        id[1][1] = tmp; // swap id[0][1] and id[1][1]
                    }

                    // insert degenerated quadrilateral
                    // the face normals are used for the vertex normals
                    m_pVertices[3*face+0].p = pVertices[pIndices[id[1][0]]].p;
                    m_pVertices[3*face+2].p = pVertices[pIndices[id[0][1]]].p;
```

```
                    m_pVertices[3*face+1].p = pVertices[pIndices[id[0][0]]].p;
                    m_pVertices[3*face+0].n = vNormal[i];
                    m_pVertices[3*face+2].n = vNormal[j];
                    m_pVertices[3*face+1].n = vNormal[i];
                    face++;
                    m_pVertices[3*face+0].p = pVertices[pIndices[id[1][0]]].p;
                    m_pVertices[3*face+2].p = pVertices[pIndices[id[1][1]]].p;
                    m_pVertices[3*face+1].p = pVertices[pIndices[id[0][1]]].p;
                    m_pVertices[3*face+0].n = vNormal[i];
                    m_pVertices[3*face+2].n = vNormal[j];
                    m_pVertices[3*face+1].n = vNormal[j];
                    face++;
                }
            }
        }
        assert(face == m_dwNumFaces);

        delete[] vNormal;

    end:
        // unlock buffers
        pMesh->UnlockVertexBuffer();
        pMesh->UnlockIndexBuffer();

        pMesh->Release();
        return ret;
    }
```

原始几何体的局部副本储存在 pVertices 中，全局副本储存在 m_pVertices 中，后者保存最终阴影体的几何形状。在计算完面法线后，再在两个三角形之间搜索任意两条重合的边。如果找到了两条重合的边，则进一步检查面的方向。如果一个面的方向与另一个面不同，则改变这个面的方向。

现在拥有了一个扩展的几何体，那么接下去的问题便是“如何渲染这个阴影体？”。答案是，借助于模板缓冲器。先考虑一个像素的情况，假设照相机不在阴影体内，那么该像素所对应的点在场景中就有 4 种可能性：如果从照相机到这一点的射线没有与阴影体相交，那么阴影多边形就不会被绘制到该像素上，并且模板缓冲器将仍为 0；如果该点位于阴影体的前方，那么在绘制阴影多边形时 z 缓冲测试失败，模板仍旧保持不变；如果该点位于阴影体的后方，那么将会渲染出相同数量的前阴影面与后阴影面，并且模板也将在经历了相同次数的递增和递减后变为 0；最后一种可能性就是该点位于阴影体的内部，在这种情况中，绘制阴影体后方的面时 z 缓冲测试失败（但前方的面不会失败），因此模板缓冲器将变为一个非 0 值，所以结果便是帧缓冲器中位于阴影内的那一部分的模板值将为非 0。为了实现这样的功能，示例程序完成了以下几个步骤。

（1）清除渲染目标（颜色缓冲器）、z 缓冲器、模板缓冲器。

（2）将场景渲染到颜色缓冲器和 z 缓冲器中。

（3）禁用写颜色缓冲器和 z 缓冲器，激活模板缓冲器，并打开 Flat 着色模式。

（4）当模板测试通过或失败时，保持模板缓冲器不变；当深度测试失败时，递增模板缓

冲器。

（5）渲染模板缓冲器。

a. 如果支持双面模板缓冲器，则打开它：

i）当深度测试失败时，递减模板缓冲器的第二面；

ii）关闭背面剔除；

iii）渲染阴影体，填充双面模板缓冲器，然后跳到步骤 6。

b. 如果不支持双面模板缓冲器：

i）渲染所有面朝前方的多边形；

ii）颠倒剔除次序，这样朝后的多边形（背面）也可以被写入模板缓冲器；

iii）当深度测试失败时，递减模板缓冲器；

iv）渲染朝后的多边形。

（6）对每个光源重复步骤 4 和步骤 5。

（7）打开 Gouraud 着色，允许写颜色缓冲器和 z 缓冲器，停用模板缓冲器，关闭 z 缓冲测试。

（8）在模板值大于或等于 1 的位置上渲染阴影，从而在场景中渲染出阴影。

（9）在步骤 8 和步骤 2 中使用 alpha 混合（可选）。

对于这类使用模板缓冲器来实现体效果的技术而言，关键点都在于模板缓冲器与 z 缓冲器之间的相互作用。一个具有阴影体的场景是分三个阶段来渲染的。首先，像往常那样渲染不带阴影的场景，并使用 z 缓冲器。然后，在模板缓冲器中标记出阴影。在这个阶段中，使用不可见的多边形来绘制阴影体前方的面，并启用 z 缓冲测试，但禁止写 z 缓冲器，每当像素通过了 z 缓冲测试后递增该像素所对应的模板缓冲器中的值，接着再以类似的方式来渲染阴影体后方的面，不过此时是递减模板值。

前述一系列步骤中的步骤（1）和步骤（2）完成第一次渲染，这次是将场景渲染到颜色缓冲器和 z 缓冲器中。然后是设置模板缓冲器，由于在填充模板缓冲器时并不需要将颜色写入颜色缓冲器，所以也无需使用 Gouraud 着色，因此使用廉价的 Flag 着色即可。由于此时不需要渲染几何体，所以同时关闭 z 缓冲器。

当模板测试通过或失败时，应保持模板缓冲器中的值不变；当深度测试失败时，应递增模板缓冲器中的值。接下去根据对双面模板缓冲器的支持情况，来执行步骤（5）a 或步骤（5）b。如果支持双面模板缓冲器，则在深度测试失败时递减模板缓冲器第二面中的值。然后在关闭背面剔除后，依照模板缓冲器中两个面的内容来渲染阴影体。

如果显卡不支持双面模板缓冲器，则执行步骤（5）b。由于已在步骤（4）中设置了模板缓冲器的测试方式，所以在步骤（5）b 中可以直接将面朝前的多边形渲染到模板缓冲器中。由于传统的模板缓冲器没有背面，所以为了渲染朝后的多边形，需要颠倒剔除次序，并将模板缓冲器设置为当深度测试失败时递减模板值。然后，通过渲染朝后的面来第二次填充模板缓冲器。有多少光源就应该重复执行步骤（4）和步骤（5）多少次。

最后留在模板缓冲器中递增的值即对应处于阴影中的像素。通过将模板缓冲器作为掩

码，在场景中渲染出阴影，从而使处于阴影中的像素变暗。以下是源代码：

```
//-----------------------------
// Step 2: Render the scene into color and z-buffer
//-----------------------------
D3DXMatrixIdentity( &matWorld);
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld);
m_pMeshBG->Render( m_pd3dDevice );

// scale, rotate, translate and render the box on the right
D3DXMatrixScaling( &matScale, 1.82f,1.65f, 1.82f );
D3DXMatrixRotationY( &matRotate, 0.59f * D3DX_PI );
D3DXMatrixTranslation( &matTranslate, 2.73f-1.85f, 0.f ,1.69f);
matTemp = matScale * matRotate * matTranslate;
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matTemp );
m_pMeshBox->Render( m_pd3dDevice );

// scale, rotate, translate and render the box on the left
D3DXMatrixScaling( &matScale, 1.69f, 3.30f, 1.69f );
D3DXMatrixRotationY( &matRotate, 0.91f * D3DX_PI );
D3DXMatrixTranslation( &matTranslate, 2.73f-3.685f, 0, 3.51f );
matTemp = matScale * matRotate * matTranslate;
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matTemp);
m_pMeshBox->Render( m_pd3dDevice );

//-----------------------------
// -----------------------------
// Step 3: Disable color writes, z-buffer writes, activate
// stencil buffer, switch on flat shading mode
//-----------------------------
m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );

// prevent the update of the color buffer
m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, FALSE );

// switch on flat shading
m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_FLAT );

// switch on stencil buffer
m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE );

// The comparison function is used to compare the reference
// value to a stencil buffer entry. This comparison applies
// only to the bits in the reference value and stencil buffer
// entry that are set in the stencil mask (set by the
// D3DRS_STENCILMASK render state). If TRUE, the stencil test
// passes.
m_pd3dDevice->SetRenderState(D3DRS_STENCILFUNC, D3DCMP_ALWAYS);

// An int reference value for the stencil test. The default
// value is 0.
m_pd3dDevice->SetRenderState( D3DRS_STENCILREF, 0x1 );

// Mask applied to the reference value and each stencil buffer
// entry to determine the significant bits for the stencil
```

```
    // test. The default mask is 0xFFFFFFFF.
    m_pd3dDevice->SetRenderState( D3DRS_STENCILMASK, 0xffffffff);
    m_pd3dDevice->SetRenderState(D3DRS_STENCILWRITEMASK, 0xffffffff);

    //-----------------------------------
    // Step 4: pass stencil buffer, when the stencil test passes or
    // fails
    //-----------------------------------
    m_pd3dDevice->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_KEEP);
    m_pd3dDevice->SetRenderState( D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP);

    //-----------------------------------
    // Step 5: increment stencil buffer, when the depth test fails
    //-----------------------------------
    m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_INCR );

    //-----------------------------------
    // Step 6: if the two-sided stencil buffer is supported by
    // hardware
    //-----------------------------------
    if( ( m_d3dCaps.StencilCaps & D3DSTENCILCAPS_TWOSIDED ) != 0 )
    {
        // same for the second side of the stencil buffer ...
        m_pd3dDevice->SetRenderState( D3DRS_TWOSIDEDSTENCILMODE, TRUE );
        m_pd3dDevice->SetRenderState( D3DRS_CCW_STENCILFUNC, D3DCMP_ALWAYS );

        //-----------------------------------
        // Step 6. a): decrement the second side of the stencil
        // buffer, when the depth test fails
        //-----------------------------------
        m_pd3dDevice->SetRenderState( D3DRS_CCW_STENCILZFAIL, D3DSTENCILOP_DECR );

        //-----------------------------------
        // Step 6. b): switch off backface culling
        //-----------------------------------
        m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );

        //-----------------------------------
        // Step 6. c): Render shadow volume
        //-----------------------------------
        if( m_pEffect != NULL )
        {
            D3DXHANDLE hTechnique = m_pEffect->GetTechniqueByName( "TShader" );
            m_pEffect->SetTechnique( hTechnique );
            m_pEffect->Begin( NULL, 0 );
            m_pEffect->Pass( 0 );

            // scale, rotate and translate the shadow of the
            // box on the right
            D3DXMatrixScaling( &matScale, 1.82f,1.65f, 1.82f);
            D3DXMatrixRotationY( &matRotate, 0.59f * D3DX_PI);
            D3DXMatrixTranslation( &matTranslate, 2.73f-1.85f, 0.f , 1.69f );
            matWorld = matScale * matRotate * matTranslate;
            matTemp = matWorld * m_matView * m_matProj;
            m_pEffect->SetMatrix("matWorldViewProj",&matTemp);
            D3DXMatrixInverse( &matTemp, NULL, &matWorld);
```

```
            D3DXVec3Transform( &vVector, &m_LighPos,&matTemp);
            m_pEffect->SetVector("vLightPos", &vVector);
            m_pShadowBox->Render( m_pd3dDevice );

            // scale, rotate and translate the shadow of the
            // box on the left
            D3DXMatrixScaling( &matScale, 1.69f, 3.30f, 1.69f );
            D3DXMatrixRotationY( &matRotate, 0.91f * D3DX_PI );
            D3DXMatrixTranslation(&matTranslate, 2.73f-3.685f, 0, 3.51f );
            matWorld = matScale * matRotate * matTranslate;
            matTemp = matWorld * m_matView * m_matProj;
            m_pEffect->SetMatrix("matWorldViewProj",&matTemp);
            D3DXMatrixInverse( &matTemp, NULL, &matWorld);
            D3DXVec3Transform( &vVector, &m_LighPos, &matTemp );
            m_pEffect->SetVector("vLightPos", &vVector );
            m_pShadowBox->Render( m_pd3dDevice );

            m_pEffect->End();
        }
        m_pd3dDevice->SetRenderState( D3DRS_TWOSIDEDSTENCILMODE, FALSE );
    }

    //--------------------------------------
    // Step 7: else if two-sided stencil buffer is not supported
    // (non-DX9 hardware)
    //--------------------------------------
    else
    {
        //--------------------------------------
        // Step 7. a): render front-facing polygons
        //--------------------------------------
        if( m_pEffect != NULL )
        {
            D3DXHANDLE hTechnique = m_pEffect->GetTechniqueByName( "TShader" );
            m_pEffect->SetTechnique( hTechnique );
            m_pEffect->Begin( NULL, 0 );
            m_pEffect->Pass( 0 );

            // scale, rotate and translate the shadow of the box
            // on the right
            D3DXMatrixScaling( &matScale, 1.82f,1.65f, 1.82f );
            D3DXMatrixRotationY( &matRotate, 0.59f * D3DX_PI );
            D3DXMatrixTranslation( &matTranslate, 2.73f-1.85f, 0.f , 1.69f );
            matWorld = matScale * matRotate * matTranslate;
            matTemp = matWorld * m_matView * m_matProj;
            m_pEffect->SetMatrix("matWorldViewProj", &matTemp);
            D3DXMatrixInverse( &matTemp, NULL, &matWorld);
            D3DXVec3Transform( &vVector, &m_LighPos, &matTemp);
            m_pEffect->SetVector("vLightPos", &vVector);
            m_pShadowBox->Render( m_pd3dDevice );

            // scale, rotate and translate the shadow of the
            // box on the left
            D3DXMatrixScaling(&matScale, 1.69f, 3.30f, 1.69f);
            D3DXMatrixRotationY(&matRotate, 0.91f * D3DX_PI);
            D3DXMatrixTranslation(&matTranslate, 2.73f-3.685f, 0, 3.51f );
```

```
    matWorld = matScale * matRotate * matTranslate;
    matTemp = matWorld * m_matView * m_matProj;
    m_pEffect->SetMatrix("matWorldViewProj", &matTemp);
    D3DXMatrixInverse( &matTemp, NULL, &matWorld);
    D3DXVec3Transform( &vVector, &m_LighPos, &matTemp );
    m_pEffect->SetVector("vLightPos", &vVector );
    m_pShadowBox->Render( m_pd3dDevice );

    m_pEffect->End();
}

//-----------------------------------
// Step 7. b): reverse culling order so the back sides can be
// written to stencil buffer
//-----------------------------------
m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CW );

//-----------------------------------
// Step 7. c): decrement stencil buffer, when depth test fails
//-----------------------------------
m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_DECR );

//-----------------------------------
// Step 7. d): render back-facing polygons
//-----------------------------------
if( m_pEffect != NULL )
{
    D3DXHANDLE hTechnique = m_pEffect->GetTechniqueByName( "TShader" );
    m_pEffect->SetTechnique( hTechnique );
    m_pEffect->Begin( NULL, 0 );
    m_pEffect->Pass( 0 );

    // scale, rotate and translate the shadow of the box on
    // the right
    D3DXMatrixScaling( &matScale, 1.82f,1.65f, 1.82f );
    D3DXMatrixRotationY( &matRotate, 0.59f * D3DX_PI );
    D3DXMatrixTranslation(&matTranslate, 2.73f-1.85f, 0.f, 1.69f );
    matWorld = matScale * matRotate * matTranslate;
    matTemp = matWorld * m_matView * m_matProj;
    m_pEffect->SetMatrix("matWorldViewProj", &matTemp);
    D3DXMatrixInverse( &matTemp, NULL, &matWorld);
    D3DXVec3Transform( &vVector, &m_LighPos, &matTemp);
    m_pEffect->SetVector("vLightPos", &vVector);
    m_pShadowBox->Render( m_pd3dDevice );

    // scale, rotate and translate the shadow of the
    // box on the left
    D3DXMatrixScaling( &matScale, 1.69f, 3.30f, 1.69f );
    D3DXMatrixRotationY( &matRotate, 0.91f * D3DX_PI );
    D3DXMatrixTranslation(&matTranslate, 2.73f-3.685f, 0, 3.51f );
    matWorld = matScale * matRotate * matTranslate;
    matTemp = matWorld * m_matView * m_matProj;
    m_pEffect->SetMatrix("matWorldViewProj",&matTemp);
    D3DXMatrixInverse( &matTemp, NULL, &matWorld);
    D3DXVec3Transform(&vVector, &m_LighPos, &matTemp);
    m_pEffect->SetVector("vLightPos", &vVector );
```

```
            m_pShadowBox->Render( m_pd3dDevice );

            m_pEffect->End();
        }
    }
```

如果不支持新型的双面模板缓冲器，那么阴影几何体对于每个光源都要被绘制两次，这样做会降低 GPU 的顶点吞吐率。双面模板就是为了解决这样的问题而设计的，双面模板缓冲器的优点就在于它将这个处理过程减少到只需渲染一次。

通过执行以下步骤来渲染阴影体：

（1）从光源处投射出一条光射线；

（2）无论一个顶点是否朝向光源都计算出光射线与其法线的点积(L.N)；

（3）根据 L.N 的值来决定将 0 或 1 存储到变量 scale 中；

（4）稍微偏移一点阴影几何体的位置值，从而使阴影不会与物体的边缘相重叠；

（5）对光射线的方向进行缩放后，用位置值减去缩放后的结果（这个过程也称为“挤压”阴影几何体）。若阴影几何体的位置不是光源方向的延伸（scale == 0），则说明它比物体更小，即它被物体遮挡住了。

这段源代码非常简单。

```
VS_OUTPUT VS (float4 Pos: POSITION, float3 Normal : NORMAL)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;

    // light to vertex ray
    float4 dir = vLightPos - Pos;

    // dot ray and normal
    float LN = dot( Normal, dir );

    // scale is 0 for light facing or 1 for non-light facing
    float scale = (LN >= 0) ? 0.0f : 1.0f;

    // reduce position slightly of shadow volume geometry
    Pos.xyz -= 0.001f*Pos;

    // reduced position - scale * light direction
    Out.Pos = mul( Pos - scale * dir, matWorldViewProj );

    return Out;
}
```

通过 Pos.xyz 所在那行及其注释，可以看到如何来稍微地偏移一个位置。要实现这样的效果还有几种其他的方法。请参阅 Eric Lengyel 的文章“The Mechanics of Robust Stencil Shadows”来了解更多的做法，该文章位于 http://www.gamasutra.com/features/20021011/lengyel_pfv.htm。

10.4.2 使用阴影体所需考虑的问题

请注意由于这个示例程序中只有一个光源，所以只通过这一个光源来计算阴影。但这并不表示在其他具有多个光源的情况下，也只能从几个光源中选出某个特定的光源来计算阴影。另外，在实际应用中判定哪些对象应投射到可见区域时，为了减少产生阴影的数量，应仅考虑那些可视的物体。

为了进一步地改进这个示例，可以使用 Direct3D 的深度偏置（depth-biasing）能力，它可以克服因物体远离照相机时，而导致深度精度降低的问题。不过当你使用 W 键或 S 键来放大或缩小时，还是会导致一些比较明显的闪烁。DirectX 9.0 提供了 D3DRS_DEPTHBIAS 与 D3DRS_SLOPESCALEDEPTHBIAS 渲染状态，用于计算一个偏移量。这个偏移量被用于将阴影体的前盖（front capping）放置到遮光板中面朝前的几何体之后。这是通过将该偏移量加到片断（fragment）的内插深度值上完成的，相加结果最终被用于深度测试。可使用 D3DPRASTERCAPS_DEPTHBIAS 与 D3DPRASTERCAPS_SLOPESCALEDEPTHBIAS 标志来检查显卡是否支持这些新能力。如果不支持该功能，则可退回到使用传统的 D3DRS_DEPTHBIAS 渲染状态。

通过更有选择性地插入四边形，比如先前所述的检查法线的方向，可以减少阴影几何体数据集的大小。另外，一个低精度多边形的遮光板模型或许已经足以计算阴影几何体了，所以可以不用考虑如何来减少阴影几何体的数据量。

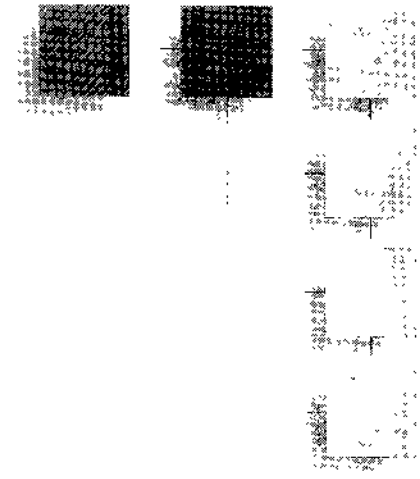
请记住，执行阴影体的一项重要要求，就是需要一个封闭体的网格。如果不是一个封闭体，则网格中的任何缝隙都有可能致使模板计数趋于不平衡，从而中断阴影体的执行。应对建模者和设计者提出这一要求，提醒他们注意其工作流程和建模风格，而不应依靠修改图形引擎来解决该问题。

另外，将阴影体与蒙皮网格（skinned mesh）结合使用是一件相当复杂的事情。Chris Brennan 在 *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*（Wordware Publishing，2002）中对此介绍了一种解决方法。

10.5 小结

本章涉及了两项非常重要的技术，它们已被广泛地应用于当前众多的游戏中，并且很有可能在未来几年中仍被继续使用。对于立方体贴图运作原理的基本理解有助于你去理解那些当前正在发展中的高级技术。模板阴影体技术则向你展示了如何使用模板缓冲器来标记出一个场景的某些区域，理解这一概念也将有助于你对许多其他技术的理解。

第 11 章



使 用 文 件

现今流行的 3D 文件格式有很多种。然而，你要能够区分出哪些文件格式是为美工所使用的，哪些是在游戏引擎中使用的。3D 工具软件所使用的文件格式有.3ds、.max、.cof 等，这些格式都是由哪些工具所产生的。它们包含了许多信息，并且都不为像游戏引擎这样的实时环境所优化。游戏需要使用更快、更简单的文件格式，根据游戏类型的不同，每个游戏使用的文件格式也各不相同。一般来说，都是一款游戏对应于一种文件格式。

在游戏生产过程中，为提高开发效率所需考虑的一个棘手问题便是：如何在 3D 图形美工与游戏引擎程序员之间交换图形信息，并为此建立一套合适的工作流程。解决该问题的典型做法就是编写一些专门的工具，这些工具可用于在游戏引擎的文件格式与 3D 图形软件的文件格式之间做导入和导出。

由于许多游戏公司都授权使用了 Quake 引擎，因此 id Software 的 Quake 系列所使用的文件格式得到了非常广泛的应用。在本章中，我将着重于介绍微软的 X 文件格式，这是 DirectX 使用的格式；而在下一章中，将介绍 Quake 3 的.md3 模型格式。我想，这些内容都比较简单，如果你能了解这两种格式，那么你也应该可以去理解和使用其他格式了。

在本章中，你将学到：

- 如何使用 3D 文件格式；
- 如何使用变换矩阵；
- 如何使用 X 文件。

11.1 3D 文件格式

定义一种 3D 场景文件格式需要解决两个基本问题：（1）如何储存构成场景的各个对象；（2）如何储存这些对象之间的关系。此外，还需要能够保存用户自定义的数据类型。场景数

据具有两种关系性：层级关系和关联关系，前者比如父子关系、兄弟关系，后者则如网格附着到框架（frame）、材质附着到网格等。另外，在设计一种文件格式时还应考虑使其具有向前和向后的兼容性。

X 文件格式可以处理所有这些问题。因此，它是一套很好的解决方案，展示了如何将 3D 模型加载到一个游戏引擎中去。

11.2 X 文件格式

Direct3D 的 X 文件格式最初是为传统的 Direct3D 保留模式而设计的。在 DirectX 6.0 问世后，针对立即模式对它作过一次扩展。让我们先在一个抽象层面上看一下，X 文件是如何解决前一节中所描述的几个问题的。

- **用户自定义的数据类型**：X 文件是由模板（template）驱动的，模板定义了如何储存一个数据对象，这样用户便可以自己来定义具体的格式。预定义的模板位于 rmxftmpl.h 和 rmxftmpl.x[①]中，模板的标识符都在 rmxfguid.h 中，通用文件 d3dfile.cpp 包含了这两个头文件。
- **层级关系**：模板所允许的数据类型称为“可选成员（optional member）”，这些可选成员作为数据对象的子对象来保存。子对象可以是另一种数据对象、或对一个先前数据对象的引用、或是一个二进制的对象。

好了，现在让我们通过一个正方形示例 square.x 来深入研究一下 X 文件。你可以使用 DirectX 9 SDK 提供的 Mesh Viewer 先来观察、摆弄一下这个模型（见图 11.1）。

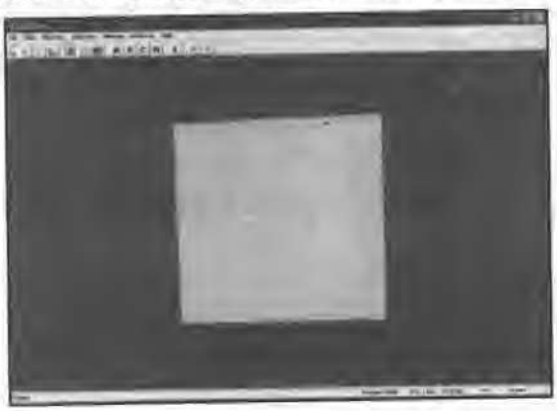


图 11.1 在 DX 9 的 Mesh Viewer 中加载 square.x

以下是这个文件的源代码：

```
// square
```

译注①：文件 rmxftmpl.x 在最新几个版本的 DirectX 9 SDK 中已经不存在了，在本书写作时，它尚出现于 DirectX 8.1 和第一个版本的 DirectX 9 SDK 中，你可以从这些老版本的 SDK 中获取这个文件，或在 Internet 上搜索这个文件。

```
Mesh Square {
    // front face and back face
    8;                          // number of vertices
    1.0; 1.0; 0.0;,             // vertice 0
    -1.0; 1.0; 0.0;,            // vertice 1
    -1.0;-1.0; 0.0;,            // vertice 2
    1.0;-1.0; 0.0;              // vertice 3
    1.0; 1.0; 0.0;,             // vertice 4
    1.0;-1.0; 0.0;,             // vertice 5
    -1.0;-1.0; 0.0;,            // vertice 6
    -1.0; 1.0; 0.0;;            // vertice 7
    4;                          // number of triangles
    3;0,1,2;,                   // triangle #1
    3;0,2,3;,                   // triangle #2
    3;4,5,6;,                   // triangle #3
    3;4,6,7;;                   // triangle #4

    MeshMaterialList {
        1;                      // one material
        4;                      // four faces
        0,                      // face #0 use material #0
        0,                      // face #1 use material #0
        0,                      // face #2 use material #0
        0;;                     // face #3 use material #0
        Material {              // material #0
            0.0;1.0;0.0;1.0;;   // face color
            0.0;                // power
            0.0;0.0;0.0;;       // specular color
            0.0;0.0;0.0;;       // emissive color
        }
    }
}
```

11.2.1　首部（Header）

起始的 4 字节是魔数[1]（magic number）"xof "。主版本号为 03，次版本号也为 03。"txt" 表示文本文件格式，"bin"表示二进制文件格式。

另外，传统的 Direct3D 保留模式所支持的传统*.x 文件格式使用了一种称为 Header 的模板。它的声明和所有其他的模板声明一样，都定义于 rmxftmpl.x 中。

还有一些其他可能的文件格式，如 tzip 和 bzip，它们分别表示 MSZip 压缩的文本文件和 MSZip 压缩的二进制文件。在这里将浮点数据类型设置为 0032，另外还可以设置为 0064。

正如你在源代码中所看到的，注释是以双斜线（就像在 C++中一样）作为起始来书写的，另外还可以用井号（#）。

11.2.2　网格

大多数文件都包含了一个 Mesh 模板，它的完整模板定义如下：

译注[1]：所谓“魔数”就是一些用于标识某种特定文件格式的字节序列，比如*.bmp 文件的魔数为"BM"、*.exe 文件的魔数为"MZ"或"PE"。

```
template Mesh {
    <3D82AB44-62DA-11cf-AB39-0020AF71E433>
    DWORD nVertices;
    array Vector vertices[nVertices];
    DWORD nFaces;
    array MeshFace faces[nFaces];
    [...]
}
```

Mesh 模板需要 Vector 模板和 MeshFace 模板，因为 Mesh 会引用到它们：

```
template Vector {
    <3D82AB5E-62DA-11cf-AB39-0020AF71E433>
    FLOAT x;
    FLOAT y;
    FLOAT z;
}

template MeshFace {
    <3D82AB5F-62DA-11cf-AB39-0020AF71E433>
    DWORD nFaceVertexIndices;
    array DWORD faceVertexIndices[nFaceVertexIndices];
}
```

模板由四部分组成。第一部分是一个独一无二的名称，可以由数字、字符、下划线构成，但不能以数字开头。第二部分是一个 UUID（Universally Unique Identifier；全球惟一标识符）。第三部分由各个数据项构成。最后一部分用于控制模板的限制程度，一个模板可以是开放的、闭合的或受限的。开放模板可以包含所有其他数据类型，封闭模板不可以包含任何其他的数据类型，受限模板只可以包含特定的数据类型。

```
// square
Mesh Square {
    // front face and back face
    8;                      // number of vertices
    1.0; 1.0; 0.0;,         // vertice 0
    -1.0; 1.0; 0.0;,        // vertice 1
    -1.0;-1.0; 0.0;,        // vertice 2
    1.0;-1.0; 0.0;          // vertice 3
    1.0; 1.0; 0.0;,         // vertice 4
    1.0;-1.0; 0.0;,         // vertice 5
    -1.0;-1.0; 0.0;,        // vertice 6
    -1.0; 1.0; 0.0;;        // vertice 7
    4;                      // number of triangles
    3;0,1,2;,               // triangle #1
    3;0,2,3;,               // triangle #2
    3;4,5,6;,               // triangle #3
    3;4,6,7;;               // triangle #4
```

以上第一个数字是使用的顶点数量，接着是设置各个顶点。在此定义了一个向前的表面和一个向后的表面。前表面使用了 4 个顶点，构成 2 个三角形，如图 11.2 所示。

Mesh 模板最后使用的是一个待展开的方括号[...]，所以它是一个开放的模板，可以包含

任何其他的数据类型。

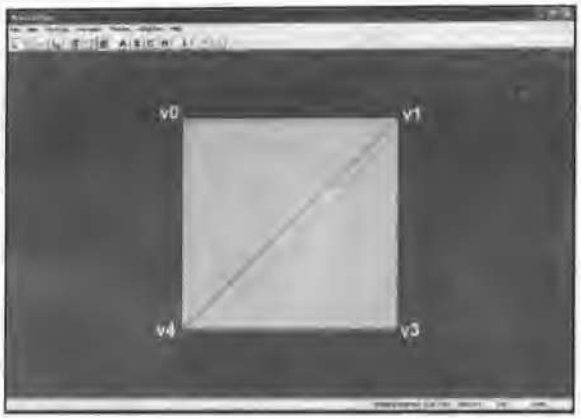


图 11.2　Square.x 中由 4 个顶点构成的 2 个三角形

11.2.3　网格材质列表

MeshMaterialList 是 Mesh 对象的一个子对象，它作为 Mesh 对象的一部分，位于其内部。在 MeshMaterialList 中需要给出面的数量和材质的数量，它将每个面与材质相关联。

```
MeshMaterialList {
    1;                      // one material
    2;                      // two faces
    0,                      // face #0 use material #0
    0;;                     // face #1 use material #0
    Material {              // material #0
        0.0;1.0;0.0;1.0;;   // face color
        0.0;                // power
        0.0;0.0;0.0;;       // specular color
        0.0;0.0;0.0;;       // emissive color
    }
}
```

rmxftmpl.x 中的模板如下：

```
template MeshMaterialList {
    <f6f23f42-7686-11cf-8f52-0040333594a3>
    DWORD nMaterials;
    DWORD nFaceIndexes;
    array DWORD faceIndexes[nFaceIndexes];
    [Material]
}
```

第 1 个变量保存该示例使用的材质数量。第 2 个变量保存面的数量，这个正方形有前后两个面，因此该变量被设为 2。faceIndexes 数组记录了面与材质的关联信息，其中每个面都指定一个材质号来与一种材质相关联。X 文件读取程序会对给出的面进行计数，从而可以确知当前读取到的是哪个面的材质号。

MeshMaterialList 模板是一个受限模板，只能包含特定的模板。在那个数组后面，只可以包含 Material 模板。

```
template Material {
    <3D82AB4D-62DA-11cf-AB39-0020AF71E433>
    ColorRGBA faceColor;
    FLOAT power;
    ColorRGB specularColor;
    ColorRGB emissiveColor;
    [...]
}
```

这个模板又包含了以下两个模板：

```
template ColorRGBA {
    <35FF44E0-6C7C-11cf-8F52-0040333594A3>
    FLOAT red;
    FLOAT green;
    FLOAT blue;
    FLOAT alpha;
}

template ColorRGB {
    FLOAT red;
    FLOAT green;
    FLOAT blue;
}
```

Material 数据对象保存着材质颜色、镜面反射的指数、镜面反射颜色、自发光颜色。待展开的方括号表示它可以包含其他数据对象。你可以在 MeshMaterialList 中引用 Material 数据对象。文件 square2.x 展示了两个被引用的 Material 数据对象。

```
xof 0303txt 0032

Material GreenMat {         // material #0
    0.0;1.0;0.0;1.0;;
    0.0;
    0.0;0.0;0.0;;
    0.0;0.0;0.0;;
}
Material RedMat {       // material #1
    1.0;0.0;0.0;1.0;;
    0.0;
    0.0;0.0;0.0;;
    0.0;0.0;0.0;;
}
// square
Mesh Square {
    // front face and back face
    8;                          // number of vertices
    1.0; 1.0; 0.0;,             // vertice 0
    -1.0; 1.0; 0.0;,            // vertice 1
    -1.0;-1.0; 0.0;,            // vertice 2
    1.0;-1.0; 0.0;              // vertice 3
    1.0; 1.0; 0.0;,             // vertice 4
    1.0;-1.0; 0.0;,             // vertice 5
    -1.0;-1.0; 0.0;,            // vertice 6
    -1.0; 1.0; 0.0;;            // vertice 7
```

```
    4;                          // number of triangles
    3;0,1,2;,                   // triangle #1
    3;0,2,3;,                   // triangle #2
    3;4,5,6;,                   // triangle #3
    3;4,6,7;;                   // triangle #4
    MeshMaterialList {
        2;                      // two material
        4;                      // four faces
        0,                      // face #0 use material #0
        0,                      // face #1 use material #0
        1,                      // face #2 use material #1
        1;;                     // face #3 use material #1
        {GreenMat}
        {RedMat}
    }
    VertexDuplicationIndices {
        8;
        8;
        0,
        1,
        2,
        3,
        4,
        5,
        6,
        7;
    }
}
```

{GreenMat}和{RedMat}是对 Material 对象的引用。模板 VertexDuplicationIndices 保存的信息记录了网格中哪些顶点是其他顶点的副本。该模板的用途是为了允许加载程序可以判断出，哪些呈现出不同外围参数的顶点其实是模型中相同的顶点。某些应用程序（比如简化网格的程序）会用到这类信息。图 11.3 展示了在 Mesh Viewer 中看到的 square2.x 文件。

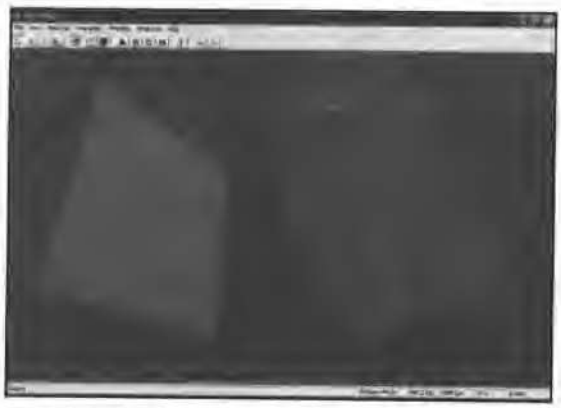


图 11.3　通过两种材质来渲染一个多边形——左边使用材质 GreenMat，右边使用材质 RedMat

11.2.4　法线

光照模型计算中需要用到法线，如前所述，法线分为面法线和顶点法线。在基于逐顶点

计算的光照模型中，需要使用顶点法线。通常顶点法线都是通过这种方式来计算的：先将共享该顶点的几个面的面法线相加，然后归一化相加的结果。square3.x 文件中的法线定义如下：

```
MeshNormals {
    8;
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;-1.000000;,
    0.000000;0.000000;-1.000000;,
    0.000000;0.000000;-1.000000;,
    0.000000;0.000000;-1.000000;;
    4;
    3;0,1,2;, // vertex normals on faces
    3;0,2,3;,
    3;4,5,6;,
    3;4,6,7;;
}

template MeshNormals {
    <f6f23f43-7686-11cf-8f52-0040333594a3>
    DWORD nNormals;
    array Vector normals[nNormals];
    DWORD nFaceNormals;
    array MeshFace faceNormals[nFaceNormals];
}
```

如上所述，所有面上共有 8 条顶点法线，它们的次序展示在粗体的“4;”之后。例如，顶点法线 4、6、7 位于第四个面上。

11.2.5 纹理

TextureFilename 数据对象用于引用纹理，它可以作为 Material 对象的子对象。

```
template TextureFilename {
    STRING filename;
}
```

对纹理文件名的引用必须包含在 Material 对象中，如下所示：

```
Material GreenMat { // material #0
    0.0;1.0;0.0;1.0;;
    0.0;
    0.0;0.0;0.0;;
    0.0;0.0;0.0;;
    TextureFilename{"wall.bmp";}
}
```

另外，还需要指定纹理坐标：

```
template MeshTextureCoords {
    <f6f23f40-7686-11cf-8f52-0040333594a3>
    DWORD nTextureCoords;
    array Coords2d textureCoords[nTextureCoords];
}

template Coords2d {
    <f6f23f44-7686-11cf-8f52-0040333594a3>
    FLOAT u;
    FLOAT v;
}
```

第一个变量保存需要关联纹理坐标的顶点数量。接下来的数组保存了纹理的 uv 坐标对。如果想要将一张纹理映射到整个正方形上，可以这么做：

```
MeshTextureCoords {
    4;                  // 4 texture coords
    1.0; 1.0;,          // coord 0
    1.0; 0.0;,          // coord 1
    0.0; 0.0;,          // coord 2
    0.0; 1.0;;          // coord 3
}
```

右下角为(1.0f, 1.0f)，左上角为(0.0f, 0.0f)，这样做无需顾及纹理的实际大小，哪怕纹理的宽度比高度大。

要看到 4 张纹理的话，则需要使用以下坐标：

```
MeshTextureCoords {
    4;                  // 4 textures
    1.0; 1.0;,          // coord 0
    -1.0; 1.0;,         // coord 1
    -1.0;-1.0;,         // coord 2
    1.0;-1.0;;          // coord 3
}
```

square3.x 示例由默认的包装纹理寻址模式驱动，关于纹理寻址模式我已在第二部分中解释过了。完整的文件如下：

```
xof 0303txt 0032

Material GreenMat {         // material #0
    0.0;1.0;0.0;1.0;;
    0.0;
    0.0;0.0;0.0;;
    0.0;0.0;0.0;;
    TextureFilename {
        "wall.bmp";         // texture map to use
    }
}

// square
Mesh Square {
    // front face and back face
```

```
8;                          // number of vertices
1.0; 1.0; 0.0;,             // vertice 0
-1.0; 1.0; 0.0;,            // vertice 1
-1.0;-1.0; 0.0;,            // vertice 2
1.0;-1.0; 0.0;              // vertice 3

1.0; 1.0; 0.0;,             // vertice 4
1.0;-1.0; 0.0;,             // vertice 5
-1.0;-1.0; 0.0;,            // vertice 6
-1.0; 1.0; 0.0;;            // vertice 7

4;                          // number of triangles
3;0,1,2;,                   // triangle #1
3;0,2,3;,                   // triangle #2
3;4,5,6;,                   // triangle #3
3;4,6,7;;                   // triangle #4

MeshNormals {
    8;
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;1.000000;,
    0.000000;0.000000;-1.000000;,
    0.000000;0.000000;-1.000000;,
    0.000000;0.000000;-1.000000;,
    0.000000;0.000000;-1.000000;;
    4;
    3;0,1,2;,
    3;0,2,3;,
    3;4,5,6;,
    3;4,6,7;;
}

MeshTextureCoords {
    8;                      // 4 texture coords
    1.0; 1.0;,              // coord 0
    1.0; 0.0;,              // coord 1
    0.0; 0.0;,              // coord 2
    0.0; 1.0;;              // coord 3

    1.0; 1.0;,              // coords 4
    1.0;-1.0;,              // coords 5
    -1.0;-1.0;,             // coords 6
    -1.0; 1.0;;             // coords 7
}

MeshMaterialList {
    1;                      // one material
    4;                      // one face
    0,                      // face #0 use material #0
    0,                      // face #1 use material #0
    0,                      // face #2 use material #0
    0;                      // face #3 use material #0

    {GreenMat}
```

```
    }

    VertexDuplicationIndices {
        8;
        8;
        0,
        1,
        2,
        3,
        4,
        5,
        6,
        7;
    }
}
```

如图 11.4 所示，前表面显示了 4 张纹理，后表面显示了 1 张纹理。在 Mesh Viewer 中，所有纹理都与绿色的材质颜色相调制。

我准备了一个稍大点的示例，其中用到的顶点数据曾在第 5 章的示例中用到过。在 Mesh Viewer 中打开 CD-ROM 的本章目录下 boidy.x 文件。图 11.5 展示了这个模型对象。

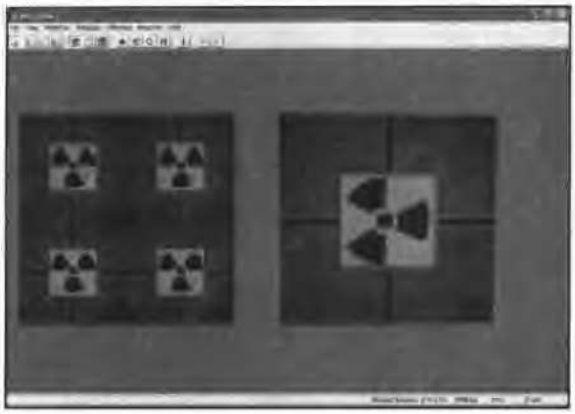


图 11.4　前表面映射了 4 张纹理，后表面映射了 1 张纹理

图 11.5　存储于在 X 文件中的第 5 章的模型对象

图 11.5 显示了该示例用到的面、顶点和顶点法线。以下是源代码：

```
xof 0302txt 0064

// boid
Mesh boid {
    // vertices
    7;                          // number of vertices
    0.00; 0.00; 0.50;,          // vertice 0
    0.50; 0.00;-0.50;,          // vertice 1
    0.15; 0.15;-0.35;,          // vertice 2
    -0.15; 0.15;-0.35;,         // vertice 3
    0.15;-0.15;-0.35;,
    -0.15;-0.15;-0.35;,
    -0.50; 0.00;-0.50;;

    10;                         // number of triangles
    3;0,1,2;,                   // triangle #1
    3;0,2,3;,                   // triangle #2
    3;0,3,6;,                   // triangle #3
    3;0,4,1;,                   // ...
    3;0,5,4;,
    3;0,6,5;,
    3;1,4,2;,
    3;2,4,3;,
    3;3,4,5;,
    3;3,5,6;;

    MeshMaterialList {
        1;                      // one material
        10;                     // ten faces
        0,0;;                   // face #1 use material #0
        0,0;;                   // face #2 use material #0
        0,0;;                   // face #3 use material #0
        0,0;;                   // face #4 use material #0
        0,0;;                   // face #5 use material #0
        0,0;;                   // face #6 use material #0
        0,0;;                   // face #7 use material #0
        0,0;;                   // face #8 use material #0
        0,0;;                   // face #9 use material #0
        0,0;;                   // face #10 use material #0

        Material {              // material #0
            1.0;0.92;0.0;1.0;;
            1.0;
            0.0;0.0;0.0;;
            0.0;0.0;0.0;;
        }

        MeshNormals {
            7;
            0.000000;0.000000;1.000000;,
            0.984784;0.000000;0.173785;,
            0.192491;0.971619;-0.137493;,
            -0.185863;0.938163;-0.292070;,
            0.185863;-0.938163;-0.292070;,
            -0.192491;-0.971619;-0.137493;,
            -0.984784;0.000000;0.173785;;
            10;
            3;0,1,2;,
            3;0,2,3;,
```

```
            3;0,3,6;,
            3;0,4,1;,
            3;0,5,4;,
            3;0,6,5;,
            3;1,4,2;,
            3;2,4,3;,
            3;3,4,5;,
            3;3,5,6;;
        }
    }
}
```

7 个顶点就有 7 条法线，这些法线分别位于 10 个面上。

由于顶点数量较少，所以光照效果不是很真实。Mesh Viewer 中的漫反射光照是基于逐顶点计算的，如果模型拥有的顶点数量较少就会导致这样的漫反射光照问题。在 Mesh Viewer 中通过 N-Patches 菜单中的“N-Patch Selected”和“Snapshot to Mesh”可以镶嵌一个模型，镶嵌操作会根据 N-Patch 算法将一些新的顶点插入到模型中。另外，在每次镶嵌过程后必须重新计算法线。图 11.6 展示了做过 4 次镶嵌后的结果。

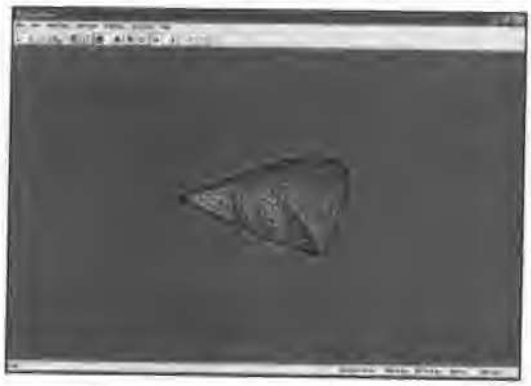


图 11.6　多次镶嵌后的模型对象

现在该物体均匀地受到了光照，你可以看到更多的面、顶点和顶点法线。快去尝试一下 Mesh Viewer 的这个功能吧。

现在你应该已对如何手动地创建 X 文件中的几何体有一个大致的认识了。接下来，让我们深入地了解一下动画方面的内容。

11.2.6　变换矩阵

为了能独立地对一个物体的不同部分进行变换，必须将模型分割成不同的框架（frame）。例如，一辆坦克可以向上、下、左、右转动它的炮管，当它辗压过野蛮的敌方阵地时，它的履带和车轮会同时滚动。因此，对于车体、炮台、炮管、履带、车轮各有一个框架。每个框架都保存着对应于坦克某一特定部分的变换矩阵。一个 Frame 数据对象可能是像下面这样的：

```
Frame Hull {
    FrameTransformMatrix {
        1.000000, 0.000000, 0.000000, 0.000000,
        0.000000, 1.000000, 0.000000, 0.000000,
        0.000000, -0.000000, 1.000000, 0.000000,
        206.093353, -6.400993, -31.132195, 1.000000;;
    }
    Mesh Hull {
        2470;
        41.310013; -26.219450; -113.602348;,
...

    Frame Wheels_L {
        FrameTransformMatrix {
            1.000000, 0.000000, 0.000000, 0.000000,
            0.000000, 1.000000, -0.000000, 0.000000,
            0.000000, 0.000000, 1.000000, 0.000000,
            -56.020325, -31.414078, 3.666503, 1.000000;;
}
        Mesh Wheels_L {
            2513;
            -4.642166; -11.402874; -98.607910;,
...

    Frame Wheels_R {
        FrameTransformMatrix {
            1.000000, 0.000000, 0.000000, 0.000000,
            0.000000, 1.000000, -0.000000, 0.000000,
            0.000000, 0.000000, 1.000000, 0.000000,
            56.687805, -31.414078, 3.666503, 1.000000;;
        }
        Mesh Wheels_R {
            2513;
            4.642181; -11.402874; -98.607910;,
            ...

    Frame Turret {
        FrameTransformMatrix {
            1.000000, 0.000000, 0.000000, 0.000000,
            0.000000, 1.000000, 0.000000, 0.000000,
            0.000000, 0.000000, 1.000000, 0.000000,
            -2.077148, 84.137527, 29.323750, 1.000000;;
        }
        Mesh Turret {
            2152;
            52.655853; -36.225544; -16.728998;,
...
```

本例使用的模板在 rmxftmpl.x 中定义为：

```
template Frame {
    <3D82AB46-62DA-11cf-AB39-0020AF71E433>
    [...]
}
template FrameTransformMatrix {
```

```
    Matrix4x4 frameMatrix;
}
template Matrix4x4 {
    array FLOAT matrix[16];
}
```

其中那种对 4×4 矩阵的声明方式是 DirectX 图形编程长久以来的习惯做法。另外，还有一个简单的示例 car.x 应该看一下，它位于“c:\dxsdk\samples\Media”目录下。boidsy2.x 是上一个 boidsy.x 示例文件的高级版本，如下所示：

```
xof 0302txt 0064

Frame BOID_Root {
    FrameTransformMatrix {
        1.000000, 0.000000, 0.000000, 0.000000,
        0.000000, 1.000000, 0.000000, 0.000000,
        0.000000, 0.000000, 1.000000, 0.000000,
        0.000000, 0.000000, 0.000000, 1.000000;;
    }

    // boid
    Mesh boid {
        // vertices
        7;                              // number of vertices
        0.00; 0.00; 0.50;,              // vertice 0
        0.50; 0.00;-0.50;,              // vertice 1
        0.15; 0.15;-0.35;,              // vertice 2
        -0.15; 0.15;-0.35;,             // vertice 3
        0.15;-0.15;-0.35;,
        -0.15;-0.15;-0.35;,
        -0.50; 0.00;-0.50;;

        10;                             // number of triangles
        3;0,1,2;,                   // triangle #1
        3;0,2,3;,                   // triangle #2
        3;0,3,6;,                   // triangle #3
        3;0,4,1;,                   // ...
        3;0,5,4;,
        3;0,6,5;,
        3;1,4,2;,
        3;2,4,3;,
        3;3,4,5;,
        3;3,5,6;;

        MeshMaterialList {
            1;                      // one material
            10;                     // ten faces
            0,0;;                   // face #1 use material #0
            0,0;;                   // face #2 use material #0
            0,0;;                   // face #3 use material #0
            0,0;;                   // face #4 use material #0
            0,0;;                   // face #5 use material #0
            0,0;;                   // face #6 use material #0
            0,0;;                   // face #7 use material #0
```

```
                0,0;;                   // face #8 use material #0
                0,0;;                   // face #9 use material #0
                0,0;;                   // face #10 use material #0

                Material {              // material #0
                    1.0;0.92;0.0;1.0;;
                    1.0;
                    0.0;0.0;0.0;;
                    0.0;0.0;0.0;;
                }

                MeshNormals {
                    7;
                    0.000000;0.000000;1.000000;,
                    0.984784;0.000000;0.173785;,
                    0.192491;0.971619;-0.137493;,
                    -0.185863;0.938163;-0.292070;,
                    0.185863;-0.938163;-0.292070;,
                    -0.192491;-0.971619;-0.137493;,
                    -0.984784;0.000000;0.173785;;

                    10;
                    3;0,1,2;,
                    3;0,2,3;,
                    3;0,3,6;,
                    3;0,4,1;,
                    3;0,5,4;,
                    3;0,6,5;,
                    3;1,4,2;,
                    3;2,4,3;,
                    3;3,4,5;,
                    3;3,5,6;;
                }
            }
        }
    }
```

11.2.7 动画

为了使一个 X 文件产生动画，必须提供一个动画集（a set of animations），每个动画集都应具有一个对某个框架的引用。示例文件 square4.x 展示了两个简单的动画，分别是在四边形的 x 轴和 y 轴方向上移动它。在 Mesh Viewer 中通过选择“File—>Open Mesh File”来加载这个文件。

```
xof 0303txt 0032

Frame SQUARE_Root {
    FrameTransformMatrix {
        1.000000, 0.000000, 0.000000, 0.000000,
        0.000000, 1.000000, 0.000000, 0.000000,
        0.000000, 0.000000, 1.000000, 0.000000,
```

```
    0.000000, 0.000000, 0.000000, 1.000000;;
}

// square
Mesh Square {
    // front face and back face
    8;                      // number of vertices
    1.0; 1.0; 0.0;,         // vertice 0
    -1.0; 1.0; 0.0;,        // vertice 1
    -1.0;-1.0; 0.0;,        // vertice 2
    1.0;-1.0; 0.0;          // vertice 3
    1.0; 1.0; 0.0;,         // vertice 4
    1.0;-1.0; 0.0;,         // vertice 5
    -1.0;-1.0; 0.0;,        // vertice 6
    -1.0; 1.0; 0.0;;        // vertice 7

    4;                      // number of triangles
    3;0,1,2;,               // triangle #1
    3;0,2,3;,               // triangle #2
    3;4,5,6;,               // triangle #3
    3;4,6,7;;               // triangle #4

   MeshMaterialList {
        1;                  // one material
        4;                  // one face
        0,                  // face #0 use material #0
        0,                  // face #1 use material #0
        0,                  // face #2 use material #0
        0;;                 // face #3 use material #0

        Material GreenMat {                    // material #0
            0.0;1.0;0.0;1.0;;
            0.0;
            0.0;0.0;0.0;;
            0.0;0.0;0.0;;

            TextureFilename{"wall.bmp";} // the texture map
        }
   }

   MeshTextureCoords {
        8;                  // 4 texture coords
        1.0; 1.0;,          // coords 0
        1.0; 0.0;,          // coords 1
        0.0; 0.0;,          // coords 2
        0.0; 1.0;;          // coords 3
        1.0; 1.0;,          // coords 4
        1.0;-1.0;,          // coords 5
        -1.0;-1.0;,         // coords 6
        -1.0; 1.0;;         // coords 7
   }

   MeshNormals {
        8;
        0.000000;0.000000;1.000000;,
        0.000000;0.000000;1.000000;,
```

```
                0.000000;0.000000;1.000000;,
                0.000000;0.000000;1.000000;,
                0.000000;0.000000;-1.000000;,
                0.000000;0.000000;-1.000000;,
                0.000000;0.000000;-1.000000;,
                0.000000;0.000000;-1.000000;;
                4;
                3;0,1,2;,
                3;0,2,3;,
                3;4,5,6;,
                3;4,6,7;;
            }

            VertexDuplicationIndices {
                8;
                8;
                0,
                1,
                2,
                3,
                4,
                5,
                6,
                7;
            }
        }
    }

    AnimationSet MoveX{
        Animation {
            AnimationKey {
                4;                  // Position keys
                5;                  // 5 keys
                0;16;1.0,0.0,0.0,0.0,
                          0.0,1.0,0.0,0.0,
                          0.0,0.0,1.0,0.0,
                          0.1,0.0,0.0,1.0;;,
                80;16;    1.0,0.0,0.0,0.0,
                          0.0,1.0,0.0,0.0,
                          0.0,0.0,1.0,0.0,
                          0.2,0.0,0.0,1.0;;,
                160;16;1.0,0.0,0.0,0.0,
                          0.0,1.0,0.0,0.0,
                          0.0,0.0,1.0,0.0,
                          0.3,0.0,0.0,1.0;;,
                240;16;1.0,0.0,0.0,0.0,
                          0.0,1.0,0.0,0.0,
                          0.0,0.0,1.0,0.0,
                          0.2,0.0,0.0,1.0;;,
                320;16;1.0,0.0,0.0,0.0,
                          0.0,1.0,0.0,0.0,
                          0.0,0.0,1.0,0.0,
                          0.1,0.0,0.0,1.0;;,
            }
            {SQUARE_Root}
        }
```

```
}

AnimationSet MoveY{
    Animation {
        AnimationKey {
            4;                  // Position keys
            5;                  // 5 keys
            0;16;1.0,0.0,0.0,0.0,
                 0.0,1.0,0.0,0.0,
                 0.0,0.0,1.0,0.0,
                 0.0,0.1,0.0,1.0;;,
            80;16;   1.0,0.0,0.0,0.0,
                 0.0,1.0,0.0,0.0,
                 0.0,0.0,1.0,0.0,
                 0.0,0.2,0.0,1.0;;,
            160;16;1.0,0.0,0.0,0.0,
                 0.0,1.0,0.0,0.0,
                 0.0,0.0,1.0,0.0,
                 0.0,0.3,0.0,1.0;;,
            240;16;1.0,0.0,0.0,0.0,
                 0.0,1.0,0.0,0.0,
                 0.0,0.0,1.0,0.0,
                 0.0,0.2,0.0,1.0;;,
            320;16;1.0,0.0,0.0,0.0,
                 0.0,1.0,0.0,0.0,
                 0.0,0.0,1.0,0.0,
                 0.0,0.1,0.0,1.0;;,
        }
        {SQUARE_Root}
    }
}
```

该示例展示了对两个不同动画集 MoveX 和 MoveY 的用法。你可以在 Mesh Viewer 中通过菜单“Animation—>Animations”来选择播放它们。这两套动画都使用了 5 个动画键（animation key），每个动画键由一个 4×4 的矩阵组成。就 MoveX 而言，在 5 个动画键中惟一发生变化的值就是第 4 行的第 1 个值——正如你所知的，该值用于控制正方形的 x 位置。{SQUARE_ROOT}是对框架的引用。在 AnimationKey 模板中的 4 表示当前使用的是“矩阵键”，该变量称为 keyType。DirectX 8.0 和 9.0 的 SDK 文档指出成员 keyType 用于表示动画键的类型，动画键可以是旋转键、缩放键、位置键或矩阵键。虽然文档叙述了 keyType 的正确范围应为 0～3，分别表示这 4 种动画键的类型。但是在 SDK 的 media 目录下的 tiny.x 示例却使用 4 来表示矩阵键[1]。

11.3　使用 X 文件

本书所有示例都使用了一些相似的函数来加载 X 文件。比如，我在立方体贴图示例中使用的加载函数便是：

译注[1]：关于这点，译者在与原作者讨论后，结论是——这是一个“很有名气的”bug，原因可能是文档编写错误，也可能是 D3DX 内部程序中的问题。但无论怎样，在当前的情况中，应使用 4 来表示矩阵键，而不是像文档所说的 3。

```
HRESULT CMyD3DApplication::LoadXFile(TCHAR* name)
{
    HRESULT hr;
    LPD3DXMESH pMeshSysMem = NULL, pMeshSysMem2 = NULL;

    if (FAILED (D3DXLoadMeshFromX(name, D3DXMESH_SYSTEMMEM,
                m_pd3dDevice, NULL, NULL, NULL, NULL, &pMeshSysMem)))
        return E_FAIL;

    D3DVERTEXELEMENT9 decl[]=
    {
        // stream, offset, type, method, semantic type (for example normal), ?
        {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
        {0, 12, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
        {0, 20, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0},
        D3DDECL_END()
    };

    hr = pMeshSysMem->CloneMesh(D3DXMESH_MANAGED, decl, m_pd3dDevice, &pMeshSysMem2);

    // compute the normals
    hr = D3DXComputeNormals(pMeshSysMem2,NULL);
    if(FAILED(hr))
        return E_FAIL;

    D3DVERTEXELEMENT9 decl2[]=
    {
       // stream, offset, type, method, semantic type (for example normal), ?
       {0, 0, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
       {0, 16, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
       {0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0},
       D3DDECL_END()
    };

    hr = pMeshSysMem2->CloneMesh(D3DXMESH_MANAGED, decl2, m_pd3dDevice, &m_pD3DXMesh);

    // cleanup
    SAFE_RELEASE(pMeshSysMem);
    SAFE_RELEASE(pMeshSysMem2);

    return S_OK;
}
```

为了加载一个 X 文件，需要使用 Direct3DX 库中的 D3DXLoadMeshFromX()函数。该函数的声明如下：

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

第 1 个参数 pFilename 应是一个指向文件路径的指针。各个示例都使用 Direct3D 实用函数 DXUtil_FindMediaFileCb()来协助加载*.x 文件的。

```
TCHAR strFile[160];
DXUtil_FindMediaFileCb(strFile, sizeof(strFile),_T("teapot.x"));
LoadXFile(strFile);
```

DXUtil_FindMediaFileCb() 会搜索整个文件系统，查找一个指定的文件。D3DXLoadMeshFromX()函数的第 2 个参数 Options 用于指定顶点缓冲器或索引缓冲器的内存池类型和一些其他标志。以上这个示例将其指定为 D3DXMESH_SYSTEMMEM，但通常其他情况下更好的选择或许是 D3DXMESH_MANAGED。由于 CloneMesh()函数会读取顶点缓冲器和索引缓冲器，所以不能在这里使用组合标志 D3DXMESH_MANAGED | D3DXMESH_WRITEONLY。

第 3 个参数应传入一个指向设备的指针。由于这里不需要任何领接数据，所以第 4 个参数为 NULL。同样，也不需要材质数据和效果实例的数据，所以接下来的 3 个参数也为 NULL。最后一个参数返回一个指向网格的指针，本例中该网格稍后将用于复制以及法线计算。另外，请关注一下 D3DXComputeNormals()函数，该函数用于计算网格的法线，但它接受的输入位置值只能由 3 个 float 组成。两个 CloneMesh()函数用于创建原网格的副本，副本将按 D3DVERTEXELEMENT9 结构定义的数据布局来保存数据。第一次复制是为了将原网格复制为一种 D3DXComputeNormals()函数期望的数据布局。第二次复制是为了将前一次的副本复制为一种新的数据布局，这种数据布局中的位置值由 4 个 float 组成的。

相对于加载一个动画 X 文件而言，加载一个非动画的 X 文件则简单得多。欲进一步地了解如何使用蒙皮(skinning)技术来产生动画，以及如何加载这样的动画 X 文件，请参考 DirectX 9 SDK 中的 SkinnedMesh 示例。

11.4　扩展 X 文件

在 X 文件背后的诸多概念中，最吸引人的一项特征应该是它的可扩展性。回想一下 rmxftmpl.x 文件，它保存了各种模板的定义。你可以在其中定义自己的模板，并按这种方式创建出你自己的 X 文件格式。例如，你可以按这种方式来实现一种特殊的 shader 模板，它可以具备本书第二部分中介绍的所有纹理功能，还可以给每项功能指定不同的参数。由此，你的文件格式可以保存各种信息，如自定义的 shader 类、碰撞检测参数、物理属性、专门的动画信息、声音文件的路径等等。

如果你看一下以下小节，你就会了解到在这方面已经获得的一些成果。我想你应该会在接下来的几个小时里考虑构建一种你自己的文件格式，记得参考或使用这种方法噢！

11.5 附加资源

或许你还想要更进一步地阅读一些文章来加深对本章所涉及主题的了解，这一节将给出一些相关的网站。

11.5.1 X 文件格式

DirectX 9 的文档本身就包含了极佳的内容可供进一步地阅读。这里还有一份由 Rich Thompson 提供的 X 文件的转储（dump），位于 http://www.xmission.com/~legalize/book/snippets/index.html#12。

11.5.2 蒙皮网格（Skinned Mesh）

一篇关于如何模块化 DX9 SDK 中的 SkinnedMesh 示例的文章，位于 http://www.flipcode.com/articles/article_dx9skinmeshmod.shtml。

一篇解释蒙皮原理的文章，位于 http://www.gamedev.net/reference/articles/article1835.asp。

一个展示了各种蒙皮网格示例的西班牙语网站，位于 http://atc1.aut.uah.es/~infind/Programming/Programming.htm。

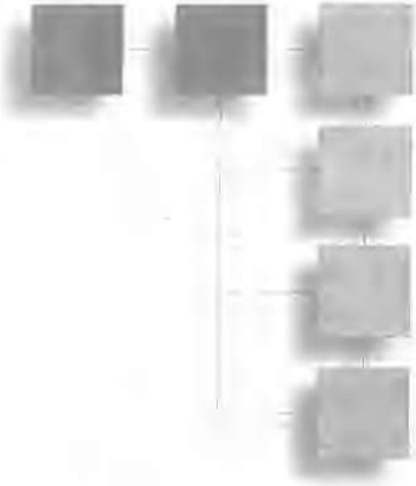
此外，在 *Game Developer* 杂志的 1998/5 刊和 2001/4 刊上有几篇 Jeff Lander 撰写的好文章，都是讨论了蒙皮技术的。

另外，Eike F. Anderson 撰写的一篇文章也很有启发性，位于 http://ncca.bournemouth.ac.uk/newhome/alumni/docs/CharacterAnimation.pdf。Michael Putz 和 Klaus Hufnagl 在另一篇文章中描述了一种更高级的做法，位于 http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-002/MPutzKHufnagl/paper.pdf。

11.6 小结

本章介绍了所有你曾经想要了解的内容，其中包括变换矩阵和微软的 X 文件格式。好吧，或许这还不是最终你想要的，但无论如何本章至少给了你一个不错的起点。我敢肯定你已经等不及要学习关于 *Quake 3* 引擎所使用的*.md3 文件格式了，那就翻到下一页，让我们继续吧。

第 12 章



使用*.md3 文件

本章将描述 *Quake 3*（雷神之锤 3）引擎所使用的*.md3 文件，还将通过一个示例程序来说明如何使用这种格式，该示例中包含了逐像素光照、凹凸映射和一个通过 HLSL shader 来实现的高效 shader 系统。

.md3 格式是一种常用的文件格式，它已经历过广泛的测试，在 *Quake 3* 和许多其他游戏中，.md3 被证明为是一种极好的模型格式。当你开始钻研.md3 格式时，你也就将深入到 *Quake 3* 的代码世界中。通过观察模型可以让人深刻地了解到 *Quake 3* 是被如何开发出来的。在你开始设计自己的模型文件格式前，先花一点时间来看一个已被广泛认可的模型文件规范总不会是一件坏事情。

Quake 3 一般在最高精细度的模型中使用 800～900 个面，在中等精度的模型中使用大约 500 个面，在最低精度的模型中使用大约 300 个面。但这并不意味着你也必须遵循这些面数限制。

这种模型文件将动画储存在不同的帧中，每个动画都具有一个固定的名字、不同的长度、以及在每次移动时不同的速度。关键帧（key frame）是一种廉价的动画处理方法。但是，关键帧动画的缺点是处理高度镶嵌的（high-tesselated）模型[1]需要大量的存储空间。

大多数 *Quake 3* 模型都拥有不同的“皮肤”（即不同的纹理贴图），这些皮肤可以在需要的时候被加载。例如，你可以对一个静止的雕像、一个士兵或一个戴着金色头盔的指挥官都使用同一个几何模型，但赋予不同的皮肤后即可体现出它们之间的区别。

12.1 文件格式探索

一个.md3 模型通常是由身体的 3 个部分和 1 个武器模型组成的。因此，需要打开 3 个模

译注[1]：即具有很多顶点的高精度模型。

型文件并将它们放到一起（如果还使用了武器的话，则需要打开 4 个文件）。在 *Quake 3* 中，将它们分别称为 head.md3、upper.md3、lower.md3，如图 12.1 所示。

在深入了解.md3 文件后，你会发现其实一个模型可以有任意多个组成部分。例如，在“*DOOM marine*”[1]的 head.md3 中就有两个网格，分别叫作 h_helmet（头盔）和 h_visor（护目镜）。你可以对模型的每个组成部分应用不同的纹理或 shader 效果，而且对于组成部分的数量也并无限制，但它们必须遵循一种命名规范，即头部（head）组件的名称以“h_”开头、上身（upper）组件以“u_”开头、下身（lower）组件以“l_”开头。

.md3 文件将角色动画存储在关键帧中。为了形象地理解关键帧动画，可以想象一下在老式的精灵引擎中使用精灵的方法，一般会有一长串的图片，每张图片都是基于前一张的变化，就像老式的电影胶片那样，由此观看者可以感觉到物体是在运动。对于.md3 文件中的多边形顶点也是同样的原理。虽然，引擎对帧的数量没有限制，但一般最多不超过 300 帧，这样才能保证可以快速地渲染出模型，并具有合理的图像质量。如果使用更多关键帧的话，则会导致相当大的存储开销。animation.cfg 文件存储着动画方面的信息，它用于指导对关键帧的调用。

图 12.1 *DOOM marine*（*Quake 3* 的原始模型）（id Software 授权）

你可以通过 3 个不同的.md3 文件来提供三种不同精细程度的模型。在 *Quake 3* 中，对这 3 种精细程度的文件命名规范是：对于最低精细度的模型，在其文件名后加一个 2；对于中等精细度的模型，在文件名后加一个 1；对于最高精细度的模型，则不需要加任何额外的数字。

[1] 译注：*Quake 3* 中的一个游戏角色。

因此，文件名 head.md3、head1.md3、head2.md3 分别对应于最高精度、中等精度和最低精度。

头部、上身、下身这 3 个躯体部分之间通过所谓的“标记（tag）”来彼此连接（attach）。每个标记其实就是一个单面的几何体（一个直角三角形），如图 12.2 所示。三角形中较长的垂直边必须面向前方，三角形的底部是一个支点（pivot point），这个支点决定了各个模型将围绕何处作旋转。标记是没有材质的，因此你也看不到它。

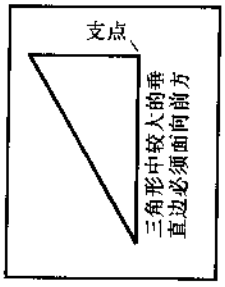


图 12.2　连接各个.md3 模型的标记

每个.md3 模型一般会有 3 个标记，这些标记的名称对应于它们所在的位置。在头颈底部，通过 tag_head 标记将头部连接到上身的上方。在脊骨底部，通过 tag_torso 标记将上身连接到腿上。武器通过 tag_weapon 连接到模型的手上。每个模型文件至少会有一个标记，例如 head.md3 有一个标记 tag_head，upper.md3 有 tag_head、tag_torso 和 tag_weapon，lower.md3 有 tag_torso。

你需要将 tag_head 放到头骨的支点上，将 tag_torso 放到脊骨底部的支点上，依此类推。每个连接.md3 文件的标记都是通过*.skin 文件中的一条记录来指定的。你可以按任何希望的方式来连接.md3 模型的各个部分。

将皮肤（即.tga 或.jpg 格式的纹理贴图文件）连接到几何体上也是通过*.skin 文件完成的。通过这种方式可以很方便地在相同的几何体上使用不同的皮肤（例如，一个模型既可用于一个乳白色的雕像，也可用于一位正在战斗的士兵）。在 *Quake 3* 中，通过这种皮肤文件还带来了另一种新的视觉进步，那便是 32 位的纹理。在以前，限制只能使用 8 位调色板，并且对一个模型只能使用一张皮肤；而现在你可以使用 32 位的颜色了，并且还能对一个模型的不同部分使用不同的皮肤。另外，如果一张纹理是正方形的，并且其宽度和高度都是 2 的幂次、大小不超过 256×256 的话，那么 *Quake 3* 引擎便可以最佳的效率来处理它（本章的示例则无所谓这一点，它可以使用任意尺寸的纹理）。请记住这点，在 *Quake 3* 中模型的每个组成部分都可以有它们自己的皮肤。

Quake 3 引擎还为.md3 模型使用了 shader 文件。记得在 *Quake 3* 发布的时候，这曾是众多令人兴奋的特征之一。通过 shader 效果可以控制纹理映射、透明程度、各种光照参数等很多参数。*Quake 3* 使用了一个叫作 models.shader 的 shader 文件来处理所有这些参数。

另外，*Quake 3* 引擎还允许你将不同的声音连接到.md3 文件格式中。声音文件要求以 22.5kHz、16 位、单声道的格式来存储。理论上文件的长度可以是任意的。

表 12.1 总结了一个标准的 *Quake 3* .md3 模型所使用的各种文件格式及其文件名。

.md3 文件存储着模型的几何数据，而.tga 或.jpg 文件存储着.md3 文件用到的纹理。shader 文件控制着纹理的映射和其他许多参数。借助于.skin 文件，你可以将多个.md3 文件标记为一组，从而构成一个完整的模型；还可以给该模型指定专门的皮肤（.tga 或.jpg 文件）。另外，为了使储存在各个.md3 文件中的模型的不同部分产生动画，还需要使用一个 animation.cfg 文件。为了能够听到模型的说话声或射击声，你还可以提供一些具有适当声音的.wav 文件。

表 12.1　　标注 Quake 3 模型使用的文件格式

文件格式	文件名
.md3 文件	通常为 head.md3、upper.md3、lower.md3
图标（肖像）文件	icon_default.tga
动画文件	animation.cfg
皮肤文件	.tga 或.jpg
标记文件	.skin
shader 文件	比如 models.shader
声音文件	.wav

为了使你可以自己来分析.md3 文件，你需要看一下 *Quake 3* 中的文件 pak0.pk3。在安装完游戏后，这个文件会位于 *Quake 3* 的一个目录中。你必须将其重新命名为 pak0.zip，然后再使用一个 Zip 解压缩程序来查看其中的目录和文件。以其中的.md3 模型 Biker 为例，它的各文件分别储存在以下几个目录中：

- .md3 文件、animation.cfg、.tga、.skin 在目录“models/players/biker”中；
- 所有模型的 shader 文件都在目录“scripts”中；
- .wav 文件在目录“sound/player/biker”中。

你可以看一下 models 目录，*Quake 3* 的模型都保存在它的子目录中，其子目录中一般都会有以下几个文件：

- animation.cfg
- head.md3
- upper.md3
- lower.md3
- head_default.skin
- upper_default.skin
- lower_default.skin
- 至少一张.tga 或.jpg 格式的纹理贴图
- icon_default.tga（一张 64×64 像素、24 位的图像，并具有一个 alpha 掩码）

如果要在“Capture the Flag（夺旗）”模式[1]中使用模型的话，还需要额外的 CTF 皮肤文件，如下所示：

- head_blue.skin
- head_red.skin
- upper_blue.skin
- upper_red.skin

译注[1]：*Quake 3* 和其他很多 FPS 游戏的一种玩法，对战方争夺一面旗帜。

- lower_blue.skin
- lower_red.skin
- 至少两张纹理贴图，每种颜色一张.tga 或.jpg
- icon_blue.tga
- icon_red.tga

为了创建你自己的 *Quake 3* 模型，文件的名字就必须遵循以下规范：

- head_角色名称.skin
- upper_角色名称.skin
- lower_角色名称.skin
- 角色的纹理
- icon_角色名称.tga

现在让我们来深入研究一下这些不同文件的细节。

12.1.1　Animation.cfg

Quake 3 模型的动画是通过“关键帧”来实现的。这种类型的动画是通过存储一个模型在各帧中的位置而实现的，然后根据这一系列的位置逐帧绘制出每个模型，即可形成动画的效果。

在 *Quake 3* 模型中，一个模型的不同变化被储存为不同的顶点和标记。因此，如果一个模型支持 246 帧不同的动画，那么组成该模型的每个.md3 文件中就需要保存 246 组不同的顶点和 246 个不同的标记（用于连接.md3 文件）。以上身为例，它通常具有 3 个标记，那么对于 246 帧的动画而言，它就需要保存 738（246×3）个不同的标记用于连接上身几何体。

附注

相对于关键帧动画而言，DirectX 中的 X 文件格式使用的却是另一种基于骨骼的动画系统，如半条命（Half-Life）、重金属（F.A.K.K.2）以及新近的虚幻 2（Unreal 2）引擎、毁灭战士 3（DOOM III）引擎，它们都是通过给模型加上一套骨骼来实现动画的。所谓“骨骼”指的是一系列绑定在模型的各个部位上的关节或骨头，通过操作这些关节或骨头来决定模型的各个部位将如何运动。这样的动画系统在运行期间只需很小的存储开销，但它要求更强大的计算能力。

Quake 3 文件中的动画通常可以是任意长度、任意帧速率的。但动画名称必须完全符合命名规范。Biker 模型的 animation.cfg 文件如下：

```
// animation config file
sex      m

// first frame, num frames, looping frames, frames per second, name of animation
headoffset -4 0 0
```

```
footsteps boot
0   30      0       25  // BOTH_DEATH1
29  1       0       25  // BOTH_DEAD1
30  30      0       25  // BOTH_DEATH2
59  1       0       25  // BOTH_DEAD2
60  30      0       25  // BOTH_DEATH3
89  1       0       25  // BOTH_DEAD3

90  40      0       20  // TORSO_GESTURE

130 6       0       15  // TORSO_ATTACK
136 6       0       15  // TORSO_ATTACK2

142 5       0       20  // TORSO_DROP
147 4       0       20  // TORSO_RAISE

151 1       0       15  // TORSO_STAND
152 1       0       15  // TORSO_STAND2

153 9       9       20  // LEGS_WALKCR
162 8       8       20  // LEGS_WALK
170 13      13      24  // LEGS_RUN
183 10      10      24  // LEGS_BACK
193 10      10      15  // LEGS_SWIM

203 7       0       15  // LEGS_JUMP
210 5       0       17  // LEGS_LAND

215 8       0       15  // LEGS_JUMPB
223 4       0       17  // LEGS_LANDB

227 10      10      15  // LEGS_IDLE
237 9       9       15  // LEGS_IDLECR

246 7       6       15  // LEGS_TURN
```

就我所知，任何 animation.cfg 文件都是按以上这种形式创建的。其中 Sex 为 m 表示男性、f 表示女性、n 表示无性别（比如机器）。Sex 决定了游戏将如何显示关于这个角色的信息（比如，称之为“他”还是“她”），以及在缺少自定义声音时将使用哪些默认声音。

接下去是两个名为 footsteps 和 headoffset 的选项变量。角色的脚步声可设置为 boot、normal、energy、mech 或 flesh。headoffset 是 HUD[1]的偏移量，以 x、y、z 坐标表示。

再下面的内容便是一段基于关键帧动画的脚本。其中前 4 列的数字会发生变化，但行的数量是固定的。第 1 列是每个动画的起始帧。第 2 列是帧的数量，表示了动画的长度。另外存在一些所谓的“循环动画”，所以第 3 列是循环帧的数量。第 4 列是动画的速度，最后一列的双斜杠后面是动画的名称。动画名称在每个模型中都是相同的。

BOTH_DEATH*动画同时用于腿部对象和上身对象。一个死亡动画的最后一帧位于

译注：即“Heads Up Display”的缩写，一般是指汽车或飞机中的各种仪表盘，在 *Quake* 中是指屏幕底部一系列的状态显示栏目，诸如生命值、当前武器等。

BOTH_DEAD*动画中。

至于接下去的 TORSO_*动画，我想你应该已经猜到了，它们是用于上身对象的。TORSO_GESTURE 可以是任意长度的，它甚至还可以具有自己的声音。TORSO_ATTACK 是对应于那些具有射程武器的攻击动画。在 *Quake 3* 中，它必须正好为 6 帧的长度，因为它需要与武器的开火相同步。动画的第一帧是瞄准，第二帧是开火与后座力，余下帧则用于恢复到第一帧。TORSO_ATTACK2 也必须是一个 6 帧的攻击序列，表示使用 gauntlet[1]的攻击。它的第一帧是攻击准备，第二帧是进行攻击，其余帧则是为了恢复到第一帧。TORSO_DROP 是武器切换的第一个部分，必须正好为 5 帧的长度。武器切换的第二个部分是 TORSO_RAISE，它也必须为 5 帧的长度。TORSO_STAND 是手持射程武器时的站立动画，TORSO_STAND2 则是手持 gauntlet 时的站立动画。

LEGS_*是用于腿部的动画。LEGS_WALKCR 是蹲伏行走的循环动画。接下去是行走（LEGS_WALK）、奔跑（LEGS_RUN）、游泳（LEGS_SWIM）、向前跳跃（LEGS_JUMP）、向前着地（LEGS_LAND）、向后跳跃（LEGS_JUMPB）、向后着地（LEGS_LANDB）、后退或向后跑（LEGS_BACK）等循环动画。还有空闲（LEGS_IDLE）、蹲伏时的空闲（LEGS_IDLECR）、以及当玩家旋转的角度足以要求腿部旋转时的转动动画（LEGS_TURN）。

通常，所有的动画帧都储存在数组中，你可以通过给出数组中的帧号来调用相应的动画。

12.1.2　.skin 文件

所有模型需要用到的标记和纹理信息都存储在模型所在目录下的.skin 文件中（请注意，示例程序 CharacterEngine 使用了一种不同于原先 *Quake 3* 的*.skin 文件格式，该示例程序位于 CD-ROM 的“Chapter12\CharacterEngine”目录下）。这些文本文件记录了连接模型的标记，以及连接的皮肤信息。.skin 文件的名字与使用它的.md3 文件的名字相同。以下是 Biker 模型的*_default.skin 文件。

- head_default.skin（头部）

  tag_head,
  h_head,models/players/biker/biker_h.tga

- upper_default.skin（上身）

  tag_head,
  tag_weapon,
  u_larm,models/players/biker/biker.tga
  u_rarm,models/players/biker/biker.tga
  u_torso,models/players/biker/biker.tga
  tag_torso,

[1] 译注：*Quake 3* 中一把外形类似电锯，但又有点儿类似电棒的武器，它是最主要的近距离作战武器。

- lower_default.skin（下身）

  l_legs,models/players/biker/biker.tga

  tag_torso,

头部的.skin 文件仅描述了一个标记和一张纹理，标记由 head.md3 文件提供，而那张纹理将用于头部。上身的.skin 文件描述了 3 个网格和 3 个标记，这 3 个网格将使用同一张.tga 纹理文件。下身的.skin 文件描述了 lower.md3 中的网格将被贴上 biker.tga 位图文件，并且它只有一个标记。

纹理应与.md3 文件位于相同的目录下。你接下去就会看到，在.md3 文件中还有一张默认纹理的名字。如果你想要使用自制的纹理，则应在一个自定义的.skin 文件中声明它们。

12.1.3 纹理与 Shader 文件

.md3 文件所使用的纹理可以具有 32 位的 RGB 颜色深度。在 *Quake 3* 引擎中，纹理的宽度和高度应为 2 的幂次，并且宽度和高度都不能超过 256 个像素。*Quake 3* 引擎根据 shader 文件中的命令来渲染纹理，这些 shader 文件位于 scripts 目录下（请注意，示例程序 CharacterEngine 使用了另一种不同于原先 *Quake 3* 的 shader 文件格式）。一个模型中的每个网格都可以具有各自不同的纹理和 shader 属性。

shader 文件是一种文本文件，就像 animation.cfg 和.skin 文件一样，你可以通过文本编辑器来操作它们。shader 文件是以一种专门的语言来编写的，由许多命令组成。shader 是一种简单的脚本，*Quake 3* 引擎用它来决定多边形表面的属性和渲染方式，最常见的做法是将 shader 用于控制关卡中墙壁的属性。shader 提供了许多命令来控制物体纹理的各项参数，由于 *Quake* 系列使用 OpenGL 作为其 3D API，所以这些命令非常类似于 OpenGL 中的纹理函数。你可以在本书的第二部分中找到这些 OpenGL 纹理函数在 Direct3D 中的等价物。你也可以浏览一下 models.shader 文件以窥其究竟，该文件位于 *Quake 3* 的 pak 文件中的 scripts 目录下。

以下是对一个 shader 文件的剖析：

```
// a comment
[the shader name]
{
    [surface attributes]

    // a stage
    {
        [stage attributes]
    }

    [more stages]
}
```

正如你所料想的，注释是通过标准 C++的双斜杠设置的。每个 shader 都以其名字作为起始，名字应全为小写，并小于 64 个字符。斜杠必须是正斜杠（/），shader 语句也必须包含

在花括号（{}）内。

shader 一共有三种属性类型。surface 属性用于控制连接着该 shader 的全部多边形表面的行为（比如，表面是水还是熔岩）。stage 或 content 属性用于控制表面的外观。而 deformation 属性则是通过某些方式来对一个表面的顶点作变形。

通过.md3 模型，你便可以专心于 stage 或 content 属性上了。就我所知，models.shader 文件只使用到了这些内容。若想获取一份 *Quake 3* 所使用的所有 shader 命令的参考，请查阅 ftp://ftp.idsoftware.com/idstuff/quake3/tools/Q3Ashader_manual.doc。这是由 id Software 编写的一份官方 shader 文档。

让我们来研究一下 model.shader 文件中对应于“DOOM marine”模型的那段命令，我已将这段 shader 脚本复制到了下面。

```
models/players/doom/phobos_f
{
    {
        map textures/effects/tinfx.tga
        tcGen environment
        blendFunc GL_ONE GL_ZERO
        rgbGen lightingDiffuse
    }
    {
        map models/players/doom/phobos_f.tga
        blendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA
        rgbGen lightingDiffuse
    }
}

models/players/doom/phobos
{
    {
        map models/players/doom/phobos_fx.tga
        blendFunc GL_ONE GL_ZERO
        tcmod scale 7 7
        tcMod scroll 5 -5
        tcmod rotate 360
        rgbGen identity
    }
    {
        map models/players/doom/phobos.tga
        blendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA
        rgbGen lightingDiffuse
    }
}

models/players/doom/f_doom
{
    {
        map models/players/doom/f_doom.tga
    }
    {
        map models/players/doom/fx_doom.tga
```

```
            tcGen environment
            rgbGen lightingDiffuse
            blendfunc gl_ONE gl_ONE
        }
    }

    models/players/doom/doom_f
    {
        {
            map models/players/doom/doom_f.tga
            rgbGen lightingDiffuse
        }
        {
            map models/players/doom/doom_fx.tga
            tcGen environment
            rgbGen lightingDiffuse
            blendfunc gl_ONE gl_ONE
        }
    }
```

这个 DOOM 模型的几何体用到了 4 个不同的纹理包。先从 phobos_f 模型开始看。

```
map textures/effects/tinfx.tga
```

这行代码的意思是要求对模型使用图 12.3 所示的纹理贴图。

图 12.3　环境贴图

这个是一张全局效果纹理，许多的模型、墙壁等都用到了这张纹理。以下代码则指示渲染器生成环境映射所需的纹理坐标。

```
tcGen environment
```

为了将纹理与漫反射光照相混合，以下两行代码设置了 OpenGL 的混合参数。

```
blendFunc GL_ONE GL_ZERO
rgbGen lightingDiffuse
```

混合函数（blendFunc）在这里作为命令关键字，它告诉 *Quake 3* 图形引擎的渲染器应如何混合各个图像层。例如，如何混合不同的纹理贴图，或者如何混合具有不同颜色或光照的纹理贴图。

blendFunc（即“blending function”，混合函数）是 shader 图形处理核心中的一个方程式。alpha 混合公式如下：

```
FinalColor = SourcePixelColor * SourceBlendFactor + DestPixelColor * DestBlendFactor
```

SourcePixelColor 通常是.tga 纹理文件中的 RGB 颜色数据，可通过 rgbGen 和 alphaGen 来修改这类数据。在 shader 中，SourcePixelColor 一般是由 MAP 命令后面的图像文件名来标识的，而 DestPixelColor 则是当前位于帧缓冲器中的颜色数据。

正如你在第 8 章中所学到的，每次（pass）纹理渲染或每个混合阶段都是以累积方式将当前数据与前一阶段传来的数据相混合。数据的组合方式依赖于每个阶段的源/目的混合系数。在 *Quake 3* 引擎中，除了 GL_ONE 和 GL_SRC_ALPHA（其中 alpha 通道为完全的白色）以外，任何 SourceBlendFactor 系数都会使源颜色变得更暗。SourceBlendFactor 有许多可选值，OpenGL 使用这些值作为纹理映射的混合系数。请查阅 shader 手册和相关 OpenGL 函数的参考（可搜索 glBlendFunc()）来了解这些参数的不同含义。请记住，Direct3D 也提供了等价的函数与参数。

由于在此使用了参数 GL_ONE，所以源颜色的值与 SourceBlendFactor 相乘后没有任何变化。目的混合系数 DestBlendFactor 为 GL_ZERO，该值就是 0。因此，目的乘法后，所有目的 RGB 数据都将变为 0，即变为黑色。

接下来，将源和目的两边的乘法结果相加，再将相加结果存入帧缓冲器中，成为下一阶段的目的。“rgbGen lightingDiffuse”表示使用顶点法线来计算光照，从而获得物体的颜色。

因此，第一阶段指示了渲染器混合纹理 tinfx.tga，并通过漫反射光照来计算颜色值，同时还创建了环境映射的纹理坐标。

然后转到下一个阶段。在第二个阶段中将另一张纹理 phobos_f.tga 与第一张纹理相混合（见图 12.4）。

```
map models/players/doom/phobos_f.tga
```

alpha 混合则使用了以下 OpenGL 参数：

```
blendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA
```

图 12.4 *Quake 3* 中“DOOM marine”的护目镜
（id Software 授权）

这些参数类似于以下 Direct3D 参数：

```
D3DBLEND_SCRALPHA D3DBLEND_INVSRCALPHA
```

请查阅第 8 章中讲述多纹理的部分，以获取对这些参数的描述。

最后的 rgbGen 指示渲染器通过定向光和 Gouraud 着色来计算表面的颜色值，并将 alpha 混合的结果与表面的颜色值相乘。

好，看完了 phobos_f 模型。现在，再来研究一下 phobos 模型。

在此需要混合一张名为 phobos_fx.tga 的纹理贴图，如图 12.5 所示。混合参数为 GL_ONE 和 GL_ZERO，它们类似于 Direct3D 中的 D3DBLEND_ONE 和 D3DBLEND_ZERO。

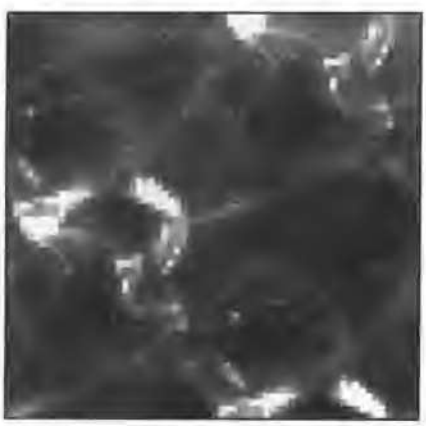


图 12.5 *Quake 3* 中 DOOM/Phobos 模型的效果纹理贴图
（id Software 授权）

命令“tcMod scale 7 7”通过乘以给定的系数来重新调整（扩大或收缩）纹理坐标的大小。这个公式沿着相应轴、按照 1/n 的比例来缩放某阶段中的纹理。“tcMod scale 0.5 0.5”将使图像扩大一倍，而“tcMod scale 2 2”则将使图像缩小一倍。“tcMod scroll”用于连续地滚动

某阶段中的纹理，每秒滚动的幅度通过偏移量 x 和 y 来指定。滚动速度以“纹理/秒”来度量，其中的纹理是指原始.tga 纹理经由前面一系列的 shader 修改后的结果。负的 x 值将纹理向左滚动，负的 y 值将纹理向下滚动。因此，“tcMod scroll 0.5 −0.5”将纹理以每秒半张纹理的速度向下和向右滚动。

“tcMod rotate”按每秒特定的度数旋转某阶段中的纹理。正的度数将以顺时针方向旋转，而负的度数则将以逆时针方向旋转。当前代码“tcMod rotate 360”是将阶段中的纹理以每秒 360 度的速度作旋转。至于最后的 rghGen，它的参数表示保持 RGB 值不变。

总而言之，第一阶段加载了一张效果纹理，然后对其作了缩放、滚动和旋转。在下一阶段中，加载了默认的纹理贴图 phobos.tga，并通过漫反射光照进行颜色计算，然后再将这张纹理与第一阶段中的纹理相混合。混合过程是将来自于第一阶段的目的纹理与 GL_ONE_MINUS_SRC_ALPHA 相乘，来自于第二阶段的源纹理与 GL_SRC_ALPHA 相乘，最后将这两个乘积相加。

整个 alpha 混合公式如下：

```
FinalColor = SourcePixelColor * GL_SRC_ALPHA + DestPixelColor * GL_ONE_MINUS_SRC_ALPHA
```

第一个参数对应于 Direct3D 中的 D3DBLEND_INVSRCALPHA，第二个参数对应于 Direct3D 中的 D3DBLEND_SRCALPHA。

你可能已经注意到了，*Quake 3* 中的 shader 语言是一种十分强大的脚本语言，它直接对应于 *Quake 3* 渲染引擎所使用的 OpenGL 函数。

我希望你对 *Quake 3* 的 shader 已有了一个大致的概念。本章附带的示例程序使用了 DirectX 9 SDK 的效果（effect）文件格式，它类似于这里的 *Quake 3* shader 文件格式，但更为先进。

12.1.4　自定义声音

你需要将声音放到“sounds/player/<modelname>”目录下。在 *Quake 3* 中，.wav 文件的格式应为 22.5kHz、16 位、单通道（单声道）。*Quake 3* 引擎要求每个声音文件都要有一个专门的名字。这些文件的长度可以是任意的，但通常较长的文件会导致一些性能问题。如果一个模型没有提供自定义的声音文件，那么会根据模型的性别不同，将模型 sarge 或 major[1]的声音文件作为默认的声音文件。表 12.2 列出了 *Quake 3* 引擎所要求的声音文件。

所有模型的脚步声音文件都位于“sounds/player/footsteps”目录下。根据 animation.cfg 文件中提供的 footsteps 参数来选择适当的文件，每种脚步声音总对应于 4 个文件。如果你在 animation.cfg 文件中指定了 energy，则即对应于文件 energy1.wav～energy4.wav。

译注 1：这两个也是 *Quake 3* 中的游戏角色。

表 12.2　　Quake 3 引擎所要求的声音文件

文件	描述
death1.wav	死亡的声音
death2.wav	死亡的声音
death3.wav	死亡的声音
drown.wav	溺死的声音
gasp.wav	在水下呼吸困难的声音
falling1.wav	坠落的声音
fall1.wav	落下或高跳的碰撞声
jump1.wav	跳跃的声音
pain100.wav	由健康值决定的痛苦声音
pain75_1.wav	75%健康值时痛苦的声音
pain50_1.wav	50%健康值时痛苦的声音
pain25_1.wav	25%健康值时痛苦的声音
taunt.wav	做挑衅姿势时的声音

接下去，我们将转至.md3 文件格式的内部。为了更进一步地认识 *Quake 3* 的.md3 文件，现在必须展开你的手术刀了。

12.1.5　.md3 格式

id Software 直到最近才以官方文档的形式对.md3 文件格式作了描述。不过据我所知，随着时间的流逝，其中的一些描述已经发生了变化。因此，以下的介绍也将顺应于这些改变。

有许多种方法可以创建.md3 文件。比如，http://www.planetquake.com/polycount 以及其他一些介绍 *Quake 3* 修改的网站都会提供免费的.md3 文件下载。并且，这些网站总会提供一份工具列表，来满足你自己动手的想法。

你还可以找到许多 3D Studio Max 或其他建模软件的导出插件。如果你对这些建模软件已经很熟悉的话，那就最好不过了。如果你像我一样对这些软件并不熟悉，那么我建议你使用 MilkShape 3D，它可以辅助你创建出具有动画的.md3 文件。它是一款共享软件，作者开发它的惟一目的就是为各种各样的 3D 射击游戏创建模型。在下载完这款软件后，你可以通过信用卡在线注册，它的最新价格为$25。在 MilkShape 网站上可以查阅到相关的指南（http://www.swissquake.ch/chumbalum-soft/index.html）。

你可以使用任何自己喜好的图像处理软件来制作各种.tga 或.jpg 格式的皮肤，比如 Adobe Photoshop 或 Jasc Paint Shop Pro。

为了查看自己的作品，你可以使用 Java MD3 Model Viewer，它位于 http://fragland.net/md3view/download.html。就现在而言，它应该还是完成这类任务最好的程序。安装过程中会有一点点棘手的问题，因此，请留意一下它的安装指导。

一个.md3 文件的内容可以分为以下几类：

- .md3 首部；
- .md3 骨骼帧（boneframe）；
- .md3 标记；
- .md3 网格。

每个.md3 网格又由以下这些部分组成：

- Mesh 首部；
- Mesh 纹理名称；
- Mesh 三角形；
- Mesh 纹理坐标；
- Mesh 顶点。

这些不同部分的次序也反映了.md3 文件的读入次序。

Md3.h

.md3 首部保存了整个.md3 文件的所有关键信息。源文件 md3.h 中包含了一个用于保存这些信息的结构。

```
typedef struct
{
    char id[4];             // id of file, always "IDP3"
    int iVersion;           // version number
    char cFileName[68];
    int iBoneFrameNum;      // Number of animation key frames
                            // in the whole model=every mesh
    int iTagNum;            // Number of tags
    int iMeshNum;           // Number of meshes
    int iMaxTextureNum;     // maximum number of unique textures used in a md3 model
    int iHeaderSize;        // size of header
    int iTagStart;          // starting position of tag frame structures
    int iMeshStart;         // starting position of mesh structures
    int iFileSize;
} MD3HEADER;
```

iVersion 变量用于保存版本号，在这里应使用 15 作为版本号。iHeaderSize 应总为 108 个字节。整个模型的动画关键帧数量应与该模型中每个网格的动画关键帧数量相等。据我了解，iMaxTextureNum 总为 0，因此，它似乎并没有被用到。这个首部结构作为读取后续不同文件内容的起点，它还可用于检查你的.md3 文件读取程序所读取的长度是否正确。

当检查完首部结构后，接着需要从.md3 文件中读取骨骼帧结构。

```
typedef struct
{
    FLOAT mins[3];          // first corner of the bounding box
    FLOAT maxs[3];          // second corner of the bounding box
    FLOAT position[3];      // position/origin of the bounding box
    FLOAT scale;            // radius of bounding sphere
```

```
    char name[16];          // name of frame ASCII character string, NULL terminated
} MD3BONEFRAME;
```

每个骨骼帧结构都为一个动画帧保存着一个包裹盒（bounding box），包裹盒可用于进行碰撞检测。name 数组是该骨骼帧的名称。如首部所示，整个模型中会有 iBoneFrameNum 个关键帧。

你或许还记得，关键帧动画是通过存储模型的不同位置来实现的，如此便可以画出一系列不同的模型。在.md3 文件中，模型的不同位置储存在各个不同的标记、包裹盒和顶点中。

存储下来的每组模型位置称为“一帧”。对于每种动画而言，每个网格保存的帧数都是相同的，每一帧都包含了标记、包裹盒和顶点。若一个模型只有一个网格，并具有 12 帧和一个标记，则有：

- 12 个包裹盒；
- 12 个标记；
- 12 组顶点。

现在来看一下标记。至少会有与骨骼帧一样多的标记。

```
typedef struct
{
    char name[64];
    FLOAT position[3];      // position of tag relative to the model that contains
                            // the tag
    FLOAT rotation[3][3];   // the direction the tag is facing
} MD3TAG;
```

标记所面对的方向储存在 3×3 矩阵中，而位置则储存在三个 float 值中。

在连接模型各部分的标记之后，便是.md3 模型中最重要的部分——网格。每个.md3 模型都可以由任意多个不同的网格组成。通常网格的数量对应于需要贴以不同纹理的部分数量。例如，“DOOM marine”的头盔就是由护目镜和头盔的其余部分两个网格组成的。你可以给护目镜同时贴上一张护目镜纹理贴图和一张环境纹理贴图，而给头盔的其余部分贴上默认的纹理贴图。

接下去来看一下网格的首部，它保存了这个网格所需加载的纹理名称、三角形、纹理坐标、顶点等方面的信息。

```
typedef struct
{
    char cId[4];
    char cName[68];
    int iMeshFrameNum;      // number of frames in mesh
    int iTextureNum;        // number of textures=skins in this mesh
    int iVertexNum;         // number of vertices in this mesh
    int iTriangleNum;       // number of triangles
    int iTriangleStart;     // start of triangle data, rel. to MD3MESHFILE
    int iHeaderSize;        // Headersize = starting position of texture data
    int iTecVecStart;       // starting position of the texture vector data
    int iVertexStart;       // starting position of the vertex data
```

```
    int iMeshSize;
} MD3MESHFILE;
```

大多数变量的名字已经解释了它们的含义。纹理的名称起始于 iHeaderSize 个字节之后，iHeaderSize 通常为 108。将纹理的数量乘以 68，可得到三角形数据的起始偏移量，即 iTriangle Start。将 iTriangleNum 乘以 TRIANGLEVERT 数据类型的大小，并加上 iTriangleStart 后，可得到纹理坐标的起始偏移量，即 iTecVecStart。将顶点的数量乘以 TEXCOORDS 结构的大小，可得到顶点数据的起始偏移量，即 iVertexStart。由于文件为每一帧保存一组顶点，所以将顶点数量与帧数量相乘，再乘以顶点数据的大小，即可到达网格的结尾处。

```
typedef struct
{
    SHORT vec[3];
    UCHAR unknown;
} MD3VERTEX;
```

MD3VERTEX 结构保存了一个短整型的数组和一个未知的 UCHAR，这个 UCHAR 估计是用于保证 4 字节对齐的。

12.2　CharacterEngine 示例程序

CharacterEngine 示例程序位于 CD-ROM 的“Chapter12\CharacterEngine”目录下，如图 12.6 所示。这个示例程序说明了以下技术：

- 如何加载一个完整的 *Quake 3* 模型的几何数据，并播放其动画；
- 如何增强这种文件格式；

图 12.6　CharacterEngine 示例程序

- 如何使用顶点 shader 和像素 shaders 来对模型着色和播放动画；
- 如何为.md3 模型实现一套 shader 级别（Level of Shading，LOS）系统。

请注意，railgun[1]模型由 Lee David Ash 提供并授权使用（可以在 http://www.planetquake.com/polycount 找到相关内容）。龙的模型由 Michael Mellor（Magarnigal）制作（可以在 http://www.planetquake.com/polycount 找到相关内容）。

图 12.7 的左上角显示了上身和下身所有可能的动画（按 A 键可显示出该菜单）。你可以使用 J、L、K、I 键来选择动画。

图 12.7　动画菜单系统

按 O 键可以访问到所有的其他选项，如图 12.8 所示。按 G 键可以打开或关闭武器，按 W 键可以打开或关闭线框模式，按数字键盘上的+和−键可以在不同的 shader 配置文件中循环切换，按 P 键和箭头键可以移动光源。

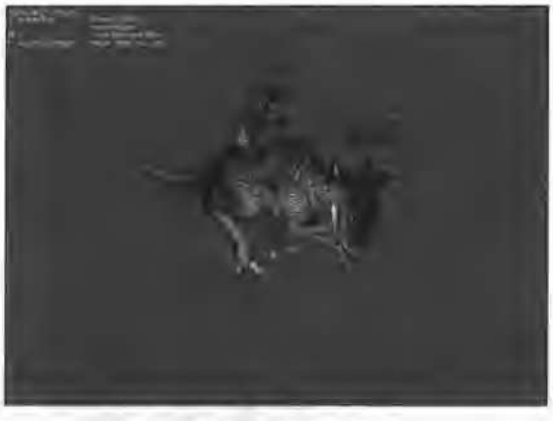


图 12.8　选项菜单

如图 12.9 所示，光的位置通过一个黄色的球体来表示。你可以来回地移动它，以便观察示例中的表面是如何反射光照的。

译注[1]：*Quake 3* 中的一种枪。

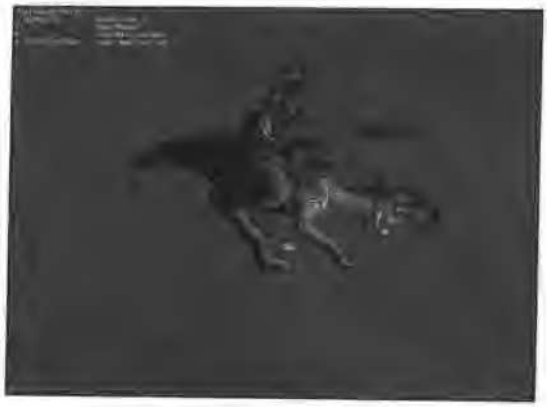


图 12.9　活动的光源

以下几个小节将一步步地带领你了解这些特征。

.md3 模型的加载与动画

所有用于.md3 文件加载、动画和着色的代码都在 md3.h 和 md3.cpp 中。这些代码由两个类构成：Q3Player 和 CMD3Model。Q3Player 是在 CMD3Model 之上的一个抽象层。它包含了一个模型所需的全部功能，通常一个模型由 4 个.md3 文件构成（在这个示例中，3 个.md3 文件用于龙和骑手，1 个用于武器）。

Q3Player 类的方法如下：

```
class Q3Player
{
public:
    Q3Player();
    ~Q3Player();

    VOID LoadPlayerGeometry(TCHAR *path);
    VOID DeletePlayerGeometry(VOID);

    VOID LoadSkins(LPDIRECT3DDEVICE9 m_pd3dDevice, char *path, char *skin);
    VOID FreeSkins();

    VOID LoadPlayerShaders(LPDIRECT3DDEVICE9 m_pd3dDevice, char *cPath, char *cSkin);
    VOID FreePlayerShaders(LPDIRECT3DDEVICE9 m_pd3dDevice);
    VOID LoadPlayerShaderProfile(char *pcFileName);

    HRESULT CreateVertexNIndexBuffer(LPDIRECT3DDEVICE9 m_pd3dDevice);
    VOID DeleteVertexNIndexBuffer();

    VOID Draw(LPDIRECT3DDEVICE9 m_pd3dDevice);

    VOID Update(FLOAT time);
    VOID LoadAnim(TCHAR *filename);
```

```
        VOID Update(FLOAT time);
        VOID LoadAnim(TCHAR *filename);
        VOID SetLowerAnim(int iAnimNumber);
        VOID SetUpperAnim(int iAnimNumber);
        VOID DumpAnimInfo();

        VOID LoadWeaponGeometry(char *cPath);
        VOID DeleteWeaponGeometry();

        VOID LoadWeaponSkins(LPDIRECT3DDEVICE9 m_pd3dDevice, char *cPath, char *skin);
        VOID FreeWeaponSkins();

        VOID LoadWeaponShaders(LPDIRECT3DDEVICE9 m_pd3dDevice, char *cPath, char *skin);
        VOID FreeWeaponShaders(LPDIRECT3DDEVICE9 m_pd3dDevice);
        VOID LoadWeaponShaderProfile(char *pcFileName);

        VOID SetWorldMatrix(D3DXMATRIX &matW) { matWorld=matW;}
        VOID SetViewProjMatrix(D3DXMATRIX matVP) { matViewProj=matVP; }

        CMD3Model md3Lower, md3Upper, md3Head, md3Weapon;

        LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclaration;

    private:
        MD3ANIM playerAnim[25];

        int playerAnimLower;
        int playerAnimUpper;

        LPDIRECT3DINDEXBUFFER9 m_pIB;
        LPDIRECT3DVERTEXBUFFER9 m_pVB;

        D3DXMATRIX matViewProj;
        D3DXMATRIX matWorld;
    };
```

在构造函数和析构函数之后，LoadPlayerGeometry()和 DeletePlayerGeometry()函数分别用于加载和删除龙与骑手的几何数据。所有的纹理都通过 LoadSkins()和 FreeSkins()函数来加载和释放。接下去 3 个函数用于加载和释放所有与 shader 相关的东西。其中，LoadPlayerShaders()和 FreePlayerShaders()分别加载和释放所有用于龙与骑手的顶点 shader 和像素 shader，而 LoadPlayerShaderProfile()用于加载 shader 的配置文件。这些配置文件为每个 shader 级别中的每个网格定义了一对特定的 shader。CreateVertexNIndexBuffer()和 DeleteVertexNIndexBuffer()用于创建和删除顶点缓冲器与索引缓冲器。所有几何数据都是每帧动态地写入顶点缓冲器和索引缓冲器中的。Draw()函数处理了所有绘制模型时所需的命令。Update()用于更新动画时间。LoadAnim()从 animation.cfg 中加载所有的动画，SetLowerAnim()和 SetUpperAnim()函数用于设置具体的动画。最后一个动画函数 DumpAnimInfo()将所有的动画数据转存到一个日志文件中。所有名字中含有 Weapon 一词的函数都等价于上述介绍的含有 Player 一词的函数。SetWorldMatrix()和 SetViewProjMatrix()函数用于设置主程序传入的矩阵。4 个不同的.md3 文件分别储存在 md3Lower、md3Upper、md3Head、md3Weapon 对象中。请注意，现在这些对

以加载 1 个、3 个或更多的.md3 文件。对于所有的顶点 shader，只有一个顶点声明。这个声明对应于顶点缓冲器中的内容。

CMD3Model 类则用于处理所有的底层事务，这个类包含了 Q3Player 类会调用的底层函数：

```
class CMD3Model
{
public:
    CMD3Model();
    ~CMD3Model();

    int LoadModelGeometry (TCHAR *filename);
    VOID DeleteModelGeometry(VOID);
    VOID DumpGeometryInfo(VOID);
    VOID UpdateFrameTime (FLOAT fTime);

    VOID DrawSkeleton (CMD3Model *md3Model,
                       LPDIRECT3DDEVICE9 m_pd3dDevice,
                       LPDIRECT3DINDEXBUFFER9 m_pIB,
                       LPDIRECT3DVERTEXBUFFER9 m_pVB,
                       D3DXMATRIX *viewProj,
                       D3DXMATRIX *world,
                       LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclaration);

    int LinkModel (TCHAR *cTagName, CMD3Model *mod);
    VOID UnLinkModel (TCHAR *cTagName);

    VOID  LoadSkins  (LPDIRECT3DDEVICE9  m_pd3dDevice,  TCHAR  *filename,  TCHAR
  *imagepath);
    VOID DeleteSkins(VOID);
    VOID DumpSkinInfo(VOID);

    VOID  LoadShaders(LPDIRECT3DDEVICE9  m_pd3dDevice,  char  *pcFileName,  char
  *pcShaderPath);
    VOID DeleteShaders(LPDIRECT3DDEVICE9 m_pd3dDevice);
    VOID DumpShaderInfo(VOID);

    VOID CalculateNormals2(MD3MESH *MD3Mesh);

    VOID ComputeDuDv(const D3DXVECTOR3& v0_pos, const D3DXVECTOR2& v0_uv,
                     const D3DXVECTOR3& v1_pos, const D3DXVECTOR2& v1_uv,
                     const D3DXVECTOR3& v2_pos, const D3DXVECTOR2& v2_uv,
                     TANGENTS& meshTangents);
    VOID AverageTriangles(int index, int tri_count, int frame, MD3MESH *MD3Mesh);
    VOID GenerateTangent(MD3MESH *MD3Mesh);

    int iFrame; // current frame to draw
    int iNextFrame; // next frame to draw
    int iFps; // frames per second

    int iStartFrame;
    int iEndFrame;

    MD3MESH *md3Meshes;
```

```
    MD3HEADER modelHeader;

private:
    TCHAR             cModelName[512];  // name for the md3 file
    MD3BONEFRAME  *md3BoneFrames;
    D3DTAG            *d3dTag;          // Direct3D specific

    CMD3Model         **md3Links;
    FLOAT             fOldTime;
    FLOAT             fNewTime;
    IDirect3DTexture9 *pMiniTextureCache[MAXTEXTURESPERMESH];
};
```

几何数据通过那些名字中包含 Geometry 一词的函数来加载，纹理通过那些名字中包含 Skin 一词的函数来加载，shader 则通过那些名字中包含 Shader 一词的函数来加载。CalculateNormals2()、ComputeDuDv()、AverageTriangles()、GenerateTangent()4 个函数通过计算法线和一个切线向量来创建纹理空间坐标系。其中，GenerateTangent()调用了 ComputeDuDv()和 AverageTriangles()。

CMD3Model 类中所有后续结构对象的类型都定义在 md3.h 文件中，并且已在前一节中作过了相关的描述。请注意这个示例使用了一个纹理 cache（高速缓冲存储器），可通过它来跟踪纹理的变化。这种纹理 cache 技术可以保证，直到网格的纹理发生变化时才会调用 SetTexture()函数。

以下几个小节将介绍 CMD3Model 类中的几个最重要的函数，其中包括：LoadModelGeometry()、LoadShaders()和 DrawSkeleton()。

1. LoadModelGeometry()

LoadModelGeometry()函数用于加载一个.md3 文件中所有的几何数据。它还计算了法线和切线，并将它们存储到*.n 和*.tan 文件中（这是两种压缩的格式）。一个.md3 文件的内容可以分为以下几类，同时这个函数也是按以下次序来读取几何数据的：

- md3 首部；
- md3 骨骼帧；
- md3 标记；
- md3 网格；
- 网格首部；
- 网格纹理名称；
- 网格三角形；
- 网格纹理坐标；
- 网格顶点。

整个函数如下：

```
1:    int CMD3Model::LoadModelGeometry(char *filename)
```

```
2:    {
3:        FILE *md3file;
4:        int i, j;
5:
6:        // check if file exist
7:        if ( CheckFile(filename) == 0)
8:            return -1;
9:
10:       // open file
11:       md3file = fopen(filename, "rb");
12:
13:       // copy name
14:       _tcscpy(cModelName, filename);
15:
16:
17:       // read header
18:       fread(&modelHeader, 1, sizeof(MD3HEADER), md3file);
19:
20:       static char ver[5]; // 译注
21:       wsprintf(ver, "%c%c%c%c", modelHeader.id[0], modelHeader.id[1],
22:                modelHeader.id[2], modelHeader.id[3]);
23:
24:       if ( strcmp(ver, "IDP3") || modelHeader.iVersion != 15)
25:       {
26:           fclose(md3file);
27:           return -2;
28:       }
29:
30:       // read boneframes
31:       md3BoneFrames = (MD3BONEFRAME *) malloc(
32:           sizeof(MD3BONEFRAME) * modelHeader.iBoneFrameNum);
33:       fread(md3BoneFrames, sizeof(MD3BONEFRAME),
34:             modelHeader.iBoneFrameNum, md3file);
35:
36:       // read tags
37:       MD3TAG *tempTag;
38:       tempTag = (MD3TAG *) malloc(
39:           sizeof(MD3TAG) * modelHeader.iBoneFrameNum * modelHeader.iTagNum);
40:       d3dTag = (D3DTAG *) malloc(
41:           sizeof(D3DTAG) * modelHeader.iBoneFrameNum * modelHeader.iTagNum);
42:       fread(tempTag, sizeof(MD3TAG),
43:             modelHeader.iBoneFrameNum * modelHeader.iTagNum, md3file);
44:
45:       for (i=0; i < modelHeader.iBoneFrameNum * modelHeader.iTagNum; i++)
46:       {
47:           _tcscpy (d3dTag[i].name, tempTag[i].name);
48:           d3dTag[i].matTag(0, 0) = tempTag[i].rotation[0][0];
49:           d3dTag[i].matTag(0, 1) = tempTag[i].rotation[0][1];
50:           d3dTag[i].matTag(0, 2) = tempTag[i].rotation[0][2];
51:           d3dTag[i].matTag(0, 3) = 0.0f;
52:           d3dTag[i].matTag(1, 0) = tempTag[i].rotation[1][0];
53:           d3dTag[i].matTag(1, 1) = tempTag[i].rotation[1][1];
```

译注：在原书和 CD-ROM 的源代码中，此处声明的都是一个包含 4 个 char 的数组，而下面的 wsprintf 将要格式化一个包含 4 个字符的字符串，所以必须要有 5 个 char 的空间才够用（因为结尾还有一个'\0'）。所以，译者在此将数组长度改为 5。

```
54:             d3dTag[i].matTag(1, 2) = tempTag[i].rotation[1][2];
55:             d3dTag[i].matTag(1, 3) = 0.0f;
56:             d3dTag[i].matTag(2, 0) = tempTag[i].rotation[2][0];
57:             d3dTag[i].matTag(2, 1) = tempTag[i].rotation[2][1];
58:             d3dTag[i].matTag(2, 2) = tempTag[i].rotation[2][2];
59:             d3dTag[i].matTag(2, 3) = 0.0f;
60:             d3dTag[i].matTag(3, 0) = tempTag[i].position[0];
61:             d3dTag[i].matTag(3, 1) = tempTag[i].position[1];
62:             d3dTag[i].matTag(3, 2) = tempTag[i].position[2];
63:             d3dTag[i].matTag(3, 3) = 1.0f;
64:         }
65:
66:         delete tempTag;
67:
68:         // init links
69:         md3Links = (CMD3Model **) malloc(
70:             sizeof(CMD3Model) * modelHeader.iTagNum);
71:
72:         for (i=0; i<modelHeader.iTagNum; i++)
73:             md3Links[i] = NULL;
74:
75:         // read meshes
76:         md3Meshes = (MD3MESH *) malloc(
77:             sizeof(MD3MESH) * modelHeader.iMeshNum);
78:
79:         LONG lMeshOffset = ftell(md3file);
80:
81:         for (i=0; i<modelHeader.iMeshNum; i++)
82:         {
83:             fseek(md3file, lMeshOffset, SEEK_SET);
84:             fread(&md3Meshes[ i ].meshHeader, sizeof(MD3MESHHEADER),
85:                   1, md3file);
86:
87:             // ----------------------
88:             md3Meshes[ i ].meshSkins = (MD3SKIN *)malloc(
89:                 sizeof(MD3SKIN) * md3Meshes[ i ].meshHeader.iTextureNum);
90:             fread(md3Meshes[ i ].meshSkins, sizeof(MD3SKIN),
91:                   md3Meshes[ i ].meshHeader.iTextureNum, md3file);
92:
93:             // ----------------------
94:             fseek(md3file,
95:                   lMeshOffset+md3Meshes[ i ].meshHeader.iTriangleStart,
96:                   SEEK_SET);
97:             md3Meshes[ i ].meshTriangles = (MD3TRIANGLE *) malloc(
98:                 sizeof(MD3TRIANGLE) * md3Meshes[ i ].meshHeader.iTriangleNum);
99:             fread(md3Meshes[ i ].meshTriangles, sizeof(MD3TRIANGLE),
100:                  md3Meshes[ i ].meshHeader.iTriangleNum, md3file);
101:
102:            // ----------------------
103:            fseek(md3file,
104:                  lMeshOffset+md3Meshes[ i ].meshHeader.iTecVecStart,
105:                  SEEK_SET);
106:            md3Meshes[ i ].meshTexCoord = (MD3TEXCOORD *) malloc(
107:                sizeof(MD3TEXCOORD) * md3Meshes[ i ].meshHeader.iVertexNum);
108:            fread(md3Meshes[ i ].meshTexCoord,
109:                  sizeof(MD3TEXCOORD),
```

```
110:                md3Meshes[ i ].meshHeader.iVertexNum, md3file);
111:
112:            // ----------------------
113:            fseek(md3file,
114:                  lMeshOffset+md3Meshes[ i ].meshHeader.iVertexStart,
115:                  SEEK_SET);
116:            md3Meshes[ i ].meshVertices = (MD3VERTEXEX *) malloc(
117:                sizeof(MD3VERTEXEX) * md3Meshes[ i ].meshHeader.iVertexNum *
118:                md3Meshes[ i ].meshHeader.iMeshFrameNum);
119:
120:            MD3VERTEX *meshTempVertices;
121:            meshTempVertices = (MD3VERTEX *) malloc(sizeof(MD3VERTEX) *
122:                md3Meshes[ i ].meshHeader.iVertexNum *
123:                md3Meshes[ i ].meshHeader.iMeshFrameNum);
124:            fread(meshTempVertices, sizeof(MD3VERTEX),
125:                md3Meshes[ i ].meshHeader.iMeshFrameNum *
126:                md3Meshes[ i ].meshHeader.iVertexNum, md3file);
127:
128:            for (j = 0; j < md3Meshes[ i ].meshHeader.iVertexNum *
129:                           md3Meshes[ i ].meshHeader.iMeshFrameNum; j++)
130:            {
131:                // The /64 is a constant scale factor that is used to
132:                // convert the md3 model verts from shorts to floats.
133:                md3Meshes[ i ].meshVertices [ j ].vVector[0] =
134:                    (FLOAT) meshTempVertices[j].sVector[0] / 64;
135:                md3Meshes[ i ].meshVertices [ j ].vVector[1] =
136:                    (FLOAT) meshTempVertices[j].sVector[1] / 64;
137:                md3Meshes[ i ].meshVertices [ j ].vVector[2] =
138:                    (FLOAT) meshTempVertices[j].sVector[2] / 64;
139:
140:                /*
141:                The normal information in the md3 file is stored as two
142:                unsigned chars to save space. These chars contain the
143:                spherical coordinates of the normal.
144:
145:                The first byte is longitude and the second byte is latitude.
146:                You may find This information in mathlib.c in the
147:                NormalToLatLong() function in the Quake3 tools source code.
148:                */
149:                // md3Meshes[ i ].meshVertices [ j ].vNormal =
150:                // fCalcNormals[meshTempVertices[j].cNormal[0]]
151:                // [meshTempVertices[j].cNormal[1]];
152:            }
153:            //----------------------
154:            // Calculate Normals and Tangents
155:            // store them in files for faster startup
156:            //----------------------
157:            FILE *File;
158:            DWORD dwNumberOfVertices;
159:            TCHAR cMeshName[68];
160:
161:            dwNumberOfVertices = md3Meshes[i].meshHeader.iMeshFrameNum *
162:                                 d3Meshes[i].meshHeader.iVertexNum;
163:
164:            // allocate memory for tangents
165:            md3Meshes[i].meshTangents = (TANGENTS *) malloc (
```

```
166:                                    sizeof(TANGENTS) * dwNumberOfVertices);
167:
168:          // Calculate Normals
169:          // store the normals in a file with the extension *.nor
170:          // read normals from this file to speed things up
171:          int z = 0;
172:
173:          // read in for example hunter\ when cModelName stores
174:          // hunter\head.md3
175:          while (!(cModelName[z - 1] == '\\'))
176:              cMeshName[z] = cModelName[z++];
177:
178:          // adds to hunter\ for example legs_1: hunter\legs_1
179:          for (int y = 0; y < strlen(md3Meshes[i].meshHeader.cName); y++)
180:              cMeshName[z++] = md3Meshes[i].meshHeader.cName[y];
181:
182:          // saves mesh name in an extra array and adds the terminating \0
183:          TCHAR cNormalName[68];
184:          memcpy (cNormalName, cMeshName, sizeof(cNormalName) );
185:          cNormalName[z] = '\0';
186:
187:          // add the appendix .n
188:          strcat (cNormalName, ".n");
189:
190:          // check if file exists
191:          if ( CheckFile(cNormalName) == 0)
192:          {
193:              CalculateNormals2(&md3Meshes[i]);
194:
195:              // open file
196:              File = fopen(cNormalName, "w+b");
197:
198:              for(int y = 0; y < dwNumberOfVertices; y++)
199:                  fwrite(
200:                  &md3Meshes[i].meshVertices[y].uiCompressedVertexNormal,
201:                      sizeof(UINT), 1, File);
202:
203:              // close file
204:              fclose (File);
205:          }
206:          else
207:          {
208:              // open file
209:              File = fopen(cNormalName, "rb");
210:
211:              for(int y = 0; y < dwNumberOfVertices; y++)
212:                  fread(
213:                  &md3Meshes[i].meshVertices[y].uiCompressedVertexNormal,
214:                      sizeof(UINT), 1, File);
215:
216:              // close file
217:              fclose(File);
218:          }
219:
220:          // Calculate tangent
221:          // store the tangent in a file with the extension *.tan
```

```
222:          // read tangents from this file to speed things up
223:          // check if file exists
224:          TCHAR cTangentName[68];
225:          memcpy (cTangentName, cMeshName, sizeof(cTangentName) );
226:          cTangentName[z] = '\0';
227:
228:          // add the appendix .t
229:          strcat (cTangentName, ".t");
230:
231:          // check if file exists
232:          if ( CheckFile(cTangentName) == 0)
233:          {
234:              GenerateTangent(&md3Meshes[i]);
235:
236:              // open file
237:              File = fopen(cTangentName, "w+b");
238:
239:              // write tangents into file
240:              for(int y = 0; y < dwNumberOfVertices; y++)
241:                  fwrite(
242:                      &md3Meshes[i].meshTangents[y].vCompressedTangent,
243:                      sizeof(UINT), 1, File);
244:
245:              // close file
246:              fclose (File);
247:          }
248:          else
249:          {
250:              // open file
251:              File = fopen(cTangentName, "rb");
252:
253:              // read tangents
254:              for(int y = 0; y < dwNumberOfVertices; y++)
255:                  fread(
256:                      &md3Meshes[i].meshTangents[y].vCompressedTangent,
257:                      sizeof(UINT), 1, File);
258:
259:              // close file
260:              fclose(File);
261:          }
262:
263:          delete meshTempVertices;
264:
265:          lMeshOffset += md3Meshes[ i ].meshHeader.iMeshSize;
266:          md3Meshes[i].iNumTextures = NULL;
267:
268:          for (j = 0; j < MAXSHADERPROFILE; j++)
269:          {
270:              md3Meshes[i].iChoosedShaderLevel[j] = NULL;
271:          }
272:
273:          for (j = 0; j < MAXSHADERLEVELINMESH; j++)
274:          {
275:              md3Meshes[i].pVertexShader[j] = 0;
276:              md3Meshes[i].pPixelShader[j] = 0;
277:          }
```

```
278:
279:            for (j = 0; j < MAXTEXTURESPERMESH; j++)
280:            {
281:                md3Meshes[i].pTexturesInterfaces[j] = 0;
282:            }
283:        }
284:
285:        // close file
286:        fclose(md3file);
287:
288:        // note: the last frame for the quake3 model is
289:        // header.numBoneFrames - 1
290:        modelHeader.iBoneFrameNum -= 1;
291:
292:        // set the start, end frame
293:        iStartFrame = 0;
294:        iEndFrame = modelHeader.iBoneFrameNum;
295:
296:        return 1;
297:    }
298:
299:
```

第 1～14 行代码检查了指定的.md3 文件是否存在。第 16～28 行代码读取并检查了 MD3HEADER 结构。标识码总为“IDP3”，最后已知的版本号为 15。如果文件的标识码或版本号不正确，则函数将返回一个表示“bad header”的返回值。模型的首部指定了骨骼帧的数量（iBoneFrameNum），第 30～34 行代码分配了存放这些骨骼帧的内存。md3.h 文件中的 MD3BONEFRAME 结构已在先前“Md3.h”一节中介绍过了。每个网格都必须具有相同数量的骨骼帧，骨骼帧的数量在模型首部中给出。

第 36～43 行代码分配了存放所有标记的内存，标记在游戏中，用于连接各个单独的.md3 对象，然后从文件中读入所有标记。第 45～66 行代码将这些标记转换为一种自定义的 D3DTAG 结构，该结构以另一种方式来储存原始 MD3TAG 格式中的 3×3 旋转矩阵和位置值。

第 68～73 行代码分配了用于“链接（link）”的内存。这些链接将用于构建一个包含所有.md3 对象的树，该树用于搜索每个模型所链接的下一个模型。例如，可以将武器链接到手上（在本例中，武器就是长矛）。DrawSkeleton()将遍历这个树，递归地绘制出每个.md3 对象，当一个模型没有再链接其他模型时，即完成了这一系列模型的绘制。

接下来这部分代码借助于 MD3MESHHEADER 结构读取网格数据。第 75～152 行代码读取了 MD3MESH 结构的所有成员。该结构如下：

```
typedef struct
{
    MD3MESHHEADER           meshHeader;
    MD3SKIN                 *meshSkins;
    MD3TRIANGLE             *meshTriangles;
    MD3TEXCOORD             *meshTexCoord;
    MD3VERTEXEX             *meshVertices;
    TANGENTS                *meshTangents;
    int                     iNumTextures;
```

```
    LPDIRECT3DTEXTURE9          pTexturesInterfaces[MAXTEXTURESPERMESH];
    LPDIRECT3DVERTEXSHADER9     pVertexShader[MAXSHADERLEVELINMESH];
    LPDIRECT3DPIXELSHADER9      pPixelShader[MAXSHADERLEVELINMESH];
    int                         iChoosedShaderLevel[MAXSHADERPROFILE];
} MD3MESH;
```

这个结构储存了首部数据、三角形、纹理坐标、顶点数据、切线、纹理接口、顶点 shader 和像素 shader 数据。另外，它还储存了你为该网格所选择的 shader 级别。其中有两点值得注意。第一，原始顶点数据储存在 3 个 16 位的值中。示例程序将其转换为 32 位的浮点值，并通过将其除以 64 来作了一个缩放。这样便于对数据进行处理，并且这样做对一个示例程序而言也是极具启发意义的。不过，一个更优雅的解决方法应该是将这些 16 位的数值传递到顶点 shader 中去，并在顶点 shader 中对其进行缩放。第二，示例程序并不使用网格所提供的法线。由于示例程序需要通过法线和切线来构建一个纹理空间坐标系，从而将光向量和其他向量变换到纹理空间中去。但是网格所提供的法线是用于逐顶点光照计算的，无法完成这项操作，所以就需要使用 CalculateNormals2()函数来完整地重新计算所有法线。

第 153～261 行代码计算并保存了法线和切线。在保存之前，还借助于 QuantiseVector()函数将法线和切线压缩为 32 位的整型值。这是在 CalculateNormals2()和 GenerateTangeut()函数中完成的。

源代码片断中剩余的代码则用于常规的初始化。

2. LoadShaders()

LoadShaders()函数加载扩展名为*.sha 的文件。这些文件保存着对 shader 文件的引用和 shader 的级别。lower_default.sha 文件的内容如下：

```
lower_mesh01,shaders/ColorNOneTexture.vsh,shaderlevel0
lower_mesh02,shaders/ColorNOneTexture.vsh,shaderlevel0
l_legs01,shaders/ColorNOneTexture.vsh,shaderlevel0
lower_mesh01,shaders/OnlyOneTexture.psh,shaderlevel0
lower_mesh02,shaders/OnlyOneTexture.psh,shaderlevel0
l_legs01,shaders/OnlyOneTexture.psh,shaderlevel0
lower_mesh01,shaders/diffuse.vsh,shaderlevel1
lower_mesh02,shaders/diffuse.vsh,shaderlevel1
l_legs01,shaders/ColorNOneTexture.vsh,shaderlevel1
lower_mesh01,shaders/diffuse.psh,shaderlevel1
lower_mesh02,shaders/diffuse.psh,shaderlevel1
l_legs01,shaders/OnlyOneTexture.psh,shaderlevel1
lower_mesh01,shaders/diffspec.vsh,shaderlevel2
lower_mesh02,shaders/diffspec.vsh,shaderlevel2
l_legs01,shaders/ColorNOneTexture.vsh,shaderlevel2
lower_mesh01,shaders/diffspec.psh,shaderlevel2
lower_mesh02,shaders/diffspec.psh,shaderlevel2
l_legs01,shaders/OnlyOneTexture.psh,shaderlevel2
```

第一列（由逗号分开）表示网格的名称。lower.md3 由三个网格组成：lower_mesh01、lower_mesh02、l_legs01。整个文件应从右向左读。对于 0 级别的 shader，这三个网格将使用

文件 ColorNOneTexture.vsh 中的顶点 shader 和文件 OnlyOneTexture.psh 中的像素 shader。在 1 级别的 shader 中，这三个网格将使用 diffuse.vsh 中的顶点 shader 和 diffuse.psh 中的像素 shader，依此类推。这样，在各个 shader 级别中，每个网格都可以找到对应的顶点 shader 和像素 shader。

另外，dragon_shader.profiles 文件存储着另一种抽象层。

```
shaderprofile0
head01,shaderlevel1
lower_mesh01,shaderlevel0
lower_mesh02,shaderlevel0
l_legs01,shaderlevel0
upper_mesh01,shaderlevel1

shaderprofile1
head01,shaderlevel1
lower_mesh01,shaderlevel0
lower_mesh02,shaderlevel0
l_legs01,shaderlevel1
upper_mesh01,shaderlevel1

shaderprofile2
head01,shaderlevel1
lower_mesh01,shaderlevel0
lower_mesh02,shaderlevel1
l_legs01,shaderlevel1
upper_mesh01,shaderlevel1

shaderprofile3
head01,shaderlevel1
lower_mesh01,shaderlevel1
lower_mesh02,shaderlevel1
l_legs01,shaderlevel1
upper_mesh01,shaderlevel1

shaderprofile4
head01,shaderlevel1
lower_mesh01,shaderlevel2
lower_mesh02,shaderlevel2
l_legs01,shaderlevel1
upper_mesh01,shaderlevel1

shaderprofile5
head01,shaderlevel0
lower_mesh01,shaderlevel1
lower_mesh02,shaderlevel0
l_legs01,shaderlevel1
upper_mesh01,shaderlevel0

shaderprofile6
head01,shaderlevel0
lower_mesh01,shaderlevel0
lower_mesh02,shaderlevel0
l_legs01,shaderlevel0
```

```
upper mesh01,shaderlevel0
```

这个文件用于为某个网格在*.sha 文件中选择一个 shader 级别。例如，在 shaderprofile5 中，网格 lower_mesh01 会使用 1 级别的 shader 所联系的顶点/像素 shader，而网格 lower_mesh02 则会使用 0 级别的 shader 所联系的顶点/像素 shader。在*.shader 文件中指定一个特定的 shaderprofile 便可以将一个网格与某个 shader 级别联系起来。在*.sha 文件中，shader 级别与某个特定的顶点/像素 shader 相联系。图 12.10 展示了这种关系。

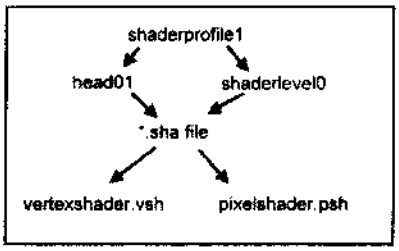


图 12.10　shader 级别（Level-of-shadeing）系统

应用程序可以循环遍历这个 shader 配置文件，根据 shader 级别的不同，为每个网格选择一种顶点 shader 和像素 shader 的组合。一般可以根据模型与照相机的距离来选择不同的 shader 级别，如此便可以实现出一套 shader 级别系统。

LoadShaders()函数用于从*.sha 文件中加载顶点 shader 和像素 shader。Q3Player 类的成员函数 LoadPlayerShaderProfile()用于加载 shader 配置文件。

LoadShaders()要求传入*.sha 文件的文件名及其所在目录的路径。以下源代码加载了某个 shader 级别中的顶点 shader 和像素 shader。

```
1:    void CMD3Model::LoadShaders(LPDIRECT3DDEVICE9 m_pd3dDevice,
2:                                char *pcFileName, char *pcShaderPath)
3:    {
4:        LOGFUNC("LoadShaders()");
5:
6:        FILE *ShaderFile;
7:        CHAR cMeshName[16];
8:        CHAR cShaderName[256];
9:        CHAR strToken[1024], cPath[1024];
10:       CHAR pcTextLine[1024];
11:       BOOL bMesh, bTag, bShader, bShaderLevel;
12:       int iShaderLevel;
13:
14:       // open *.sha file
15:       if ( CheckFile(pcFileName) == 0)
16:           return;
17:
18:       ShaderFile=fopen(pcFileName, "rt");
19:
20:       while ( !feof(ShaderFile) )
21:       {
```

```
22:             bMesh = FALSE;
23:             bTag = FALSE;
24:             bShader = FALSE;
25:             bShaderLevel = FALSE;
26:
27:             // scans a whole text line
28:             fscanf(ShaderFile, "%s", pcTextLine);
29:
30:             while (pcTextLine && pcTextLine[0])
31:             {
32:                 // returns in strToken a word of a textline
33:                 // returns in pcTextLine the next word of this textline
34:                 // Reads only words consisting of chars, numbers,
35:                 // points and underscores
36:                 ParseTextLine(pcTextLine, strToken);
37:
38:                 // get mesh name
39:                 if (bMesh == 0 && bShaderLevel == 0)
40:                 {
41:                     _tcscpy(cMeshName, _strlwr(strToken));
42:                     bMesh = 1;
43:                 }
44:                 else if ( ( strstr(_strlwr(strToken), ".vsh") ||
45:                             strstr(strToken, ".psh")) &&
46:                             bMesh == 1 && bShader == 0)
47:                 {
48:                     _tcscpy(cShaderName, strToken);
49:                     bShader = 1;
50:                 }
51:
52:                 // get shaderlevel
53:                 else if ((strstr(strToken, "shaderlevel")) &&
54:                     bShaderLevel == 0 && bMesh == 1 && bShader == 1)
55:                 {
56:                     iShaderLevel = ParseNumber(strToken);
57:                     bShaderLevel = 1;
58:                 }
59:             }
60:
61:             // create vertex and pixel shaders
62:             if (bShader)
63:             {
64:                 wsprintf(cPath, "%s\\%s", "shaders", cShaderName);
65:
66:                 // Quest for the right mesh
67:                 int i = 0;
68:                 while (!strstr(md3Meshes[i++].meshHeader.cName, cMeshName))
69:                 {
70:                     if (modelHeader.iMeshNum < i)
71:                     {
72:                         LOG("Didn't find a Mesh with the Name: " +
73:                             string(cMeshName), Logger::LOG_ERR);
74:                         fclose(ShaderFile);
75:                         return;
76:                     }
77:                 }
```

```
78:                 i--;
79:
80:                 if (strstr(cShaderName, ".vsh"))
81:                 {
82:                     TCHAR strVertexShaderPath[512];
83:                     LPD3DXBUFFER pCode;
84:                     HRESULT hr;
85:
86:                     DWORD dwFlags = 0;
87:
88:     #if defined( _DEBUG ) || defined( DEBUG )
89:                     dwFlags |= D3DXSHADER_DEBUG;
90:     #endif
91:
92:                     hr = DXUtil_FindMediaFileCb(strVertexShaderPath,
93:                             sizeof(strVertexShaderPath), cShaderName);
94:                     hr = D3DXAssembleShaderFromFile(strVertexShaderPath,
95:                             NULL, NULL, dwFlags, &pCode, NULL)
96:                     hr = m_pd3dDevice->CreateVertexShader(
97:                             (DWORD*)pCode->GetBufferPointer(),
98:                             &md3Meshes[i].pVertexShader[iShaderLevel]);
99:
100:                    if(!(FAILED(hr)))
101:                    {
102:                        CHAR strData[128];
103:                        wsprintf(strData,
104:                "Created Vertex Shader: %s for Mesh: %s with handle #%d",
105:                                cShaderName,
106:                                cMeshName,
107:                                md3Meshes[i].pVertexShader[iShaderLevel]);
108:                        LOG(string(strData), Logger::LOG_DATA);
109:                    }
110:                    else
111:                    {
112:                        LOG ("Could not load Vertex Shader " +
113:                             string(cShaderName) + "for Mesh " +
114:                             string(cMeshName), Logger::LOG_ERR);
115:                        md3Meshes[i].pVertexShader[iShaderLevel] = 0;
116:                    }
117:                }
118:                if (strstr(cShaderName, ".psh"))
119:                {
120:                    TCHAR strPixelShaderPath[512];
121:                    LPD3DXBUFFER pCode;
122:                    HRESULT hr;
123:
124:                    hr = DXUtil_FindMediaFileCb(strPixelShaderPath,
125:                             sizeof(strPixelShaderPath), cShaderName);
126:                    hr = D3DXAssembleShaderFromFile(strPixelShaderPath,
127:                             NULL, NULL,NULL, &pCode,NULL);
128:                    hr = m_pd3dDevice->CreatePixelShader(
129:                             (DWORD*)pCode->GetBufferPointer(),
130:                             &md3Meshes[i].pPixelShader[iShaderLevel]);
131:
132:                    if(!(FAILED(hr)))
133:                    {
```

```
134:                        CHAR strData[128];
135:                        wsprintf(strData,
136:                    "Created Pixel Shader: %s for Mesh: %s with handle #%d",
137:                                cShaderName,
138:                                cMeshName,
139:                                md3Meshes[i].pPixelShader[iShaderLevel]);
140:                        LOG(string(strData), Logger::LOG_DATA);
141:                    }
142:                    else
143:                    {
144:                        LOG ("Could not load Pixel Shader " +
145:                            string(cShaderName) +" for Mesh " +
146:                            string(cMeshName), Logger::LOG_ERR);
147:                        md3Meshes[i].pPixelShader[iShaderLevel] = 0;
148:                    }
149:                }
150:            }
151:        }
152:        fclose(ShaderFile);
153:    }
```

第 15～20 行代码加载了 shader 文件。在第 28 行代码中，文本分析算法通过 fscanf()从 shader 文件中读入一行内容。utility.cpp 文件中的 ParseTextLine()函数逐词地输出这行文本，它只读取由字符、数字、点和下划线组成的词。对于字符串“lower_mesh01, shaders/OnlyColor.vsh, shaderlevel0”，分析结果如下：

	cTokenString 中的内容：	pcTextLine 中的内容：
第 1 次调用：	lower_mesh01	,shaders/OnlyColor.vsh,shaderlevel0
第 2 次调用：	,	shaders/OnlyColor.vsh,shaderlevel0
第 3 次调用：	shaders	/OnlyColor.vsh,shaderlevel0
第 4 次调用：	/	OnlyColor.vsh,shaderlevel0
第 5 次调用：	OnlyColor.vsh	,shaderlevel0
第 6 次调用：	,	shaderlevel0
第 7 次调用：	shaderlevel0	空

该函数用于分析整个示例程序中的每个 ASCII 文本文件。第 39～58 行代码分析了 ParseTextLine()返回的词。在这个示例中，从文本行中提取出 3 个词：网格名称、shader 名称和 shader 级别。网格名称总是这么几个相同的词，因此很容易分辨出来。shader 名称以扩展名*.vsh 或*.psh 作为标识符。接下来 shader 级别通过搜索读入词中的字符串“shaderlevel”，然后读取其后的内容即可获得 shader 级别。这个分析算法可用于分析所有的*.sha、*.skin 和 *.profile 文件。

在创建顶点 shader 和像素 shader 之前，第 68～78 行代码先在.md3 对象中搜索一个网格。顶点/像素 shader 的句柄和 shader 级别都储存在网格结构 MD3MESH 中。这个示例使用的 shader 是用汇编写的，而非 HLSL。虽然 HLSL 编译器可以生成汇编代码，但直接在汇编层面上编写代码可以最优化 shader 的性能，而 HLSL 编译器可能无法产出最优化的汇编代码。

为了使用一个不带效果（effect）文件格式的顶点 shader 或像素 shader（这也是我在先前所有示例中使用的 shader），需要完成以下几个步骤：

- 使用 DXUtil_FindMediaFileCb()搜索 shader；
- 使用 D3DXAssembleShaderFromFile()对文件进行汇编；
- 使用 CreateVertexShader()或 CreatePixelShader()创建顶点/像素 shader；
- 使用 SetVertexDeclaration()设置顶点 shader 的声明；
- 使用 SetVertexShaderConstant*()或 SetPixelShaderConstant*()设置顶点/像素 shader 的常量；
- 使用 SetVertexShader()或 SetPixelShader()设置顶点/像素 shader。

另外，请留意一下 LOG()函数，它用于将日志信息写入文件 app.log。整个日志类位于 logger.h 和 logger.cpp 中。它的作者 Paul Nettle 已经将这个类向公众发布了。

CMD3Model 类中的 LoadSkins()函数和 Q3Player 类中的一些 Load*()函数差不多，都是以相同的模式执行的。

3. DrawSkeleton()

DrawSkeleton()使用 DrawIndexedPrimitive()将一个模型的所有网格数据绘制到显示缓冲器中。它通过每个“对下一个.md3 文件”的链接来递归地调用。递归过程会在遍历到 head.md3 时结束，因为这个模型没有再链接其他模型。

该函数向顶点 shader 提供了内插值、worldviewproj 矩阵、世界矩阵、混合的位置值、两个压缩的法线、两个压缩的切线。

```
1:      void CMD3Model::DrawSkeleton(CMD3Model *md3Model,
2:          LPDIRECT3DDEVICE9 m_pDevice, LPDIRECT3DINDEXBUFFER9 m_pIB,
3:          LPDIRECT3DVERTEXBUFFER9 m_pVB, D3DXMATRIX *matViewProj,
4:          D3DXMATRIX *matWorld,
5:          LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclaration)
6:      {
7:        //------------------------
8:        // draw one of the up to four md3 models
9:        // 1. lower.md3
10:       // 2. upper.md3
11:       // 3. optional: railgun.md3
12:       // 4. head.md3
13:       //------------------------
14:       D3DXMATRIX matClip, matTemp;
15:       int i, j; // counter
16:       int iCurrMesh, iCurrOffsetVertex, iNextCurrOffsetVertex;
17:
18:       // interpolation factor
19:       FLOAT fPol = md3Model->iFps *
20:         (md3Model->fNewTime - md3Model->fOldTime);
21:
22:       // interpolation constant
23:       D3DXVECTOR4 intpol(fPol, 1.0f , 0.5f, 1.0f);
24:       m_pDevice->SetVertexShaderConstantF(37, (float*)&intpol, 1);
```

```
25:
26:          for (iCurrMesh = 0; iCurrMesh < md3Model->modelHeader.iMeshNum;
27:            iCurrMesh++)
28:          {
29:           MD3VERTEXBUFFERSTRUCT* pVertexBuffer;
30:           m_pVB->Lock(0, 0, (VOID**)&pVertexBuffer, D3DLOCK_DISCARD);
31:
32:           WORD *pIndices; // fill index buffer
33:           m_pIB->Lock(0, 0, (VOID **)&pIndices, D3DLOCK_DISCARD);
34:           DWORD dwIndexBufferCounter = 0;
35:
36:           iCurrOffsetVertex = md3Model->iFrame *
37:               md3Model->md3Meshes[iCurrMesh].meshHeader.iVertexNum;
38:           iNextCurrOffsetVertex = md3Model->iNextFrame *
39:               md3Model->md3Meshes[iCurrMesh].meshHeader.iVertexNum;
40:
41:           // fill index buffer
42:           for (i = 0; i < md3Model->md3Meshes[iCurrMesh].meshHeader.
43:                iTriangleNum; i++)
44:               for (j = 0; j < 3; j++)
45:                   pIndices[dwIndexBufferCounter++] = (WORD) md3Model->
46:                       md3Meshes[iCurrMesh].meshTriangles[i].index[j];
47:
48:           for(i = 0; i < md3Model->md3Meshes[iCurrMesh].meshHeader.
49:               iVertexNum; i++)
50:           {
51:               // interpolated vertices -----------------------
52:               // pseudo code: CurrVertex.vPosition + fPol *
53:               //            (NextVertex.vPosition - CurrVertex.vPosition);
54:               pVertexBuffer[i].vPosition =
55:                   md3Model->md3Meshes[iCurrMesh].
56:                       meshVertices[iCurrOffsetVertex + i].vVector + fPol *
57:                       (md3Model->md3Meshes[iCurrMesh].meshVertices
58:                       [iNextCurrOffsetVertex + i].vVector-md3Model->
59:                       md3Meshes[iCurrMesh].meshVertices
60:                       [iCurrOffsetVertex + i].vVector);
61:
62:               pVertexBuffer[i].uiCompressedVertexNormal =
63:                   md3Model->md3Meshes[iCurrMesh].meshVertices
64:                   [iCurrOffsetVertex + i].uiCompressedVertexNormal;
65:               pVertexBuffer[i].uiCompressedVertexNormal2 =
66:                   md3Model->md3Meshes[iCurrMesh].meshVertices
67:                   [iNextCurrOffsetVertex + i].uiCompressedVertexNormal;
68:
69:               // texture coordinates -----------------------
70:               pVertexBuffer[i].vTexCoord = md3Model->md3Meshes[iCurrMesh].
71:                                                   meshTexCoord[i].texvec;
72:
73:               // tangent vectors, two for interpolation
74:               pVertexBuffer[i].vCompressedTangent =
75:                   md3Model->md3Meshes[iCurrMesh].meshTangents
76:                       [iCurrOffsetVertex + i].vCompressedTangent;
77:               pVertexBuffer[i].vCompressedTangent2 =
78:                   md3Model->md3Meshes[iCurrMesh].meshTangents
79:                       [iNextCurrOffsetVertex + i].vCompressedTangent;
80:           }
```

```
81:
82:             m_pVB->Unlock();
83:             m_pIB->Unlock();
84:
85:             // world * view * proj matrix
86:             D3DXMATRIX matTemp;
87:             D3DXMatrixMultiply(&matClip, matWorld, matViewProj);
88:             D3DXMatrixTranspose(&matTemp,&matClip);
89:             m_pDevice->SetVertexShaderConstantF(8, (float*)&matTemp, 4);
90:
91:             // world matrix
92:             D3DXMatrixTranspose(&matTemp, matWorld);
93:             m_pDevice->SetVertexShaderConstantF(0, (float*)&matTemp, 4);
94:
95:             m_pDevice->SetVertexDeclaration( m_pVertexDeclaration );
96:
97:             // set vertex shader
98:             m_pDevice->SetVertexShader(md3Model->md3Meshes[iCurrMesh].
99:                 pVertexShader[md3Model->md3Meshes[iCurrMesh].
100:                    iChoosedShaderLevel[iShaderProfile]]);
101:
102:            // texture caching
103:            if(md3Model->md3Meshes[iCurrMesh].iNumTextures > 0)
104:            {
105:                for (i = 0; i < md3Model->md3Meshes[iCurrMesh].iNumTextures;
106:                     i++)
107:                {
108:                    if( pMiniTextureCache[i] != md3Model->md3Meshes
109:                        [iCurrMesh].pTexturesInterfaces[i])
110:                    {
111:                        pMiniTextureCache[i] = md3Model->md3Meshes
112:                            [iCurrMesh].pTexturesInterfaces[i];
113:                        m_pDevice->SetTexture(i, pMiniTextureCache[i] );
114:                    }
115:                }
116:            }
117:
118:            // set pixel shader
119:            m_pDevice->SetPixelShader(md3Model->md3Meshes[iCurrMesh].
120:                pPixelShader[md3Model->md3Meshes[iCurrMesh].
121:                    iChoosedShaderLevel[iShaderProfile]]);
122:
123:
124:
125:            // set the vertex buffer
126:            // == specify the source of stream 0
127:            m_pDevice->SetStreamSource( 0, m_pVB, 0,
128:                                        sizeof(MD3VERTEXBUFFERSTRUCT));
129:
130:            // set the index buffer
131:            m_pDevice->SetIndices(m_pIB);
132:
133:            // ... rendering
134:            m_pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
135:                0,
136:                0,
```

```
137:                // number of vertices
138:                md3Model->md3Meshes[iCurrMesh].meshHeader.iVertexNum,
139:                0,
140:                // number of primitives
141:                md3Model->md3Meshes[iCurrMesh].meshHeader.iTriangleNum);
142:          }
143:
144:          // interpolate position of tags of lower.md3 (md3lower),
145:          // upper.md3 (md3upper) and optional railgun.md3 (md3weapon)
146:          // Not: head.md3
147:          CMD3Model *modelLink;
148:
149:          D3DXQUATERNION quatFromMatrix, quatFromMatrix2, quatResult;
150:          D3DXMATRIX matFrame, matNextFrame;
151:
152:          int iModelFrame = md3Model->iFrame * md3Model->modelHeader.iTagNum;
153:          int iModelNextFrame = md3Model->iNextFrame *
154:                                md3Model->modelHeader.iTagNum;
155:
156:          for (i=0; i < md3Model->modelHeader.iTagNum; i++)
157:          {
158:           // pointer to a model class
159:           modelLink = md3Model->md3Links[i];
160:
161:           if (modelLink)
162:           {
163:                // rotation matrix
164:                matFrame = &md3Model->d3dTag[ iModelFrame + i].matTag.m[0][0];
165:                matNextFrame = &md3Model->d3dTag[ iModelNextFrame + i].
166:                                    matTag.m[0][0];
167:
168:                // quaternion slerp interpolation
169:                D3DXQuaternionRotationMatrix(&quatFromMatrix, &matFrame);
170:                D3DXQuaternionRotationMatrix(&quatFromMatrix2, &matNextFrame);
171:                D3DXQuaternionSlerp(&quatResult, &quatFromMatrix,
172:                                          &quatFromMatrix2, fPol);
173:                D3DXMatrixRotationQuaternion(&matTemp, &quatResult);
174:
175:                // interpolated position vector
176:                // Pseudo code: Position[0] + fPol * (nextPosition[0] - Position[0]);
177:                matTemp[12] = md3Model->d3dTag[ iModelFrame + i].matTag.
178:                     m[3][0] + fPol * (md3Model->d3dTag[ iModelNextFrame + i].
179:                          matTag.m[3][0] - md3Model->d3dTag[ iModelFrame + i].
180:                          matTag.m[3][0]);
181:
182:                matTemp[13] = md3Model->d3dTag[ iModelFrame + i].matTag.
183:                     m[3][1] + fPol * (md3Model->d3dTag[ iModelNextFrame + i].
184:                          matTag.m[3][1] - md3Model->d3dTag[ iModelFrame + i].
185:                          matTag.m[3][1]);
186:
187:                matTemp[14] = md3Model->d3dTag[ iModelFrame + i].matTag.
188:                     m[3][2] + fPol * (md3Model->d3dTag[ iModelNextFrame + i].
189:                          matTag.m[3][2] - md3Model->d3dTag[ iModelFrame + i].
190:                          matTag.m[3][2]);
191:                matTemp[15] = 1.0f; matTemp[3] = matTemp[7] = matTemp[11] = 0;
192:
```

```
193:                D3DXMatrixMultiply(&matTemp ,&matTemp, matWorld);
194:                DrawSkeleton(modelLink, m_pDevice, m_pIB, m_pVB, matViewProj,
195:                    &matTemp, m_pVertexDeclaration);
196:            }
197:        }
198:    }
```

第 103～116 行代码将纹理对象的引用缓存在 pMiniTextureCache 数组中，这样可以减少 SetTexture()的调用次数。

第 163～190 行代码通过一个四元数对模型的位置做球形内插计算（Spherical Linear Interpolation，球形线性插值），其中对链接模型不同部分的标记位置做了插值计算。

第 194～195 行代码递归地调用了 DrawSkeleton()函数。

请注意，第 51～60 行代码中对位置向量所做的插值计算是通过 CPU 完成的，而压缩过的法线和切线则是在顶点 shader 中作解压缩和插值计算的（也就是通过 GPU 完成插值计算的）。若能以类似于法线和切线的插值计算方法来处理位置向量的话，则是对本例的一项很好的改进。*Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*（Wordware, 2002）中有一篇 Dean Calver 撰写的文章"Vertex Decompression in a Shader"，其中描述了一种借助于包裹盒来压缩位置值的技术，也值得参考一下。

4. HLSL Shader

示例程序使用的 ASCII 文件格式的汇编 shader 来源于 HLSL 编译器，它们是通过 HLSL 编译器 fxc.exe 编译 HLSL shader 后得到的。通过获取 HLSL 编译器的汇编输出，程序员便拥有了一次优化 shader 的机会；并且更重要的是，程序员可以借此控制编译器的输出。

可以在 VC 的"自定义生成"中调用 HLSL 编译器，命令如下：

```
fxc.exe /T vs_1_1 /E VS /Fc diffuse.vsh $(InputName).fx
```

在这行命令中，编译器选项"/Fc diffuse.vsh"用于指示编译器通过 HLSL 文件"$(InputName).fx"来创建汇编文件 diffuse.vsh。其中，宏"$(InputName).fx"会被替换成输入文件的名称。"/E"选项指示编译器 HLSL shader 的入口函数名为"VS"，"/T"选项表示要求使用 vs_1_1 版本的顶点 shader 配置文件。所有的 HLSL shader 都是通过类似于这样一行命令来编译的。

本章示例 CharacterEngine 中的 diffuseSpecular.fx 和 diffuseSpecular.fxp 取自于第 9 章中的代码。diffuse.fx 和 diffuse.fxp 中的 HLSL shader 就是这些 shader 的低版本。我删除了镜面光照的代码。其他所有的 HLSL shaders 都是第 9 章中所示 shader 的较低版本。

仔细地观察 HLSL 编译器生成的汇编 shader 可以作为学习汇编 shader 编程的起点。我在此推荐我编写的汇编 shader 指南，它位于 http://www.gamedev.net。不过我想指出的是，由于 HLSL 编译器可以生成极好的汇编代码，并且 HLSL 拥有更易于阅读的类 C 语法，所以学习汇编 shader 编程不再是必须的了。

12.3 更多的改进

对于这个模型浏览器，你还可以做出许多改进。其中较明显的一处改进便是减少应用程序的存储开销。以下列出了一些其他可做的改进：

- 在顶点 shader 中解压缩并处理原始的.md3 位置数据，或者使用一种更好的压缩方式来处理它；
- 对法线和切线实施更高程度的压缩；
- 在头盔和装甲上使用 Cook-Torrance 光照计算[1]；
- 在武器上使用一种能量等离子（energy plasma）shader；
- 对整个模型使用卡通 shader，从而赋予模型一种卡通的外观和感觉。

12.4 附加资源

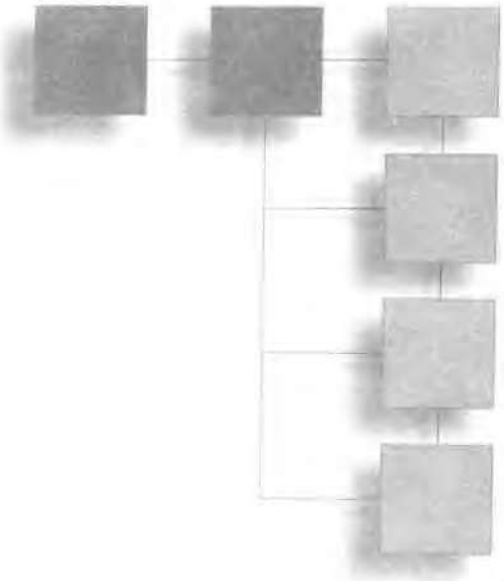
一个最佳的.md3 文件浏览器，位于 http://fragland.net/md3view/index.html。

一份最佳的.md3 文件格式的描述，位于 http://linux.ucla.edu/~phaethon/q3a/formats/md3format.html。

12.5 小结

在本章中，你学习到了许多关于 *Quake 3* 的.md3 文件格式的知识。现在已临近了本书的尾声，你应该已经拥有了一个不错的 Direct3D 编程基础。但请先等等，接下去还有一些内容！查阅一下附录，里面还有一些其他有用的附加信息，包括一份“C++入门”和一份“数学入门”。另外，也别错过了附录 E“游戏程序设计资源”，其中有一份网站列表，通过查阅它们，你可以学习到更多与本书主题相关的知识。最后，祝编程愉快！

译注 1：Cook-Torrance 光照模型将表面看作由许多微小凹凸的平面组成，每个微平面作为一个理想的反射镜，如此便可以产生真实的镜面反射。



第 4 部分

附　　录

- 附录 A　Windows 游戏编程基础
- 附录 B　C++入门
- 附录 C　数学入门
- 附录 D　使用 D3DXCreateTextureFromFileEx()创建纹理
- 附录 E　游戏程序设计资源
- 附录 F　随书光盘内容

附 录 A

Windows 游戏编程基础

游戏程序员需要知道 Windows 编程中哪些相关的内容？嗯……，坦率地说，不是很多。我认为一个游戏程序员主要需要知道 Windows 编程中 3 个方面的知识：

- 如何创建窗口；
- 如何使用窗口消息过程；
- 如何借助 Visual C/C++来使用资源（图标、菜单、快捷方式等）。

市面上有很多关于 Windows 编程的文章和书籍，请看一下附录 E“游戏程序设计资源”来获取这些内容。就本章而言，我们先来研究一下其中的一些主题。

A.1 如何看待窗口

你应该会对编写一个既可在全屏模式下运行又可在窗口模式下运行的游戏比较感兴趣。我将在此讨论这两种模式的编程，不过在开始为第一个用于呈现游戏画面输出的窗口编写代码之前，你需要先学习一些普通的 Windows 概念。

A.2 Windows 如何与你的游戏交互

一个 DOS 游戏在请求一些东西，如资源、输入和输出时，不必共享地使用这些资源、也不必在将其“据为己有”前询问其他程序。然而，一个基于 Windows 的游戏必须在 Windows 给它发送消息前处于等待状态。这些消息是由 Windows 通过调用一个特殊的函数传递到你的程序中的。每当收到一条消息后，你的程序需要能执行一个适当的响应操作。消息的到来是随机的，因此每个 Windows 游戏都必须不断地检查是否有新的消息。

Windows 不仅引入了消息机制，它还对资源（如显卡、硬盘等）的使用进行授权。当编写窗口模式或全屏模式的应用程序时，你必须考虑到这一点。有几种方法可以从 Windows 处获取对硬件的完全控制。对于全屏的游戏而言，它们尤其需要对硬件的完全控制。而在窗口模式下则需考虑共享资源的问题。

在进一步学习具体的 Windows 编程方式前，我需要定义一些重要的概念。

A.3 窗口的组成部分

每个窗口都一个边框，它定义了窗口的边界，并被用于调整窗口的大小。另外还有一个标题栏，它包括位于最左边的系统菜单和位于最右边的最小化/最大化/关闭按钮。图 A.1 展示了来自 ATI 的示例程序 FaceMorph。

图 A.1 一个窗口的组成部分

你的游戏应显示在客户区中。

A.4 窗口框架

现在我将逐步向你介绍一个 Windows 程序的开发过程，该程序将具备所有的 Windows 应用程序都会必备的特征，它将包含以下两个函数：

- WinMain()；
- 一个窗口过程，如 WndProc()。

WinMain()函数完成了以下这些常规的步骤：

（1）定义一个窗口类；

（2）向 Windows 注册该窗口类；

（3）创建一个该类的窗口；

（4）显示这个窗口；

（5）开始运行消息循环。

WinMain()首先通过定义一个窗口类、注册该类并创建一个窗口来初始化应用程序。然后显示该窗口，并进入一个消息获取和派发的循环。最后，当收到一个 WM_QUIT 消息时，程序就将终止消息循环，并将 WM_QUIT 的 wParam 参数值作为返回值，退出游戏。

图 A.2 展示了第一个（稍显单薄的）示例程序，名为 winskel.exe：

图 A.2　窗口框架

以下是源代码：

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
char szWinName[] = "Crusher"; /* name of window class */

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
    HWND hwnd;
    MSG msg;

    /* Step 1: Define a window class. */
    WNDCLASS wcl;
    wcl.hInstance = hThisInst;      /* handle to this instance */
```

```
    wcl.lpszClassName = szWinName; /* window class name */
    wcl.lpfnWndProc = WndProc;     /* window function */
    wcl.style = 0;                      /* default style */
    /* icon style */
    wcl.hIcon = LoadIcon(NULL, MAKEINTRESOURCE( IDI_APPLICATION));
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW); /* cursor style */
    wcl.lpszMenuName = NULL;   /* no menu */
    wcl.cbClsExtra = 0;            /* no extra */
    wcl.cbWndExtra = 0;            /* 0 information needed */
    /* Make the window background white. */
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH) ;

    /* Step 2: Register the window class. */
    if(!RegisterClass (&wcl))
        return 0;

    /* Step 3: Now that a window class has been registered,
    a window can be created. */
    hwnd = CreateWindow(szWinName,      /* name of window class */
                        "FAKE 4",       /* title */
                        WS_OVERLAPPEDWINDOW, /* window style - normal */
                        CW_USEDEFAULT,  /* X coordinate - let Windows decide */
                        CW_USEDEFAULT,  /* y coordinate - let Windows decide */
                        CW_USEDEFAULT,  /* width - let Windows decide */
                        CW_USEDEFAULT,  /* height - let Windows decide */
                        HWND_DESKTOP,   /* no parent window */
                        NULL,           /* no menu */
                        hThisInst,      /* handle of this instance of the program */
                        NULL            /* no additional arguments */
                        );

    /* Step 4: Display the window. */
    ShowWindow(hwnd, nWinMode);

    /* Step 5: Create the message loop. */
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);        /* allow use of keyboard */
        DispatchMessage(&msg);         /* return control to Windows */
    }
    return msg.wParam;
}

//-----------------------------------------------
// Name: WndProc
// Desc: This function is called by Windows and is passed
// messages from the message queue
//-----------------------------------------------
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_DESTROY: /* terminate the program */
        PostQuitMessage(0);
        break;
```

```
        default: /* Let Windows process any messages not
                    specified in the preceding switch statement. */
            return DefWindowProc(hwnd, message, wParam, lParam);
        }
        return 0;
    }
```

该程序使用的窗口过程名为 WndProc()。Windows 会调用它来与你的程序通信。为了表示这样的调用，需将它声明为一个回调函数（callback function）。程序由 WinMain()开始执行，该函数传入 4 个参数。

```
int WINAPI WinMain(
    HINSTANCE hInstance,      // handle to current instance
    HINSTANCE hPrevInstance,  // handle to previous instance
    LPSTR lpCmdLine,          // command line
    int nCmdShow              // show state
);
```

hInstance 表示游戏的当前实例。hPrevInstance 在新版本的 Windows 中已被废除不用了，因此就基于 Win32 的应用程序而言，此参数总为 NULL。

lpCmdLine 是一个指向字符串的指针，该字符串保存了应用程序启动时的所有命令行参数，但不包括程序本身的名称。如果要检索整个命令行，可使用 GetCommandLine()函数。nCmdShow 指定了窗口显示方式的标志（见表 A.1）。

表 A.1 列出了典型的显示标志。

表 A.1　　典型的显示标志

标志	描述
SW_HIDE	隐藏窗口，并激活另一个窗口
SW_MAXIMIZE	最大化指定的窗口
SW_MINIMIZE	最小化窗口
SW_RESTORE	从一个最大化或最小化状态，将窗口还原为它的原来大小和位置

附注

什么是句柄（handle）？为什么基于 Windows 的程序如此频繁地使用句柄？在 Windows 中，一个句柄是一个指向指针的指针[①]，因此它指向的是一个地址，该地址存储着一个表格或列表。你可以使用句柄所指向的地址来访问与该句柄相关联的对象。这种间接访问是必须的，因为内存管理程序经常会在内存中各处移动对象（比如为了紧缩内存[②]），有了这种间接性，它就不用再通知应用程序“某个对象的地址已被改变了”，并且同时保证对象的句柄依然有效。

译注①：所谓“二级指针”。

译注②：通常是为了减少内存碎片而进行的维护性操作。

> 应用程序使用 Windows 授予它们的句柄来使用各种不同类型的对象，这些对象被称为“Windows 对象”；这个称谓与 C++对象是没有任何关系的。你可以在基于 C 语言编写的程序中使用 Windows 对象，也可以在基于 C++语言的程序中使用它们。

像往常一样查阅一下由 Visual C/C++或 http://msdn.microsoft.com 提供的 Win32 文档以获取更多信息。现在让我们来做一遍这 5 个步骤，以最终显示出一个窗口。

A.4.1 步骤 1：定义一个窗口类

首先，Windows 程序必须先注册一个窗口类。注册一个窗口类就是告诉 Windows 所需窗口的外形和功能。在当前上下文中，“类”这个词不同于 C++中的含义，它意谓着风格（style）或类型（type）。

hInstance 字段被赋值为 hThisInst 所指定的当前实例句柄。lpszClassName 指向窗口类的名字，在本例中它指向一个字符串“Crusher”。lpfnWndProc 被赋值为窗口过程的地址。风格指定为 0，表示没有任何默认的风格。接着指定了应用程序的图标（icon），以及鼠标移入客户区时的箭头光标（cursor）。在这里不需要使用菜单，但需要在客户区中有一个白色的背景。

可以在 WNDCLASS 结构中使用不同的类风格。为做到这一点，可将风格值赋予 WNDCLASS 结构中的 style 成员。对于游戏程序员来说，大多数窗口风格都很少用到，但是值得关注的两个风格是：

- CS_NOCLOSE，禁用窗口菜单上的“关闭”按钮；
- CS_DROPSHADOW，在一个窗口上启用阴影（drop-shadow）效果，这个效果可通过 SPI_SETDROPSHADOW 进行开关，并为下一代中代号为“Whistler[1]”的 Windows 操作系统所支持。

至于其他风格也不见得会对你的使用有什么好处。

另外还必须为鼠标光标和应用程序的图标定义各自的默认外形。对于这些资源，你可以使用自定义的版本，也可以使用某种内建的风格，即如本框架所做的那样。LoadIcon()用于加载图标，它的原型如下所示：

```
HICON LoadIcon(HINSTANCE hInst, LPCSTR lpIconName);
```

第一个参数是一个模块（module）的实例句柄，表示需要加载的图标包含在哪个可执行文件中。若像本例那样加载一个标准图标的话，此参数可为 NULL。第二个参数是一个以'\0'结尾的字符串，包含了将要加载的图标资源名称。对此有两种做法：一，使用 MAKEINTRESOURCE 宏创建一个值，该值的低字（low-order word）包含资源标识符，高字（high-order word）为 0；二，使用某个预定义的图标，这需要将参数 hInst 设为 NULL，参数 lpIconName 设为表 A.2 所示值之一。

译注[1]：即现今的 Windows XP。

附注

LoadIcon()只能用于加载一个与系统度量（system metric）值 SM_CXICON 和 SM_CYICON 大小一致的图标。

表 A.2 lpIconName 参数值

值	描述
IDI_APPLICATION	默认的应用程序图标
IDI_ASTERISK	同 IDI_INFORMATION
IDI_ERROR	手状图标
IDI_EXCLAMATION	同 IDI_WARNING
IDI_HAND	同 IDI_ERROR
IDI_INFORMATION	星号图标
IDI_QUESTION	询问标志图标
IDI_WARNING	感叹号图标
IDI_WINLOGO	Windows 标志图标

你可能想制作你自己的图标，那会很有趣的，我们很快就会学习到它的制作方法。

如果 LoadIcon()成功，则返回值为一个新加载的图标句柄；如果失败，返回值为 NULL。

如果想加载一个鼠标光标，应使用 LoadCursor()。该函数的原型如下：

```
HCURSOR LoadCursor(HINSTANCE hInst, LPCSTR lpCursorName);
```

第一个参数通常是一个模块的实例句柄，表示要加载哪个可执行文件中的光标。第二个参数是一个以'\0'结尾的字符串，包含了将要加载的光标资源名称。对此也有两种做法：一，使用 MAKEINTRESOURCE 宏创建一个值，该值的低字包含资源标识符，高字为 0；二，使用某个预定义的光标，这需要将参数 hInst 设为 NULL，参数 lpCursorName 设为表 A.3 所示值之一。

表 A.3 预定义的光标[1]

值	描述
IDC_APPSTARTING	标准箭头和沙漏（后台运行）
IDC_ARROW	标准箭头（正常选择）

译注：为了帮助读者更形象地了解每个光标所对应的图案，每项描述后的括号内注明了该光标在“控制面板->鼠标->指针”中所使用的名称。

续表

值	描述
IDC_CROSS	十字准线（精确定位）
IDC_HAND	手状，Windows 98/Me、Windows 2000/XP 有效（链接选择）
IDC_HELP	箭头与问号（帮助选择）
IDC_IBEAM	I 光标（选定文本）
IDC_ICON	Windows NT 4.0 及其后续版本废除
IDC_NO	中间有斜杠的圆（不可用）
IDC_SIZE	Windows NT 4.0 及其后续版本废除，应使用 IDC_SIZEALL
IDC_SIZEALL	指向东、南、西、北的四向箭头（移动）
IDC_SIZENESW	指向东北、西南的双向箭头（沿对角线调整 2）
IDC_SIZENS	指向南、北的双向箭头（垂直调整）
IDC_SIZENWSE	指向西北、东南的双向箭头（沿对角线调整 1）
IDC_SIZEWE	指向东、西的双向箭头（水平调整）
IDC_UPARROW	垂直箭头（候选）
IDC_WAIT	沙漏（忙）

在本例中，使用 GetStockObject()来获取一个背景颜色画刷的句柄。一个画刷就是一种采用预定大小、颜色或图案来绘制屏幕的资源。该函数可用于获取一个备用的（stock）画笔、画刷、字体或调色板的句柄。

```
HGDIOBJ GetStockObject(int fnObject);
```

fnObject 用于指定一种备用对象的类型。游戏程序员不会经常用到这个函数。表 A.4 列出了一些可能会比较有用的标志。

表 A.4　　画刷

值	描述
BLACK_BRUSH	黑色画刷
DKGRAY_BRUSH	深灰色画刷
GRAY_BRUSH	灰色画刷
LTGRAY_BRUSH	淡灰色画刷
NULL_BRUSH	空画刷
WHITE_BRUSH	白色画刷

Windows 数据类型

框架程序没有广泛地使用如 int、char*那样的 C/C++数据类型，而是使用了 Windows 的

数据类型，这些数据类型都是定义在 Windows.h 或与其相关的头文件中的 typedef。你必须在所有的 Windows 程序中包含 Windows.h。

一些很常用的数据类型有 HANDLE（void 指针）、HWND、BYTE（8 位大小）、WORD（16 位的无符号短整数）、DWORD（32 位的无符号长整数）、UINT（32 位的无符号整数）、LONG（32 位的长整数）、BOOL（整数）、LPSTR（指向字符串的指针）和 LPCSTR（指向常量字符串的指针）。正如你所见，一些句柄类型都以一个 H 开头的，但它们都具有与 HANDLE 一样的大小。一个 HANDLE 只是一个标识某种资源的值而已。

Windows 还定义了几个结构，如 MSG 和 WNDCLASS。你马上将会看到 MSG 的作用就是保存一个 Windows 消息。

A.4.2 步骤 2：注册窗口类

RegisterClass()注册一个窗口类，该窗口类稍后将用于 CreateWindow()或 CreateWindowEx()的调用。一个应用程序注册的所有的窗口类会在程序终止时被注销掉。

```
ATOM RegisterClass(CONST WNDCLASS *lpWndClass);
```

函数返回一个非零值表示是否成功。ATOM 是一个 WORD 的 typedef。

A.4.3 步骤 3：创建一个该类的窗口

定义并注册好一个窗口类后，你的游戏就就可以使用 CreateWindow()来真正地创建对应于该窗口类的窗口了。

```
HWND CreateWindow(
    LPCTSTR lpClassName,   // registered class name
    LPCTSTR lpWindowName,  // title of window
    DWORD dwStyle,         // window style
    int x,                 // horizontal position of window
    int y,                 // vertical position of window
    int nWidth,            // window width
    int nHeight,           // window height
    HWND hWndParent,       // handle to parent or owner window
    HMENU hMenu,           // menu handle or child identifier
    HINSTANCE hInstance,   // handle to application instance
    LPVOID lpParam         // window-creation data
);
```

可以对 CreateWindow()中的大多数参数使用默认值或指定为 NULL。事实上，x、y、nWidth、nHeight 这些参数在大多数应用程序中都简单地指定为宏常量 CW_USEDEFAULT，这是告诉 Windows 自动为窗口选择一个合适的大小和位置。然而在一个游戏程序中，你可能希望将窗口设为预定的宽度和高度，那么就需要用到 AdjustWindowRect()，这个函数可以根据期望的客户矩形大小来辅助计算出对应的窗口矩形大小。

在这里必须将父窗口句柄 hWndParent 指定为 HWND_DESKTOP[1]。若窗口不包含菜单，那 hMenu 应为 NULL。若不需要附加信息（这也是最常见的情况），那么 lpParam 也为 NULL。

注意

如果一个应用程序调用 CreateWindow()来创建一个 MDI（Multiple Document Interface，多文档界面）窗口，lpParam 必须是一个指向 CLIENTCREATESTRUCT 结构的指针。

还有余下的 4 个参数需要显式地设置。lpClassName 必须是一个指向窗口类名的指针。lpWindowName 指向一个表示窗口标题的字符串。dwStyle 用于指定窗口的风格，WS_OVERLAPPEDWINDOW 宏指定了一个标准窗口，它具有一个系统菜单、一个边框、最小化按钮、最大化按钮和关闭按钮。作为一名游戏程序员，你可能需要自己来设置一些与标准窗口不太一样的风格。比如游戏只能按 Esc 键来退出，那么它将不会用到关闭按钮；如果游戏专为 640×480 或 800×600 大小的窗口作了优化，并且它只支持这样的显示大小，那么它将不会用到最小化/最大化按钮。

提示

CreateWindow()和 CreateWindowEx()有什么区别？区别就是后者可以创建具有扩展风格的窗口。

表 A.5 列出了一系列不同的窗口风格可供选择。

表 A.5　　窗口风格

值	描述
WS_BORDER	创建一个具有薄边框的窗口
WS_CAPTION	创建一个具有标题栏的窗口（包含了 WS_BORDER 风格）
WS_CHILD	创建一个子窗口，具有该风格的窗口不能拥有菜单栏。该风格不能与 WS_POPUP 风格一起使用。
WS_CHILDWINDOW	等同于 WS_CHILD 风格
WS_DISABLED	创建一个初始时禁用的窗口，一个禁用窗口无法接受用户的输入。在窗口被创建出来后可使用 EnableWindow()来启用窗口。
WS_DLGFRAME	创建一个具有边框的窗口，该风格通常用于对话框。具有该风格的窗口不能拥有标题栏。
WS_ICONIC	创建一个初始时最小化的窗口。等同于 WS_MINIMIZE 风格。
WS_MAXIMIZE	创建一个初始时最大化的窗口。

译注[1]：就是 NULL，一般使用它作为顶层窗口的父窗口句柄。

续表

值	描述
WS_MAXIMIZEBOX	创建一个具有最大化按钮的窗口。不能与 WS_EX_CONTEXTHELP 风格组合使用。还必须同时指定 WS_SYSMENU 风格。
WS_MINIMIZE	创建一个初始时最小化的窗口。等同于 WS_ICONIC 风格。
WS_MINIMIZEBOX	创建一个具有最小化按钮的窗口。不能与 WS_EX_CONTEXTHELP 风格组合使用。还必须同时指定 WS_SYSMENU 风格。
WS_OVERLAPPED	创建一个重叠窗口。一个重叠窗口拥有一个标题栏、一个边框。等同于 WS_TILED 风格。
WS_OVERLAPPEDWINDOW	创建一个具有 WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU、WS_THICKFRAME、WS_MINIMIZEBOX 和 WS_MAXIMIZEBOX 风格的重叠窗口。等同于 WS_TILEDWINDOW 风格。
WS_POPUP	创建一个弹出窗口。该风格不能与 WS_CHILD 风格一起使用。
WS_POPUPWINDOW	创建一个具有 WS_BORDER、WS_POPUP 和 WS_SYSMENU 风格的弹出窗口。WS_CAPTION 和 WS_POPUPWINDOW 风格必须组合使用才能使窗口菜单可见。
WS_SIZEBOX	创建的窗口具有一个可调整大小的边框。等同于 WS_THICKFRAME 风格。
WS_SYSMENU	创建的窗口在其标题栏上有一个窗口菜单。必须同时指定 WS_CAPTION 风格。
WS_THICKFRAME	创建的窗口具有一个可调整大小的边框。等同于 WS_SIZEBOX 风格。
WS_TILED	创建一个重叠窗口。重叠窗口具有标题栏和边框。等同于 WS_OVERLAPPED 风格。
WS_TILEDWINDOW	创建一个具有 WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU、WS_THICKFRAME、WS_MINIMIZEBOX 和 WS_MAXIMIZEBOX 风格的重叠窗口。等同于 WS_OVERLAPPEDWINDOW 风格。
WS_VISIBLE	创建一个初始时可见的窗口。

WS_VISIBLE 可使用 ShowWindow()或 SetWindowPos()进行开关。

通用文件 d3dapp.cpp 中使用的风格为：

```
m_dwWindowStyle = WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX
    | WS_VISIBLE;
```

若移除了 WS_THICKFRAME，那么将无法改变窗口的大小。WS_VISIBLE 标志表示这个窗口会在创建后立即可见，而无需再调用 ShowWindow()来使之可见。由于指定了

WS_CAPTION，窗口还有一个标题栏。这里没有指定 WS_MAXIMIZEBOX，所以最大化按钮是灰的。

CreateWindow()返回被创建出来的窗口的句柄，或者当创建失败时返回 NULL。如果没有设置 WS_VISIBLE 标志，还需要后续调用 ShowWindow()来使该窗口可见。

A.4.4 步骤 4：显示窗口

当你决定要显示窗口的时候，可使用 ShowWindow()。

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

hWnd 指定了要显示的窗口句柄。nCmdShow 指定了该窗口将如何显示出来。第一次显示窗口时，应将 WinMain()的 nCmdShow 传递给 ShowWindow()的 nCmdShow 参数，在后续的使用中也可使用表 A.6 中的任意一项。

ShowWindow()函数返回窗口原先的显示状态。若窗口原来是已显示的，则返回非 0 值；若窗口原先是未显示的，则返回 0。

表 A.6 显示属性

值	描述
SW_FORCEMINIMIZE	即使其拥有者线程被挂起，也将最小化该窗口。这个标志应仅用于从另一个非拥有者线程来最小化窗口的情况（只能在 Windows 2000 及更高版本中使用）。
SW_HIDE	隐藏窗口，并激活另一个窗口。
SW_MAXIMIZE	最大化指定的窗口
SW_MINIMIZE	最小化指定的窗口，并激活下一个 Z 轴次序上的顶层窗口
SW_RESTORE	激活并显示窗口。若窗口已被最小化或最大化，系统会将其恢复为原来的大小和位置。应用程序应在恢复一个最小化窗口时指定这个标志。
SW_SHOW	激活窗口并以它的当前大小和位置来显示。

A.4.5 步骤 5：创建消息循环

WinMain()的最后部分是消息循环，消息循环用于接收并处理 Windows 发来的消息。当一个应用程序运行时，它会不断地收到消息，消息在被读取和处理前存放在应用程序的消息队列中。每当应用程序准备读取另一条消息时，可调用 GetMessage()。

```
BOOL GetMessage(
    LPMSG lpMsg,          // message information
```

```
    HWND hWnd,              // handle to window
    UINT wMsgFilterMin,     // first message
    UINT wMsgFilterMax      // last message
);
```

lpMsg 用于存放接收到的消息。所有 Windows 消息都是 MSG 结构类型的数据，如下所示：

```
typedef struct
{
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

消息数据在 message 字段中，附加信息在 wParam 和 lParam 中。MSG 中的 hwnd 字段和 message 字段标识了一个 MSG 结构所表示的消息以及该消息所影响的窗口。它的 wParam 字段和 lParam 字段存储了与消息所涉及的事件种类相关的信息，以及事件的来源。例如，如果事件是由键盘输入引发的，wParam 和 lParam 字段标识了被按下的键，还告知是否同时还按下了一个命令键。结构中还有一个用 ms（毫秒）表示的时间戳 time 和一个以 POINT 结构保存的鼠标坐标 pt。

```
typedef struct tagPOINT
{
    LONG x, y;
} POINT;
```

wMsgFilterMin 和 wMsgFilterMax 指定了一个想要接收的消息范围。如果希望应用程序接收所有的消息，则应将 min 和 max 同时指定为 0。

当用户终止程序时 GetMessage()返回 0，这也将导致消息循环终止；否则它返回非 0。

附注

Javier F. Otaegui 有另一个有趣的方法来摆脱窗口消息泵（参阅 http://www.gamedev.net/reference/articles/article1249.asp）。他使用了第二个线程来执行窗口消息程序。

你不会在所有情况中都使用 GetMessage()，有几种情况你将需要使用 PeekMessage()。

```
BOOL PeekMessage(
    LPMSG lpMsg,            // message information
    HWND hWnd,                  // handle to window
    UINT wMsgFilterMin,         // first message
    UINT wMsgFilterMax,         // last message
    UINT wRemoveMsg             // removal options
);
```

这里多了一个参数——wRemoveMsg，这个参数用于指定消息将被如何处置。表 A.7 列出了该参数的一些可能值。

通常应该使用 PM_REMOVE。默认情况下会处理所有类型的消息。Windows 98/ME/2000 带来了一个新功能，可以让你指定仅处理某一个或某一类的消息。表 A.8 列出了更多的 PeekMessage()标志。

表 A.7　　PeekMessage()标志

值	描述
PM_NOREMOVE	执行完 PeekMessage()后不从队列删除消息
PM_REMOVE	执行完 PeekMessage()后从队列删除消息

表 A.8　　PeekMessage() II 标志

值	描述
PM_QS_INPUT	处理鼠标和键盘消息
PM_QS_PAINT	处理绘制消息
PM_QS_POSTMESSAGE	处理所有 post 而来的消息，包含定时器和热键（hot key）消息
PM_QS_SENDMESSAGE	处理所有 send 而来的消息

那么 PeekMessage()和 GetMessage()到底有什么区别呢？为什么在某些情况下应该使用 PeekMessage()？在 Windows 内部，GetMessage()和 PeekMessage()执行了相同的代码。两者最主要的差别体现在当没有消息可返回给应用程序的时候。在这种情况中，GetMessage()将应用程序置于“睡眠”状态，而 PeekMessage()则返回给应用程序一个 FALSE 值。你可能不会希望你的游戏在下一个消息到达消息队列之前一直处于“睡眠”状态吧，因此更好的做法是在游戏激活（active）时使用 PeekMessage()，而在游戏暂停时使用 GetMessage()。伪代码如下：

```
BOOL bGotMsg;
MSG msg;
While(WM_QUIT != msg.message)
{
    if (m_bActive)
        bGotMsg = PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE);
    else
        bGotMsg = GetMessage(&msg, NULL, 0U, 0U);
...
```

继续我们先前的讨论，在 while(GetMessage...)程序中有两个函数调用：

```
...
TranslateMessage(&msg);     /* allow use of keyboard */
DispatchMessage(&msg);      /* return control to Windows */
...
```

TranslateMessage()将 Windows 产生的虚拟键码翻译为字符消息。每当读取到一条消息后，如果必要，对它进行翻译。然后 DispatchMessage()就会将消息发送到窗口过程，在本例中窗口过程为 WndProc()。图 A.3 展示了消息队列、WinMain()中的消息泵[1]（message pump）、窗口过程三者之间的关系。GetMessage 表示 GetMessage()和 PeekMessage()。

当消息循环终止时，WinMain()将 msg.wParam 的值返回给 Windows。该值包含了程序终止时所产生的返回值[2]。

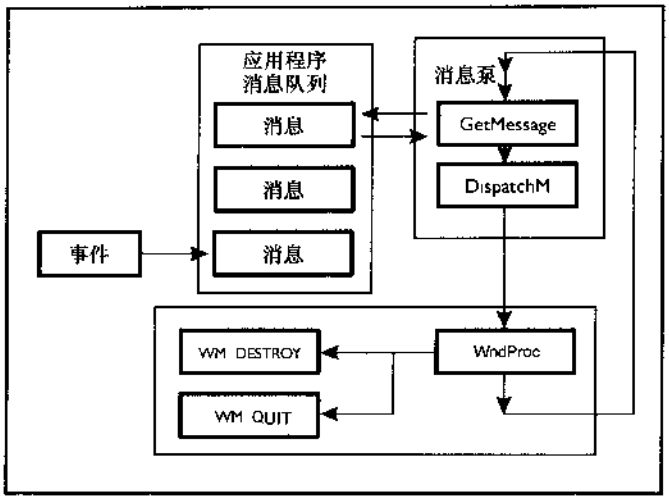


图 A.3 消息泵

A.4.6 窗口过程

示例程序的第二个函数是窗口过程，在注册窗口类时用到了这个函数名（我使用 WndProc()作为函数名）。该过程接收 MSG 结构的前 4 个成员作为参数。示例的窗口过程只响应一条消息——WM_DESTROY。这个消息是在用户终止程序时发送的。PostQuitMessage()会发送一条 WM_QUIT 消息到应用程序，WM_QUIT 将导致 GetMessage()/PeekMessage()返回 FALSE，从而终止消息循环。若一切正常，将会返回 WM_QUIT 中的 msg.wParam。

WndProc()接收到的其他消息则都通过一个对 DefWindowProc()的调用传递给 Windows 系统。DefWindowProc()调用了默认的窗口过程，默认窗口过程处理了那些应用程序未处理的窗口消息，该函数可确保每个消息都会得到处理。

译注[1]："消息循环"的一种别称。

译注[2]：即传递给 PostQuitMessage()的参数。

A.5 为游戏而优化的窗口框架

几乎每本讲述 Windows 编程的书籍都会提及上述这种通用的窗口框架，在钻研了这种窗口框架后，毋庸置疑，你肯定会希望看看一个专为游戏而作优化的版本。

```
#include <windows.h>
#include "resource.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
VOID Render();
char szWinName[] = "MyWin"; /* name of window class */
BOOL bActive = TRUE;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
LPSTR lpszArgs, int nWinMode)
{
    HWND hwnd;
    MSG msg;

    /* Step 1: Define a window class. */
    WNDCLASS wcl;
    wcl.hInstance = hThisInst;          /* handle to this instance */
    wcl.lpszClassName = szWinName;      /* window class name */
    wcl.lpfnWndProc = WndProc;          /* window function */
    wcl.style = 0;                      /* default style */
    /* icon style */
    wcl.hIcon = LoadIcon(hThisInst, MAKEINTRESOURCE(IDI_ICON1));
    /* cursor style */
    wcl.hCursor = LoadCursor( hThisInst, MAKEINTRESOURCE(IDC_HULLA));
    wcl.lpszMenuName = NULL;            /* no menu */
    wcl.cbClsExtra = 0;                 /* no extra */
    wcl.cbWndExtra = 0;                 /* information needed */
    /* Make the window background white. */
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

    /* Step 2: Register the window class. */
    if(!RegisterClass (&wcl))
        return 0;
    DWORD dwWindowStyle = WS_SYSMENU;
    DWORD dwCreationWidth = 640;
    DWORD dwHeight = 480;
    RECT rc;
    SetRect(&rc, 0, 0, dwWidth, dwHeight);
    AdjustWindowRect(&rc, dwWindowStyle, TRUE);

    /* Step 3: Now that a window class has been registered,
        a window can be created. */
    hwnd = CreateWindow(szWinName,      /* name of window class */
                        "FAKE 4",       /* title */
                        dwWindowStyle,  /* window style - normal */
                        /* X coordinate - let Windows decide */
                        CW_USEDEFAULT,
                        /* y coordinate - let Windows decide */
```

```
                        CW_USEDEFAULT,
                        (rc.right - rc.left),    /* width */
                        (rc.bottom - rc.top),    /* height */
                        HWND_DESKTOP,            /* no parent window */
                        NULL,                    /* no menu */
                        /* handle of this instance of the program */
                        hThisInst,
                        NULL                     /* no additional arguments */
                        );

    /* Step 4: Display the window. */
    ShowWindow(hwnd, nWinMode);

    /* Step 5: Create the message loop. */
    BOOL bGotMsg;
    while (WM_QUIT != msg.message)
    {
        if( bActive)
            bGotMsg = PeekMessage (&msg, NULL, 0U, 0U, PM_REMOVE);
        else
            bGotMsg = GetMessage (&msg, NULL, 0U, 0U);
        if(bGotMsg)
        {
            /* allow use of keyboard */
            TranslateMessage(&msg);
            /* return control to Windows */
            DispatchMessage(&msg);
        }
        else
        {
            if(bActive)
                Render();
        }
    }
    return msg.wParam;
}

//-----------------------------------------------
// Name: WndProc
// Desc: This function is called by Windows and is passed
// messages from the message queue
//-----------------------------------------------
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_DESTROY: /* terminate the program */
        PostQuitMessage(WM_QUIT);
        break;
    case WM_KEYDOWN:
        switch (wParam)
        {
            case VK_ESCAPE:
                PostQuitMessage(WM_QUIT);
                break;
```

```
                case VK_F1:
                {
                    bActive = FALSE;
                    MessageBox( hwnd,
                                "Here comes your help text",
                                "Help for FAKE 4", MB_ICONQUESTION|
                                MB_OK | MB_SYSTEMMODAL );
                    bActive = TRUE;
                }
            }
            break;
        }
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    //-----------------------------------------------
    // Name: Render
    // Desc: dummy function to show the use of the enhanced skeleton
    //-----------------------------------------------
    VOID Render()
    {
    };
```

第一处增强的地方就在步骤 3 之前。

```
DWORD dwWindowStyle = WS_POPUP | WS_CAPTION;
DWORD dwWidth = 640;
DWORD dwHeight = 480;
RECT rc;
SetRect(&rc, 0, 0, dwWidth, dwHeight);
AdjustWindowRect(&rc, dwWindowStyle, TRUE);
```

定义了一种只有标题栏的窗口风格，客户区为 640×480 像素大小。SetRect()设置了指定矩形的坐标。AdjustWindowRect()基于想要的客户矩形大小来计算所应具备的窗口矩形大小，由此你才能真正获得一个客户区为 640×480 大小的窗口。真正的改进是步骤 5 中的消息循环。

```
/* Step 5: Create the message loop. */
while (WM_QUIT != msg.message)
{
    if( bActive)
        bGotMsg = PeekMessage (&msg, NULL, 0U, 0U, PM_REMOVE);
    else
        bGotMsg = GetMessage (&msg, NULL, 0U, 0U);
    if(bGotMsg)
    {
```

译注：此处代码有一个小 bug，msg 结构在第一次进入 while 循环时，由于并未初始化，所以不能保证 msg.message 肯定不等于 WM_QUIT，因此就有可能会造成循环一次都不会执行。修正方法很简单，使用 do-while 结构即可，如下：

```
do
{
    ...;
} while( WM_QUIT != msg.message );
```

```
            TranslateMessage(&msg);     /* allow use of keyboard */
            DispatchMessage(&msg); /* return control to Windows */
        }
        else
        {
            if(bActive)
                Render();
        }
    }
```

当应用程序处于激活状态（active）时，调用 PeekMessage()，此时如果消息队列中没有消息，该函数返回 FALSE。当应用程序是非活动状态（inactive）时，调用 GetMessage()，此时如果消息队列中没有消息，该函数使应用程序进入“睡眠”状态。只有在收到消息时才调用 TranslateMessage()和 DispatchMessage()。另外也只有在应用程序处于激活状态时才调用 Render()。我为应用程序处于非激活状态的情况举一个小例子[1]（见图 A.4）。

图 A.4 窗口框架 2

```
case VK_F1:
{
    bActive = FALSE;
    MessageBox( hwnd, "Here comes your help text",
                "Help for FAKE 4", MB_ICONQUESTION|MB_OK | MB_SYSTEMMODAL );
    bActive = TRUE;
}
```

译注[1]：更好的例子可以考虑一下通过响应 WM_ACTIVATEAPP 或 WM_ACTIVATE 来设置 bActive 变量。

```
break;
```

如果出现了该消息框，bActive 为 FALSE，Render()也不会被调用。

就这些了，我希望你已对如何编写一个用于游戏的窗口代码有所认识。你不会用到很多窗口方面的功能，因为在全屏的游戏中你必须自己动手构建对话框和其他 GUI 对象，通常 Windows 的对话框窗口和 GUI 对象并不适用于游戏。

A.6 窗口资源

在 Windows 编程刚开始的时候，所有程序员只能以手工方式来设置资源。如今 Visual C/C++集成开发环境帮了很大的忙。你只需要按以下步骤来创建及包含资源即可。

（1）选择“File—>New—>Resource Script”。

（2）给你的资源脚本起一个好名字，如 winskel（见图 A.5）。点击 OK。新的资源文件将被命名为 winskel.rc。

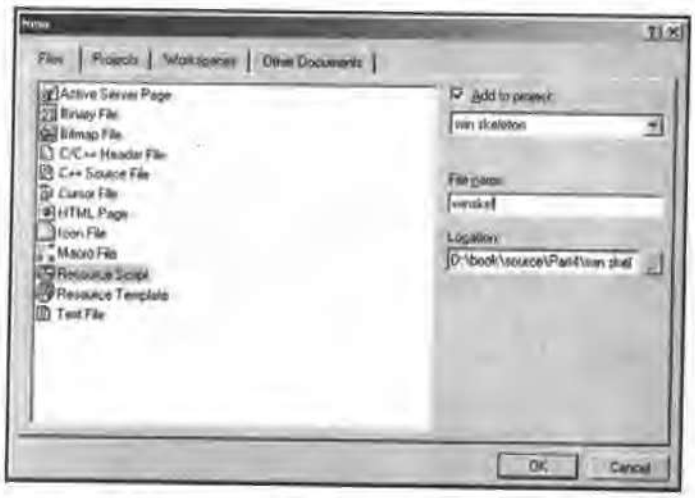


图 A.5 创建一个资源脚本（Resource Script）

（3）然后在 workspace 中点击“Resource View”标签页，然后右键点击 winskel 资源。

（4）点击 Insert（见图 A.6）。将会出现一个对话框，由此你可以选择一种资源类型（见图 A.7）。

我借用标准的图标 directx.ico，这个图标使用蓝色作为背景，该蓝色背景对于前景颜色而言是透明的。

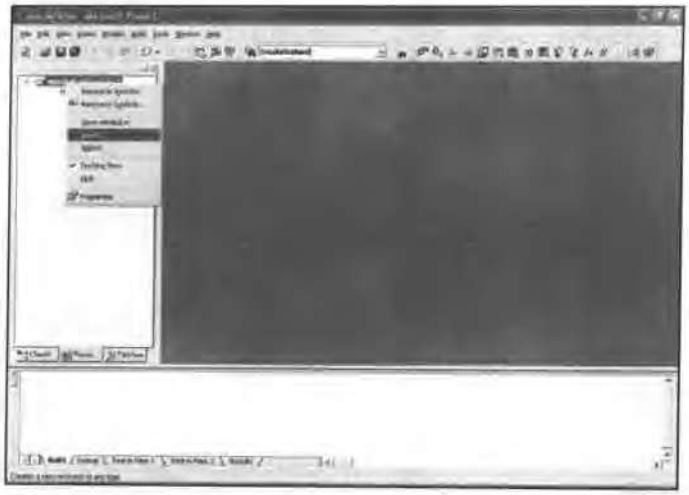


图 A.6　插入一个资源

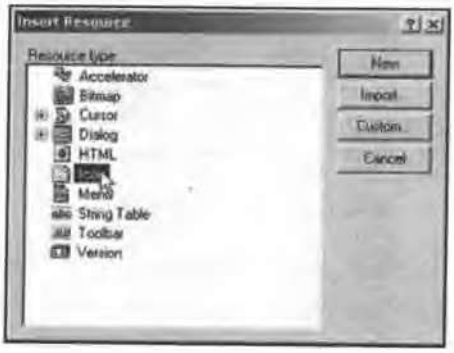


图 A.7　创建一个图标

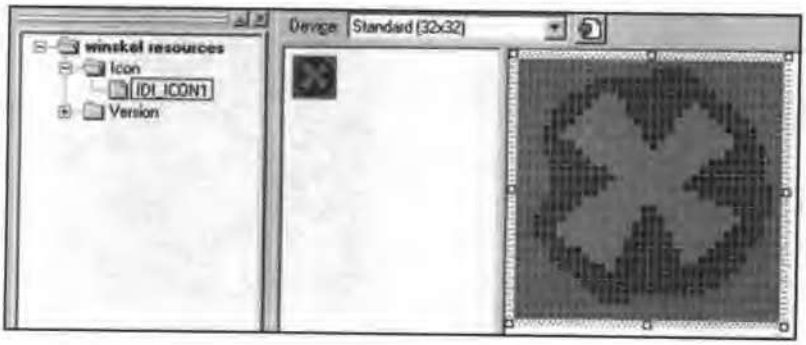


图 A.8　给 icon 上色

（5）为了加载一个图标，必须包含文件 resource.h，并使用以下代码来加载它：

```
wcl.hIcon = LoadIcon(hThisInst, MAKEINTRESOURCE(IDI_ICON1)); /* icon style */
```

（6）我还决定再创建一个版本控制（version control）资源（见图 A.9）。版本控制可以帮助你设置程序的不同版本，并可在.exe 文件中保存版权信息。

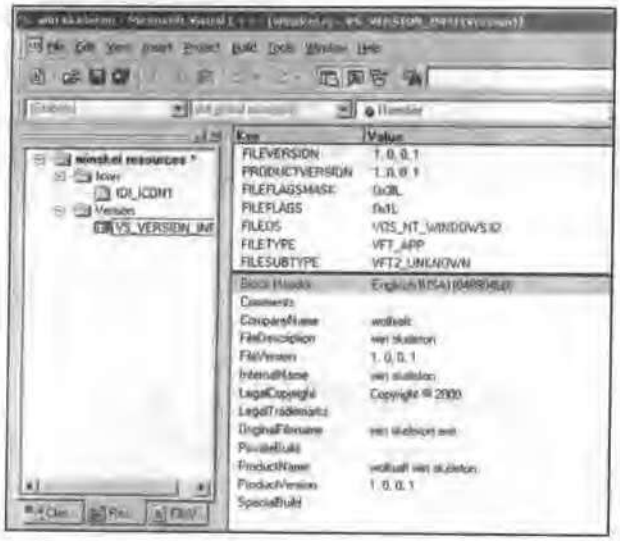


图 A.9　创建一个版本控制资源

（7）有一个别致的光标还会更棒。在 workspace 中点击“Resource View”标签页，右键单击 winskel 资源，然后选择 Cursor（见图 A.10）。

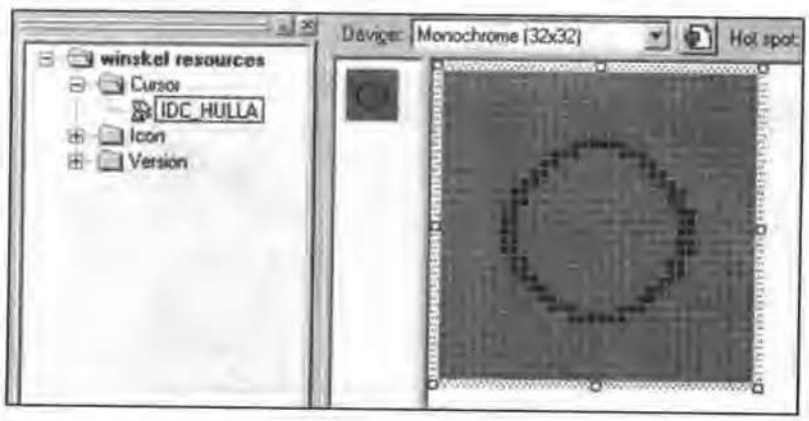


图 A.10　光标作品

现在这个简单的框架程序中已拥有 3 个资源（见图 A.11）。

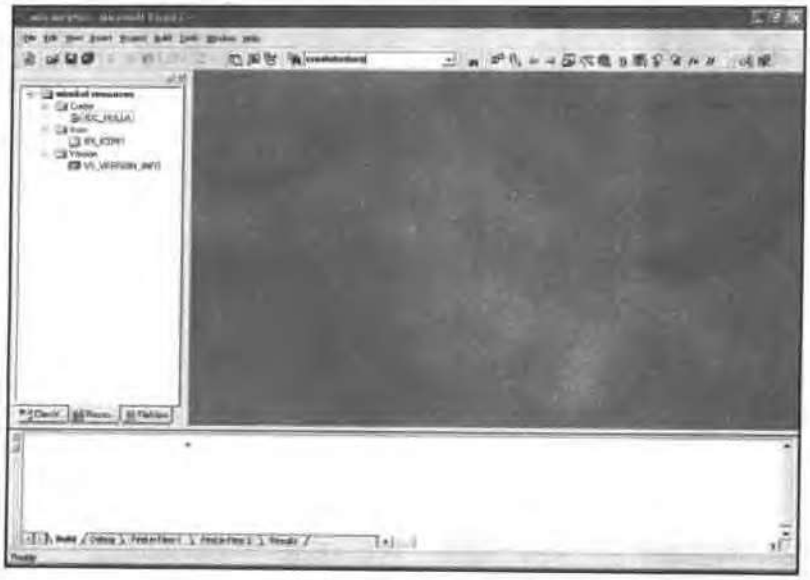


图 A.11　3 个资源

现在你已了解了如何创建第一个带有资源的 Windows 程序。尽管你不得不在 Visual C/C++编辑器中手动键入该程序的所有代码，但在创建资源时还是得到了来自于 IDE 的很多帮助。

附 录 B

C++ 入 门

数年以前，在新闻组中有很多关于在游戏里应使用 C 还是 C++的讨论。一派认为 C 可以产生更快的代码，并以著名的游戏作为证据，例如 *Quake* 和 *Unreal*。另外一派则认为 C++更为优雅，有利于更好地组织程序的结构，并且更适用于在开发团队中使用。

实践永远是一个好方法，通过实践可以得出一个比较实用的解决方案：既有 C 的速度，又有 C++易于构建游戏代码结构的能力。我假定许多游戏程序员没有使用到太多的 C++特性。如果看一下 DirectX SDK 的示例，你就会发现这些示例中并存着许多 C++代码和传统的 C 代码。

本附录主要讨论以下内容：

- 关于面向对象程序设计的特征与使用的一些简要解释；
- 对一些 C++关键字和数据类型的概述，这些关键字和数据类型不是 C 中没有的，就是在 C 和 C++中有不同用法的；
- 创建和使用类，或者说如何将 C 语言的结构演变成 C++的类，这是 C++中面向对象程序设计的基石之一；
- 其他 C++程序设计技术，例如访问限定符、访问成员方法和成员变量的操作符、构造函数、析构函数、拷贝构造函数和 this 指针。

B.1 什么是面向对象程序设计

从面向对象的程序设计观点来看，一个程序描述了一个由各个对象互相作用的系统。这与面向过程的程序设计观点相反，面向过程的观点认为程序描述的是一系列将要被执行的步骤。

面向对象的程序设计包括了一些关键性的概念，其中最基本的是抽象，抽象可以简化大

型程序的编写；另外一个概念是封装，封装可以使程序的修改和维护变得非常容易；最后是类继承的概念，这是一种强大的分类工具，它可使程序的扩展变得非常容易。[1]

作为一种面向对象的语言，你必须先要了解C++能从这些概念中获得什么好处，然后才能理解C++。

B.1.1 抽象

抽象是一种忽略细节的过程，这种忽略是为了将注意力集中到本质问题上。高级程序设计语言都支持一种高层次的抽象。一个比较典型的例子就是拿汇编语言编写的程序与用C语言编写的程序作一个比较：汇编源程序包含了很多的细节，它描述了让计算机完成一项任务所要做的各种操作；而C语言源程序则可以用更抽象的方式来描述计算机所要做的事情，这些抽象让程序更加清晰、更易于理解。

所有过程化语言都支持过程化抽象，也就是把代码片断放入一个函数，然后以相同的方式来重用这些代码。例如，考虑一个矩阵乘法：

```
FLOAT pM[16];
ZeroMemory( pM, sizeof(D3DXMATRIX) );
for( WORD i=0; i<4; i++ )
    for( WORD j=0; j<4; j++ )
        for( WORD k=0; k<4; k++ )
            pM[4*i+j] += pM1[4*i+k] * pM2[4*k+j];
memcpy( pOut, pM, sizeof(D3DXMATRIX) );
```

你不必在所有需要完成4×4矩阵乘法的地方都重写这段代码。相反，你可以把这些代码组织成一个函数，从而抽象它，就像下面这样：

```
D3DXMATRIX* D3DXMatrixMultiply (D3DXMATRIX* pOut,
                                CONST D3DXMATRIX* pM1,
                                CONST D3DMATRIX* pM2)
{
    FLOAT pM[16];
    ZeroMemory( pM, sizeof(D3DXMATRIX) );
    for( WORD i=0; i<4; i++ )
        for( WORD j=0; j<4; j++ )
            for( WORD k=0; k<4; k++ )
                pM[4*i+j] += pM1[4*i+k] * pM2[4*k+j];
    memcpy( pOut, pM, sizeof(D3DXMATRIX) );
    return (pOut);
}
```

调用D3DXMatrixMultiply()可使源代码更易于阅读和理解，同时也会节省不少打字时间。

数据抽象可以让你忽略关于如何表述一种数据类型的细节。C语言提供了int、float之类

[1] 译注：面向对象程序设计还有另一块非常重要的基石——多态。译者认为，面向对象有三块基石：封装、继承和多态，而抽象是任何程序设计范型都具备的概念。

的数据类型，这便允许你直接输入诸如 42.5 这样的数值，而不是输入十六进制字节。并且令人欣慰的是，C 程序员不需要操心诸如二进制浮点算术之类的问题。但是在另一方面，由于 C 不支持对字符串的抽象，所以这就要求你将字符串作为一系列的字符来进行处理。

C 通过结构和 typedef 来支持对用户自定义数据类型的抽象。程序员可使用结构将信息的几个部分作为一个整体来操作，而不是分别操作。

```
struct Mesh
{
    char cName[68];                    // 65 chars, 32-bit aligned == 68 chars in file
    int iNumVertices;                  // Mesh vertices
    int iMeshFrameNum;                 // animation frames in mesh
    VMESHFRAME *vMeshFrames;           // stores vertices per animation frame in
                                       // vertices[FrameNum][VertexNum]
    int iNumTriangles;                 // Mesh triangles
    TRIANGLEVERT *pTriangles;
    int iNumTextures;                  // Mesh textures
    TEXNAMES *pTexNames;
    LPDIRECT3DTEXTURE8* pTexturesInterfaces;
    TEXCOORDS *pfTextureCoords;        // Mesh tex coordinates
    TEXCOORDS *pfEnvTextureCoords;     // not used here
};
```

用户自定义的数据类型可使对变量的操作更易于理解，例如：

```
Mesh.iNumVertices = 5;
```

还可以用 typedef 来定义数据类型或数据结构，但这在概念上没有什么特别的优势[1]：

```
typedef int TRIANGLEVERT[3];
typedef char TEXNAMES[68];
typedef D3DXVECTOR3 *VMESHFRAME;
typedef struct
{
    float u;
    float v;
} TEXCOORDS;

typedef struct
{
    char cName[68];                    // 65 chars, 32-bit aligned == 68 chars in file
    int iNumVertices;                  // Mesh vertices
    int iMeshFrameNum;                 // animation frames in mesh
    VMESHFRAME *vMeshFrames;           // stores vertices per animation frame in\
                                       // vertices[FrameNum][VertexNum]
    int iNumTriangles;                 // Mesh triangles
    TRIANGLEVERT *pTriangles;
    int iNumTextures;                  // Mesh textures
    TEXNAMES *pTexNames;
    LPDIRECT3DTEXTURE8* pTexturesInterfaces;
```

译注[1]：这是因为 typedef 并不会引入新的数据类型，只是给原有类型取一个新的名字而已；在语言中，typedef 后的类型名与原有类型是完全等价的，它主要的意义在于方便书写。

```
    TEXCOORDS *pfTextureCoords;          // Mesh tex coordinates
    TEXCOORDS *pfEnvTextureCoords;       // not used here
} MD3MESH;
```

现在MD3MESH是一个抽象数据类型了，它可作为函数的参数来与函数一起组合使用，这些函数处理了诸如填充特定字段之类的细节工作。

这种传统的过程化方法可以让你认识到数据抽象和过程抽象这两种不同的技术。接下来的小节将描述一种更为优雅的方法，说明如何使用类来进行面向对象的程序设计，其中也会涉及到这两种技术。

B.1.2 类

类通过结合过程化抽象和数据抽象来描述一个高层次的实体。例如，考虑一个CMD3类，它包含了.md3模型所需要的所有几何体数据和纹理接口指针：

```
class CMD3Model
{
private:
    int                 iNumMeshes;
    MD3MESH             *pMd3Meshes;
    MD3BONEFRAM         *md3BoneFrame;
    MD3TAG              **md3Tags;
    FILE                *LogFile;     // the log file for md3 geometry data.txt
                                      // and md3 textures.txt
    LPDIRECT3DVERTEXBUFFER8 m_pVB; // Buffer to hold vertices
    long ReadLong(FILE* File);     // file i/o helper
    short ReadShort(FILE* File);
    FLOAT ReadFloat(FILE* File);
    void CreateVB(LPDIRECT3DDEVICE8 lpD3Ddevice);        // vertex buffer system
    void DeleteVB();
    void CreateTextures(LPDIRECT3DDEVICE8 lpD3Ddevice);  // texture system
    void DeleteTextures();

public:
    // class methods
    BOOL CreateModel( char *fname, LPDIRECT3DDEVICE8 lpD3Ddevice);
    void DeleteModel();
    void InitDeviceObjects(LPDIRECT3DDEVICE8 lpD3Ddevice);
    void DeleteDeviceObjects();
    void Render(LPDIRECT3DDEVICE8 lpD3Ddevice);
    CMD3Model(); // constructor/destructor
    ~CMD3Model();
};
```

通过类可以使你很容易地在程序与计算机之间创建一个额外的分离层。这样，自定义的高层实体就拥有了许多优点，这些优点就如同浮点数相对于字节操作、打印语句相对于MOV指令那样的进步。

这个类只能通过5个方法进行访问：CreateModel()、DeleteModel()、InitDeviceObjects()、

DeleteDeviceObjects()、Render()，没有办法从类的外面来访问类中的变量。这也就是面向对象程序设计的另外一个优点——封装。

附注

关键字 protected 允许派生类可以看到成员，但不会被类的用户看到。这样便可以提供给派生类比类用户更多的信息。

B.1.3 封装

封装是一个隐藏类的内部事务的过程，它用以支持抽象和实施抽象，这样就要求类的接口（公有部分）和类的实现（私有部分）具有明显的区别。这样的区别通过 priavte 标签和 public 标签来表示。

一个类的接口描述了该类能做什么，而它的实现描述了这个类将如何去做。用户是从对象能执行什么操作的角度来看待它，而不是从它的数据结构来看待它。

附注

C++封装不提供安全性的保证。程序员仍有可能通过&和*操作符访问到类的私有数据。

一个正确的封装类是通过它的函数来包装或隐藏其数据的，你只有通过调用这些函数才能访问到它的数据。可以看一下图 B.1 的示例，左面是公有数据成员，右面是私有数据成员。

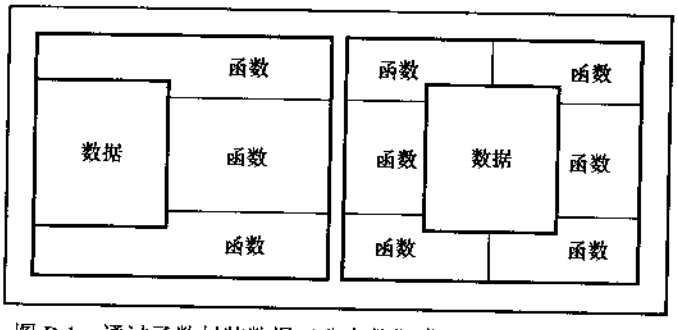


图 B.1 通过函数封装数据（公有数据成员和私有数据成员）

数据隐藏的目标是使每个模块尽可能不依赖于其他模块。在理想情况下，一个模块可以不需要知道其他模块所使用的数据结构，而只需通过其他模块的接口就可以访问到它们。

你在前面已经看到了，在 CMD3 示例中，即使类的接口以后会发生变化，使用封装类也比直接访问数据结构好。因为通过封装类你可以在现有接口上添加改动，例如，旧的接口是 IDirect3D7，那么以后还可以添加一个 IDirect3D8 的接口，这样所有使用旧接口的程序仍然可以正常运行。

在这些理论之后，我们需要一些实践了。

B.1.4 声明一个类

让我们从附录A结束的地方继续，创建一个C++版的窗口框架程序。让我们从下面这些代码开始。

```
#ifndef APP_H
#define APP_H
class CApp
{
protected:
    TCHAR* m_strWindowTitle;
    DWORD m_dwWidth;
    DWORD m_dwHeight;
private:
    BOOL m_bActive;
    HRESULT Render();
public:
    // used in Fake.cpp to create, run and pause fake
    virtual HRESULT Create (HINSTANCE hInstance);
    virtual LRESULT MsgProc (HWND hWnd, UINT uMsg,
                             WPARAM wParam, LPARAM lParam);
    virtual WPARAM Run();
    virtual VOID Pause(BOOL bPause);
    // internal constructor
    CApp();
};
#endif
```

这个类的声明位于app.h头文件中，该文件在源代码目录win skeleton++中。

以下的预处理语句用于条件编译。

```
#ifndef APP_H
#define APP_H
...
#endif
```

这些预处理器指示可以防止在一个多模块的程序中多次包含同一个头文件。类声明看上去和结构声明很相似，但是它同时包含了函数和数据，而不是仅包含数据。这个类声明了以下这些内容：

- 每个CApp实例的内容，包括窗口标题、宽度、高度、渲染的状态；
- 用于处理数据的5个方法原型。

我在App.cpp中提供了成员函数的定义：

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <tchar.h>
```

```
#include "App.h"
#include "resource.h"
//-----------------------------------
// global this pointer
//-----------------------------------
static CApp* g_pCApp = NULL;
//-----------------------------------
// Name: FAKEApp()
// Desc: Constructor
//-----------------------------------
CApp::CApp()
{
    g_pCApp = this;
    m_bActive = FALSE;
    m_strWindowTitle = _T("FAKE 4");
    m_dwWidth = 640;
    m_dwHeight = 480;
}
//-----------------------------------
// Name: WndProc()
// Desc: static msg handler which passes messages to the application class
// in the fake.cpp file
//-----------------------------------
LRESULT CALLBACK WndProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    return g_pCApp->MsgProc (hWnd, uMsg, wParam, lParam);
}
//-----------------------------------
// Name: Create()
// Desc: static msg handler which passes messages to the application class
// in the fake.cpp file
//-----------------------------------
HRESULT CApp::Create(HINSTANCE hThisInst)
{
    HWND hwnd;
    /* Step 1: Define a window class. */
    WNDCLASS wcl;
    wcl.hInstance         = hThisInst;      /* handle to this instance */
    wcl.lpszClassName     = _T("Crusher");  /* window class name */
    wcl.lpfnWndProc       = WndProc;        /* window function */
    wcl.style             = 0;              /* default style */
    /* icon style */
    wcl.hIcon = LoadIcon(hThisInst, MAKEINTRESOURCE(IDI_ICON1));
    /* cursor style */
    wcl.hCursor           = LoadCursor( hThisInst, MAKEINTRESOURCE(IDC_HULLA));
    wcl.lpszMenuName  = NULL;               /* no menu */
    wcl.cbClsExtra        = 0;              /* no extra */
    wcl.cbWndExtra        = 0;              /* information needed */
    /* Make the window background white. */
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH) ;
    /* Step 2: Register the window class. */
    if(!RegisterClass (&wcl))
        return 0;
    DWORD dwWindowStyle = WS_POPUP | WS_CAPTION | WS_VISIBLE | WS_SYSMENU;
    RECT rc;
    SetRect(&rc, 0, 0, m_dwWidth, m_dwHeight);
```

```
    AdjustWindowRect(&rc, dwWindowStyle, TRUE);
    /* Step 3: Now that a window class has been registered,
                a window can be created. */
    hwnd = CreateWindow(_T("Crusher"), /* name of window class */
                        m_strWindowTitle, /* title */
                        dwWindowStyle, /* window style - normal */
                    CW_USEDEFAULT, /* X coordinate - let Windows decide */
                        CW_USEDEFAULT, /* y coordinate - let Windows decide */
                        (rc.right - rc.left), /* */
                        (rc.bottom - rc.top), /* */
                        HWND_DESKTOP, /* no parent window */
                        NULL, /* no menu */
                        hThisInst, /* handle of this instance of the program */
                        NULL /* no additional arguments */
                    );
    /* Step 4: Display the window. */
    //ShowWindow(hwnd, nWinMode); // obsolete because of the use of WS_VISIBLE
    return S_OK;
}
//-----------------------------------
// Name: MsgProc()
// Desc: Message handling function
//-----------------------------------
LRESULT CApp::MsgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_CLOSE:
        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
//-----------------------------------
// Name: Render()
// Desc: Render the app
//-----------------------------------
VOID CApp::Render()
{
    // do anything
}
//-----------------------------------
// Name: Pause()
// Desc: Sets bActive
//-----------------------------------
VOID CApp::Pause(BOOL bPause)
{
    m_bActive = bPause? 0 : 1;
}
//-----------------------------------
// Name: Run()
// Desc: Message Loop and render method
//-----------------------------------
WPARAM CApp::Run()
{
```

```
        BOOL bGotMsg;
        MSG msg;
        PeekMessage (&msg, NULL, 0U, 0U, PM_REMOVE);
        /* Step 5: Create the message loop. */
        while (WM_QUIT != msg.message)
        {
            if( m_bActive)
                bGotMsg = PeekMessage (&msg, NULL, 0U, 0U, PM_REMOVE);
            else
                bGotMsg = GetMessage (&msg, NULL, 0U, 0U);
            if(bGotMsg)
            {
                TranslateMessage(&msg);          /* allow use of keyboard */
                DispatchMessage(&msg);      /* return control to Windows */
            }
            else
            {
                if(m_bActive)
                    Render();
            }
        }
        return msg.wParam;
    }
```

你可能已经注意到，我使用 Pause()封装了 m_bActive 变量，不过现在先让我们看一下构造函数。

B.1.5 构造函数

构造函数可确保对象总是包含有效的数据。它是一种特殊的初始化函数，当实例化一个类的时候就会自动调用它。构造函数可以防止由于使用了未初始化的对象而导致的错误，并且还可以检查一个值是否超过了有效范围。构造函数必须与它所在的类同名。虽然不是强制要求必须要写一个构造函数，但一般来说使用它总是一个不错的主意。

在 CApp()中初始化了所有的成员：g_pCApp、m_bActive、m_strWindowTitle、m_dwWidth、m_dwHeight。构造函数创建了对象，但它不返回任何值。

在一个类中可以定义多个构造函数，每个构造函数具有不同的参数列表，这些称为“重载构造函数”。如果需要用多种方法对对象进行初始化，那么这就很有用了。过一会儿我就会告诉你如何重载构造函数。

B.1.6 析构函数

与构造函数相对应的就是析构函数。它的用途是在销毁对象前执行一些必须的清理工作。析构函数的名字应与类名相同，但有波浪线（~）作为前缀。它是在对象离开作用域时被自动调用的一个成员函数。

那么什么是作用域呢？这里有一个简单的示例：

```
int number = 123;
VOID ShowMeANumber()
{
    int number = 456;
    Showme (::number);
    Showme (number);
}
```

输出结果为：

```
123
456
```

这里有一个全局变量 number 和一个局部变量 number。在上面所示的情况下，C 语言总是使用局部变量，而在 C++中则可以使用作用域操作符::来获取全局变量。所以，作用域就是用于区分名字含义的范围。

作用域操作符也可以用于访问成员函数，在这种情况下，它出现在 C++程序的类名和函数名中间：

```
VOID CApp::Render()
```

这表示指定的函数是一个特定类的成员函数。

现在回到析构函数。在这里我没有使用析构函数，那是因为通常只有比较复杂的类才需要使用到它，这些类一般都会使用析构函数来释放一些动态分配的资源。析构函数是不能重载的，它没有参数并且也没有返回值。

this 指针

在 win skeleton++的构造函数中有一个有趣的变量——g_pCApp，它使用了一个称为“this”的指针。当调用一个对象的成员函数时，编译器就会先将该对象的地址赋予 this 指针，然后再调用那个函数。每当成员函数需要访问类的数据成员时，就会隐式地使用 this 指针。输入下面代码来创建这样一个示例：

```
void CApp::Render (BOOL active)
{
    BOOL bPause;
    bPause = active;
}
myApp.Render(FALSE);
```

这等价于：

```
void CApp::Render (BOOL active)
{
    BOOL bPause;
    this->bPause = active;
```

```
        // or (*this).pause = active
    }
    myApp.Render(FALSE);
```

这两个 Render()中的语句都是等价的。

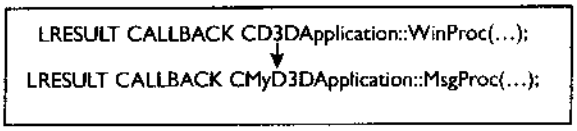


图 B.2　消息过程

也可以使用 this 指针避免自赋值。例如：

```
void String::operator= (const String &other)
{
    if (&other == this)
        return;
    ...
}
```

当试图进行自赋值时，这个方法什么都不会做就直接退出了。作为这段代码的小结，现在提出这样的要求：为了封装所有的数据，应将成员变量都标记为私有属性，如此每个成员变量都必须通过成员函数来访问。要确保每个变量都具有一个有效的值，可以使用构造函数来初始化它们。

接下来我将引出下一个面向对象程序设计的基础理论：类层次和继承。

B.1.7　类层次和继承

面向对象程序设计的一个卓越特性就是类层次。在 C++中可以通过从一个基类派生出一些新类，从而将它们定义为基类的子类型或者特殊的分类。这在 C 中是做不到的，在 C 中必须要把所有类型都看作是互相完全独立的类型。

定义一个类层次的好处在于可以共享代码与接口。一个派生类可以共享基类中的代码和接口，示例程序就同时使用了这两种共享。

1．代码继承

可以通过从基类派生出一个类，从而减少冗余代码。在游戏中管理不同种族数据库的程序就可以作为一个很好的说明代码继承的示例。

使用一个称为 alien 的 C 结构并不是个好主意，因为每一个种族都会有一些不同的信息。并且也无法定义出一个包含所有种族信息的结构类型，因为要在一个结构中包含所有可能的字段比较浪费空间。联合（union）可能会有些用，不过联合会导致 C 程序难以阅读和理解。

在 C++中通过一个基类和派生类很容易做到这些。下面是这个示例的伪代码。

```
class CAlien
{
public:
    CAlien();
    VOID Attack(HUMAN human);
    VOID RunTo(WAYPOINT waypoint);
    VOID SetHealth(DWORD dwHealth);
    DWORD GetHealth ();
private:
    char cName[32];
    DWORD dwHealth;
};
```

这是一个简化的示例，这个 CAlien 类只包含了很少的值，而在实际情况下，可能会有更多的值。假设有这么几种不同的种族，其中有的种族可以飞行、有的只能蹲着行走、有的只会水中游泳、而有的甚至可以远距传物，其中的某些族人可以咬人、手持喷火武器或者施展魔法。

```
class CMagician : public CAlien
{
public:
    Magician();
    VOID SetGlow(BOOL bGlow);
private:
    BOOL bGlow;
};
```

或者

```
class CFrankenstein : public CAlien
{
public:
    Frankenstein();
    VOID StruckByLightning(BOOL bStruck);
private:
    BOOL bStruck;
};
```

CMagician 和 CFrankenstein 都是基类 CAlien 的派生类。为了声明一个派生类，需要以这样的方式来书写：类名称 + 分号 + 关键字 public + 基类名称。

附注

使用 private 关键字进行继承的话，可以使基类的公有和保护成员都成为派生类的私有成员，这样它们都只能让派生类访问。

在派生类的声明中，可以在基类的基础上声明更多的成员，这些成员描述了与基类不同的特性。这样，CFrankenstein 的实例除了包含它自己的数据成员外，还包含了所有 CAlien 的数据成员（见图 B.3）。

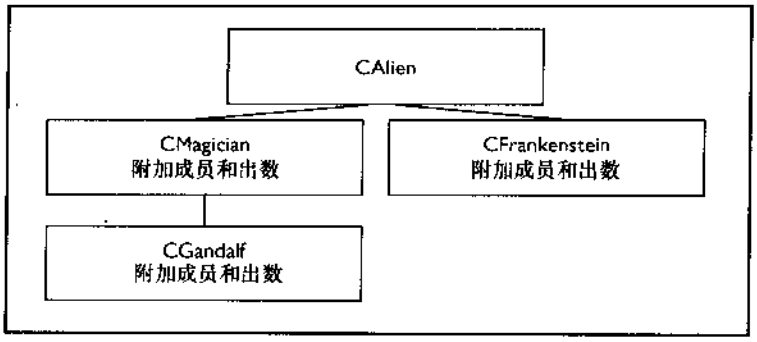


图 B.3　类层次

派生类的成员函数不能访问基类的私有成员，所以它必须通过基类的公有接口[1]来访问基类的私有成员。

如果还存在着不同的魔法师，他们具有不同的魔法或不同的等级，这就需要从 CMagician 派生出特殊的魔法师：

```
class CGandalf : CMagician
{
public:
    CGandalf();
    VOID TellWisdoms(HUMAN bHuman);
    SetWeakness(WORD wWeakness);
private:
    char cWisdoms[32][128];
    WORD wWeakness;
};
```

这意味着现在 CMagician 既是派生类又是基类，它从 CAlien 类派生，但同时又作为 CGandalf 类的基类。如此，通过这种方式你就可以根据需要定义出许多继承层次。在 win skeleton++示例的 Fake.cpp 中，仅从 CApp 派生出了一个 CmyFrame。

```
#define STRICT
#include <windows.h>
#include <tchar.h>
#include "App.h"
//-----------------------------------------------------
// Name: class App
// Desc: Main class to run this application. Most functionality is inherited
// from the App base class.
//-----------------------------------------------------
class CMyFrame : public CApp
{
    public:
    LRESULT MsgProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
    CMyFrame(); // constructor
};
```

译注[1]：还包括保护接口。

```
//-----------------------------------------------------------
// Name: WinMain()
// Desc: Entry point to the program. Initializes everything, and goes into a
// message-processing loop. Idle time is used to render the scene.
//-----------------------------------------------------------
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )
{
    CMyFrame d3dApp;
    if( FAILED( d3dApp.Create( hInst ) ) )
        return 0;
    return d3dApp.Run();
}
//-----------------------------------------------------------
// Name: CMyD3DApplication()
// Desc: Constructor
//-----------------------------------------------------------
CMyFrame::CMyFrame()
{
    // Override base class members
    m_strWindowTitle = _T("Fake 4 ++");
    m_dwWidth = 400;
    m_dwHeight = 300;
}
//-----------------------------------------------------------
// Name: MsgProc()
// Desc: message procedure
//-----------------------------------------------------------
LRESULT CMyFrame::MsgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    if (message == WM_KEYDOWN)
    {
        switch (wParam)
        {
            case VK_ESCAPE:
                PostQuitMessage(WM_QUIT);
                break;
            case VK_F1:
            {
                Pause(TRUE);
                MessageBox( hWnd, "Here comes your help text",
                            "Help for FAKE 4++", MB_ICONQUESTION|MB_OK |
                            MB_SYSTEMMODAL );
                Pause(FALSE);
            }
            break;
        }
    }
    return CApp::MsgProc( hWnd, message, wParam, lParam );
}
```

CFrame 使用了自己的 MsgProc()和自己的构造函数，在这里覆写（overwritte）了 3 个基类成员。

为代码共享而设计一个类层次时，应将大多数代码放在基类中。通过这种方法许多类可以重用这些代码，而派生类则用于表示基类的特殊化或者扩展版本。

2. 接口继承

你也可以只继承基类成员函数的名字而非代码，这称为“接口继承”。这种想法背后的思想是什么呢？

考虑那个用于派生出不同种族类的 CAlien 类，通过让所有派生类共享基类的接口，便可将这些派生类当作一个普通的 CAlian 对象来使用。

关于类和继承的讨论到此为止，我不想再进一步深入这方面的内容了。对于我没有提及的话题，你可以参考一下附录 E“游戏程序设计资源”中给出的资源，然后花些时间来学习它们。

B.1.8 虚函数

当通过基类的指针调用虚函数时，会执行该函数的派生类版本[1]，这与普通成员函数的调用行为不同。

指向派生类对象的指针可以赋值给基类指针，并且可以通过这个基类指针来调用虚函数。如果函数是虚的，并且同时在基类和派生类中出现，那么会根据基类指针实际所指的对象调用正确的派生类函数。

在基类成员函数声明前增加关键字 virtual 就可以声明一个虚函数。在通用文件框架 d3dapp.h 头文件中的所有公有接口方法都是虚函数。

```
virtual HRESULT ConfirmDevice(D3DCAPS8*,DWORD,D3DFORMAT) { return S_OK; }
virtual HRESULT OneTimeSceneInit()                        { return S_OK; }
virtual HRESULT InitDeviceObjects()                       { return S_OK; }
virtual HRESULT RestoreDeviceObjects()                    { return S_OK; }
virtual HRESULT FrameMove()                               { return S_OK; }
virtual HRESULT Render()                                  { return S_OK; }
virtual HRESULT InvalidateDeviceObjects()                 { return S_OK; }
virtual HRESULT DeleteDeviceObjects()                     { return S_OK; }
virtual HRESULT FinalCleanup()                            { return S_OK; }
```

在你的派生类中可以改写这些函数。例如：

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    return S_OK;
}
```

在 d3dapp.cpp 中的基类调用了这个方法：

```
HRESULT CD3DApplication::Resize3DEnvironment()
{
    ...
    // Initialize the app's device-dependent objects
```

译注[1]：通过基类的引用调用虚函数时，也会执行函数的派生类版本。

```
    hr = RestoreDeviceObjects();
    if( FAILED(hr) )
        return hr;
    ...
    return S_OK;
}
```

所以在基类方法 Resize3Denvironment()中调用 RestoreDeviceObjects()时，实际上调用的是派生类 CMyD3DApplication 中的 RestoreDeviceObjects()函数。

在基类中声明成员函数时才需要使用 virtual 关键字。

虚函数的使用也称为“动态绑定（dynamic binding）”。这是因为编译器必须在运行时对语句求值，否则它不知道对象的实际类型是什么。

通常你可能会问自己一个问题：“嗯，这很不错，不过这种解析会有多少开销？”，实际上调用一个虚函数仅比调用普通函数稍微多花一点时间而已[1]。

小结一下：虚函数是一种从基类调用派生类函数的优雅方法。动态绑定通常只需要很小的开销；而普通的成员函数调用是通过所谓的“静态绑定（static binding）”的，这种调用一般比虚函数调用更快一点，但没有虚函数的一项优点——多态性。

B.1.9 多态性

不需要指定对象的精确类型就能正确地调用对象成员函数的能力称为“多态性”。这意味着一条语句可以调用许多不同的函数。假定你需要检查所有种族的健康值以便和所有人类的健康值进行比较。

在 C 中不得不使用一个 switch 语句来找出种族的精确类型。代码如下：

```
DWORD ComputeHealth (struct ALIEN *alien)
{
    switch (alien->type)
    {
    case Magician:
        return alien.dwhealth;
        break;
    case Frankenstein:
        return alien.dwhealth;
        break;
    ...
    };
}
```

而在 C++中只需调用以下函数：

```
DWORD ComputeHealth (CALIEN *alien)
{
```

译注[1]：另外虚函数机制应该还需付出空间开销，一般每个具有虚函数的类都会用一张表格来记录其各虚函数的信息，不过这也取决于C++的实现。

```
    return alien->GetHealth();
}
```

C++版本的 ComputeHealth()自动调用了正确的函数，而不需要你来检查指针到底指向什么对象。正如在虚函数一节中所说的，这需要一点的开销，但是可以省下整个 switch 语句。

B.1.10 内联（inline）函数

游戏程序员可以通过内联函数优化性能。关键字 inline 可以将函数的一个副本插入到所有调用它的地方。如果在一个程序有 21 处调用了一个内联函数，那么计算机就会在.exe 文件中插入该函数的 21 份副本。

内联可以消除函数调用时的额外开销。这可以让程序运行得更快，但是由于内联会产生一个函数的多个副本，所以程序也会更大些。内联与用#define 指示定义的宏有些类似，它们的差别在于，编译器会处理内联函数，而宏只是通过简单的文本替换来实现的。此外，内联函数的功能更强大。编译器不仅会执行类型检查，而且内联函数还具有与常规函数一样的行为，并且没有宏可能会产生的副作用。下面是一个示例：

```
#define MAX (A, B) ((A) > (B)? (A) : (B))
inline int max (int a, int b)
{
    if (a > b)
        return a;
    return b;
}
```

你会喜欢这项 C++特征的，不是吗？好，根据经验来看，一般你喜欢的所有事情都需要你投入更多的注意力、做更多的操作。对于 inline 关键字来说也是如此。

编译器只有在其成本/收益分析表示有利可图时才会进行内联。而关键字__forceinline 则不会管成本/收益分析的结果而强制进行内联，这就需要依靠程序员的判断能力了，所以如果使用不当，反而可能会导致性能损失。

__forceinline 除了能让程序员绕过编译器的成本/收益分析来强制内联一个函数外，它仍无法回避由以下原因而导致的无法内联：

- 函数或它的调用者通过/Ob0 编译选项编译（这是调试生成的默认选项）；
- 函数和它的调用者使用不同的异常处理类型（例如在一个函数中使用 C++异常处理，而另一个使用结构化异常处理）；
- 函数具有一个可变参数列表；
- 函数使用内联汇编代码，并且没有通过/Og、/Ox、/O1 或/O2 选项来编译；
- 函数以值的方式返回一个不能展开（unwindable）[1]的对象，并且没有通过/GX、/EHs 或/EHa 选项来编译；

译注[1]：这里“不能展开”是指在发生 C++异常时不能直接展开，而需要调用对象的析构函数。

■ 函数以值传递的方式接收一个拷贝构造对象，并且通过/GX、/EHs 或/EHa 选项来编译；

■ 函数是递归的，并且没有伴随#pragma(inline_recursion, on)编译指示。这个 pragma 指示在默认情况下递归函数就可以内联 8 次递归调用。如果要改变内联深度，可使用#pragma(inline_depth, n)编译指示。

看一下你所用的 Visual C/C++编译器参考手册中关于编译指示的含义。如果编译器不能将一个声明为__forceinline 的函数内联，那么它会产生一个 1 级警告（4714）。

在文件 d3dfont.cpp 中有一个使用内联的典型例子。

```
inline FONT2DVERTEX InitFont2DVertex( const D3DXVECTOR4& p, D3DCOLOR color,
                                      FLOAT tu, FLOAT tv )
{
    FONT2DVERTEX v; v.p = p; v.color = color; v.tu = tu;
    v.tv = tv; return v;
}
```

这是 C++语言第一处增强的地方，继续读下去来看看另外几处。

B.1.11 C++相对与 C 增强的内容

C++相对于 C 引入了一些简单的增强和改进。在这一节中将会介绍下列 C++增强的部分：

■ 缺省函数参数；

■ 更加灵活的变量声明位置；

■ const 关键字；

■ 枚举；

■ 函数重载与操作符重载。

1. 缺省函数参数

一个 C++函数原型可以列出某些参数的缺省值。

```
VOID Pause(BOOL =TRUE);
```

可以将这个函数的声明放在头文件中。如果你提供了自己的参数，编译器就会用它们替换声明中的缺省值。

来看一个稍复杂的示例。

```
VOID DoSomething(DWORD w = 2, FLOAT f = 1.5);
```

你可以通过以下方式来调用它：

```
DoSomething( 2, 2.5 );
```

在这个示例中，第一个值（w）是 2，第二个值（f）是 2.5。另外一种调用方式为：

```
DoSomething(3);
```

第一个值（w）是 3，第二个值（f）是 1.5。

2. 变量声明的位置

这是 C++中的一个很大的改进，我想你可能已经知道了。C 要求必须在块的开始部分声明所有变量，而 C++则允许你在代码的任何地方声明变量。在 C 中一个函数的伪代码可能是：

```
VOID Myfunction ()
{
    int z;
    AnotherFunction(d);
    doSomethingWith(z);
}
```

在 C++中同样的函数，则为：

```
VOID Myfunction ()
{
    AnotherFunction(d);
    int z;
    doSomethingWith(z);
}
```

说它是一个很大的改进，这是因为将变量与使用它的代码互相靠近可使代码更具可读性。在 C++中另一个不错的常用表达式是：

```
for (int z = 0; z < 5; z++)
{
    // do something
}
```

但是不允许使用下列语句：

```
if (int z == 0)
while (int j == 0)
```

因为，它们毫无意义！

3. const 变量

就经验来说，对于任何使用常量表达式或#define 表达式的地方都可以用一条 const 常量定义语句来代替。

在 C 和 C++中对 const 修饰符的使用有一点区别。在 C 中可以初始化一个用 const 修饰的变量，并在程序中使用它的值；而在 C++中 const 修饰符可以用于指针、引用、函数等多种不同的地方。

大多数 C 程序使用#define 预处理指示定义常量：

```
#define MAX_HEIGHT 800
```

#define 指示的缺点是不能产生类型安全的常量，这是因为它只是一个预处理器指示，而非编译器语句。当使用#define 产生一个常量时，这个常量是没有数据类型的，所以无论是在设计时、运行时、还是调试时，编译器都无法确定如何在这个常量上施行正确的操作。

相反，如果在 C++中定义一个 const，那么这就执行了一次真正的编译器语句，例如：

```
const int MAX_HEIGHT=800;
```

也可以将 const 用于指针声明上：

```
char * const ptr = cString;
```

这样定义可以修改指针所指向的值：

```
*ptr = 'a'; // ok
```

但是不能修改指针本身（常量值）：

```
ptr = cString2; // error
```

另外一种情况，如果指针指向一个常数：

```
const char *ptr = cString;
ptr = cString2;
*ptr = 'a';
```

在第二行中改变指针本身是可以的，但是改变它所指的值则会收到一条错误信息。这样还可以通过 const 修饰符来防止一个函数修改其参数：

```
int readonly (const struct NODE *nodeptr);
```

如上所述，const 是一个有效的修饰符，可以使你的程序更具类型安全，所以应尽量地使用它。

4. 枚举

枚举是一种整型数据类型，用于定义一组有名字的常量清单。枚举中的每个元素都有一个整型数值，在默认情况下，后一个元素的值总比前一个元素大。如果没有指定其他值，那么第一个元素的值总为 0。下面的示例说明了在 C++程序中如何定义一个枚举：

```
enum color {yellow, orange, red, green }; // values 0, 1, 2, 4
enum day {Monday, Tuesday, Wednesday, Thursday = 8, Friday}; // values 0, 1, 2, 8, 9
```

可以将一个枚举转换成整数，但是如果将一个整数转换成枚举的话，则必须使用强制类型转换。将一个整数强制转换成枚举是不安全的，因为如果整数的值超过了枚举的范围，或者枚举存在重复的值，那么转换的结果就是未定义的。

5. 函数重载与操作符重载

C++中有两种重载：函数重载和操作符重载。这是 C++语言中主要的特征，它们都比较有用。

（1）函数重载

为了实现函数重载，就需要使用两个或更多的函数，它们具有相同的名字，但不同的参数列表。编译器通过参数匹配来决定调用哪一个版本，它会比较传递给函数的参数个数和参数类型。

在 C++中经常使用函数重载，因为它没有任何运行时的开销，以及其他的特殊开销。

函数重载的一种特殊情况就是重载构造函数。在 d3dvec.inl 中可以找到这样一段代码：

```
inline
_D3DVECTOR::_D3DVECTOR(D3DVALUE f)
{
    x = y = z = f;
}
inline
_D3DVECTOR::_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z)
{
    x = _x; y = _y; z = _z;
}
inline
_D3DVECTOR::_D3DVECTOR(const D3DVALUE f[3])
{
x = f[0]; y = f[1]; z = f[2];
}
```

在 d3dmath.inl 中，D3DXVECTOR3 的初始化是典型的函数重载：

```
D3DXINLINE
D3DXVECTOR3::D3DXVECTOR3( CONST FLOAT *pf )
{
#ifdef D3DX_DEBUG
    if(!pf)
        return;
#endif
    x = pf[0];
    y = pf[1];
    z = pf[2];
}
D3DXINLINE
D3DXVECTOR3::D3DXVECTOR3( CONST D3DVECTOR& v )
{
    x = v.x;
    y = v.y;
    z = v.z;
}
D3DXINLINE
D3DXVECTOR3::D3DXVECTOR3( FLOAT fx, FLOAT fy, FLOAT fz )
{
    x = fx;
```

```
    y = fy;
    z = fz;
}
```

这些重载函数使用不同的参数完成相同的任务，即给 x、y、z 三个变量赋值。

整个向量/数学库就是使用函数和操作符重载构建的。通过函数重载，就可以让成员函数使用相同的名字来执行不同但相似的任务。甚至还可以让整组函数使用同一个名字。

（2）操作符重载

当使用类对象时，可以通过重载操作符来重新定义它们的行为，例如加法操作符（+）、减法操作符（-）、赋值操作符（=）、增量和减量操作符（++和 --）等等。这些自定义的操作符实际上就是经过装扮的函数。

在 C++中重载一个操作符需要定义一个操作符重载函数（通常是一个成员函数）。重载操作符的函数应包含关键字 operator。

操作符重载的语法为：

```
ReturnType operator @ (arguments)
{
    // code
} // global function
ReturnType classname::operator @ (arguments)
{
    // code
} // class function
```

上面的@标记表示一个操作符。使用的语法如下：

```
y = x @ y
```

也可以这样使用：

```
z = operator @ (x, y)
```

或者，这样：

```
z = x.operator @(y);
```

一个实际应用中的示例如：

```
D3DXINLINE D3DXVECTOR3&
D3DXVECTOR3::operator += ( CONST D3DXVECTOR3& v )
{
    x += v.x;
    y += v.y;
    z += v.z;
return *this;
}
```

以下语句将 D3DXVECTOR3 结构 v 加到+=操作符左侧的 D3DVECTOR3 结构 z 中：

```
z += v;
```

另外还有一个示例是：

```
D3DXINLINE D3DXVECTOR3
operator * ( FLOAT f, CONST struct D3DXVECTOR3& v )
{
    return D3DXVECTOR3(f * v.x, f * v.y, f * v.z);
}
```

这个操作符重载的语法原型是：

```
z = operator * (f, v);
```

在这个例子中，返回值 D3DXVECTOR3(f * v.x, f * v.y, f * v.z)是一个临时结构。对此，编译器可能会通过一种称为“返回值优化（returned value optimization）”的优化技术来避免这个临时结构的产生及其所带来的开销。编译器可以直接在接受返回值的对象上创建返回对象[1]。

好了，我想你现在应该是有点概念了吧。可以再看一下 d3dxmath.inl 文件，里面还有更多的示例。

临近这份简短 C++入门的尾声了。我已尝试着将大多数对于游戏程序员而言重要的 C++特征集中在一起了，然而还是留有很多特征没有说到，你必须自己去看一下引用和多重继承的用法。所以，继续学习吧！

B.2 附加资源

有一本HTML/PDF/PalmPilot格式的免费C++书籍，位于http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html。

另外还有一些其他的 C++在线教程，分别位于 http://www.intap.net/~drw/cpp/index.htm、http://www.cplusplus.com/doc/tutorial、http://devcentral.iftech.com/articles/C++/default.php。

译注[1]：这里的意思是指，如果调用代码是将函数的返回值赋予一个对象的话，那么正常情况中，首先函数会构建一个临时对象（暂称为 T）并返回它，然后再通过 T 赋值给调用代码中用于接受返回值的对象（暂称为 R）。可见其中 T 的产生是没有必要的，因此返回值优化就将 T 直接构建在 R 上。

附录 C

数 学 入 门

这章附录提供了一些补充信息来加强你对本书第一、二部分中所涉及到的数学和 3D 概念的理解。我首先将告诉你如何旋转 3D 空间中的点，并以此作为学习 3D 中数学知识的开始。

C.1 3D 中的点（Point）

首先设想一下在 2D 空间中旋转一个点。两根轴构成了一个平面，就像一张纸一样，这个平面称为"xy 平面"，因为它是由两根主轴定义出来的（见图 C.1）。

为了将一个点按逆时针旋转 θ 弧度，可使用下面的公式：

$$x_{new} = x * \cos(\theta) - y * \sin(\theta)$$
$$y_{new} = x * \sin(\theta) + y * \cos(\theta)$$

好，现在让我们转到 3D，只是再加第三根轴而已，称之为 z 轴；我想这对于一根轴来说是一个不错的名字。2D 和 3D 之间的不同之处在于 z 轴引入了两个新的主平面，"xz 平面"和"yz 平面"。现在的要点就是，你可以将一个完整的 3D 旋转分解为分别在每个平面中旋转指定的角度。图 C.2 描述了与图 C.1 所示相同的旋转，只是这次是在 3D 中进行的。

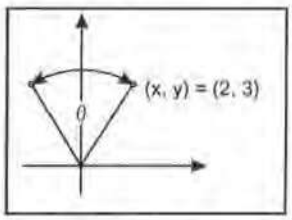


图 C.1 二维坐标系

在 3D 系统中将一个点 w(x, y, z)按逆时针旋转 θ 弧度，可使用下面的公式：

$$x_{new} = x * \cos(\theta) - y * \sin(\theta)$$
$$y_{new} = x * \sin(\theta) + y * \cos(\theta)$$
$$z_{new} = z$$

这几乎与前面的公式相同，但这里多了一个 z 值。

点的计算结果 x_{new}、y_{new}、z_{new} 可用于下一次旋转的初始值。下一次旋转可能是绕 x 轴进行的（见图 C.3）。

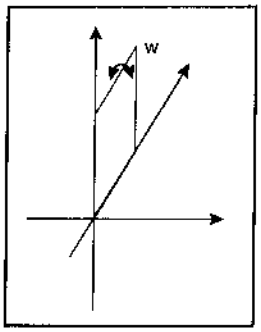


图 C.2　三维坐标系

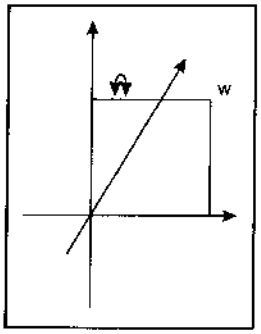


图 C.3　绕 x 轴旋转

将一个点 w(x, y, z)绕 x 轴按逆时针旋转 θ 弧度，你可使用下面的公式：

```
xnew = x
ynew = y * cos(θ) - z * sin(θ)
znew = y * sin(θ) + z * cos(θ)
```

现在再看一下绕 y 轴的旋转（见图 C.4）。

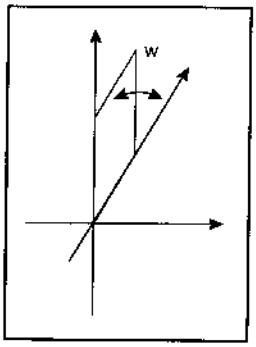


图 C.4　绕 y 轴旋转

将一个点 w(x, y, z)绕 y 轴按逆时针旋转 θ 弧度，你可使用下面的公式：

```
xnew = x * cos(θ) - z * sin(θ)
ynew = y
znew = x * sin(θ) + z * cos(θ)
```

为了在 3D 空间中旋转一个点 w(x, y, z)，你需要：

（1）在 xy 平面旋转它，并在 x_{new}、y_{new}、z_{new} 中保存结果；

（2）再在 yz 平面旋转上一次的结果点，并保存在 x_2、y_2、z_2 中保存结果；

（3）再在 zx 平面旋转上一次的结果点，并保存在 x_{new}、y_{new}、z_{new} 中保存结果；

伪代码片断如下：

```
xnew = x * cos(θ) - y * sin(θ)
ynew = x * sin(θ) + y * cos(θ)
znew = z
x2 = xnew
y2 = ynew * cos(θ) - znew * sin(θ)
z2 = ynew * sin(θ) + znew * cos(θ)
xnew = x2 * cos(θ) - z2 * sin(θ)
ynew = y2
znew = x2 * sin(θ) + z2 * cos(θ)
```

对旋转一个3D点的小结就是：为了在3D空间中旋转一个点，你必须依次绕三根轴作旋转。

在3D空间中平移一个点相对简单得多。朝一个方向移动一个点时，只需将该点加上3个偏移量（表示方向），3个偏移量则分别对应x、y、z。比如，将点(2, 3, 0)平移到(8, 7, 0)，只需要将(6, 4, 0)加到该点的对应分量上即可。

C.2 向量（Vector）

向量是一个数学实体，用于描述一个方向和一个大小（比如速度）。向量与标量（scalar）是不同的，标量只描述大小。比如，“温度32摄氏度”就是一种典型的标量值，这条信息告诉你的一切只是温度在那个点上。所以很难用它来描述一架飞机的飞行方向以及它飞得有多快，而这应该是由向量来做的事情。

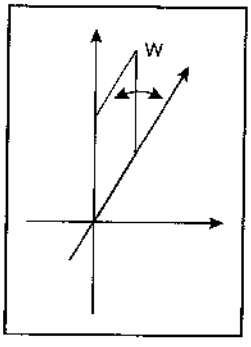
你需要来区分一下“一般用途的向量（general-purpose vector）”（或称“受限向量”）和在游戏中使用的所谓“自由向量”。

C.2.1 受限向量（Bound Vector）

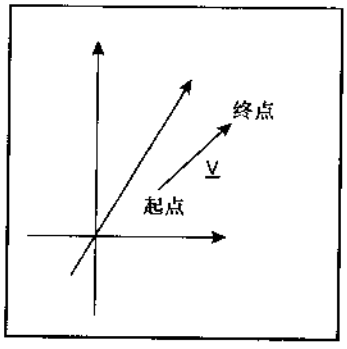
一个受限向量V具有如下属性，如图C.5所示：

- 一个起点；
- 一个终点；
- 一个方向；
- 一个大小。

一个受限向量在坐标系中看上去可能如图C.6所示。



图C.5 受限向量



图C.6 一个在坐标系中的受限向量

一个受限向量由两套坐标组成。在这两套坐标所表示的点之间画一条直线，便可看到这

些向量的方向。两点之间的距离表示大小，第一套坐标称为“起点”，第二套坐标称为“终点”。

C.2.2 自由向量（Free Vector）

对于游戏程序员而言，向量是没有起点的。如果将所有的点都以相同的速度朝一个方向移动，并给每个点各自使用一个向量的话，将会很低效。其实使用一个向量（自由向量）就足以描述所有点的速度和方向了。因此，你不必为一个自由向量指定起点。

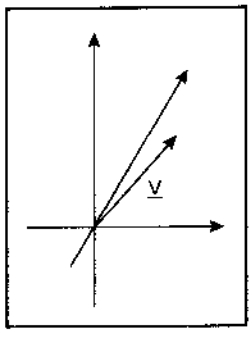


图 C.7 一个坐标系统中的自由向量

根据定义，所有自由向量的起点都在原点，因此描述一个自由向量只需要 3 个数字就可以了（见图 C.7）。

呃，你可能会说“那听上去像是 3D 空间中的一个点”。因此我觉得有必要在这里说明一下，在 3D 中一个向量和一个点有什么区别。一个点仅表示一个位置；而一个自由向量（如图 C.7 所示）还表示了大小和方向，比如，向量(2,3,2)的大小就是从原点(0, 0, 0)到终点(2, 3, 2)的距离。

如果你只需要终点，那么它就是一个 3D 点。如果你还需要大小和方向，那么它就是一个向量。你也可以说，一个自由向量描述了一个 3D 点，但那是一种更为哲学化的观点。另外，一个 3D 点也可以被描述为一个长度为 0 的零向量（null vector）。

和实数一样，可以在向量上使用算术操作，例如可以对它进行加法、减法、乘法。

1. 向量加法：U + V

为了将两个向量相加，就需要将它们的分量分别相加：

```
U (1, 3, 2) + V (2, 1, 2)
  = Z (1 + 2, 3 + 1, 2 + 2)
  = Z (3, 4, 4)
```

在 2D 空间中理解这个加法可能会更容易，因此为了尽可能简单，我就省略了 z 值。

如果你想以一种更形象的方式来做向量的加法（如图 C.8 所示），则可按以下步骤进行：先平移第二个向量 V，使其起点与向量 U 的终点重合；作线段连接 U 的起点和移动后 V 的终点，该线段即为 U + V 的和。

向量加法是很有用的，比如可以用它来表示一艘船的行径。试想在驾驶一艘船朝一个方向前进的过程中，就可能会由于一些自然因素造成行驶方向的略微偏移[译注]。加上再多的向量也都是采用同一种方法。接下去还有另一种的重要算术就是减法。

译注：意思是，为了模拟船只的行径受到诸如风力、潮流之类的影响，可以使用一个向量来表示驾驶的方向，再用另一个向量来表示诸如风力之类的自然因素，那么两向量相加的结果即为最终的实际行驶方向。

2. 向量减法：U-V

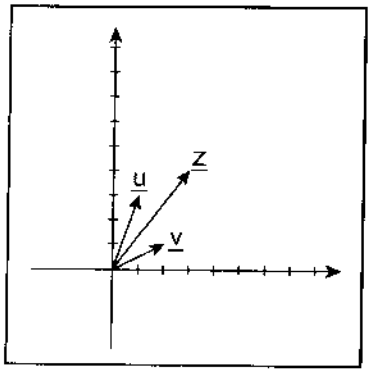


图 C.8　向量加法

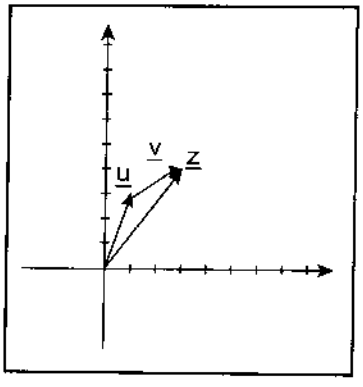


图 C.9　图形化的向量加法

减法是通过将两个向量中对应的分量相减来完成的。向量减法是向量加法的逆操作。例如：

```
U (1, 3, 2) - V (2, 1, 2)
  = Z (1 - 2, 3 - 1, 2 - 2)
  = Z (-1, 2, 0)
```

为了简单起见，这里只用到了 x 值和 y 值。这个方程式如图 C.10 所示。

可以通过一种更形象的方式来理解两个向量的减法：将一个向量 V 乘以–1，再加上 U（见图 C.11）。

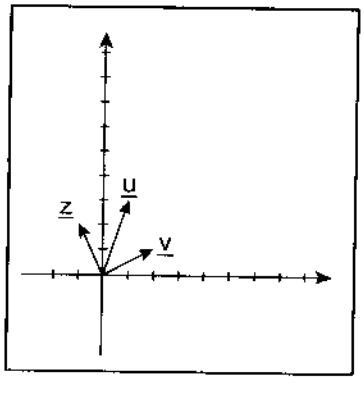


图 C.10　向量减法

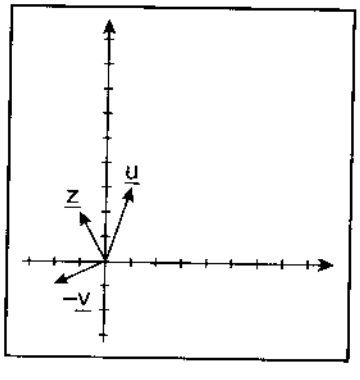


图 C.11　图形化向量减法

另外，向量减法可以抵消向量加法的作用。

3. 向量乘法

有 3 种向量乘法：

- 标量积（scalar-product）；
- 点积（dot-product）；
- 叉积（cross-product）。

（1）标量积

将一个向量乘以一个标量（实数）将改变向量的大小，也就是说将改变向量的起点到终点的距离。

```
U (1, 3, 2) * 2
  = Z (1 * 2, 3 * 2, 2 * 2)
  = Z (2, 6, 4)
```

简化图如图 C.12 所示。

Z 为 U 的 2 倍大小，标量积的结果还是一个向量。

（2）点积

两个向量的点积通过点符号“•”表示，点积的结果是一个实数值。例如：

```
U • V
```

假定你想计算两个 2D 向量 U 和 V 之间的夹角（见图 C.13）。

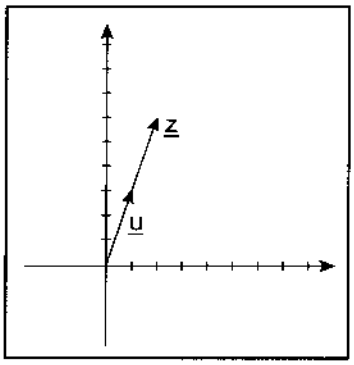


图 C.12　标量积

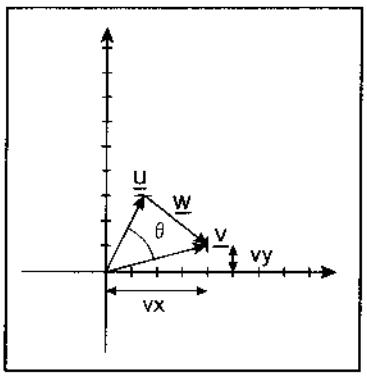


图 C.13　点积的角度

如果由 U 和 x 轴形成的夹角是α，由 V 和 x 轴形成的夹角是 β，那么可以说：

$$\cos_\alpha = u_x / ||U||$$
$$\sin_\alpha = u_y / ||U||$$
$$\cos_\beta = v_x / ||V||$$
$$\sin_\beta = v_y / ||V||$$

可以通过勾股定理计算出向量的大小，这条定理指出，一个直角三角形两直角边长度的平方和等于斜边长度的平方。用数学符号表示就是 $a^2+b^2=c^2$。或者，设直角边为 x 和 y，斜边为 m，那么该定理可表示为 $x^2+y^2=m^2$。

所以向量的长度（大小）就可以通过下面式子得到：

```
||V|| = sqrt (x² + y²)
```

U 和 V 之间的夹角就是 α 和 β 的差：

$$\theta = \alpha - \beta$$

余弦定理指出对于一个由向量 U、V、W 构成的三角形，||U||、||V||、||W||表示 3 条边的长度，则可得：

```
U·V = ||U|| * ||V|| * cos(α - β)
    = ||U|| * ||V|| * (cos α * cos β + sin α * sin β)
    = ||U|| * ||V|| * (ux / ||U|| * vx / ||V|| + uy / ||U|| * vy / ||V||)
    = ||U|| * ||V|| * (ux * vx + uy * vy) / (||U|| * ||V||)
    = ux * vx + uy * vy
```

通过解这个方程，便可求出两个向量之间的夹角：

```
cos(θ) = U·V / ||U|| * ||V||
U (1.5, 3)
V (4, .)
||U||    = sqrt(ux²+ uy²)
         - sqrt (2.25 + 9)
         = 3.4
||V||    = sqrt(vx² + vy²)
         - sqrt (16 + 1)
         = 4.12
U·V      = ux * vx + uy * vy
         - 1.5 * 4 + 3 * 1
         = 9
cos (θ) = 9 / 3.4 * 4.12
cos (θ) = 0.64
θ - 49.99°
```

现在，让我们来考虑一下 3D 向量。对于一个由向量 U、V、W 构成的三角形，其中||U||、||V||、||W||表示三条边的长度，则可得：

```
W = U - V
U - V = U² + V² - 2 * U · V
```

第二个式子可以变换为：

```
2 * U·V = U² + V² - (U - V)
```

可通过以下式子来求各向量的大小：

```
||U||² = ux² + uy² + uz²
||V||² - vx² + vy² + vz²
||U - V||² = (ux - vx)² + (uy - vy)² + (uz - vz)²
```

让我们根据点积的定义解上面的方程，有：

```
2 * U·V  = ux² + uy² + uz² + vx² + vy² + vz² - (ux - vx)² - (uy - vy)² - (uz - vz)²
```

```
= ux² + uy² + uz² + vx² + vy² + vz² - ux² - vx² + 2 * ux * vx -
  uy² - vy² + 2 * ux * vy - uz2 - vz² + 2 * uz * vz
= 2 * ux * vx + 2 * uy * vy + 2 * uz * vz
= 2 * (ux * vx + uy * vy + uz * vz)
```

得出：

```
U·V = ux * vx + uy * vy + uz * vz
```

为了求出这些向量之间的夹角，可通过：

```
cos(θ) = U·V / ||U|| * ||V||
U (1.5, 3, 4)
V (4, 1 ,2)
||U||    = sqrt(ux² + uy² + uz²)
         = sqrt (2.25 + 9 + 16)
         = 4
||V||    = sqrt(vx² + vy² + vz²)
         = sqrt (16 + 1 + 2)
         = 4.4
U·V      = ux * vx + uy * vy + uz * vz
         = 1.5 * 4 + 3 * 1 + 4 * 2
         = 17
cos (θ) = 17 / 4 * 4.4
cos (θ) = 0.96
θ = 15.00°
```

点积可用于求两个向量之间的夹角，此外，它还可以用于对多边形进行着色以及产生许多其他的效果。

叉积

对两个向量求叉积可以得到一个新的向量，它垂直于由原来两个向量所构成的平面，所以这个新向量又称为该平面的“法向量”，它指向由另外两个向量所构成的平面的方向[1]。

使用叉积可以确定多边形到底面向哪里，这需要使用多边形的两条边来生成一个法向量，可以对任意表面内的两个向量求叉积来生成这个法向量。不过和点积不同，叉积不满足交换律：

```
U × V = -(V × U)
```

U 和 V 叉积的大小(||U×V||)可以通过||U||*||V||*sin(θ)得到，结果向量的方向同时垂直于 U 和 V（见图 C.14）。

如果把叉积的结果除以两个向量的大小，就可以得到夹角的正弦值：

```
sin (θ) = U × V / ||U|| * ||V||
```

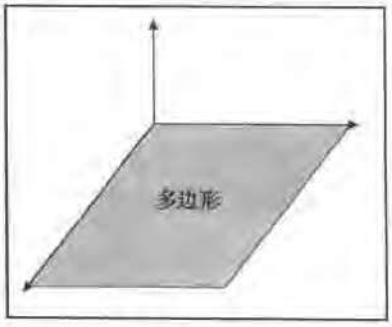


图 C.14　叉积

译注[1]：这个法向量与原来的两个向量方向符合右手法则，即，如果 W=U×V，那么右手除拇指外的四指并拢指向 U 所指的方向，掌心朝向 V 所指的方向，此时摊开的拇指即指向 W 的方向。

不过，一般不需要以这种方式来求两个向量的夹角，因为使用点积做会更快一些。

C.2.3 单位向量（Unit Vector）

另一类重要的向量是单位向量。一个单位向量的大小是 1。如果使自由向量成为一个单位向量，那么就可以得到任意方向上大小为 1 的向量。所有可能的自由单位向量的终点形成了一个球体。例如对一个照相机来说，如果将原点作为起点，那么通过改变向量值就可以来旋转照相机了。

如何计算一个单位向量？或者说，如何对向量做归一化（normalize）呢？为了计算一个单位向量，需将向量除以其大小（或者说其长度）：

```
N = V / ||V||
```

根据勾股定理来使用 3D 版的距离公式可以计算出这些向量的大小：

$$x^2 + y^2 + z^2 = m^2$$

向量的长度可以通过下面式子求出：

$$||V|| = \text{sqrt}\ (x^2 + y^2 + z^2)$$

最终的长度是平方和的平方根（勾股定理）。在数学上用特定的符号来表示一个向量的大小，即对一个大写字母左右两边各括上两条竖线，如‖V‖。完成单位向量计算的函数可能是这样的：

```
D3DXVECTOR3* D3DXVec3Normalize(D3DXVECTOR3* v1, const D3DXVECTOR3* v2)
{
    *v1 = *v2/(float) sqrt(v2->x * v2->x + v2->y * v2->y + v2->z * v2->z);
    return (v1);
}
```

它将向量除以它的大小，大小可通过勾股定理的平方根求得。

C.3 矩阵

矩阵是由数字组成的矩形数组。这些数字可以表示任意东西，例如方程或常数。此外，你也可以把矩阵看成一组行向量或一组列向量。通常我们将矩阵作为常规 3D 变换的一种压缩形式，所以在本书中关于矩阵的讨论都集中于：将矩阵用于表示世界变换、照相机变换、投影变换等变换。例如，一个 4×4 世界矩阵包含了 4 个向量，分别表示了物体在世界空间中的 x、y、z 单位轴向量的坐标，以及物体在世界空间中的位置坐标。

```
ux uy uz 0
```

```
vx vy vz 0
wx wy wz 0
tx ty tz 1
```

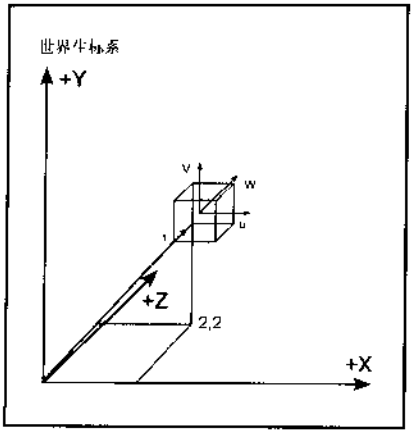


图 C.15　u、v、w 向量

u、v、w 向量表示所谓的“刚性体（rigid body）”。这个矩阵定义了从对象空间[1]到世界空间的映射。请看一下图 C.15 中的图形化表示。

为了描述立方体的位置，矩阵可以是：

```
1 0 0 0
0 1 0 0
0 0 1 0
2 2 2 1
```

这个立方体按照坐标系统的方向放置。第一行包含了物体的局部 x 轴在世界空间中的坐标，第二行包含了物体的局部 y 轴在世界空间中的坐标，第三行包含物体的局部 z 轴在世界空间中的坐标。这些向量都是单位向量，大小都为 1。最后一行包含了物体原点在世界空间中的坐标，可以通过该原点对立方体进行平移。

单位矩阵是一种主对角线上全部为 1 的特殊矩阵：

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

单位矩阵表示一组与世界坐标轴对齐的局部坐标轴。局部 x 轴的 x、y、z 坐标分量在世界空间中分别为 1、0、0，其原点向量为（0,0,0）。因此，局部模型的 x 轴直接对应于世界空间中的 x 轴，局部模型的 y 轴、z 轴也是如此，所以这是一个不会产生任何影响的矩阵。如果要将一个对象在模型空间中的位置直接等同于它在世界空间中的位置，只需要把世界变换矩阵设置为单位矩阵就可以了。可以像下面这样在 Direct3D 中设置该矩阵和所有其他矩阵：

```
D3DMATRIX mat;
mat._11 = 1.0f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 = 1.0f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 0.0f; mat._42 = 0.0f; mat._43 = 0.0f; mat._44 = 1.0f;
```

C.3.1　矩阵与向量的乘法

变换通常是通过一个 4×4 矩阵乘法操作来完成的。变换引擎用一个 4×4 矩阵乘以一个表示 3D 数据的向量（例如一个顶点、一个法向量），结果就是变换后的向量。通过标准的线性

[1] 译注：又称“局部空间”或“模型空间”。

代数可以完成该操作：

```
变换矩阵        原向量                                    变换后的向量
a b c d         x          ax + by + cy + dw       =      x'
e f g h   ×     y     =    ex + fy + gz + hw       =      y'
i j k l         z          ix + jy + kz + lw       =      z'
m n o p         w          mx + ny + oz + pw       =      w'
```

在变换一个向量前，必须先构造一个变换矩阵，这个矩阵含有将向量转换到新坐标系的数据。如果要进行一系列变换的话，则要为每个动作（缩放、旋转、平移）各创建一个临时矩阵，然后对向量分别施以这些变换。另外，还可以将这些矩阵乘到一起，乘积结果即表示了这些动作组合的效果（矩阵串联）。最终这个乘积矩阵称为“变换矩阵”，你可以用它来对一个乃至上百万个向量进行变换。重用这个矩阵可以很有效地节省下运算时间。

C.3.2 矩阵加法和减法

将两个矩阵相加非常简单，只需要将第一个矩阵的第一列第一个数字加上第二个矩阵的第一列第一个数字，依此类推。减法操作也是类似的。

```
1  2  3  4          17 18 19 20   18 20 22 24
5  6  7  8     +    21 22 23 24 =     26 28 30 32
9 10 11 12          25 26 27 28   34 36 38 40
13 14 15 16   29 30 31 32   42 44 46 48
```

C.3.3 矩阵乘法

为了解释矩阵乘法，最好看一个例子。

```
1  2  3  4          17 18 19 20
5  6  7  8          21 22 23 24
9 10 11 12   *25 26 27 28
13 14 15 16   29 30 31 32
```

方法是将第一个矩阵的第一行乘以第二个矩阵的第一列，然后把结果加起来。接着再将第一个矩阵的第一行与第二个矩阵的第二列、第三列、第四列依次以相同的方式计算。

```
1 * 17 + 2 * 21 + 3 * 25 + 4 * 29,  1 * 18 + 2 * 22 + 3 * 26 + 4 * 30,  等等
```

然后再将第一个矩阵的第二行乘第二个矩阵的第一列，把结果加起来，再将第一个矩阵的第二行乘以第二个矩阵的第二列，如此一直计算下去。最后可以得到 4×4 矩阵的 16 个值。

C.3.4 平移矩阵

为了在 3D 空间中移动一个物体，通常需要设置物体世界矩阵中的第 4 行，或者把世界矩阵乘以一个平移矩阵。

```
1 0 0 0
0 1 0 0
0 0 1 0
x y z 1
```

平移矩阵保存的是对象需要移动距离的数据，而不是坐标。模型对象本身保存着待变换的 x、y、z 位置值。

C.3.5 缩放矩阵

这种矩阵可以缩放坐标，所以可用于使一个物体变大或变小。

```
x 0 0 0
0 y 0 0
0 0 z 0
0 0 0 1
```

你也可以直接设置 mat._11、mat._22、mat._33 这 3 个值来达到缩放的目的。

C.3.6 旋转矩阵

有三种旋转矩阵，分别绕 y、x、z 轴旋转。

1. 绕 y 轴旋转

Direct3D 函数 D3DXMatrixRotationY()可以构建一个绕 y 轴旋转的矩阵。我没有源代码，但它应该如以下代码所示：

```
VOID D3DXMatrixRotationY( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._11 = cosf( fRads );
    mat._13 = -sinf( fRads );
    mat._31 = sinf( fRads );
    mat._33 = cosf( fRads );
}
```

这等价于矩阵：

```
cos fRads     0    -sin fRads        0
0             1    0                 0
sin fRads     0    cos fRads         0
0             0    0                 1
```

2. 绕 x 轴旋转

D3DXMatrixRotationX()把对象绕 x 轴旋转，这里的 fRads 等于要旋转的弧度值。它的源代码应如下所示：

```
VOID D3DXMatrixRotationX( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._22 = cosf( fRads );
    mat._23 = sinf( fRads );
    mat._32 = -sinf( fRads );
    mat._33 = cosf( fRads );
}
```

这等价于矩阵：

```
1  0                0              0
0  cos fRads        sin fRads      0
0  -sin fRads       cos fRads      0
0  0                0              1
```

3. 绕 z 轴旋转

D3DXMatrixRotationZ()把对象绕 z 轴旋转，这里的 fRads 等于要旋转的弧度值。它的源代码应如下所示：

```
VOID D3DXMatrixRotationZ( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._11 = cosf( fRads );
    mat._12 = sinf( fRads );
    mat._21 = -sinf( fRads );
    mat._22 = cosf( fRads );
}
```

这等价于矩阵：

```
cos fRads     sin fRads     0 0
-sin fRads    cos fRads     0 0
0             0             1 0
0             0             0 1
```

这些矩阵是 Direct3D 会用到的。需要记住的非常重要一点，矩阵乘法是不可交换的。这意味着在大多数情况下，([a] * [b])不等于([b] * [a])。最终的变换公式为：

```
|W| = |M| * |T| * |X| * |Y| * |Z|
```

其中，M 表示模型的矩阵，T 表示平移矩阵，X、Y、Z 表示旋转矩阵。Direct3D 会在内部使用合并结果的矩阵 W。

C.4 小结

本附录给出的内容足以让你能够看懂本书所涉及到的数学知识。然而，要成为高级游戏程序员的话，还有更多的东西需要学习。数学是解决大多数与游戏编程相关问题的关键，具有良好数学背景对于一个优秀的游戏程序员来说是必不可少的。

附 录 D

使用 D3DXCreateTextureFrom FileEx()创建纹理

本书中的大部分例子都是使用 D3DUtil_CreateTexture()函数创建纹理的，这个函数位于 d3dutils 库文件中。其实它只是对一个复杂函数 D3DCreateTextureFromFileEx()的简单抽象而已。这个函数太复杂了，以至于你可能会迷失在它的参数中，而 D3DUtil_CreateTexture()则赋予了这些参数一些最常用的值，由此便可以在更高的层次上为 Direct3D 设备创建纹理。本附录将解释 D3DXCreateTextureFromFileEx()的参数。

```
HRESULT D3DXCreateTextureFromFileEx(
    LPDIRECT3DDEVICE9 pDevice,      // pointer to device interface
    LPCTSTR pSrcFile,               // pointer to string with filename
    UINT Width,                     // width in pixels
    UINT Height,                    // height in pixels
    UINT MipLevels,                 // number of mip levels requested
    DWORD Usage,                    // usage of texture
    D3DFORMAT Format,               // describes requested pixel format
    D3DPOOL Pool,                   // memory class
    DWORD Filter,                   // how the texture is filtered
    DWORD MipFilter,                // how the texture is mip filtered
    D3DCOLOR ColorKey,              // color to replace with transparent black
    D3DXIMAGE_INFO *pSrcInfo,       // info structure regarding texture
    PALETTEENTRY *pPalette,         // 256 color palette
    LPDIRECT3DTEXTURE9 *ppTexture   // pointer to texture interface
);
```

pDevice 表示与纹理关联的设备。pSrcFile 保存一个 ANSI 或 UNICODE 的字符串[1]，表示要创建纹理的文件名。还可以指定纹理的宽度和高度，如果指定 0 或 D3DX_DEFAULT（定

译注[1]：LPCTSTR 其实是“const TCHAR *”的 typedef，TCHAR 则根据是否定义了 UNICODE 预处理器标志被定义成不同的字符类型：未定义 UNICODE 时为 char，定义了 UNICODE 时为 wchar_t。

义在 d3dutil.cpp 中），则会从源文件中获取尺寸信息。

这个方法还可以为你生成 mipmap。一组 mipmap 是由许多内容相同但分辨率不同的纹理组成的，尺寸上相差 2 的指数倍。那么为什么要对同一张纹理使用不同的分辨率呢？当物体对于玩家来说非常远的时候，玩家是看不到纹理上的细节的，因此就不需要在物体上映射最高分辨率的那张纹理。因此出于性能考虑，更适宜的映射操作应该是：当物体在最远处时映射最小的纹理；在中等距离处时映射中等大小的纹理；在最近处时映射最大、最细致的纹理。Direct3D 能自动生成和使用这些纹理，你只需要生成 mipmap 即可。可通过 MipLevels 参数来生成 mipmap，将其指定为 0 或 D3DX_DEFAULT 时，就可以从源文件创建出一个完整的 mipmap 链；如果指定为其他数字的话，那么就会创建出指定数量的 mipmap。在 d3dutil.cpp 中使用了默认值 D3DX_DEFAULT，从而便可以在你的引擎中创建并使用整个 mipmap 链。

Usage 参数告诉 Direct3D 引擎是否将纹理表面用作渲染目标。有许多给人印象深刻的效果都是通过在一个纹理表面上进行渲染来实现的。在 DirectX SDK 中一个最常见的例子便是在一个旋转的立方体上渲染一段视频。通常，你可以将这个参数指定为 0 或 D3DUSAGE_RENDERTARGET。如果需要渲染到纹理表面，必须先使用 CheckDeviceFormat() 检查显卡的能力，然后把 Pool 参数设为默认的 D3DPOOL_DEFAULT，接着再用 SetRenderTarget()来设置渲染目标。而 d3dutil.cpp 中的 D3DUtils_CreateTexture()函数将该参数设为 0。

Format 参数定义了希望使用的表面格式，这样就可以返回与源文件不同的格式了。如果设为未知格式（D3DFMT_UNKNOWN），则从源文件中获取格式；也可以在 D3DXCreateTextureFromFileEx()中直接给出期望的格式。如果不准备使用特殊的格式，那么可将该参数设为 D3DFMT_UNKNOWN，即使用源文件的格式。

Pool 参数定义了纹理驻留的存储器类型。通用文件 d3dutil.cpp 中使用了 D3DPOOL_MANAGED，表示使用最便利的托管存储器类型。

通过 Filter 可以控制如何过滤图像（也就是说如何从源图像中采样像素，这将直接影响到图像的质量）。当纹理大小与源文件大小不同时，这就很有用了。如果纹理是源文件大小的一半，那么可以将该参数指定为 D3DX_FILTER_BOX，这样便会将源文件中 2×2 正方形区域内的像素平均值作为纹理像素值。如果纹理是源文件大小的 1/4，那么可指定 D3DX_FILTER_LINEAR，这样将通过计算源图像中距离它最近的 4 个像素的平均值来得出纹理中的每个像素点，这也是该函数中非常有用的特征之一。而 D3DUtil_CreateTexture()使用 D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR 作为参数来调用 D3DXCreateTexture FromFileEx()，这将使纹理和源文件具有相同的尺寸，D3DX_FILTER_TRIANGLE 参数告诉 Direct3D 引擎，源文件中的每个像素对于纹理来说都具有相同的重要性（也是最慢的一种过滤方式），并且应当使用镜像过滤，即将 u 值、v 值、w 值作镜像。

MipFilter 参数使用了与 Filter 参数相同的选项。在使用 mipmap 时，处理不同尺寸贴图的视觉质量甚至比对单张纹理进行过滤还要重要。D3DUtil_CreateTexture()在这里使用了与 Filter 相同的参数。

ColorKey 参数用于设置一张纹理中的透明区域。它是一个 D3DCOLOR 值，纹理中所有等于该值的像素都将被替换成透明的黑色。缺省情况，不使用透明区域。

pSrcInfo 可以设置为一个指向 D3DXIMAGE_INFO 结构的指针，它用于保存函数返回的源文件宽度、高度、mip 级的个数和格式等信息。在 D3DUtil_CreateTexture()中没有使用到它。

如果使用到了调色板纹理（256 色），pPalette 将会以 PALETTEENTRY 结构返回一系列的调色板条目。但是 32 位的纹理具有更好的视觉质量，通过对 *Quake 2* 和 *Quake 3* 模型的对比就可以看出这种差别：前者只使用了 8 位的纹理，而后者则能够处理更光滑的 32 位纹理。我们都应采用 32 位的质量，所以在 D3DUtil_CreateTexture()调用 D3DXCreateTextureFromFileEx()时不需要获取这个调色板结构。

ppTexture 指针返回一个 IDirect3DTexture9 纹理接口指针，表示所创建出来的纹理。

如果 D3DXCreateTextureFromFileEx()成功，它返回 D3D_OK。

小结

仔细研究一下 D3DXCreateTextureFromFileEx()函数并用它来做一些试验是很值得的。因为它提供了大量的操作可用于在加载纹理时改变纹理的外观、质量等属性，你会惊异于在自己的代码中竟有如此多原先必须要自己编写的功能都已在这个函数中实现了。

附 录 E

游戏程序设计资源

你可能需要查阅一些正规的基础信息，以使自己可以与游戏程序设计快速的发展同步。那么我在这里推荐一些互联网资源，它们中可能就有你正在寻找的。

E.1 常规

http://www.bluesnews.com。提供了有关许多最新游戏的信息和截屏，这些内容会有助于你了解在游戏程序中哪些是可能做到的。有时它还会提供一些在游戏开发过程中泄漏出来的驱动程序和信息，不过这些内容可能不仅是专门面向开发者的。

http://www.gameversity.com。一个在线学院，面向所有希望学习游戏程序设计的人们。

http://www.realtimerendering.com。一个很好的关于图形程序设计的链接清单。

E.2 DirectX 图形

http://msdn.microsoft.com/directx。这是所有 DirectX 图形的源头，一定要在这里阅读一下 Philip Taylor 的文章。

http://www.direct3D.info。这是本书的相关网站，你可以在这里找到更新和更正。此外，这里还会有一些关于 Direct3D 的有趣教程。

http://www.gamedev.net。在这里可以找到许多 DirectX 图形方面的教程，其中有一些是我写的。

http://www.flipcode.com。一个游戏程序员的网站，每天我都会去看看上面的图片、代码、每日提示。

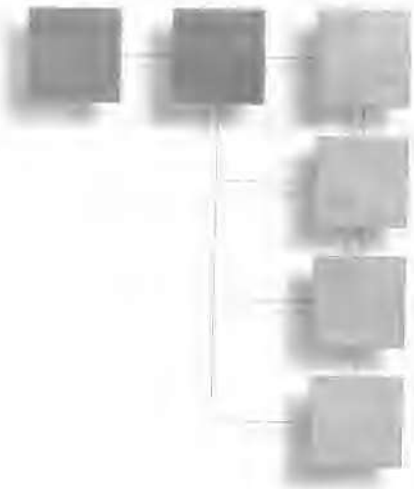
http://www.zen-x.net。ZEN-X 是一个有源代码的 DirectX 图形引擎，你可以通过阅读这份源代码学到很多东西。

http://www.cbloom.com。这是 Charles Bloom 的网站，他提供了一些非常有趣的文件和示例下载。

http://www.swissquake.ch/chumbalum-soft。MilkShape 3D 的主页，这是你能得到的最好的建模工具之一。

http//discuss.microsoft.com/archives/directxdev.html。这是一个很有必要了解的 HOW-TO 数据库。

附 录 F



随书光盘内容

你已经读过这本书，现在让我们最后来看一下会用到的软件。本书的 CD-ROM 包含了书中每个示例程序的源代码，另外还有一些有用的工具和应用程序，我用它们创建了书中讲述的效果。

如果你没有浏览过 CD-ROM 上的内容，那么现在就去撕开 CD-ROM 的封套并把它放入计算机的光盘驱动器吧。以下是这张 CD-ROM 中应用程序的清单[①]。

F.1 DirectX 9.0 SDK

最主要的软件就是微软的DirectX 9 SDK。为了使用本书的所有项目，必须首先安装SDK。关于如何安装 DirectX 9.0 SDK 的帮助，可以参阅本书的“前言”。

F.2 ATI RenderMonkey

光盘中包含了 ATI 神奇的 RenderMonkey 软件，使用这个软件可以创建出那些足以让人“目瞪口呆”的实时 shader 效果。这是一个 beta 版本。

译注①：由于软件在不断更新，这些软件中，DirectX SDK、RenderMonkey、Cg 各自都有最新的版本，你可以分别从以下网址来获取最新版本：

- DirectX SDK，http://www.microsoft.com/directx
- RenderMonkey，http://www.ati.com/developer/rendermonkey
- Cg，http://developer.nvidia.com/page/cg_main.html

F.3 NVIDIA Cg 工具包

Cg 是 NVIDIA 用于图形编程的新型高级语言。Cg 工具包是基于 Cg 语言的，它可以让程序员通过实时像素 shader 和顶点 shader 创建出令人惊叹的视觉效果。

F.4 教学演示动画

光盘中还包括了一些我创建的 Flash 动画，向你演示了如何在 Visual C/C++6 和 Visual C++.NET 中进行相关的设置，以便于你可以着手编写 Direct3D 应用程序。