



 Addison Wesley

教程

— 可编程实时图形权威指南

The Cg Tutorial

The Definitive Guide to Programmable Real-Time Graphics

Randima Fernando Mark J. Kilgard 著
洪伟 = 刘亚妮, 李骑 丁莲珍 译

*Foreword by Bill Mark,
Lead Designer, Cg Language*

The image features a close-up of a woman's face, with a focus on her eyes and nose. A semi-transparent yellow grid is overlaid on the left side of the image. Overlaid text includes "Foreword by Bill Mark, Lead Designer, Cg Language" and several lines of Cg shading language code related to lighting and reflection calculations for a cornnea or iris.

Cg

Cg（用于图形的C语言）是用于快速创建特殊效果和在多种平台上体验实时电影质量图像的一个完整的编程环境。通过提供一个全新的抽象层，Cg使得开发人员更加直接地面对OpenGL、DirectX、Windows、Linux、Mac OS X 和一些游戏平台（例如Xbox），而不需要直接使用图形硬件的汇编语言进行编程。Cg是由NVIDIA公司与Microsoft公司密切合作开发的，它与OpenGL API 和 Microsoft DirectX 9.0 的 HLSL 都兼容。

ISBN 7-115-12430-2



人民邮电出版社网址 www.ptpress.com.cn

教程

—可编程实时图形权威指南

本书解释了如何在当今的可编程GPU构架上实现基本和高级的技术。

主要涉及的方面包括：

- 3D 变换
- 每个顶点和每个像素的光照
- 顶点混合与关键帧插值
- 环境贴图
- 凹凸贴图
- 雾
- 性能优化
- 投影纹理
- 卡通着色
- 合成

“Cg是释放新一代可编程图形硬件能力的关键。本书是对Cg的最权威的介绍，而且将是任何编写高质量实时图形的人所必需的。本书将教你如何使用Cg来创建在以前的实时应用程序中从来没有出现过的效果。”

——Larry Gritz, *Advanced RenderMan* (Morgan Kaufman, 2000年) 一书的作者

“一本非常重要和及时的书：像素级的程序性纹理——云、火和水的动画，以及所有有关的程序技巧——终于从电影屏幕来到了桌面电脑。通过一种像C的语言来使用这种计算的力量将引导我们进入一个令人激动的图形新领域。”

——Ken Perlin, 纽约大学教授

ISBN7-115-12430-2/TP·4076
定价：38.00 元

Cg 教 程

——可编程实时图形权威指南

Randima Fernando Mark J.Kilgard 著
洪伟 刘亚妮 李骑 丁莲珍 译



人民邮电出版社

图书在版编目 (CIP) 数据

Cg 教程：可编程实时图形权威指南 / () 基尔加德 (Kilgard,R.F.M.J.) 著；洪伟等译。
—北京：人民邮电出版社，2004.9

ISBN 7-115-12430-2

I . C... II. ①基...②洪... III. 图形软件, Cg IV. TP391.41

中国版本图书馆 CIP 数据核字 (2004) 第 079205 号

版 权 声 明

Randima Fernando Mark J.Kilgard: The Cg Tutorial The Definitive Guide to Programmable Real-Time Graphics
ISBN: 0321194969

Copyright © 2003 by NVIDIA Corporation

This translation of The Cg Tutorial The Definitive Guide to Programmable Real-Time Graphics, First Edition is published by arrangement with Pearson Education Limited.

本书中文简体字版由 Addison Wesley 出版公司授权人民邮电出版社出版。未经出版者书面许可，对书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

Cg 教 程

—— 可编程实时图形权威指南

◆ 著 Randima Fernando Mark J.Kilgard

译 洪 伟 刘亚妮 李 骑 丁莲珍

责任编辑 俞 彬

◆ 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 010-67132705

北京汉魂图文设计有限公司制作

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：19

彩插：8

字数：348 千字

2004 年 9 月第 1 版

印数：1-3 500 册

2004 年 9 月北京第 1 次印刷

著作权合同登记 图字：01 - 2003 - 3576 号

ISBN 7-115-12430-2/TP • 4076

定价：38.00 元

本书如有印装质量问题，请与本社联系 电话：(010) 67129223

图 1

Cg 着色器在 Maya4.5 中使用 Maya Cg 插件

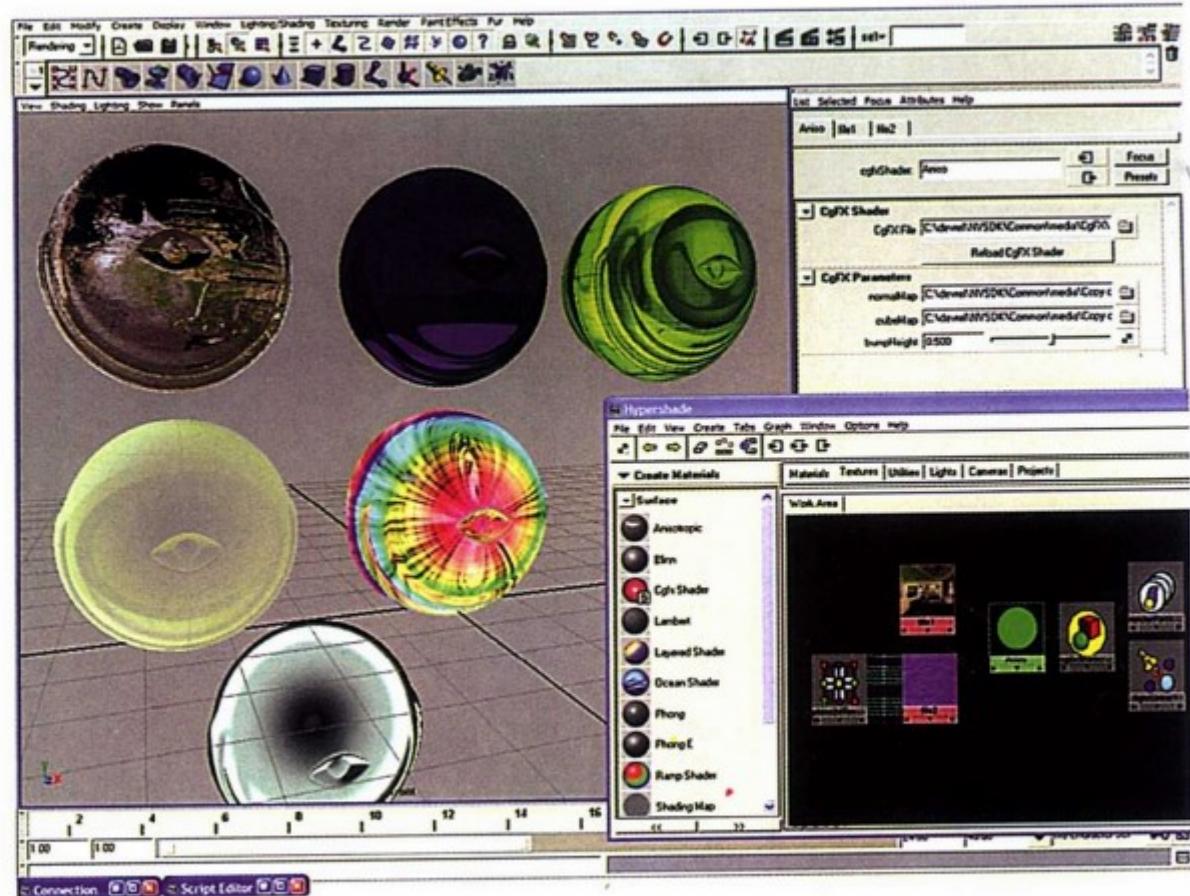


图 2

使用 Maya 和 Cg 在 NVIDIA Quadro FX 上渲染的各种形状的珠子

感谢 Alias\Wavefront 提供图像



图 3

Discreet 3ds max 5 和 3ds max Cg 插件

在 3ds max DirectX 视图中应用和显示的 Cg 着色器通过独立的显示程序也被显示在一个 OpenGL 窗口中。

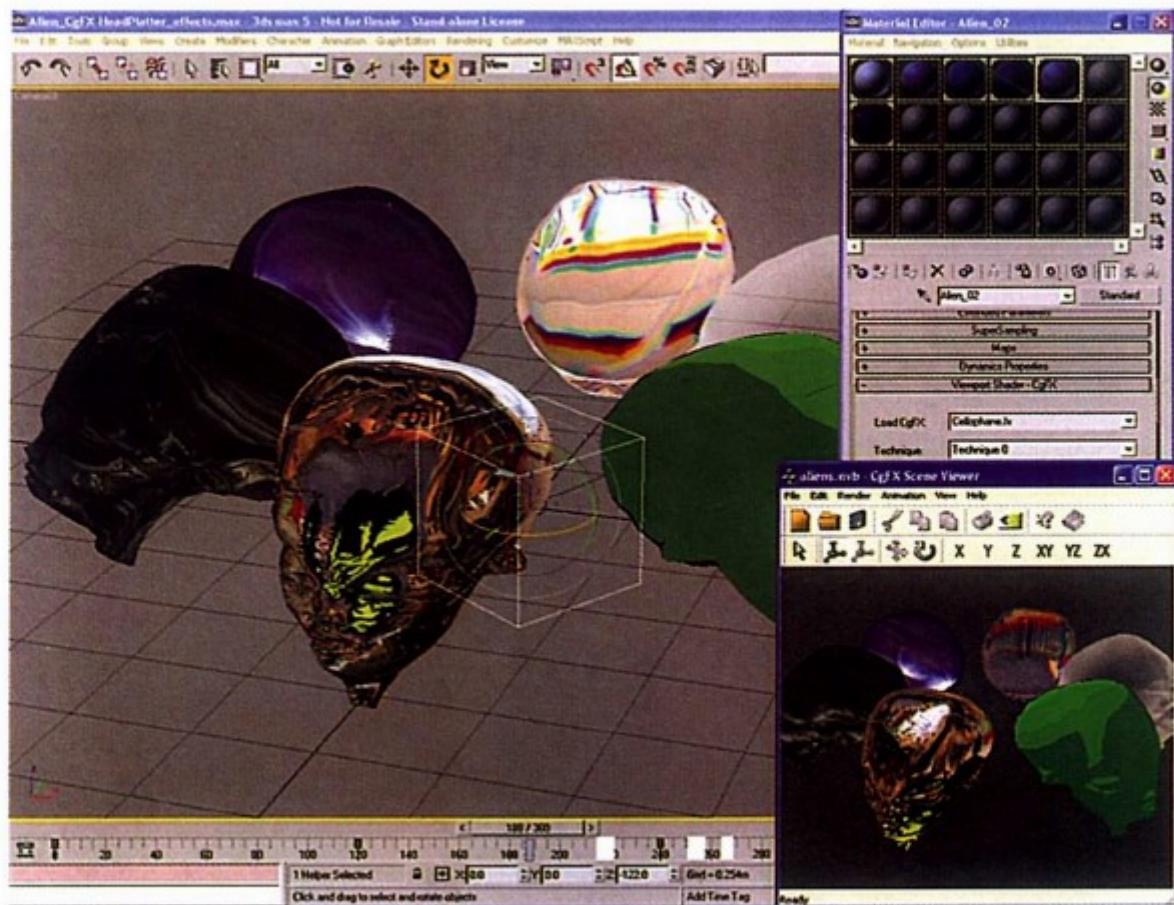


图 4

3ds max Cg 插件和连接编辑器

3ds max Cg 插件允许美工人员通过连接编辑器调整参数和选择渲染技术。这个编辑器在 3ds max 场景和 Cg 着色器的应用程序驱动的变量之间建立了一个动态连接。

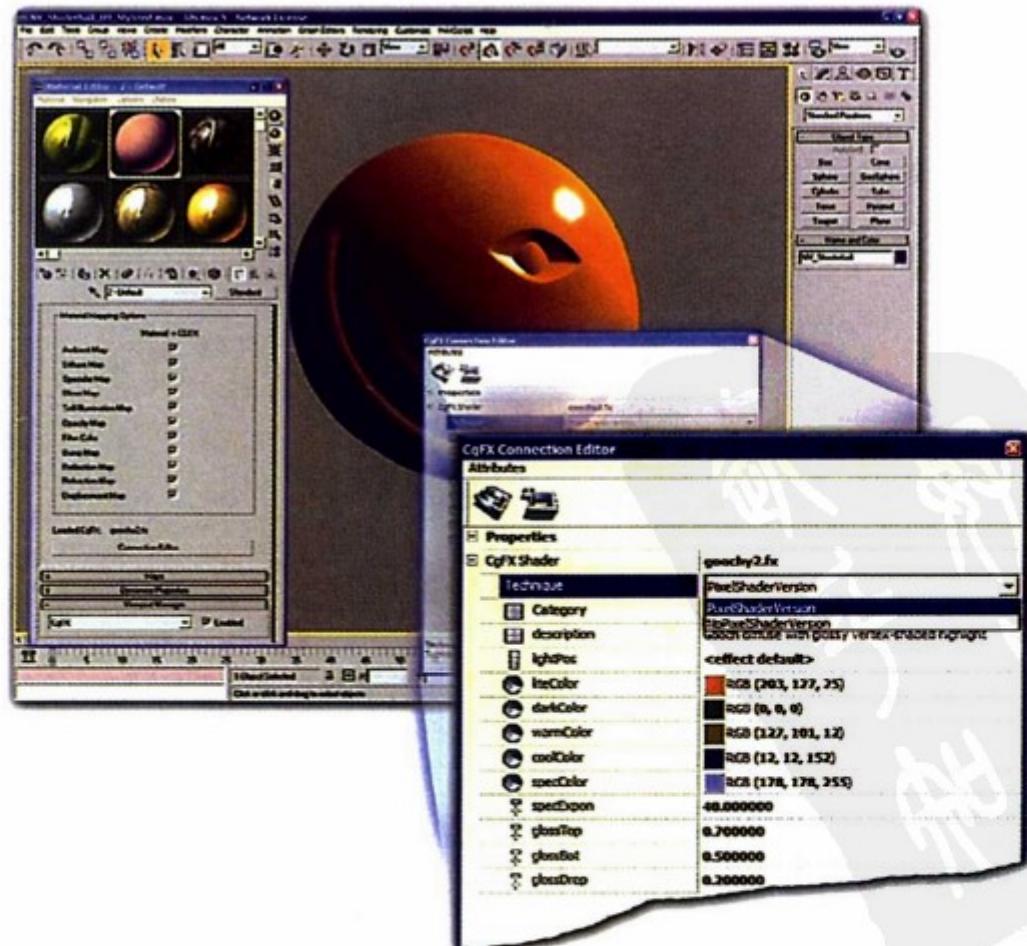


图 5

Softimage/XSI v.3.0 集成了
了Cg

感谢 Softimage Co. 和 Avid
Technology, Inc 提供的图像。

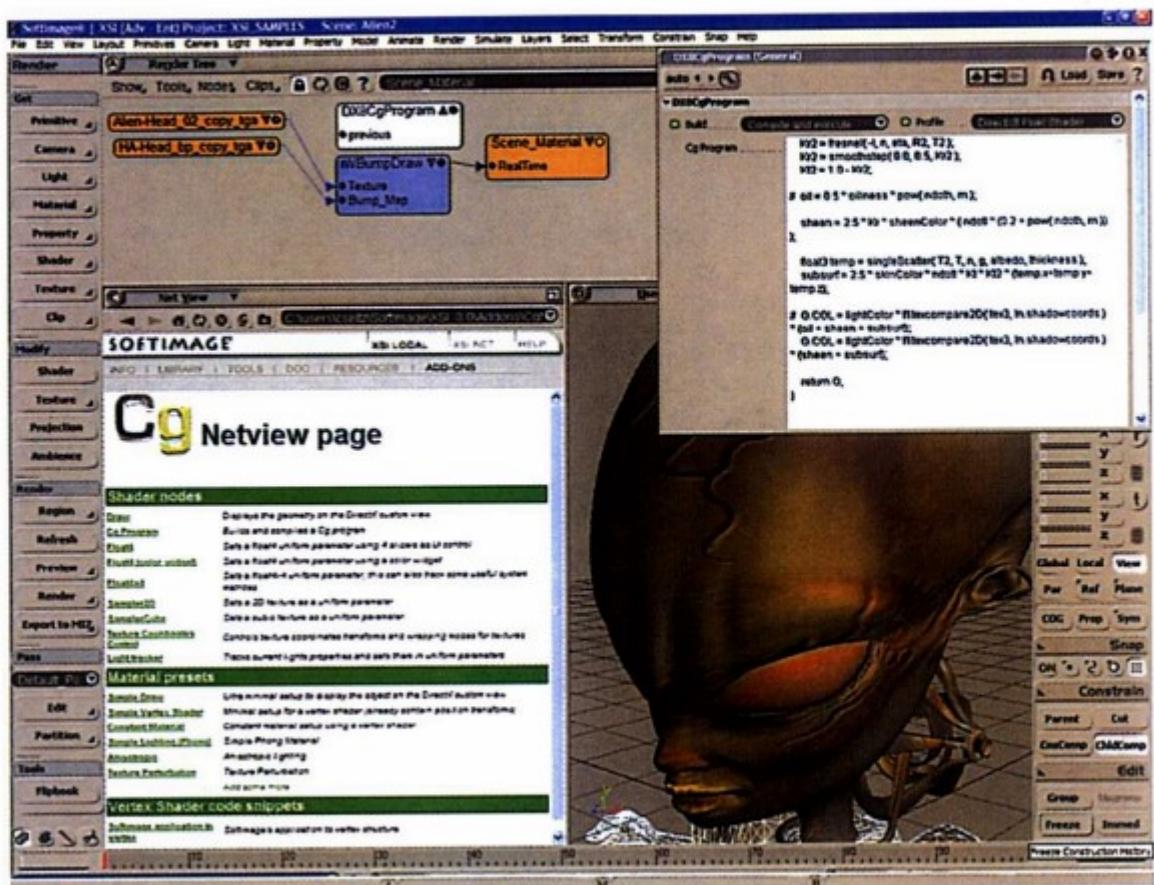


图 6

创建重影效果

C3E5v_twoTextures 顶点程序移动了一个单独的纹理坐标位置两次，使用两个不同的位移来生成两个稍稍分开的纹理坐标集。然后，C3E6f_twoTextures 在两个偏移位置对一个纹理图像进行存取，并平等地混合这两个纹理结果。有关的详细描述可以参看第 3 章。

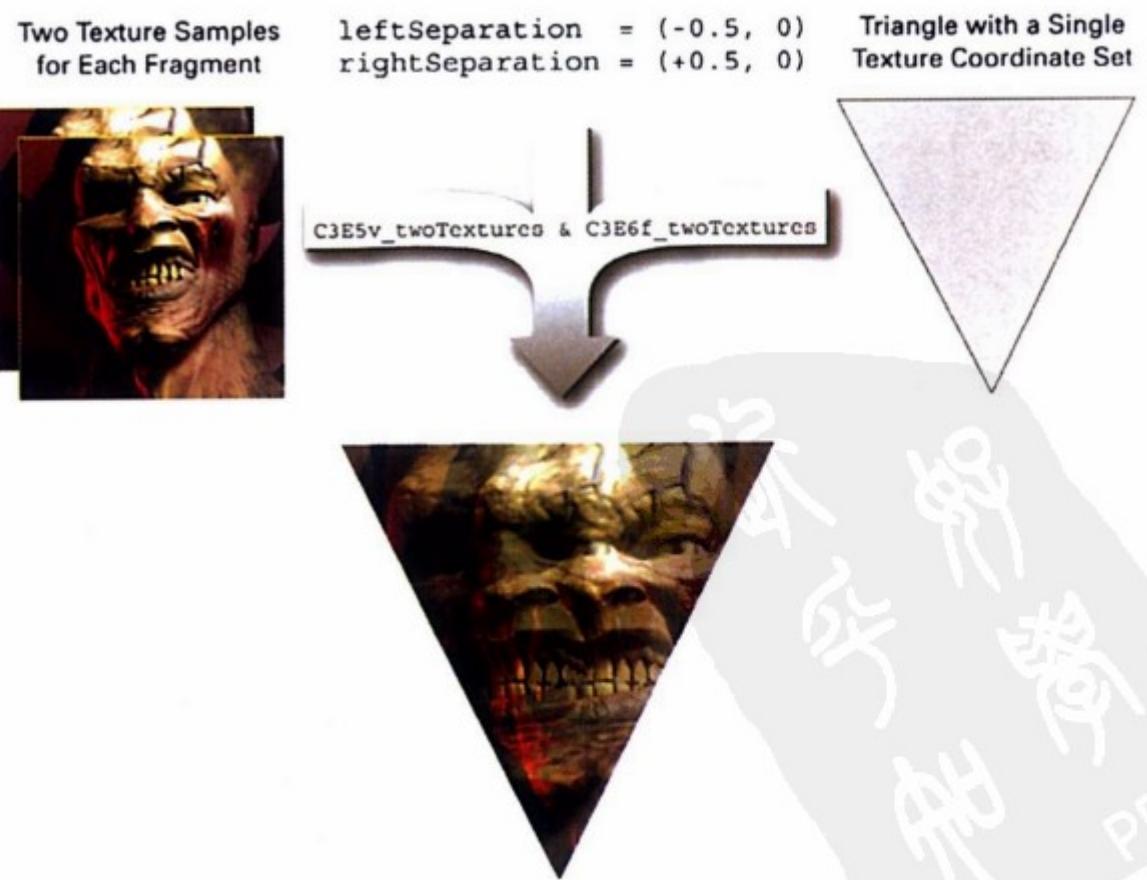


图 7

环境光照、漫反射光照和镜面反射光的组合

(详见第 5 章)

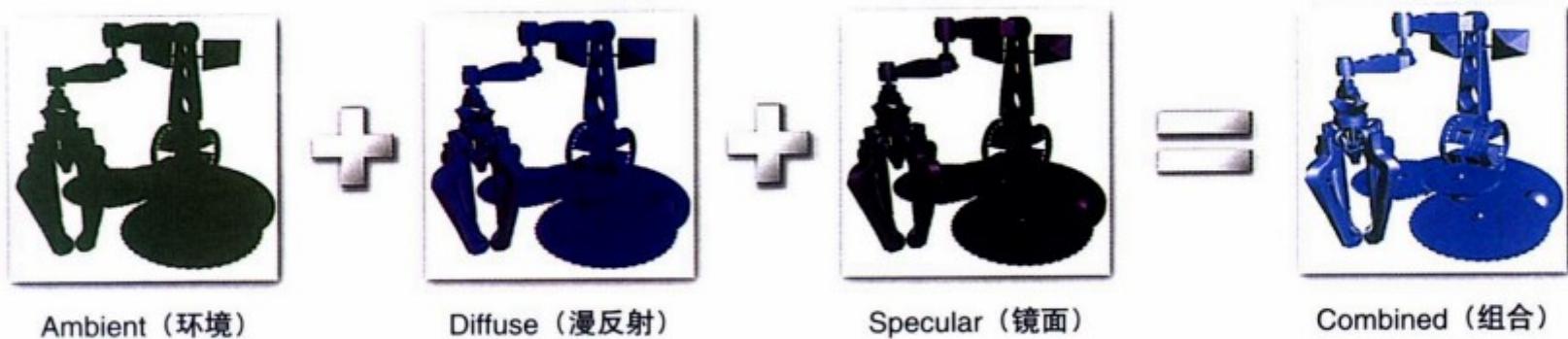


图 8

六个纹理图像组合起来形成一个立方贴图

(详见第 7 章)

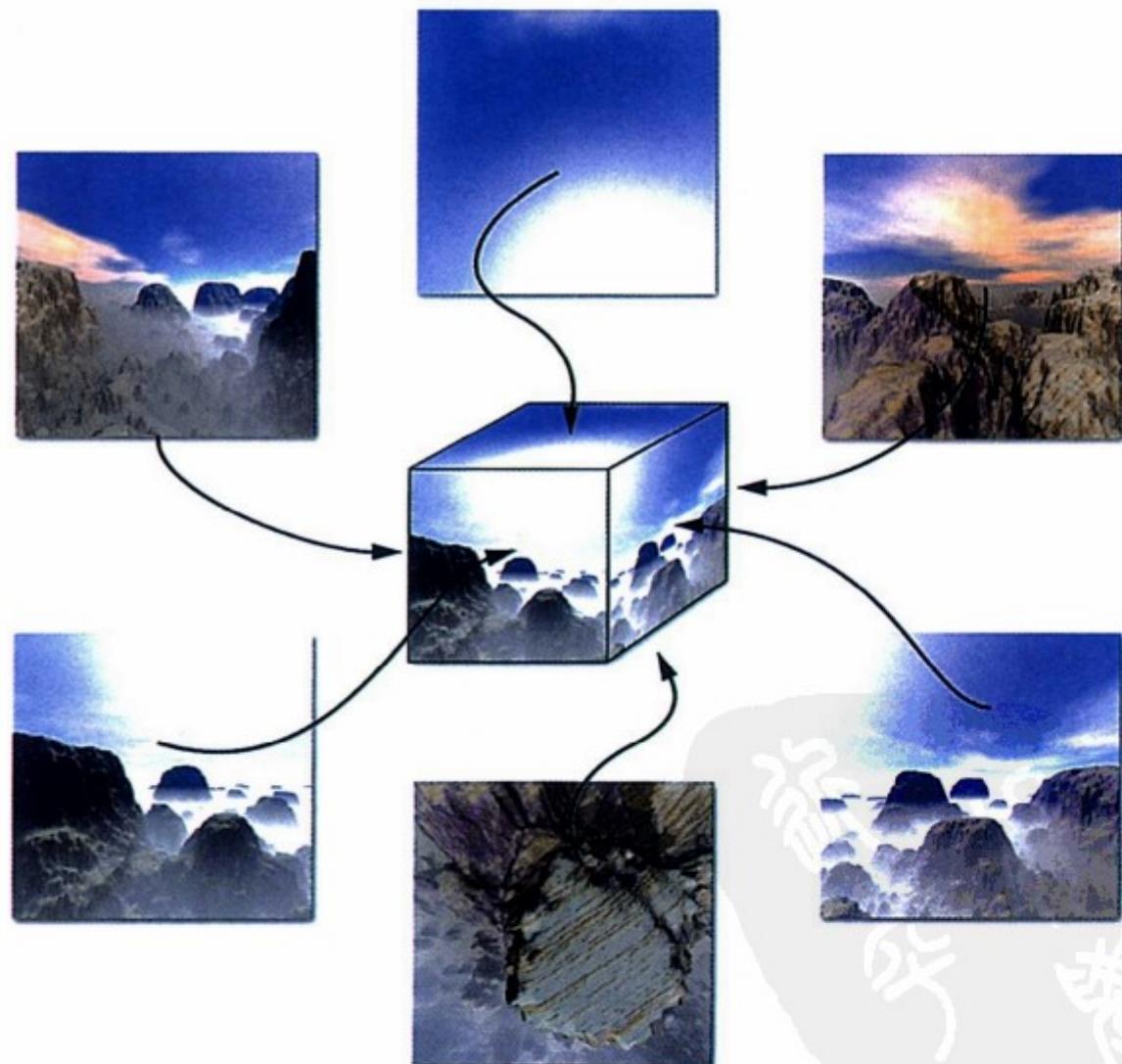


图 9

反射环境贴图

(详见第 7 章)



图 10

折射环境贴图

(详见第 7 章)

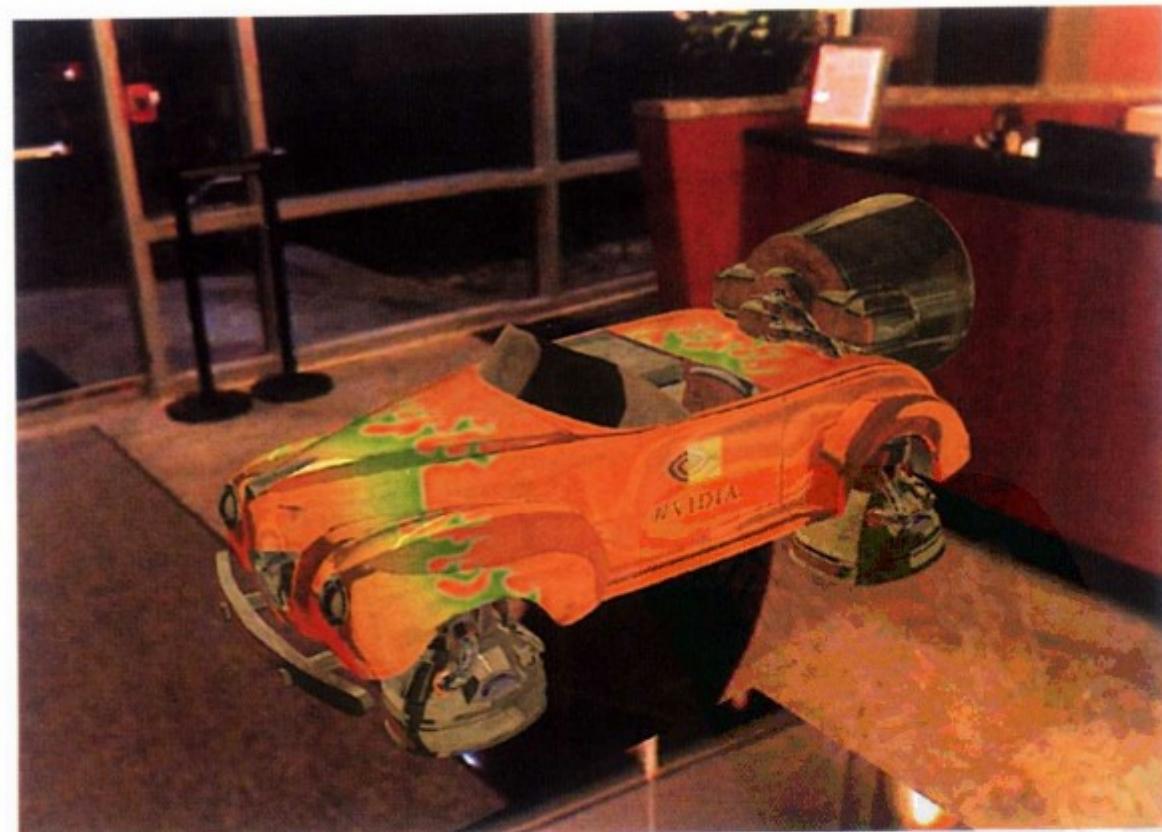


图 11

菲涅耳效果和颜色色散
(详见第 7 章)

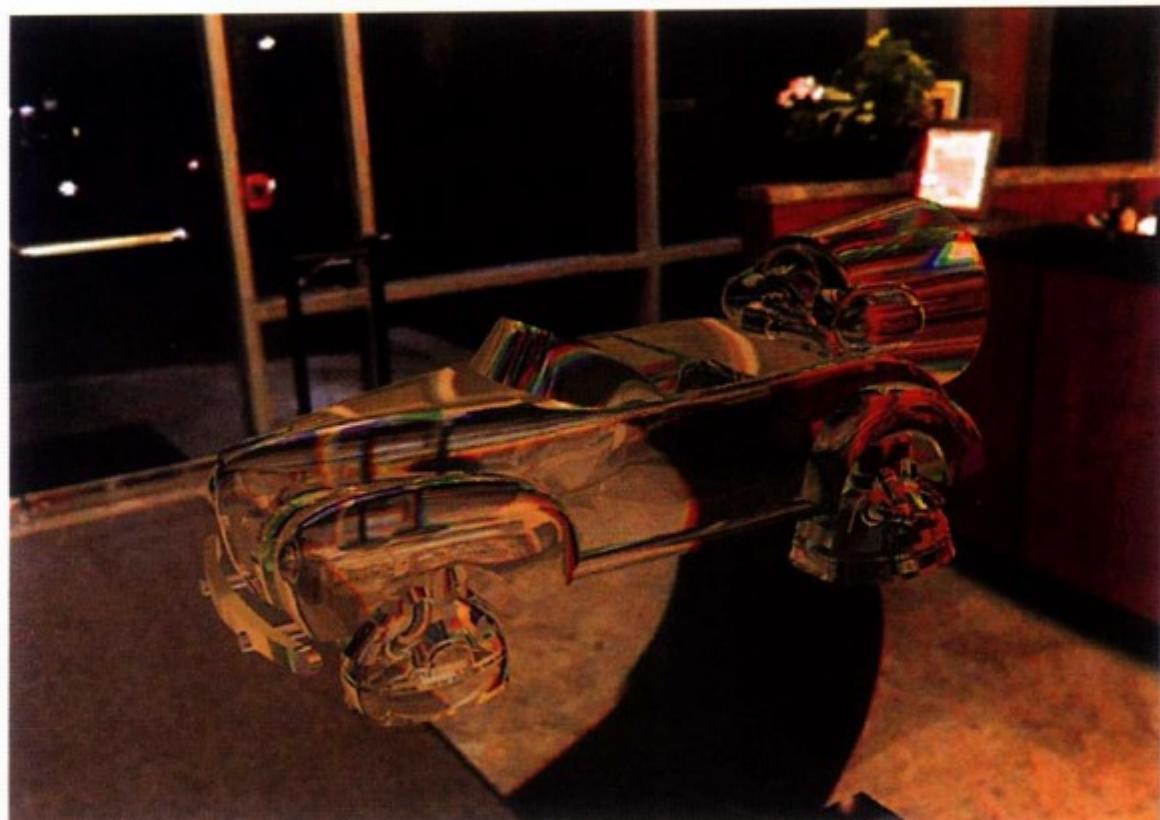


图 12

一个灰度高度域转化成为一个范围压缩的向量贴图
(详见第 8 章)

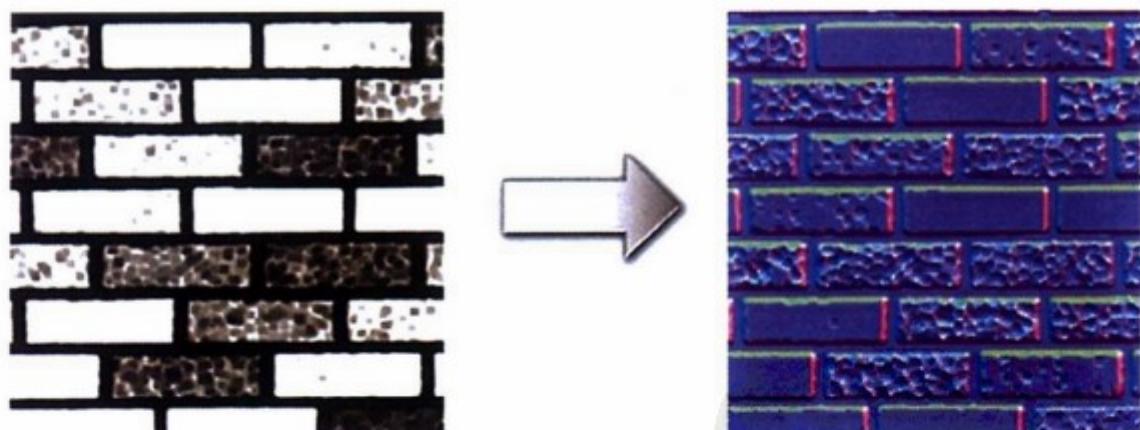


图 13

凹凸贴图一个网格

同一个三角形存在于 2D 纹理空间（左边）和 3D 物体空间（中间和右边）中。有关的详细描述可以参看第 8 章。

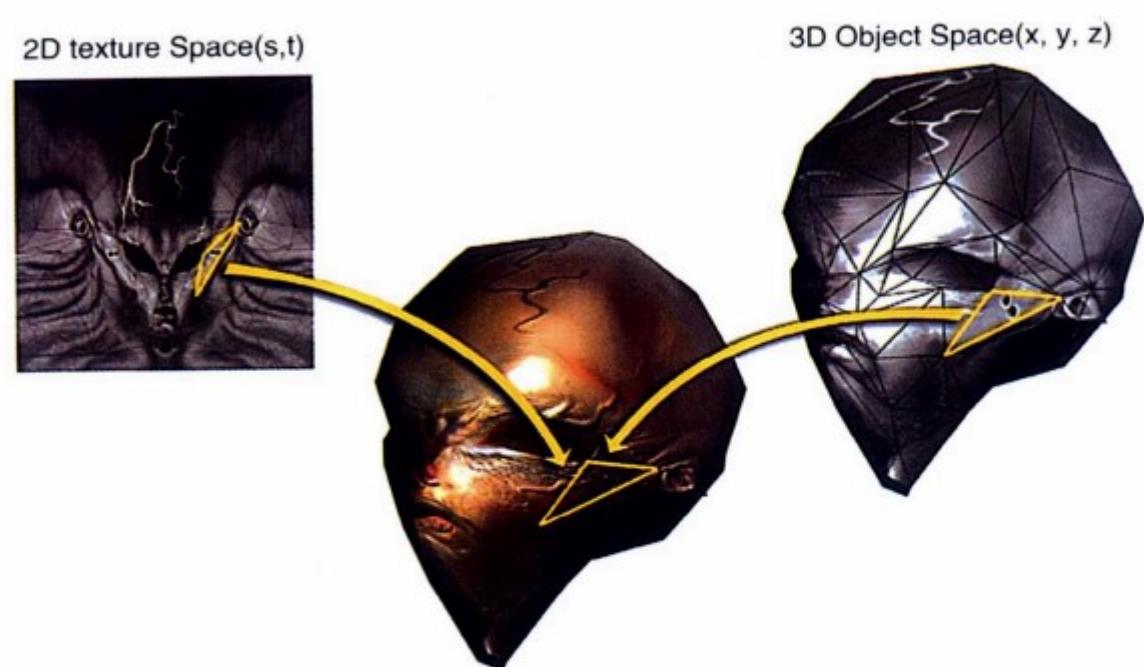


图 14

非真实性渲染

一个卡通着色的光线枪和对应的使用了漫反射和镜面反射着色的图像。有关的详细描述可以参看第 9 章。

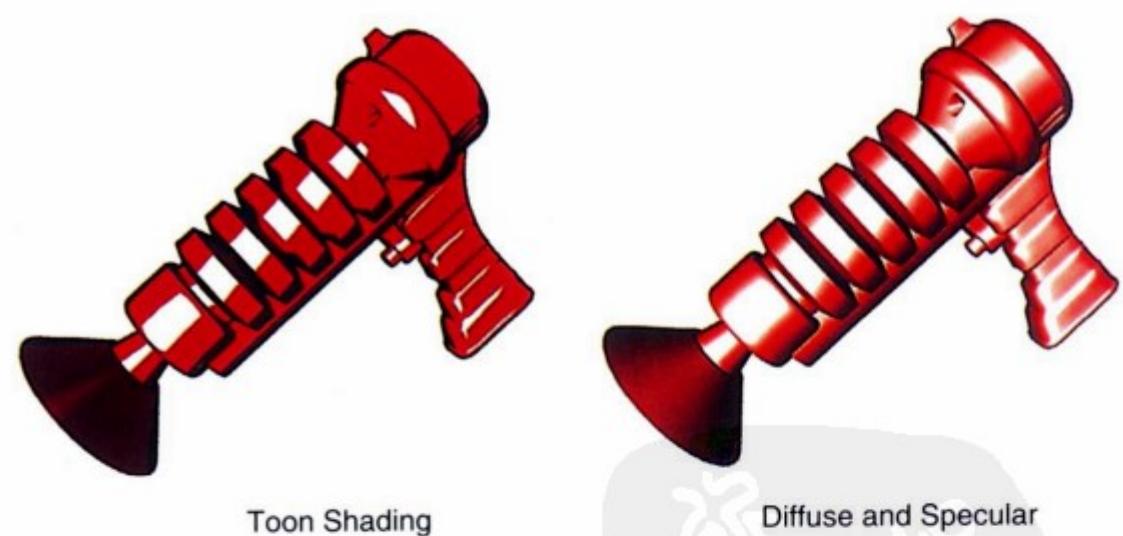


图 15

由用于表皮的着色器、骨骼蒙皮和形状混合所驱动的 Dawn

皮肤着色器使用了颜色、镜面反射和血液特征贴图的一个组合来生成非常真实的皮肤。另外，一系列的立方贴图——用于漫反射和“高亮度”皮肤光照——生成了场景光照的微妙之处。对于翅膀，一个透明的着色器根据视线和光线的角度，修改了从它们反射的颜色和穿过它们的光的量。着色器还操纵了顶点来计算骨骼蒙皮和形状混合动画。

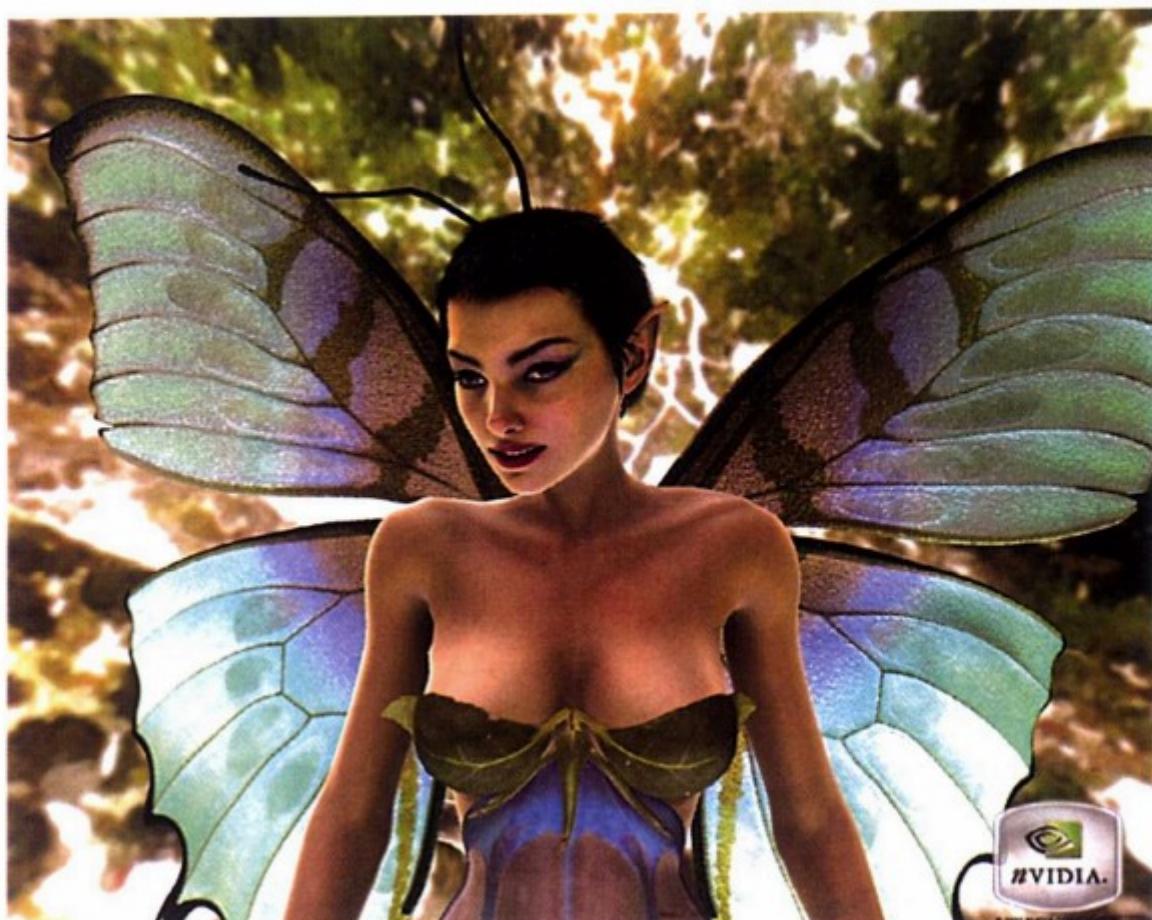


图 16

NVIDIA 时间机器演示

在场景中的每个老化的材质都有一个与之广联的独立的着色器。每个着色器程序化控制应用程序的颜色、法向量、镜面反射、反射、反射率和汽车窗框的贴图。

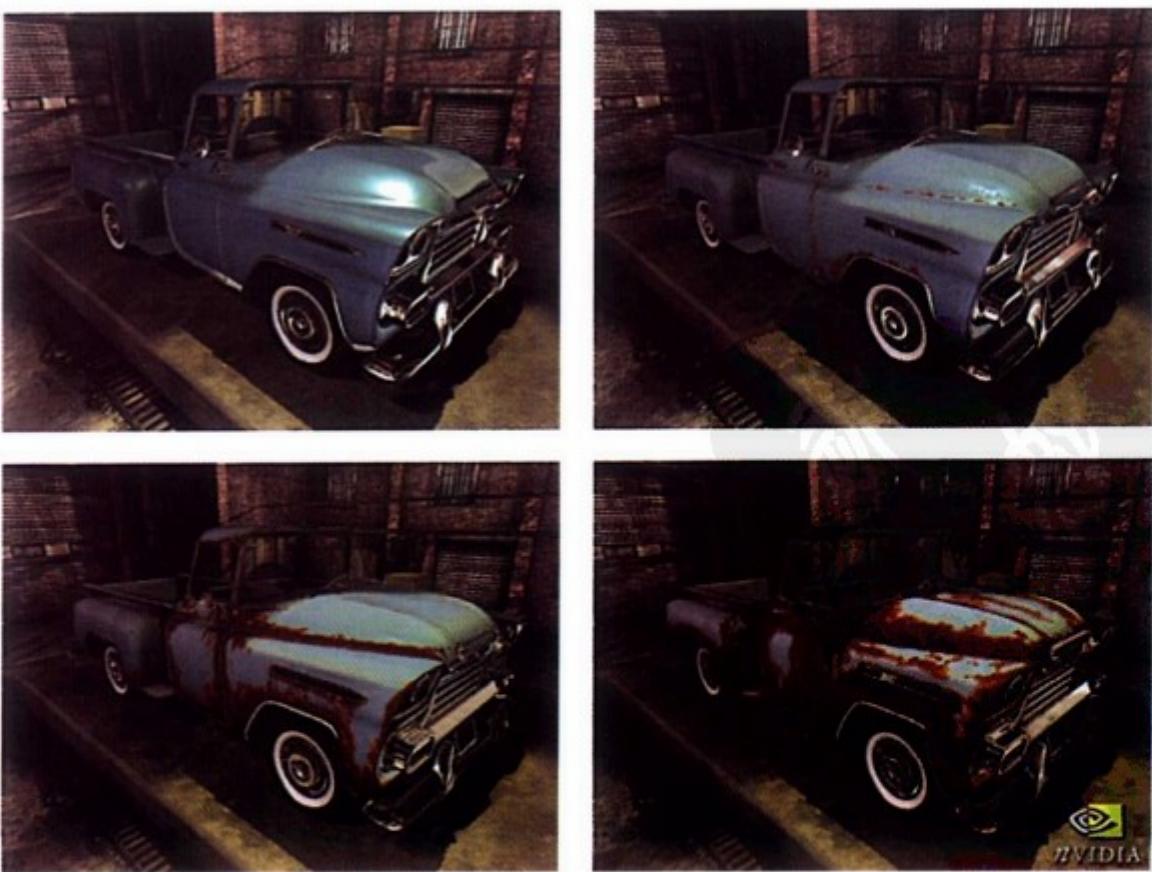


图 17

NVIDIA 跳舞的 Ogre 演示

在一个GeForce FX演示中，顶点和片断程序驱动由Spellcraft Studios 创建的实时动画物体。一个高级的皮肤着色器通过一个颜色、凹凸和反射纹理贴图的组合使用了真正的Blinn凹凸映射。光照效果包括阴影贴图的阴影和物体自我遮挡。



图 18

Yeti 的 Gun Metal 游戏，使用 Cg 来创建高级视觉效果



图 19

Iritor 在线使用了 Cg 着色器



图 20

程序性木头纹理着色器

Arkadiusz Waliszewski 的 Cg 程序在一个由程序生成的三维木头纹理上实现了每个像素正确的光照。浏览 www.cgshaders.org 可以找到彩图 20 到 23 中的着色器的详细资料。

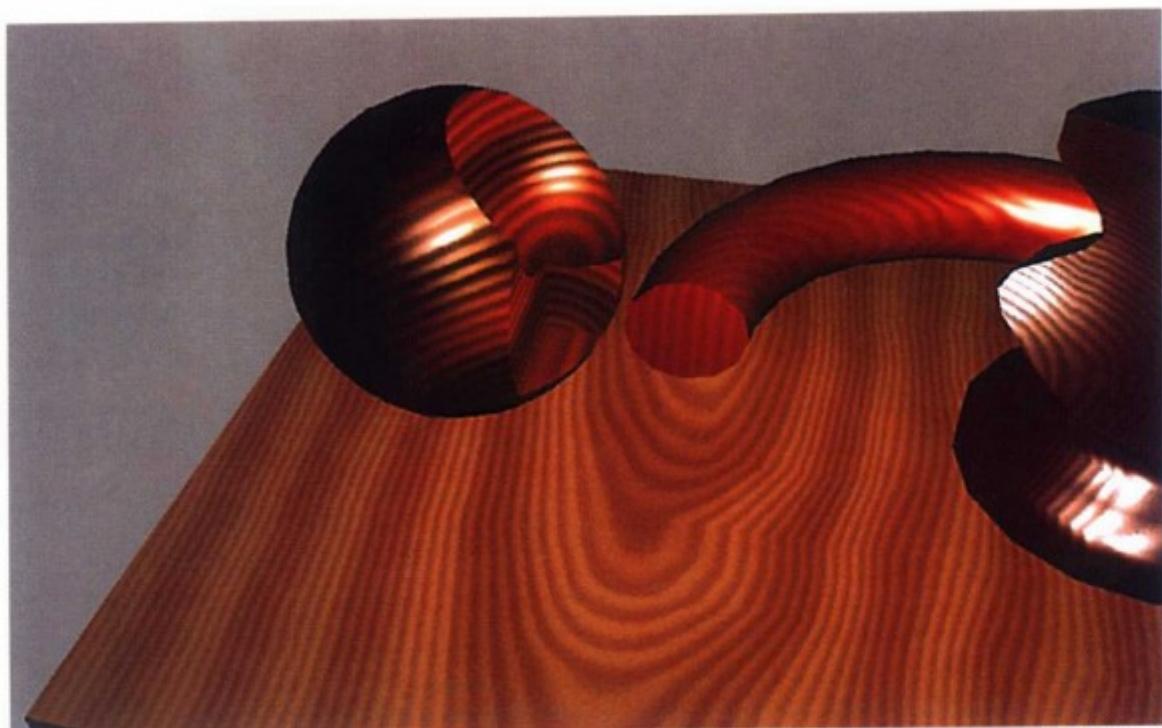


图 21

景深

带有动态的每个像素光照的场景深度效果，由 Arkadiusz Waliszewski 在一个 Cg 程序中实现。



图 22

动态反应扩散纹理

来自自由 Mark Harris 编写的一个 Cg 程序实现的一个模拟器, 这一系列图像展示了两种化学药品的反应和后来的扩散过程。

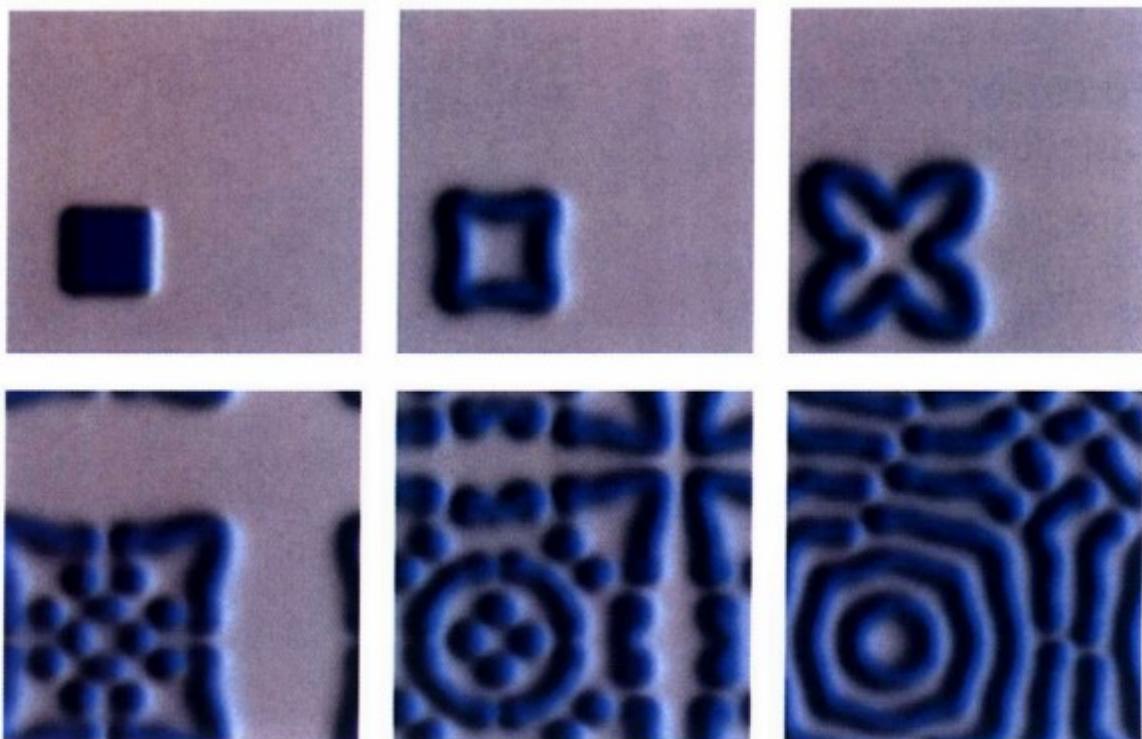


图 23

浮雕纹理贴图

由 Fabio Policarpo 用一个 Cg 程序实现。这个基于图像的渲染技术从一个预先生成的立方纹理贴图使用了物体的法向量和深度值来用一个简单的约束框表现大而密集的网格。



图 24

凹凸反射贴图

来自 NVIDIA Cg 浏览器的 Cg 程序，使用了法向量贴图和一个立方环境贴图来使得飞碟表面看起来有凹凸感和反光。



图 25

细节向量贴图

一个简单多边形网格的底层线框图增强了放置两层法向量贴图来创建一个精细的和一个粗糙的凹凸映射效果的结果。由一个来自 NVIDIA Cg 浏览器的 Cg 程序生成的。



图 26

水的交互

Cg 顶点和像素程序被用来动态展示水的高度，并把高度值转换为一个表面法向量贴图用于反射着色。在图形处理器上，Cg 着色器计算一个水的物理模拟，然后这个模拟被用于生成表面法向量贴图。由来自 NVIDIA Cg 浏览器的一个 Cg 程序生成的。



图 27

一个程序性地形演示

为了节省有限的内存，这个顶点程序把排列和梯度表合并到一个 float4 的数组中。由来自 NVIDIA Cg 浏览器基于 Ken Perlin 的源代码实现了 Cg 噪音生成的。

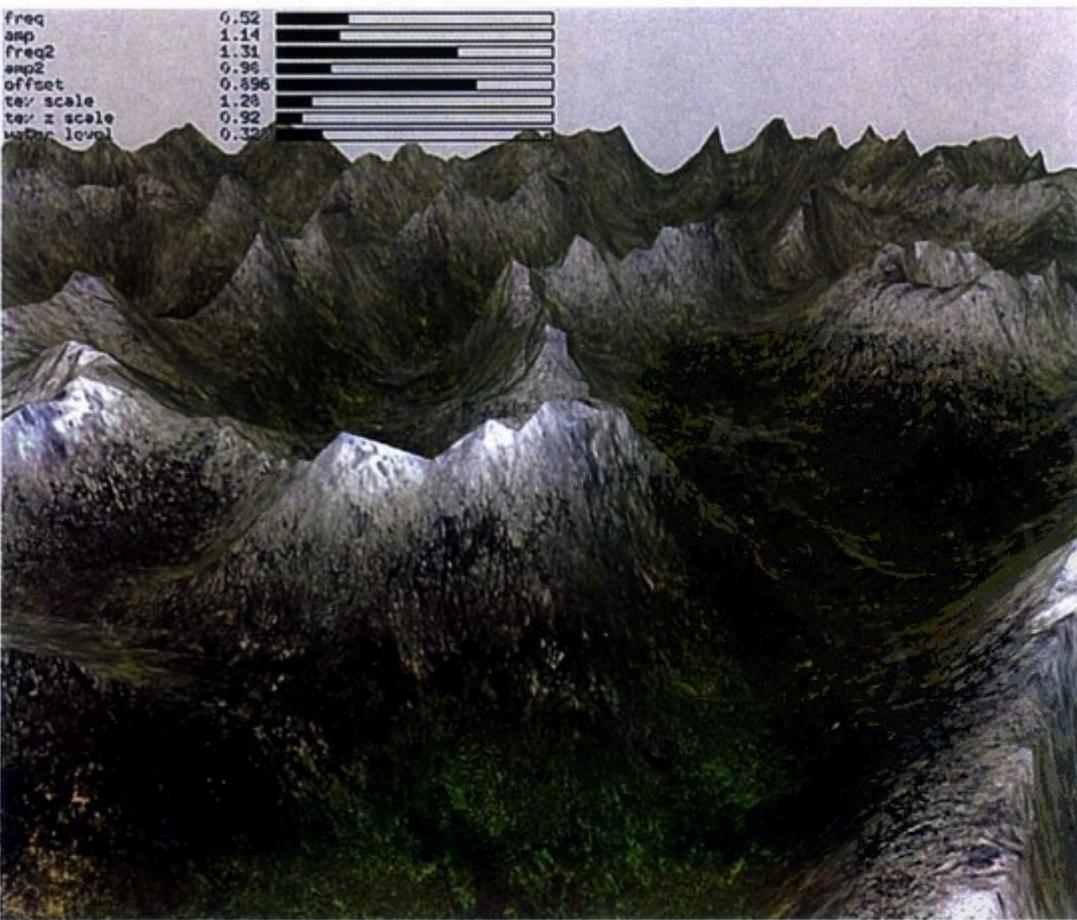


图 28

逼真的菲涅耳反射效果

几个不同的贴图物体使用一个单独的Cg着色器着色后，看起来就像是擦光的玻璃。每个物体使用了稍微不同的参数，包括变换折射指数。



图 29

NVIDIA 的 Cg 浏览器接口

屏幕显示了 Cg 浏览器应用程序的三个视图：一个示例的层次树（左边），用于一个各向异性光照示例的 Cg 代码（中间），和一个把各向异性光照着色器应用到一个飞碟模型上的交互显示。

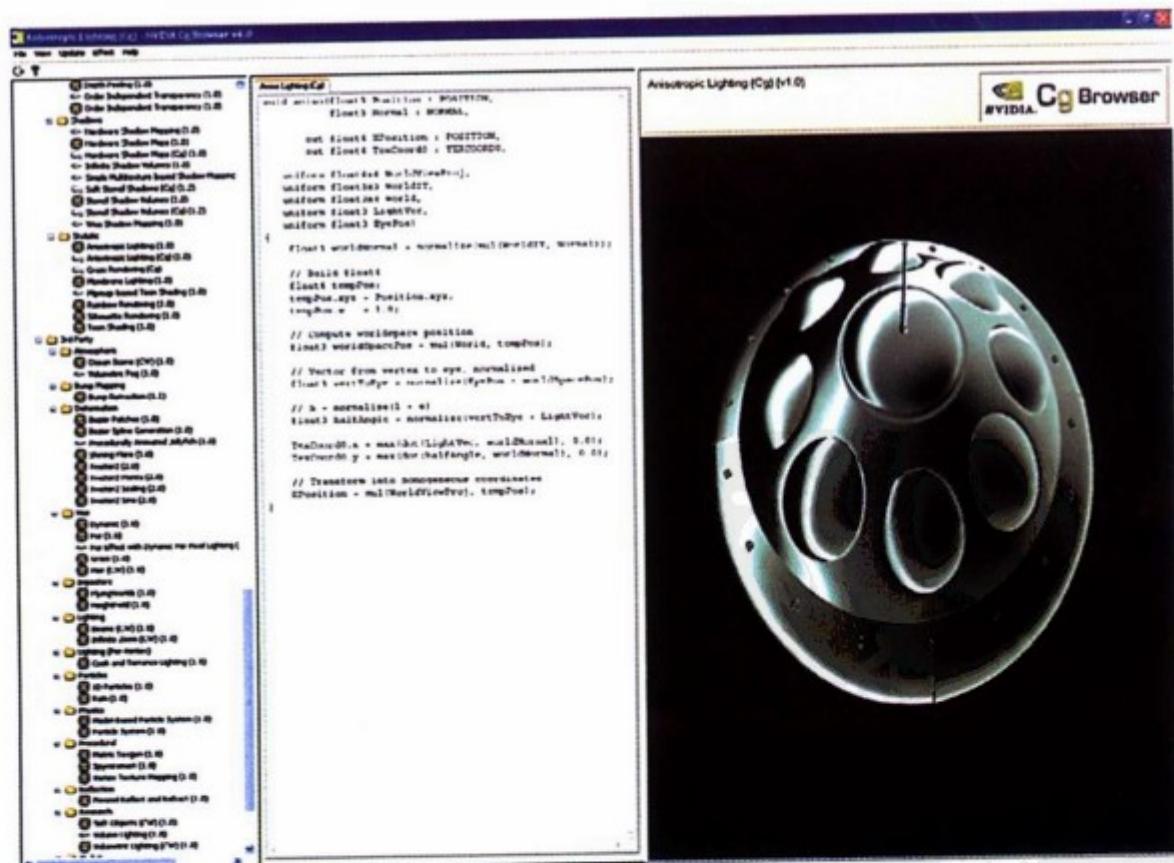


图 30

使用了 Gooch Cg 着色器的机械模型

Gooch着色用暖色调和冷色调替换了明到暗的过渡，在这种情况下能够使得工程师看到一个复杂机器部件的所有区域的形状和曲率。

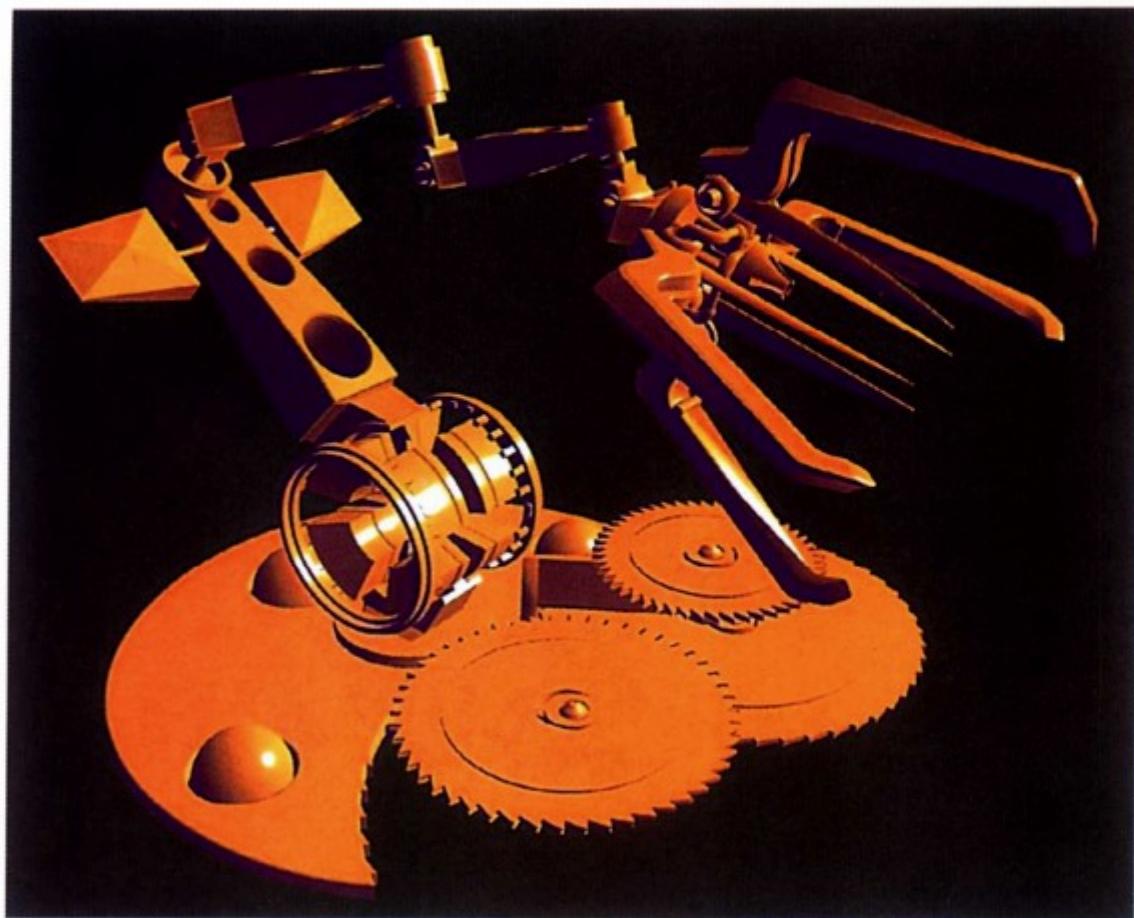


图 31

使用 CgFX 应用多个着色器到一个光线枪模型

同一个CgFX文件被应用到枪管和枪体上，但使用了不同的反射和颜色参数。另外的参数可以在一个Cg显示程序的连接编辑器中看到。



内容提要

Cg 是最早的为可编程图形硬件设计的高级编程语言，本书教你如何编写 Cg 程序。

本书共 10 章，第 1 章简要介绍 Cg 语言。随后的每一章分别介绍 Cg 中的一个概念和技术的说明，包括最简单的 Cg 程序，参数、纹理和表达式，如何进行顶点变换，如何用 Cg 实现光照模型，如何用 Cg 顶点程序实现模型的动画和变换，如何实现环境贴图，如何实现凹凸贴图，雾、卡通光照、投影聚光、阴影贴图和复合技术，目前可用的 Cg 的顶点和片断的简要描述（Profile）等内容。每章的结尾提供了习题，以帮助你进一步学习，探索更多的 Cg 知识。本书另外还附有 5 个关于 Cg 基础知识的附录。

本书适合开发三维游戏和应用软件的程序员及项目管理人员、实时三维美工人员和学习计算机图形学的学生阅读，也可供任何对学习实时渲染技术当前发展动态有兴趣的人士阅读参考。



序 言

实时计算机图形硬件从支持一些固定的算法到完全可编程，正在经历着一个显著的转变。同时，图形处理器（GPU）的性能也在以非常快的速率增加，甚至比中央处理器（CPU）的性能增长速度还要快，因为图形处理器可以有效地利用图形算法中大量存在的并行计算。这些在图形处理器灵活性和性能方面的改进在将来很可能会继续，这些改进将能允许开发者编写愈加复杂和多样的程序在图形处理器上执行。

直到最近，大部分的图形处理器程序都是用汇编语言编写的，但是历史已经显示了当开发者们有机会用高级语言编程的时候，他们能够创造性和有效地使用这些可编程的硬件。我们设计 Cg 的目标是在为开发人员提供与汇编语言一样的灵活性和高性能的同时，提供高级编程语言的表达力和易用性。C 语言在中央处理器上的应用已经实现了这些目标，证明了它巨大的成功。因此，我们选择 C 语言的语法和基本原理作为我们定义 Cg 的起点。特别是，这个选择给了我们信心—Cg 语言将能够实现任何图形处理器程序，而不是把开发人员限制在一个预先定义的为着色计算特别设计的框架中。

Cg 同时也是建立在一些从早期的计算机图形学方面的工作中获得的想法和教训之上的。我个人在可编程图形硬件方面的兴趣是被北卡大学的像素流项目（PixelFlow）所激发的，特别是被其中由 Marc Olano 和其他人为 PixelFlow 所编写的着色编程语言编译器所激发。从许多方面看，Cg 语言最直接的原型是由 Kekoa Proudfoot、Pat Hanrahan、我和其他一些人在斯坦福为单片机图形处理器编写的一个实时可编程着色系统。我们从编写斯坦福系统所获得的这些知识对设计 Cg 语言是至关重要的。不实时的电影渲染依赖可编程的着色已经很久了。由 Pat Hanrahan 在 Pixar 公司所设计的 RenderMan 语言是 Cg 语言一个主要的灵感，特别是在选择

Cg 语言的内置函数的时候。

创造一个可以被 OpenGL 和 Direct3D API 所广泛支持的图形处理器编程语言的期望使我们在设计 Cg 的时候和 Microsoft 进行了协作。Microsoft 所实现的语言被称为 HLSL，被作为 DirectX9 的一部分发行。在 Microsoft，Craig Peeler 和 Loren McQuade 对语言的设计有深远的影响。在本书中所使用的例子，HLSL 和 Cg 上都工作得很好。

能够在短短的不到一年的时间里设计和实现 Cg 语言，只可能是因为我们有非常有才能的一组人在一起工作。其中两个人作出了特别的独一无二的贡献。Steve Glanville 在编译器和语言方面的广博的专家级的意见对 Cg 的设计是至关重要的，此外 Cg 编译器第一个版本的前台大部分都是由他实现的。Kurt Akeley 的作为系统设计师的经验对整个项目的作用是无法衡量的。

在这本书里，Randy 和 Mark 组织安排了一系列的 Cg 应用指南，在介绍 Cg 的同时讲述了许多被先进的图形处理器所支持的实时光照技术。通过把这两组概念集中在一组练习中，本书使你能够体验到编写和试验你自己的图形处理器程序的乐趣。享受你的学习旅程吧！

Bill Mark
Cg 语言主设计师
德克萨斯大学奥斯汀分校助理教授

前　言

从前，实时的计算机图形问题全是关于顶点、三角形和像素的。实际上，现在的情况仍然是这样。但是，一个程序员可以控制的处理层次和那些图形原语的表现已经有了相当大的进步。直到几年前，程序员们仍然不得不依赖中央处理器来处理所有这些用来生成计算机图像的变换和光栅化算法。随着时间的推移，硬件工程师们开始在特别设计的高性能的三维图形硬件上执行这些算法。程序员们开始通过标准的三维编程接口来访问这些硬件提供的图形功能，例如 OpenGL（由 Silicon Graphics [SGI] 公司开发）和 Direct3D（由 Microsoft 公司开发），而不再需要直接实现这些算法。一开始，这些昂贵的三维图形硬件只出现在天价的 UNIX 工作站和飞行模拟器上。现在，通过摩尔定律的奇迹，图形硬件加速已经给低价的 PC 机和游戏机带来了极大的益处。

尽管通过使用精密的图形硬件来执行顶点变换、三角形光栅化和像素更新等复杂的任务所获得的性能远远超过了只使用中央处理器编程所获得的性能，作为代价，实时三维图形程序员放弃了相当大的控制权来换取这种高速度。开发人员被限制在仅能使用一些硬件所能处理的调色板功能固定的图形操作。有时，一个熟练和专注的程序员可以巧妙地使用这些图形编程接口和图形硬件来完成某些不一般的任务，但是这是一项非常困难和耗时的工作。

当图形硬件工程师们开始使他们专门设计的由像素驱动的硬件向性能实时化发展的同时，非实时的计算机图形软件包，例如 Pixar 公司的 PhotoRealistic RenderMan，正在用令人惊奇的计算机生成的特殊效果改变电影和电视的表现力。由于电影和大部分电视节目是预先录制的，使得这些媒体非常适合非实时的渲染。计算机为电影和视频生成的图像虽然不是实时渲染的，而是非常仔细地一帧一帧地构造的，这项工作使用一般标准的中央处理器通常需要几个小时、几天或几周

时间来完成。使用通用的中央处理器的优势是程序员和美工师可以利用中央处理器创造各种他们可以想象出的效果，而不是满足于实现复杂的硬件算法。这些所谓的非实时渲染系统所欠缺的相对速度，被它们通过渲染的效果和真实性来弥补了。

非实时渲染系统的灵活性和通用性是上一代的三维图形硬件所缺乏的关键特性。换句话说，上一代三维图形硬件缺乏可编程性。

认识到这个限制，计算机图形设计师已经设计了新一代的图形硬件，能够允许一个空前灵活的可编程性。现在，许多可编程的光照技术已经成功地被使用在非实时渲染上，并且开始进入实时图形技术领域。

非实时渲染系统的开发人员创造一种被称为光照语言的专门的计算机语言来表达那些使得物体表面看上去像美工师预期的那样的图形操作。一种为可编程图形硬件设计的光照语言提供了同样功能，但是在实时图形硬件范畴内提供的。就像传统的程序员从 C++ 和 Java 获得巨大的帮助那样，图形程序员和美工人员将受益于这样一种高级编程语言。使用为图形硬件设计的高级语言能够自动把程序员的意图转换成图形硬件可以执行的形式。

本书是关于 Cg 的，Cg 是最早的为可编程图形硬件设计的高级编程语言。NVIDIA 和 Microsoft 密切合作一起开发了 Cg。Cg 是帮你释放可编程图形硬件强大力量的最轻便和高效的方法。本书是一本教你如何编写 Cg 程序的指南。

我们预期的读者

我们试图以一种对初学者和高级读者都有价值的方式来编写这本书。如果可编程图形这个世界对你来说是完全陌生的话，这本书将给你的学习打下一个坚实的基础。如果你遇到一个对你来说不熟悉的词汇或概念而本书又没有足够的解释，请参考每章最后的补充阅读部分。

这本书主要的读者对象是开发三维游戏和应用软件的程序员、开发这些项目的管理人员、实时三维美工人员和学习计算机图形学的学生，或者任何对实时渲染技术当前发展动态有兴趣的人。虽然你应该相对熟悉编程语言的一些概念，但是通过本书学习 Cg 语言，你不需要是一个有经验的程序员。如果你非常熟悉 C 语言或者从它派生出来的语言，例如 C++ 或 Java，Cg 将会非常容易掌握。Cg 程序相对来说比较短小，通常少于一页，因此即使是一个美工人员或初级程序员也能够

从这本指南中掌握 Cg 语言的要点，并学会编写非常有趣的 Cg 程序。

计算机图形编程会涉及许多数学知识。理解基本的代数和三角几何学将会有助于你理解本书的几个部分。你也需要熟悉基本的计算机图形顶点变换和光照模型所涉及的数学知识。你不需要知道 OpenGL 或者 Direct3D，但是了解其中一个编程接口是非常有帮助的。本书所讲述的所有 Cg 例子都能很好地运行在 OpenGL 或 Direct3D 下，除非有特别的说明。一些例子需要高级的 Cg 功能，这些功能在旧的图形处理器上也许不能工作。

本书的结构

第 1 章将简单介绍 Cg 语言。随后的每一章都是一个简短介绍某个特别 Cg 概念和技术的说明。这本指南的各个章节是互相依赖的，所以我们建议读者按照章节的顺序阅读本书。

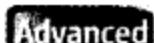
- 第 1 章将展示 Cg 和实时可编程图形硬件的基础。
- 第 2 章将介绍一些最简单的 Cg 程序。
- 第 3 章将解释参数、纹理和表达式。
- 第 4 章将说明如何进行顶点变换。
- 第 5 章将包含如何用 Cg 实现光照模型。
- 第 6 章将描述如何用 Cg 顶点程序实现模型的动画和变换。
- 第 7 章将用 Cg 解释环境贴图。
- 第 8 章将展示如何实现凹凸贴图。
- 第 9 章将讨论一些高级主题：雾、卡通光照、投影聚光、阴影贴图和复合技术。
- 第 10 章将解释目前可用的 Cg 的顶点和片断的简要描述（Profile），并且为提高 Cg 程序的性能提供建议。

这本书将帮助你起步，但是它不可能包含关于 Cg 的你最终想知道的所有东西。这本指南补充了包含在 Cg 开发工具集中的其他文档（例如 Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics）的不足。请参考用户手册和其他 Cg 文档的信息。

编排体例

本书所提供的各种各样的要素都被特意编排以方便阅读理解。代码例子是用 Courier 字体通过轻微增强显示的。变量和关键字用的是粗体的 Courier 字体。

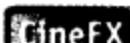
另外，我们用如下的图标来标识一些特殊的主题。



高级主题：为专业读者提供额外的深入见解，如果跳过不看的话不会影响阅读的连续性。



警告：指出一个精妙之处或概念，以避免在编写 Cg 代码的时候出现错误。



CineFX：描述的算法和特性只能在 NVIDIA CineFX 体系结构上应用，或者在拥有类似高级性能的体系结构上应用。



代码提示：给出了一些关于好的编码实践的指导方针。



性能提示：指出了一些使用 Cg 获得最优的图形处理器性能的方法。

尝试例子

我们已经设计了一个附随的软件框架以便你可以直接开始工作，即使你对 OpenGL、Direct3D、C 或者 C++没有一点了解。我们的目标是分离 Cg 语言，以让你可以自由方便地尝试它。当然，当你开始要用 Cg 实现一个现实世界的应用程序的时候，你的项目也许会需要一些关于 OpenGL、Direct3D、C 和 C++的知识。

这个附随的软件框架能够允许你试验本书所带的各种各样的 Cg 例子，而不用担心图形编程接口、C、或 C++代码。这个程序的最新版本可以从本书的支持网站上免费下载。附录 A 解释了如何下载最新版本的 Cg 和附随的实验辅导应用程序。

这些实验辅导应用程序使你能够非常容易地调整本书所带的各种例子，你能

够看到如何改变某个 Cg 例子的同时可以立刻影响渲染的三维结果。如果可能的话，就近找一台支持 Cg 的计算机尝试一下所有的例子程序。通过我们的软件，你可以只写 Cg 程序而不必担心一些细节，例如如何加载一个三维模型和纹理。当你想要知道所有这些细节的时候，可以查看源代码，所有这些都是可以免费下载获得的，这样你就可以看到 Cg 是如何与 C++ 和 OpenGL 或者是 Direct3D 相互作用的了。Cg 工具箱也附带了一些你可以学习的简单的例子程序。

每章的结尾包括了一些建议的习题，以帮助你进一步学习，探索更多的 Cg 知识。

致谢

我们非常感谢在 NVIDIA 的为 Cg 做出贡献的众多同事，感谢他们帮助我们编写了本书。在 2001 年和 2002 年，Bill Mark、Steve Glanville、Mark Kilgard 和 Kurt Akeley 一起工作定义了最初的 Cg 语言。David Kirk、Jensen Huang、Dwight Diercks、Matt Papakipos 和 Nick Triantos 意识到了用于图形处理器的高级语言的需求，并提供了使 Cg 在短短的一年多的时间内就变为现实所需的所有资源。Geoff Berry、Michael Bunnell、Chris Dodd、Cass Everitt、Wes Hunt、Craig Kolb、Jayant Kolhe、Rev Lebaredian、Nathan Paymer、Matt Pharr、Doug Rogers 和 Chris Wynn 开发了 Cg 编译器，标准库函数和运行技术。Sim Dietrich、Ashu Rege 和 Sébastien Dominé 一起研究最初的 CgFX 技术。Chris Seitz 在项目的各个方面给了我们巨大的支持，他帮助我们的地方实在是太多了，以至于无法一一列出，没有他的这些帮助本书将不可能存在。John Spitzer 提供了非常清晰的预见，是 Cg 开发工作的最基本的资源，他和他的小组给了我们巨大的帮助，使本书的出版成为了可能。Sanford Russell 富有感染力的鼓励帮助我们开始编写本书。Cyril Zeller 创建本书所附带的便利的辅导程序框架，并为附录 B 贡献了许多资料。Sim Dietrich 在附录 C 和我们分享了 CgFX 的知识。Kevin Bjorke 帮我们编写了高级章节里的合成技术部分。Teresa Saffaie、Catherine kilkenny 和 Debra Valentine 审阅了本书，并使本书更加易于理解。Caroline Lie、Spender Yuen、Dana Chan、Huey Nguyen 和 Steve Burk 借出了他们的创造力和想象力，帮我们设计和美化了本书的封面和图片。NVIDIA 的演示小组（在 Mark Daly 指导下 Curtis Beeson、Dan Burke、Joe Demers、Eugene d'Eon、Steve Giesler、Simon Green、Daniel Hornick、Gary King、Dean Lupini、Hubert Nguyen、

Bonnie O'Clair、Alexei Sakhartchouk 和 Thant Tessman) 为本书的彩页做出了贡献。

我们非常感谢 Jason Allen、Geoff Berry、Michael Bunnell、Sim Dietrich、Chris Dodd、Gihani Fernando、Simon Green、Larry Gritz、Eric Haines、Wes Hunt、Gary King、Craig Kolb、Jayant Kolhe、Eric Lengyel、Cameron Lewis、Gilliard Lopes、Viet-Tam Luu、Kurt Miller、Tomas Akenine-Möller、Russell Pflughaupt、Matt Pharr、John Spitzer、Nick Triantos、Eric Werness、Matthias Wloka、Cyril Zeller 和那些匿名的审稿人，感谢他们在审阅本书过程中所提出宝贵的评论意见。每一组评论意见都使本书更加清楚和准确。

Microsoft 和 NVIDIA 相互协作在标准硬件光照语言的语法和语义上达成了一致。正是因为这种努力，DirectX9 的高级光照语言 HLSL 和 Cg 其实是同一种语言。我们特此感谢 Craig Peeler、Loren McQuade、Dave Aronson、Anuj Gosalia、Chas Boyd 和 Mike Toelle 的工作。

我们对在北卡大学和斯坦福大学进行硬件光照语言研究工作者表示感谢。显然，Pixar 公司的 RenderMan 光照语言为 NVIDIA 为大量生产的图形硬件开发一种实时语言提供了大量的灵感。Ken Perlin 从事像素流编辑器的工作，他早期在 Cg 编译器方面的测试工作并没有被提及，现在是他获得称赞的时候了。

在硬件方面，我们对 Erik Lindholm 和 Henry Moreton 的基础工作表示感谢，他们设计了 NVIDIA GeForce3 图形处理器中的用户可编程顶点处理引擎。OpenGL 和 Direct3D 对普通可编程顶点处理的支持，以及 Cg 对此的支持都受惠于他们的这项工作。

为了不断推出更快和可编程性更强的图形处理器，NVIDIA 的架构、硬件和软件工程师们所付出的努力工作和奉献曾经是现在也是 Cg 最好的证明。我们对在 NVIDIA 工作的所有工程师所付出的努力表示感谢，他们的努力使得实时可编程光照对所有人都变成了现实。

我们感谢 Addison-Wesley 出版小组使本书成为了现实。我们尤其感谢 Chris Keane 促使我们的手稿成为现实。

最后，我们感谢众多的 Cg 开发者所提供的反馈、错误报告、耐心和热情。

目 录

第1章 简介	1
1.1 什么是 Cg	1
1.1.1 为可编程图形硬件设计的语言	2
1.1.2 Cg 的数据流模型	2
1.1.3 图形处理器的特殊性和中央处理器的通用性	3
1.1.4 Cg 性能的基本原理	3
1.1.5 与传统编程语言共存	4
1.1.6 Cg 的其他方面	5
1.1.7 Cg 程序的有限执行环境	6
1.2 顶点、片段和图形流水线	7
1.2.1 计算机图形硬件的发展史	7
1.2.2 四代计算机图形硬件	8
1.2.3 图形硬件流水线	12
1.2.4 可编程图形流水线	15
1.2.5 Cg 提供了顶点和片段的可编程能力	19
1.3 Cg 的发展史	19
1.3.1 Microsoft 和 NVIDIA 协作开发了 Cg 和 HLSL	21
1.3.2 非交互的着色语言	21
1.3.3 三维图形的编程接口	23
1.4 Cg 环境	24
1.4.1 标准三维编程接口：OpenGL 和 Direct3D	24
1.4.2 Cg 编译器和运行库（Runtime）	26

1.4.3 CgFX 工具箱和文件格式	28
1.5 练习	31
1.6 补充阅读	31
第 2 章 最简单的程序	33
2.1 一个简单的顶点程序	33
2.1.1 输出结构 (Output Structure)	34
2.1.2 标识符	35
2.1.3 结构成员	36
2.1.4 向量	36
2.1.5 矩阵	37
2.1.6 语义	38
2.1.7 函数	39
2.1.8 输入和输出语义是不同的	39
2.1.9 函数体	40
2.2 编译你的例子	42
2.2.1 顶点程序 Profile	42
2.2.2 Cg 编译错误类别	43
2.2.3 依赖 Profile 的错误	44
2.2.4 标准：多重入口函数	45
2.2.5 下载和配置顶点和片段程序	46
2.3 一个简单的片段程序	47
2.4 用顶点和片段示例程序渲染	50
2.4.1 用 OpenGL 渲染一个三角形	51
2.4.2 用 Direct3D 渲染一个三角形	51
2.4.3 获得同样的结果	52
2.5 练习	54
2.6 补充阅读	54
第 3 章 参数、纹理和表达式	55
3.1 变量	55
3.1.1 Uniform 变量	55
3.1.2 const 类型限制符	58

3.1.3 Varying 参数	58
3.2 纹理样本	60
3.2.1 样本对象	61
3.2.2 纹理采样	62
3.2.3 在对一个纹理采样的时候，发送纹理坐标	62
3.3 数学表达式	63
3.3.1 操作符	63
3.3.2 依赖于 profile 的数值数据类型	65
3.3.3 标准库内置的函数	68
3.3.4 二维扭曲	71
3.3.5 重影效果	74
3.4 练习	78
3.5 补充阅读	79
第 4 章 变换	81
4.1 坐标系统	81
4.1.1 物体空间	82
4.1.2 齐次坐标	83
4.1.3 世界空间	83
4.1.4 建模变换	83
4.1.5 眼睛空间	84
4.1.6 视变换	85
4.1.7 剪裁空间	86
4.1.8 投影变换	86
4.1.9 标准化的设备坐标	87
4.1.10 窗口坐标	88
4.2 理论应用	88
4.3 练习	89
4.4 补充阅读	89
第 5 章 光照	91
5.1 光照和光照模型	91
5.2 实现基本的每个顶点的光照模型	93

5.2.1 基本的光照模型	93
5.2.2 一个基本的每个顶点光照的顶点程序	99
5.2.3 每个顶点光照的片段程序	108
5.2.4 单个顶点光照结果	108
5.3 单个片段光照	108
5.3.1 实现每个片段的光照	110
5.3.2 用于每个片段光照的顶点程序	111
5.3.3 用于每个片段光照的片段程序	111
5.4 创建一个光照函数	113
5.4.1 声明一个函数	113
5.4.2 一个光照函数	115
5.4.3 结构	115
5.4.4 数组	116
5.4.5 流控制	118
5.4.6 计算漫反射和镜面反射光照	119
5.5 扩展基本模型	119
5.5.1 距离衰减	120
5.5.2 增加一个聚光灯效果	122
5.5.3 平行光	126
5.6 练习	127
5.7 补充阅读	128
第 6 章 动画	129
6.1 随时间运动	129
6.2 一个有规律搏动的物体	130
6.2.1 顶点程序	131
6.2.2 位移计算	132
6.3 粒子系统	134
6.3.1 初始化条件	135
6.3.2 向量化计算	136
6.3.3 粒子系统的参数	136
6.3.4 顶点程序	137

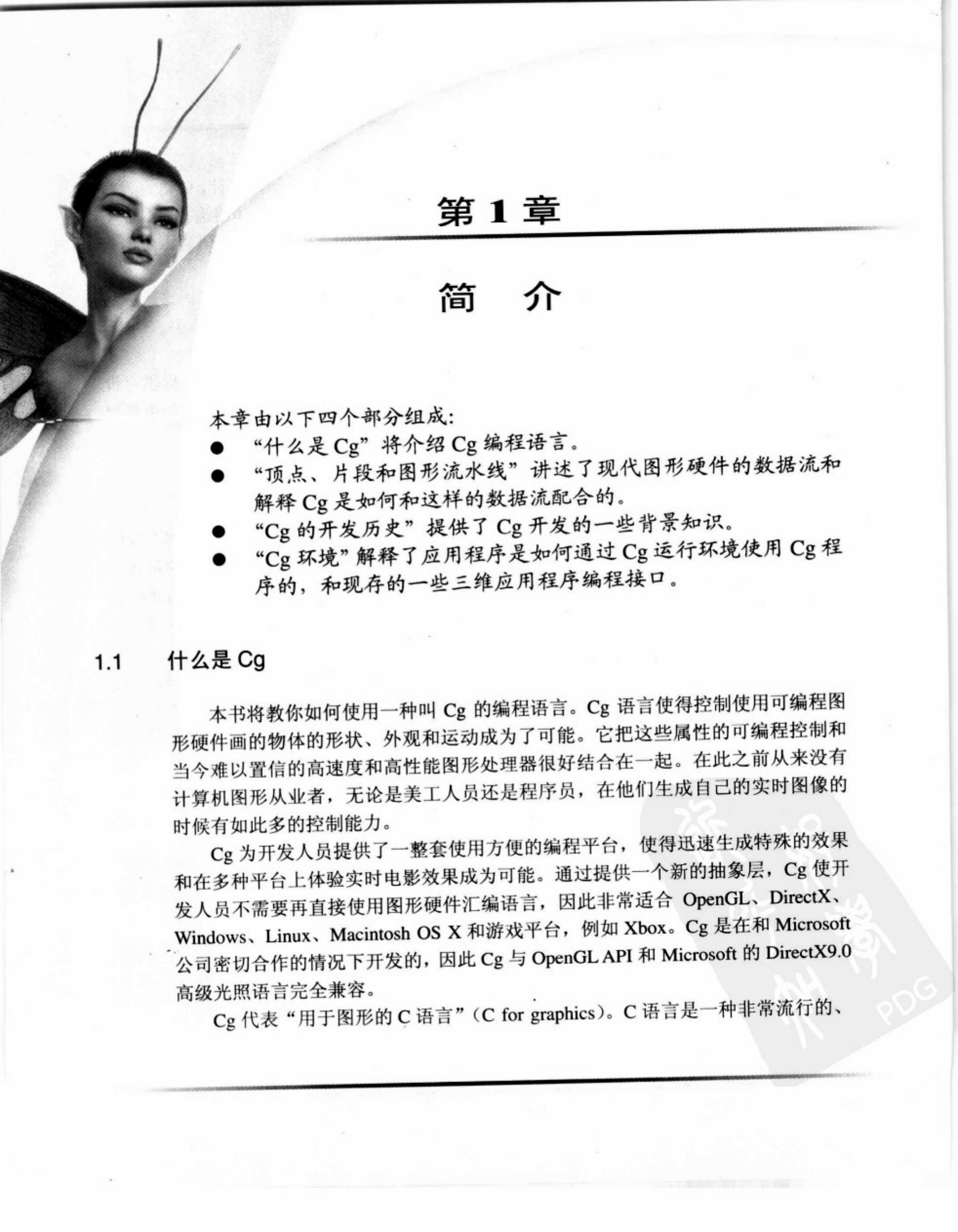
6.3.5 修饰你的粒子系统	139
6.4 关键帧插值	140
6.4.1 关键帧的背景知识	140
6.4.2 插值方法	143
6.4.3 基本的关键帧插值	145
6.4.4 带光照的关键帧插值	146
6.5 顶点混合	147
6.5.1 顶点混合理论	147
6.5.2 在顶点程序中的顶点混合	149
6.6 练习	151
6.7 补充阅读	152
第 7 章 环境映射技术	153
7.1 环境映射	153
7.1.1 立方贴图纹理	154
7.1.2 生成立方贴图	155
7.1.3 环境映射的概念	155
7.1.4 计算反射向量	155
7.1.5 环境映射的一些假设	157
7.2 反射环境映射	158
7.2.1 应用程序指定的参数	159
7.2.2 顶点程序	159
7.2.3 片段程序	163
7.2.4 控制贴图	164
7.2.5 顶点程序与片段程序	164
7.3 折射环境映射	165
7.3.1 折射的物理原理	166
7.3.2 顶点程序	168
7.3.3 片段程序	170
7.4 菲涅耳效果和颜色色散	171
7.4.1 菲涅耳效果	171
7.4.2 颜色色散	172

7.4.3 应用程序指定的参数	173
7.4.4 顶点程序	174
7.4.5 片段程序	175
7.5 练习	177
7.6 补充阅读	178
第 8 章 凹凸映射	179
8.1 凹凸映射一个砖墙	179
8.1.1 砖墙的法向量贴图	180
8.1.2 把凹凸贴图存储成法向量贴图纹理	181
8.1.3 对一个砖墙的简单凹凸映射	183
8.1.4 带镜面反射的凹凸映射	187
8.1.5 凹凸映射其他几何图形	190
8.2 凹凸映射一个砖铺的地板	192
8.3 凹凸映射一个圆环	197
8.3.1 圆环的数学表示	197
8.3.2 凹凸映射的圆环的顶点程序	200
8.4 凹凸映射纹理的多边形网格	202
8.4.1 考察单独一个三角形	202
8.4.2 一些告诫	204
8.4.3 推广到一个多边形的网格	206
8.5 把凹凸映射和其他效果结合在一起	206
8.5.1 印花贴图 (Decal Map)	206
8.5.2 光泽贴图	207
8.5.3 投射自己的几何阴影 (Geometric Self-Shadowing)	207
8.6 练习	208
8.7 补充阅读	209
第 9 章 高级论题	211
9.1 雾	211
9.1.1 均匀的雾	212
9.1.2 雾的属性	213
9.1.3 雾的数学运算	213

9.1.4 直觉化公式	215
9.1.5 用 Cg 创建均匀的雾	216
9.2 非真实性渲染	218
9.2.1 卡通着色	218
9.2.2 实现卡通着色	219
9.2.3 集成在一起	222
9.2.4 卡通着色技术存在的一些问题	224
9.3 投影贴图	224
9.3.1 投影纹理如何工作	225
9.3.2 实现投影纹理贴图	227
9.3.3 投影纹理贴图的代码	228
9.4 阴影映射	231
9.5 合成	233
9.5.1 把输入映射到输出像素	234
9.5.2 基本的合成操作	235
9.6 练习	238
9.7 补充阅读	239
第 10 章 Profile 和性能	241
10.1 Profile 描述	241
10.1.1 DirectX8 的顶点着色器 Profile	241
10.1.2 OpenGL 的基本 NVIDIA 顶点程序 Profile	242
10.1.3 OpenGL 的 ARB 顶点程序 Profile	242
10.1.4 DirectX 9 的顶点着色器 Profile	243
10.1.5 OpenGL 高级 NVIDIA 顶点程序 Profile	243
10.1.6 DirectX 8 的像素着色器 Profile	243
10.1.7 用于 OpenGL 的基本 NVIDIA 片段程序 Profile	244
10.1.8 DirectX 9 像素着色器 Profile	245
10.1.9 OpenGL 的 ARB 片段程序 Profile	245
10.1.10 OpenGL 高级 NVIDIA 片段程序 Profile	245
10.2 性能	246
10.2.1 使用 Cg 标准库	246

10.2.2 充分利用统一参数	247
10.2.3 使用顶点程序与片段程序	247
10.2.4 数据类型和它们对性能的影响	248
10.2.5 充分利用向量化	248
10.2.6 使用纹理来编码函数	249
10.2.7 自由使用重组 (Swizzling) 和取反 (Negation)	249
10.2.8 只对必须着色的像素进行着色	250
10.2.9 简短的汇编代码并不是更快的性能所必须的	250
10.3 练习	251
10.4 补充阅读	251
附录 A Cg 入门	253
A.1 获得本书的配套示例	253
A.2 获得 Cg 工具箱	253
附录 B Cg 运行库	255
B.1 什么是 Cg 运行库	255
B.2 为什么使用 Cg 运行库	255
B.2.1 未来的证明	255
B.2.2 不存在依赖问题	256
B.2.3 输入参数管理	256
B.3 Cg 运行库是如何工作的	256
B.3.1 头文件 (Header File)	257
B.3.2 创建一个环境	258
B.3.3 编译一个程序	258
B.3.4 载入一个程序	258
B.3.5 修改程序的参数	259
B.3.6 执行程序	260
B.3.7 释放资源	260
B.3.8 处理错误	261
B.4 更多的细节	261
附录 C CgFX 文件格式	263

C.1 什么是 CgFX	263
C.2 格式纵览	263
C.2.1 技巧 (Technique)	264
C.2.2 过程 (Pass)	265
C.2.3 渲染状态	265
C.2.4 变量和语义	266
C.2.5 注解 (Annotation)	266
C.2.6 一个 CgFX 文件的示例	267
C.3 支持 CgFX 格式的 Cg 插件	269
C.4 学习更多有关 CgFX 的知识	269
附录 D Cg 关键字	271
附录 E Cg 标准库函数	273
E.1 数学函数	273
E.2 几何函数	277
E.3 纹理贴图函数	278
E.4 导数函数	280
E.5 调试函数	280



第 1 章

简介

本章由以下四个部分组成：

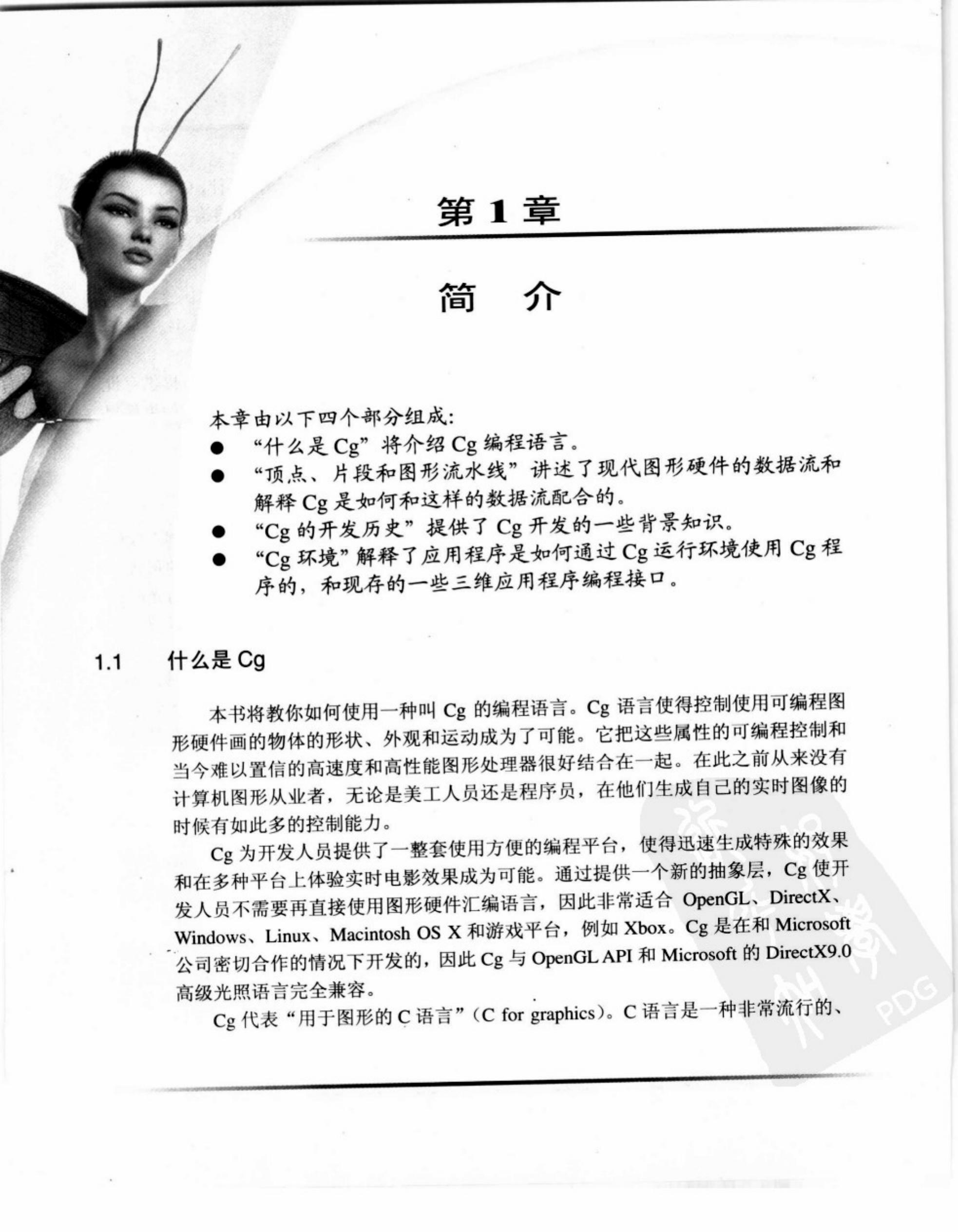
- “什么是 Cg” 将介绍 Cg 编程语言。
- “顶点、片段和图形流水线” 讲述了现代图形硬件的数据流和解释 Cg 是如何和这样的数据流配合的。
- “Cg 的开发历史” 提供了 Cg 开发的一些背景知识。
- “Cg 环境” 解释了应用程序是如何通过 Cg 运行环境使用 Cg 程序的，和现存的一些三维应用程序编程接口。

1.1 什么是 Cg

本书将教你如何使用一种叫 Cg 的编程语言。Cg 语言使得控制使用可编程图形硬件画的物体的形状、外观和运动成为了可能。它把这些属性的可编程控制和当今难以置信的高速度和高性能图形处理器很好结合在一起。在此之前从来没有计算机图形从业者，无论是美工人员还是程序员，在他们生成自己的实时图像的时候有如此多的控制能力。

Cg 为开发人员提供了一整套使用方便的编程平台，使得迅速生成特殊的效果和在多种平台上体验实时电影效果成为可能。通过提供一个新的抽象层，Cg 使开发人员不需要再直接使用图形硬件汇编语言，因此非常适合 OpenGL、DirectX、Windows、Linux、Macintosh OS X 和游戏平台，例如 Xbox。Cg 是在和 Microsoft 公司密切合作的情况下开发的，因此 Cg 与 OpenGL API 和 Microsoft 的 DirectX9.0 高级光照语言完全兼容。

Cg 代表“用于图形的 C 语言”(C for graphics)。C 语言是一种非常流行的、



第 1 章

简介

本章由以下四个部分组成：

- “什么是 Cg” 将介绍 Cg 编程语言。
- “顶点、片段和图形流水线” 讲述了现代图形硬件的数据流和解释 Cg 是如何和这样的数据流配合的。
- “Cg 的开发历史” 提供了 Cg 开发的一些背景知识。
- “Cg 环境” 解释了应用程序是如何通过 Cg 运行环境使用 Cg 程序的，和现存的一些三维应用程序编程接口。

1.1 什么是 Cg

本书将教你如何使用一种叫 Cg 的编程语言。Cg 语言使得控制使用可编程图形硬件画的物体的形状、外观和运动成为了可能。它把这些属性的可编程控制和当今难以置信的高速度和高性能图形处理器很好结合在一起。在此之前从来没有计算机图形从业者，无论是美工人员还是程序员，在他们生成自己的实时图像的时候有如此多的控制能力。

Cg 为开发人员提供了一整套使用方便的编程平台，使得迅速生成特殊的效果和在多种平台上体验实时电影效果成为可能。通过提供一个新的抽象层，Cg 使开发人员不需要再直接使用图形硬件汇编语言，因此非常适合 OpenGL、DirectX、Windows、Linux、Macintosh OS X 和游戏平台，例如 Xbox。Cg 是在和 Microsoft 公司密切合作的情况下开发的，因此 Cg 与 OpenGL API 和 Microsoft 的 DirectX9.0 高级光照语言完全兼容。

Cg 代表“用于图形的 C 语言”(C for graphics)。C 语言是一种非常流行的、

开发于 20 世纪 70 年代的通用编程语言。因为它的流行和简洁的设计，C 语言为后续的几种编程语言提供了坚实的基础。例如，C++ 和 Java 的语法和结构都是基于 C 的。Cg 语言本身也是基于 C 语言的。如果你熟悉 C 语言或者是从 C 语言派生出来的其他语言，Cg 将会非常容易掌握。

另一方面，如果你不熟悉 C 语言，或者甚至没接触过一般的编程语言，但是你非常喜欢计算机图形并且想学一些新东西，无论如何都请继续阅读本书。Cg 程序都非常短小和易于理解。

本章的大部分都在讲述背景知识，将为你理解 Cg 和有效地使用 Cg 提供有价值的上下文关系。另一方面，你将发现通过实践 Cg 是非常容易学习的。如果你想马上进入 Cg 教程，那么可以随时直接跳到第 2 章去。

1.1.1 为可编程图形硬件设计的语言

Cg 不同于 C、C++ 和 Java，因为它非常特别。没有人会用 Cg 写电子制表软件或者文字处理应用程序。相反，Cg 的目标是为使用图形硬件渲染的物体的形状、外观和运动提供可编程控制的能力。从广义上讲，这种类型的语言被称为光照语言。但是，Cg 可以做光照以外的很多事情。例如，Cg 程序可以实现物理模拟、混合和其他非光照任务。

你可以把 Cg 看作是一个非常详细的通过使用可编程图形硬件渲染物体的“处方”。例如，你可以编写一个 Cg 程序来使得物体的外表看起来非常凹凸不平，或者使一个虚拟的人物运动起来。以后，在第 1.3 节中，你将学习许多光照语言的历史和 Cg 是如何融入这个历史中的。

1.1.2 Cg 的数据流模型

除了专门为图形设计以外，Cg 和其他光照语言与传统的编程语言是不同的，因为它们是基于数据流模型的。在这样一个模型里，计算的发生是为了响应流经一序列处理过程的数据。

Cg 程序运行在渲染一幅图像时被处理的顶点和片段上（如果你不知道什么是片段，现在就把它当作像素）。你可以把 Cg 程序想象成一个黑箱，顶点和片段从一边流入，经过某些变换以后，从另一边流出。但是，这个箱子并不是一个真正的黑箱，因为你可以通过编写 Cg 程序来决定它到底起什么作用。

在渲染三维场景的时候，每当一个顶点被处理或者光栅器产生一个片段，你对应的顶点或者片段 Cg 程序就会被执行。第 1.3 节会进一步解释 Cg 的数据流模型。

大部分最新的个人计算机和所有最近的游戏平台都包括一个图形处理器 (GPU)，来专门处理图形任务，例如变换和光栅化三维模型。你的 Cg 程序实际上是在你计算机中的图形处理器上执行的。

1.1.3 图形处理器的特殊性和中央处理器的通用性

无论是个人计算机或者游戏平台是否拥有一个图形处理器，它都必须有一个中央处理器来运行操作系统和应用程序。中央处理器是以多种用途为目的设计的。中央处理器所执行的程序（例如，文字处理器和统计软件包）是用多用途的语言编写的，例如 C++ 或者 Java。

因为图形处理器是专门设计的，它执行图形任务要比多用途的中央处理器快许多，例如渲染三维场景。新的图形处理器可以在一秒钟内处理好几千万的顶点和光栅化几亿或者甚至是几十亿片段。未来的图形处理器会更快。这绝对比中央处理器处理类似数量的顶点和片段的速度快很多。但是，图形处理器不能像中央处理器那样运行任意的多种多样的程序。

图形处理器的这种专用性和高性能是 Cg 为什么存在的原因。多用途的编程语言对处理顶点和片段这项专门的任务来说太自由了。相反，Cg 语言是专用于这项任务的。Cg 同样也提供了一个和图形处理器的执行模型相符合的抽象执行模型。你将在第 1.2 小节学习图形处理器独特的执行模型。

1.1.4 Cg 性能的基本原理

为了支持想象的交互性，一个三维应用程序需要维护每秒显示 15 帧或更多图像的刷新率。通常，我们认为 60 帧/s 或更高的刷新率为实时，在这样的刷新率下与应用程序的交互看起来就像同时发生的。计算机的显示屏也许有 100 万或更多的像素需要重画。对一个三维场景而言，图形处理器一般对屏幕上的一一个像素需要处理好几次，因为场景中的物体总是相互重叠的，或者是为了提高每个像素的表现。这就意味着实时三维应用程序每秒钟需要更新几亿个像素。随同需要处理的像素是由顶点组成的三维模型，这些顶点在被正确地变换之后，

才能被组成多边形、线段和点，然后光栅化成像素。这就需要每秒钟变换上千万的顶点。

而且，在这种图形处理之外还需要中央处理器做相当大的工作来为每个新图像更新显示。事实上，我们同时需要中央处理器和图形处理器专门面向图形的功能。以一个可交互的刷新率和三维应用程序和游戏所要求的质量标准来渲染场景同时需要这两种处理器。这意味着一个开发人员可以用 C++ 写一个三维应用程序或游戏，然后使用 Cg 来充分利用图形处理器的额外的图形处理能力。

1.1.5 与传统编程语言共存

Cg 决不是用来代替现存的多用途编程语言。Cg 是一种辅助语言，是专门为图形处理器设计的。使用 C 或者 C++ 等传统编程语言为中央处理器编写的程序可以使用 Cg 的运行程序（将会在第 1.4.2 小节进行解释）来加载在图形处理器上执行的 Cg 程序。Cg runtime（Cg 运行库）是一个用来加载、编译、操作和设置 Cg 程序在图形处理器上执行的标准子程序集合。应用程序使用 Cg 程序来指示图形处理器如何完成这些可编程的渲染效果，这些效果在中央处理器上不可能达到图形处理器所能获得的渲染速度。

Cg 使得一种专门类型的并行处理成为可能。当你的中央处理器执行一个传统的应用程序的时候，这个应用程序将通过 Cg 编写的程序来协调图形处理器对顶点和片段进行并行处理。如果一个实时光照语言是这样一个主意的话，为什么没有人早一点发明 Cg 呢？这个问题的答案与计算机硬件的发展有关。在 2001 年以前，大部分计算机图形硬件，当然是那种不是很昂贵的用于个人计算机和游戏控制台的图形硬件，采用硬件连线方式实现顶点和片段的处理任务。硬件连线方式是指这些算法是固定在硬件里的，与可以被图形应用程序使用的可编程方式相反。即使是这些硬件连线方式的图形算法可以被图形应用程序通过多种方式设置，应用程序仍然不能重新调整硬件来实现硬件的设计者没有预期到的任务。幸运的是这种情况已经改变了。

图形硬件的设计已经发展了，而且近来的图形处理器中的顶点和片段处理单元是完全可编程的。在可编程图形硬件到来之前，没有理由为它提供一种编程语言。现在，既然这种硬能够被使用了，使得编程使用这种硬件更容易些的需求就变得非常明显。Cg 使得编写图形处理器程序就像 C 使得编写中央处理器程序那样变得更加容易。

在 Cg 存在之前，展示图形处理器可编程能力只能靠低级的汇编语言。汇编语言（例如 DirectX8 的顶点，像素着色器和一些 OpenGL 扩展）所需要的这种密码似的指令语法和硬件寄存器操作，对大部分开发人员来说都是一项痛苦的工作。由于图形处理器技术使得更长和更复杂的汇编语言程序成为可能，这使得对高级语言的需求越来越明显。为了达到优化性能的目的需要大量的低级编程，现在这些程序被升级为优化代码输出和处理冗长乏味的指令调度的编译器。图 1-1 是一个用来表现皮肤的复杂汇编语言片段程序的一小部分。明显地，这个程序非常难理解，特别是引用了许多特殊的硬件寄存器。

```

    . .
    DEFINE LUMINANCE = {0.299, 0.587, 0.114, 0.0};
    TEX H0, f[TEX0], TEX4, 2D;
    TEX H1, f[TEX2], TEX5, CUBE;
    DP3X H1.xyz, H1, LUMINANCE;
    MULX H0.w, H0.w, LUMINANCE.w;
    MULX H1.w, H1.x, H1.x;
    MOVH H2, f[TEX3].wxyz;
    MULX H1.w, H1.x, H1.w;
    DP3X H0.xyz, H2.xzyw, H0;
    MULX H0.xyz, H0, H1.w;
    TEX H1, f[TEX0], TEX1, 2D;
    TEX H3, f[TEX0], TEX3, 2D;
    MULX H0.xyz, H0, H3;
    MADX H1.w, H1.w, 0.5, 0.5;
    MULX H1.xyz, H1, {0.15, 0.15, 1.0, 0.0};
    MOVX H0.w, H1.w;
    TEX H1, H1, TEX7, CUBE;
    TEX H3, f[TEX3], TEX2, 1D;
    MULX H3.w, H0.w, H2.w;
    MULX H3.xyz, H3, H3.w;
    . .

```

图 1-1 汇编语言代码片段

相反，注释清晰的 Cg 程序非常简便、清晰、容易调试和容易再使用。在提供了与低级汇编代码一样的性能的同时，Cg 能给你高级语言（例如 C）的优点。

1.1.6 Cg 的其他方面

Cg 是一种非常简单容易使用的编程语言。这使得它比现代多用途语言更简单，

例如 C++。因为 Cg 是专门用来变换顶点和片段的，它现在还没有包含许多大量软件工程任务所需要的复杂的性能。不像 C++ 和 Java，Cg 不需要支持类和其他在面向对象编程中大量使用的特点。现在的 Cg 实现不提供指针乃至内存分配（虽然以后的实现会提供，并且关键字已经适当保留了）。Cg 当然没有对文件输入和输出进行支持。基本上，这些限制并不是语言上永久的限制，也不是当今高性能图形处理器的标志性功能。由于技术发展到允许图形处理器提供许多通用编程能力，你可以预期到 Cg 将会适当成长。因为 Cg 是紧密基于 C 的，未来对 Cg 的更新可能也会从 C 和 C++ 采用许多语言特性。

Cg 提供了数组和结构。它拥有所有现代语言的流控制：循环、条件和函数调用。

Cg 天生就支持向量和矩阵，因为这些数据类型和相关的数学操作是图形学的基础，并且大部分的图形硬件直接支持向量数据类型。Cg 有一个函数库叫做标准函数库（Standard Library），这个函数库非常适合图形学所需要的各种操作。例如，Cg 标准函数库包括一个反射（reflect）函数用来计算反射向量。

Cg 程序的执行是相对独立的。这意味着对一个特殊的顶点或片段的处理对在同一时间处理的其他顶点或片段没有影响。执行一个 Cg 程序没有任何副作用。在顶点和片段之间缺乏相互依赖使得 Cg 程序非常适合使用高度流水和并行的硬件来执行。

1.1.7 Cg 程序的有限执行环境

当你使用一种语言在现代操作系统上为先进的中央处理器编写一个程序的时候，你可以预期到差不多任何程序只要程序本身正确，就可以正常地编译和执行。这是因为中央处理器是设计用来执行多用途程序的，对这样的程序综合系统拥有太多的资源。

但是，图形处理器是专用的而不是多用途的，并且图形处理器的特征集还在发展中。并不是你用 Cg 写的所有东西都可以被编译然后在一个所给的图形处理器上执行。Cg 包含了一个硬件的概念“配置”（profile），当你在编译一个 Cg 程序的时候，你必须指定其中一个配置。每一个配置对应一种特别的硬件和图形 API 的组合。例如，一个特定的片段配置也许会限制你每个片段不能使用多于四个纹理。

随着图形处理器的发展，Cg 将会支持更多的与性能更强的图形处理器体系结构相对应的 profile。在不久的将来，当图形处理器变得功能越来越全面的时候，profile 将会变得越不重要。但是，现在的 Cg 程序员将需要限制程序来确保它们可

以在现有的图形处理器上编译和执行。通常，未来的 profile 将会是目前的 profile 的超集，因此所有为现在的 profile 编写的程序可以在未来的 profile 上不经修改直接编译。

这种情形看起来会带来一些限制，但是实际上本书所显示的 Cg 程序可以在上千万的图形处理器上工作并生成引人注目的渲染效果。另一个限制程序大小和范围的原因是，你的 Cg 程序越小越有效，它们运行地就会越快。实时图形学常常需要在增长的环境复杂度、动画速度和改善光照中取得平衡。因此通过明智地 Cg 编程最大化渲染效果始终是一件好事。记住 profile 所强加的限制是当前的图形处理器的限制所造成的，而不是 Cg 本身所造成的。Cg 语言本身功能是非常强大的，足够表达目前所有的图形处理器还没能支持的光照技术。随着时间的过去，图形处理器的功能将会发展到 Cg profile 将能够运行令人惊讶的复杂 Cg 程序。Cg 是一种适于现在和未来图形处理器的语言。

1.2 顶点、片段和图形流水线

为了把 Cg 放在一个合适的上下文中，你需要理解图形处理器是如何渲染图像的。这个部分将解释图形硬件是如何发展的，然后将探索当今的图形处理器硬件的渲染流水线。

1.2.1 计算机图形硬件的发展史

计算机图形硬件正在以令人不可思议的速度前进。如图 1-2 所示，一共有三种力量驱使了这种创新的步伐。首先，半导体工业自身每 18 个月就使一个微芯片上的晶体管数目增加一倍。计算机能力的这个固定的加倍速度，在历史上被称为摩尔定律，意味着越来越便宜和越来越快的计算机硬件，并且可以作为我们这个时代的标准。

第二种力量是模拟我们周围的世界所需要的大量的计算。人的眼睛和大脑以一种令人惊讶的速度和敏锐接触来理解三维世界的图像。我们永远不可能使得计算机图形成为现实的替代物。现实世界实在是太真实了。计算机图形的实践者们还是勇敢地站起来接受挑战。幸运地是，生成图像是一个使人为难的并行问题。我们所谓的“使人为难的并行”是指，图形硬件的设计者们可以把创造真实图像

这个问题重复地分解成许多比较小和比较容易解决的问题。然后，硬件工程师们就可以并行地安排大量的晶体管来执行所有的不同工作。



图 1-2 推动图形硬件不断创新的力量

我们所说的第三种力量是，我们拥有全部的对可以被真实地模拟和娱乐的持续不变的期盼。这种力量把计算机硬件资源持续加倍发展的源头连接到模拟比以前更真实的虚拟现实的任务上。

正如图 1-2 所显示的那样，这些敏锐的观察使我们有信心预言计算机图形硬件将会变得更快。这些创新使我们集体对交互性更好和更引人注目的三维体验的欲望更强烈。满足这种需求是激发我们开发 Cg 语言的动力。

1.2.2 四代计算机图形硬件

在 20 世纪 90 年代中期，世界上最快的图形硬件由多块一起工作的芯片来渲染图像，并在屏幕上显示它们。最复杂的计算机图形系统由许多的芯片组成，遍布在几块板子上。随着时间的推移和半导体技术的提高，硬件工程师们把这种复杂的多片设计的所有功能都集成在一个单独的图形芯片里。这一进步导致了巨大的集成和大规模的系统。

获悉现在的图形处理器已经在每个微芯片的晶体管数目上超过了中央处理器，你也许会很吃惊。晶体管的数量是一种对有多少计算机硬件投入到一个微芯片的粗略的度量。例如，Intel 公司在他们的 2.4GHz 奔腾 4 处理器中封装了 5500 万个晶体管，而 NVIDIA 公司在最初的 GeForceFX 图形处理器中使用了超过 1.25

亿个晶体管。当遗留下来的“视频图形阵列控制器”这个术语不再是个人计算机中的图形硬件的精确描述的时候，NVIDIA 在 20 世纪 90 年代后期引入了“图形处理器”这个术语。IBM 在 1987 年提出了视频图形阵列（VGA）硬件。在那个时候，视频图形阵列控制器就是我们现在所谓的帧缓存。这意味着由中央处理器来负责所有像素的更新。现在，中央处理器已经很少直接操作像素了。相反，图形硬件设计师们把像素的更新设计在图形处理器里了。

到目前为止，工业界的观察家们已经确定了四代图形处理器的发展。每一代图形处理器都提供了更好的性能和更进步的可编程能力。每一代图形处理器也都影响和集成了两个主要的三维编程接口：OpenGL 和 DirectX。OpenGL 是一个为 Windows、Linux、UNIX 和 Macintosh 上三维编程服务的开放式标准。DirectX 是 Microsoft 的不断发展的多媒体编程接口，其中包括的 Direct3D 是用来进行三维编程的。

一、图形处理器之前的图形加速

在引入图形处理器之前，像 Silicon Graphics（SGI）和 Evans&Sutherland 这样的公司设计了专用的昂贵的图形硬件。由这些公司开发的图形系统引入了许多现在仍然认可的概念，例如顶点变换和纹理映射。这些系统对计算机图形的开发过程非常重要，但是因为它们实在是太昂贵了，作为为个人计算机和视频游戏控制台设计的单一图形处理器，它们没能在市场上获得成功。现在，图形处理器的功能比以前任何的系统都要强大，而且更加便宜。

二、第一代图形处理器

第一代图形处理器（一直到 1998 年）包括 NVIDIA 的 TNT2、ATI 的 Rage 和 3dfx 的 Voodoo3。这些图形处理器能够光栅化变换前的三角形和使用一或两个纹理。它们还实现了 DirectX6 的特征集。当运行大部分的三维和二维的应用程序的时候，这些图形处理器完全把中央处理器从更新单独的像素中解脱出来了。但是，这一代的图形处理器需要忍受两个明显的限制。首先，它们缺乏变换三维物体顶点的能力；相反，顶点的变换是在中央处理器中进行的。第二，它们只有一个有限的数学操作集合来结合纹理计算光栅化后像素的颜色。

三、第二代图形处理器

第二代图形处理器（1999 年～2000 年）包括 NVIDIA 的 GeForce256 和

GeForce2、ATI 的 Radeon 7500 和 S3 的 Savage3D。这些图形处理器从中央处理器那里承担了顶点变换和光照的工作 (T&L)。在这一代之前，快速的顶点变换是高端工作站区别于个人计算机的关键性能之一。OpenGL 和 DirectX7 都支持硬件顶点变换。虽然用来结合纹理和给像素上色的数学操作在这一代包括了立方图纹理和带符号数学操作，能够完成的工作仍然很有限。换句话说，这一代的图形处理器能够进行更多的设置，但仍然不是真正的可编程。

四、第三代图形处理器

第三代图形处理器（2001 年）包括 NVIDIA 的 GeForce3 和 GeForce4Ti、Microsoft 的 Xbox 和 ATI 的 Radeon 8500。这一代图形处理器提供了顶点编程能力，而不是仅仅提供更多的可设置性。这些图形处理器让应用程序指定一序列的指令来处理顶点，而不是支持由 OpenGL 和 DirectX7 指定的传统的变换和光照模型。虽然提供了相当多的像素级设置功能，但是这些模式还没有强大到可以被认为是真正的可编程。因为这些图形处理器支持顶点编程，但是缺乏真正的像素编程能力，这代图形处理器是过渡期产品。DirectX8 和 OpenGL 的 ARB_vertex_program 扩展都展示了提供给应用程序的顶点级的可编程能力。DirectX8 的像素着色引擎和各个开发商制定的 OpenGL 扩展展示了这代图形处理器片段级的可设置能力。

五、第四代图形处理器

第四代也就是现在的图形处理器(2002 年到现在)包括 NVIDIA 的使用 CineFX 体系结构的 GeForce FX 系列和 ATI 的 Radeon 9700。这些图形处理器同时提供了顶点级和像素级的可编程能力。这个级别的可编程能力使得把复杂的顶点变换和像素着色操作从中央处理器转移到图形处理器成为可能。DirectX9 和各种各样的 OpenGL 扩展功能显示了这些图形处理器的顶点级和像素级的可编程能力。这是这一代的图形处理器使 Cg 感兴趣的地方。表 1-1 列出了选出的 NVIDIA 的图形处理器所代表的不同的图形处理器时代。

表 1-1 各个时代的 NVIDIA 图形处理器的特点和性能

年代	时间	产品名称	制程 (μm)	晶体管 (百万个)	反走样填充率 (百万个)	多边形率 (百万个)	备注
第一代	1998 年后	RIVA TNT	0.25	7	50	6	1
第一代	1999 年前	RIVA TNT2	0.22	9	75	9	2

续表

年代	时间	产品名称	制程 (μm)	晶体管 (百万个)	反走样填充率 (百万个)	多边形率 (百万个)	备注
第二代	1999 年后	GeForce 256	0.22	23	120	15	3
第二代	2000 年前	GeForce2	0.18	25	200	25	4
第三代	2001 年前	GeForce3	0.15	57	800	30	5
第三代	2002 年前	GeForce4 Ti	0.15	63	1200	60	6
第四代	2003 年前	GeForce3 FX	0.13	125	2000	200	7

备注：

1. 两个纹理单元 DirectX 6。
2. 4 倍速 AGP。
3. 固定功能的顶点硬件、寄存器结合器、立方贴图、DirectX 7。
4. 性能、双倍速率内存 (DDR)。
5. 顶点程序、矩形纹理映射、纹理着色器、DirectX 8。
6. 性能、反走样。
7. 巨大的顶点和片段可编程能力、浮点像素、DirectX 9、8 倍速 AGP。

这个表格使用了以下一些术语：

- 制程——在半导体制造过程中使用来制作每个微芯片的最小特征的大小，以微米 (μ) 为单位。
- 晶体管——芯片设计和制造的复杂度的一个近似度量，以百万为单位 (M)。
- 反走样填充率——一个图形处理器填充像素的能力，使用每秒钟填充的 32 位 RGBA 像素的数目来度量的，假设使用两个像素的反走样，一般以百万为单位。斜体的数字表示这个填充率是降级的，因为对应的硬件缺乏真正的反走样渲染。
- 多边形率——一个图形处理器画三角形的能力，一般用每秒钟百万三角形来衡量。

表格的备注部分突出了每种设计中最重要的改进。性能速度不能很好的和其他硬件厂商的设计比较。

未来的图形处理器将近一步推广当前图形处理器的可编程的方方面面，并且 Cg 将使得这种额外的可编程性更加容易使用。

1.2.3 图形硬件流水线

一个流水线是一系列可以并行和按照固定顺序进行的阶段。每个阶段都从它的前一阶段接收输入，然后把输出发给随后的阶段。就像一个在同一时间内，不同阶段不同的汽车一起制造的转配线，传统的图形硬件流水线以流水的方式处理大量的顶点、几何图元和片段。

图 1-3 显示了当今图形处理器所使用的图形硬件流水线。三维应用程序传给图形处理器一序列的顶点组成不同的几何图元：典型的多边形、线段和点。正如图 1-4 中所示，有许多种方法来指定几何图元。

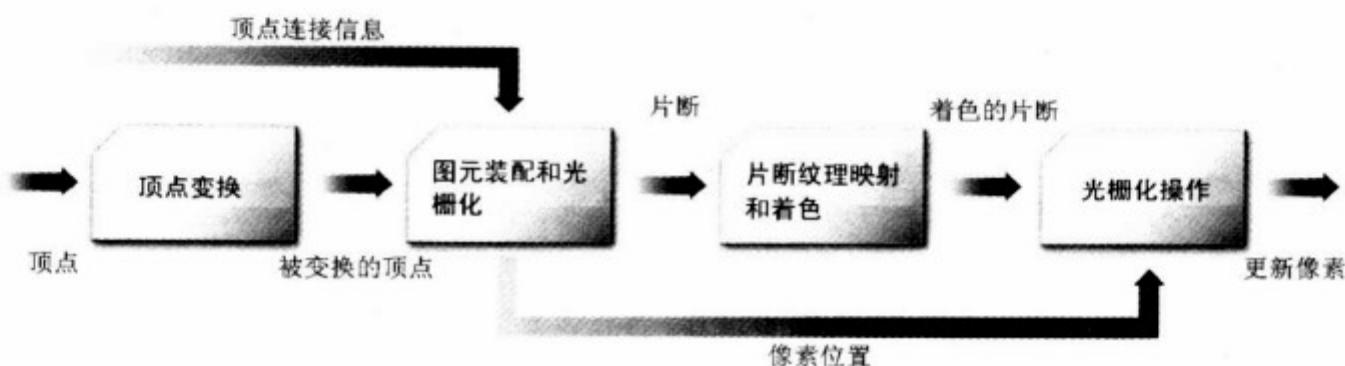


图 1-3 图形硬件流水线

每个顶点除了有位置信息外，还有一些其他属性，例如颜色、第二（反射）颜色、一个或多个纹理坐标集和一个法向量。法向量指示了物体表面在该顶点的方向，它是专门用来计算光照的。

一、顶点变换

顶点变换（Vertex transformation）是图形硬件流水线中的第一个处理阶段。顶点变换在每个顶点上执行一系列的数学操作。这些操作包括把顶点位置变换到屏幕位置以便光栅器使用，为贴图产生纹理坐标，以及照亮顶点以决定它的颜色。我们将在后续的几章解释这些任务中的大部分。

二、图元装配（Primitive Assembly）和光栅化（Rasterization）

经过变换的顶点流按照顺序被送到下一个被称为图元装配和光栅化的阶段。首先，在图元装配阶段根据伴随顶点序列的几何图元分类信息把顶点装配成几何图元。这将产生一序列的三角形、线段和点。这些图元需要经过剪裁到可视平截

体（三维空间中一个可见的区域）和任何有效的应用程序指定的剪裁平面。光栅器还可以根据多边形的朝前或朝后来丢弃一些多边形。这个过程被称为挑选（culling）。

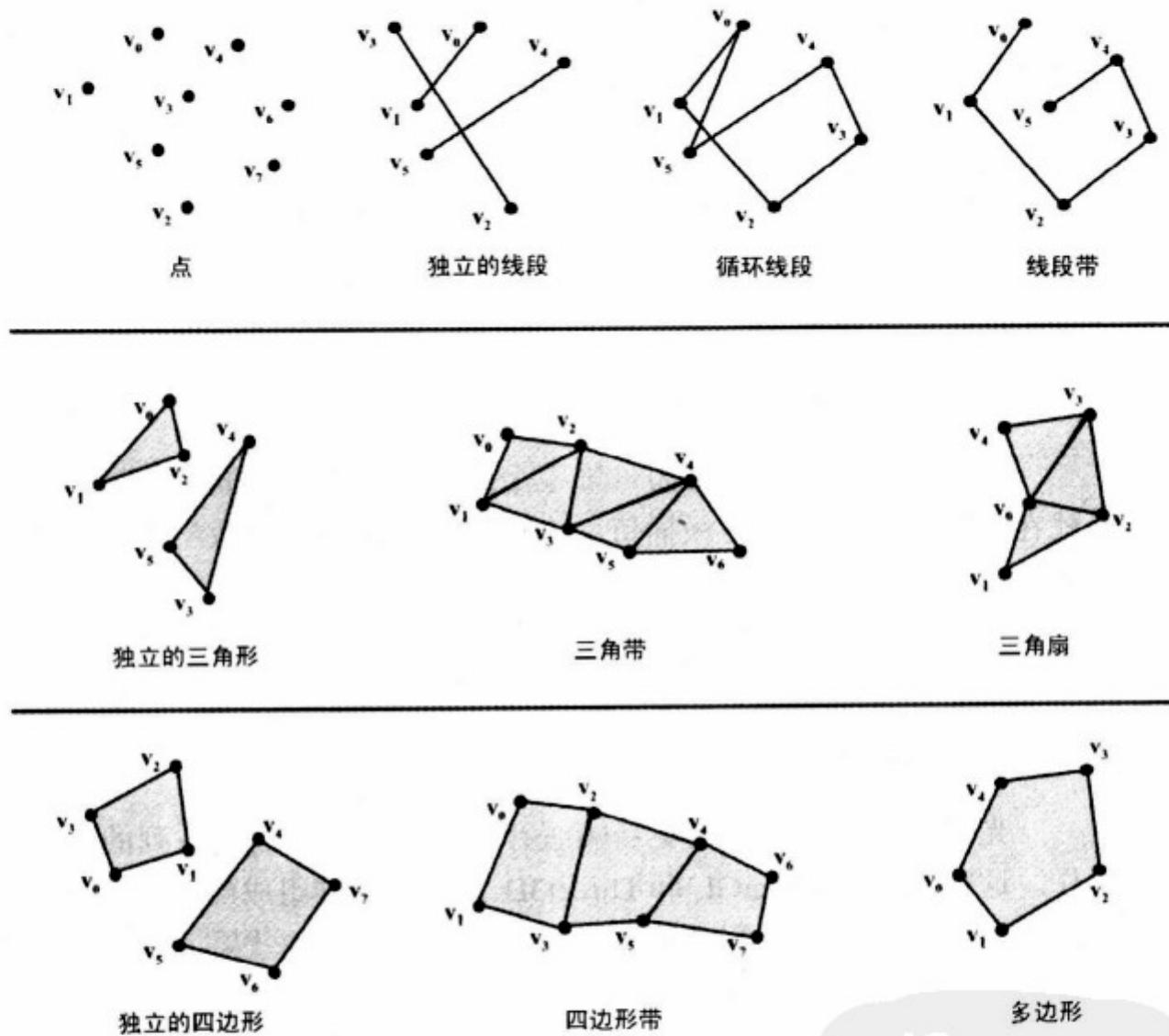


图 1-4 几何图元的类型

经过剪裁和挑选剩下的多边形必须被光栅化。光栅化是一个决定哪些像素被几何图元覆盖的过程。多边形、线段和点根据为每种图元指定的规则分别被光栅化。光栅化的结果是像素位置的集合和片段的集合。当光栅化后，一个图元拥有的顶点数目和产生的片段之间没有任何关系。例如，一个由三个顶点组成的三角形可以占据整个屏幕，因此需要生成上百万的片段。

如果你不能确切地知道片段是什么的话。前面，我们已告诉过你可以把一个片段看成是像素，但是，在现在这个时候，片段和像素之间的区别变得非常重要

了。术语像素是图像元素的简称。一个像素代表帧缓存中某个指定位置的内容，例如颜色，深度和其他与这个位置相关联的值。一个片段（Fragment）是更新一个特定像素潜在需要的一个状态。

之所以术语片段是因为光栅化会把每个几何图元（例如三角形）所覆盖的像素分解成像素大小的片段。一个片段有一个与之相关联的像素位置、深度值和经过插值的参数，例如颜色，第二（反射）颜色和一个或多个纹理坐标集。这些各种各样的经过插值的参数是得自变换过的顶点，这些顶点组成了某个用来生成片段的几何图元。你可以把片段看成是潜在的像素。如果一个片段通过了各种各样的光栅化测试（在光栅操作阶段，这些测试将被简单介绍），这个片段将被用于更新帧缓存中的像素。

三、插值、贴图和着色

当一个图元被光栅化为一堆零个或多个片段的时候，插值、贴图和着色阶段就在片段属性需要的时候插值，执行一系列的贴图和数学操作，然后为每个片段确定一个最终的颜色。除了确定片段的最终颜色，这个阶段还确定一个新的深度，或者甚至丢弃这个片段以避免更新帧缓存对应的像素。允许这个阶段可能丢弃片段，这个阶段为它接收到的每个输入片段产生一个或不产生着色的片段。

四、光栅操作

光栅操作阶段在最后更新帧缓存之前，执行最后一系列的针对每个片段的操作。这些操作是 OpenGL 和 Direct3D 的一个标准组成部分。在这个阶段，隐藏面通过一个被称为深度测试的过程而消除。其他一些效果，例如混合和基于模板的阴影也发生在这个阶段。

光栅操作阶段将根据许多测试来检查每个片段，这些测试包括剪切、alpha、模板和深度等测试。这些测试涉及了片段最后的颜色或深度，像素的位置和一些像素值（例如像素的深度值和模板值）。如果任何一项测试失败了，片段就会在这个阶段被丢弃，而更新像素的颜色值（虽然一个模板写入的操作也许会发生）。通过了深度测试就可以用片段的深度值代替像素的深度值了。在这些测试之后，一个混合操作将把片段的最后颜色和对应像素的颜色结合在一起。最后，一个帧缓存写操作用混合的颜色代替像素的颜色。图 1-5 显示了这一系列的操作。

图 1-5 显示了光栅操作阶段本身实际上也是一个流水线。实际上，所有以前介绍的阶段都可以被进一步分解成子过程。

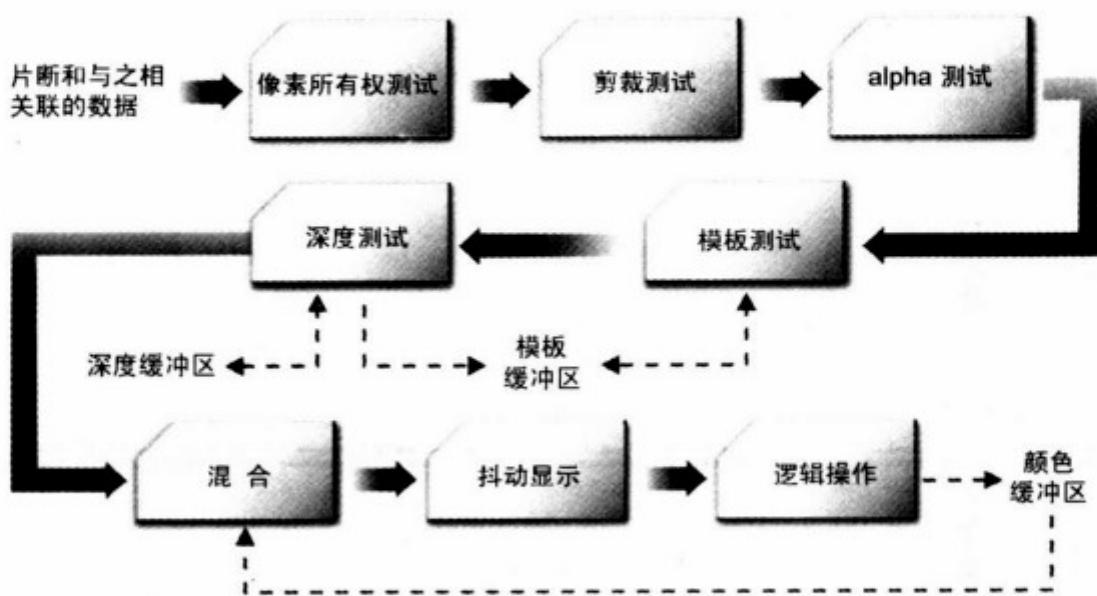
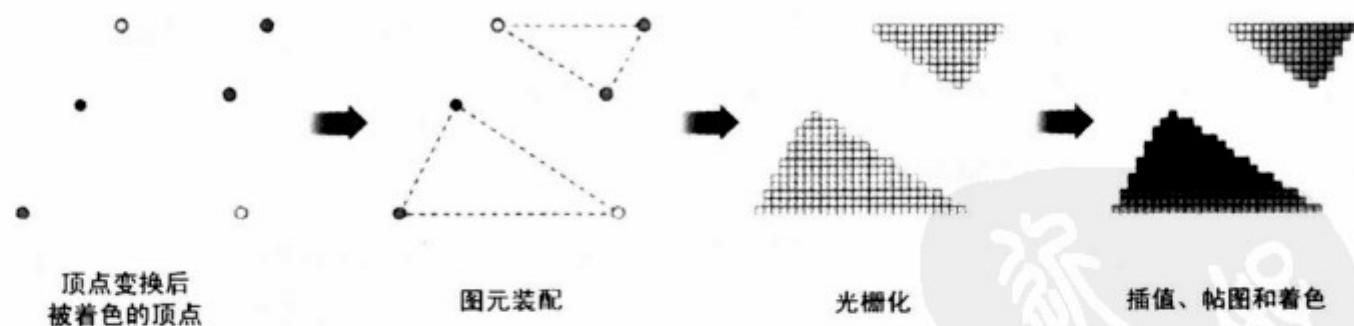


图 1-5 标准 OpenGL 和 Direct3D 光栅操作

五、形象化图形流水线

图 1-6 描写了图形流水线的各个阶段。在本图中，两个三角形被光栅化了。整个过程从顶点的变换和着色开始。下一步，图元装配阶段从顶点创建三角形，如虚线所示。之后，光栅用片段填充三角形。最后，从顶点得到的值被用来插值，然后用于贴图和着色。注意仅仅从几个顶点就产生了许多片段。



1.2.4 可编程图形流水线

当今图形硬件设计上最明显的趋势是在图形处理器内提供更多的可编程性。图 1-7 显示了一个可编程图形处理器的流水线中的顶点处理和片段处理阶段。

图 1-7 比图 1-3 展示了更多的细节，更重要的是它显示了顶点和片段处理被分

离成可编程单元。可编程顶点处理器是一个硬件单元，可以运行你的 Cg 顶点程序，而可编程片段处理器则是一个可以运行你的 Cg 片段程序的单元。

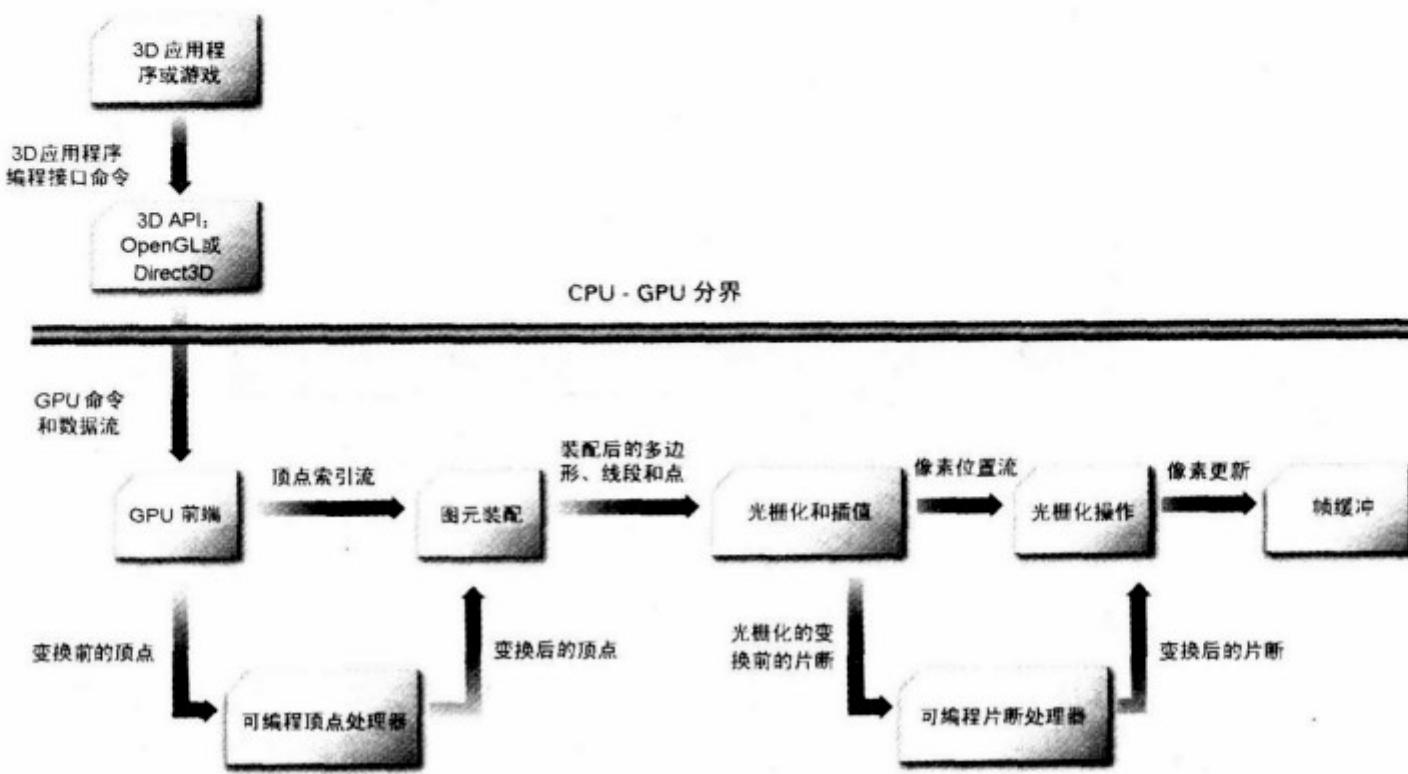


图 1-7 可编程图形流水线

正如第 1.2.2 小节中解释的，图形处理器的设计已经发展了，使得图形处理器中的顶点和片段处理器已经从可配置转变为可编程了。随后的两个小节的描述将介绍可编程顶点和片段处理器的重要功能特征。

一、可编程顶点处理器

图 1-8 显示了一个典型的可编程顶点处理器的流程图。顶点处理的数据流模型从载入每个顶点的属性（例如位置、颜色、纹理坐标等）到顶点处理器开始。然后，顶点处理器重复地取出下一个指令并执行它，直到顶点程序结束。这些指令经常存取几个不同的包含向量值的寄存器库，例如位置、法向量或者颜色。顶点属性寄存器是只读的，包含了应用程序指定的顶点属性集。临时寄存器能够进行读写操作，可以被用来计算中间结果。输出结果寄存器只能进行写操作。程序负责把结果写入这些寄存器。当顶点程序结束的时候，输出结果寄存器包含了最新的变换后的顶点。在三角形组成和光栅化后，每个寄存器的值经过插值后传递给片段处理器。

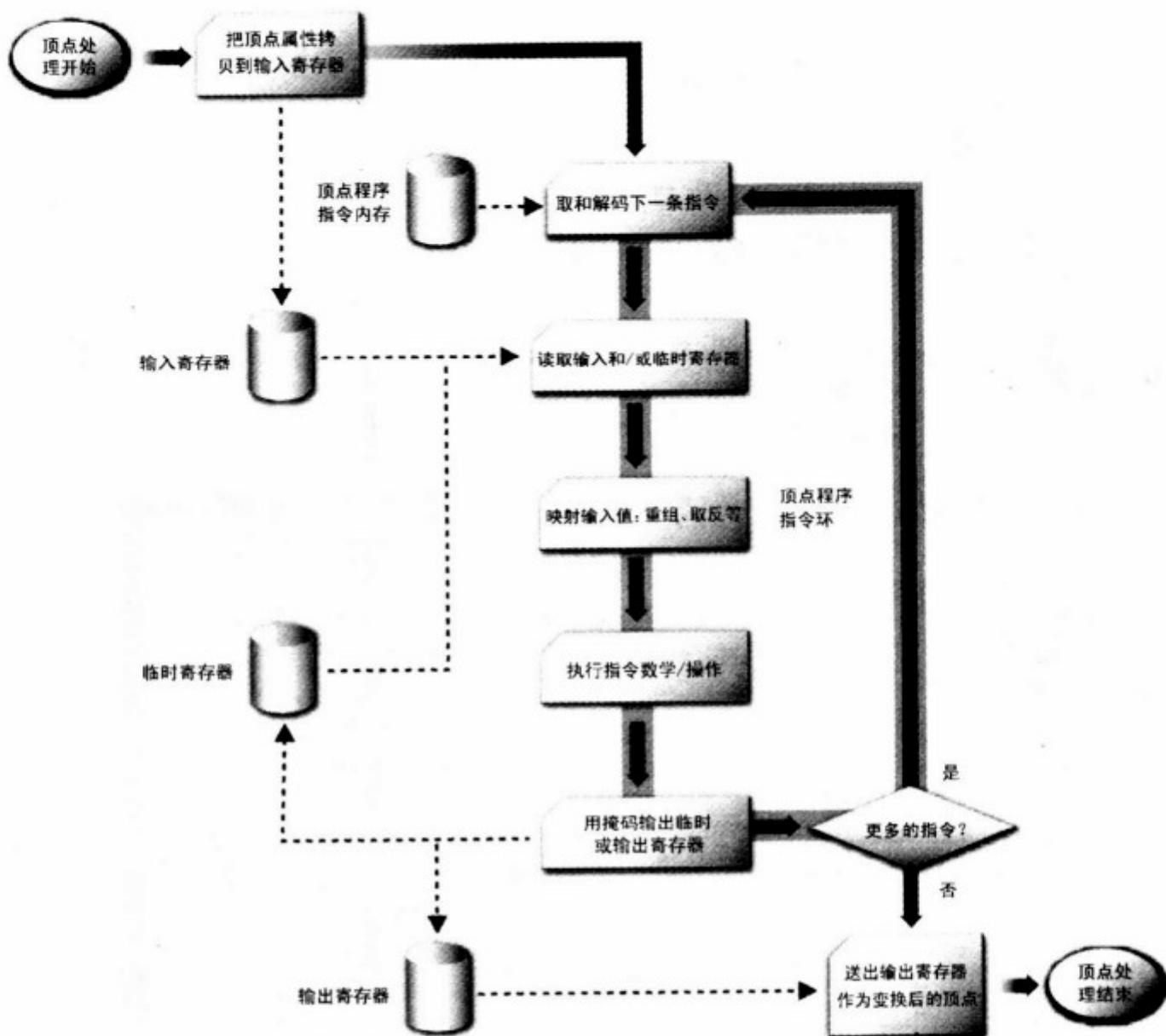


图 1-8 可编程顶点处理器流程图

大部分的顶点处理仅使用了有限的操作。作用于二维、三维或四维浮点向量的向量数学操作是非常需要的。这些操作包括加、乘、乘加、内积、最小值和最大值。硬件对向量取反和移位（任意重新安排向量分量位置的能力）的支持能够推广向量数学指令来提供取反、减法和外积。分量写掩码控制了所有指令的输出。把倒数和平方根倒数与向量乘法和内积分别结合起来，能够提供向量比例除法和向量正规化。指数、对数和三角近似法使光照、雾和几何计算变得容易了。专门的指令能够使得光照和衰减容易计算。

进一步的功能，例如常量的相对地址和流控制对分支和循环的支持，在最新

的可编程顶点处理器中可以得到。

二、可编程片段处理器

可编程片段处理器需要许多和可编程顶点处理器一样的数学操作，但是它们还需要支持纹理操作。纹理操作使得处理器可以通过一组纹理坐标存取纹理图像，然后返回一个纹理图像过滤的采样。

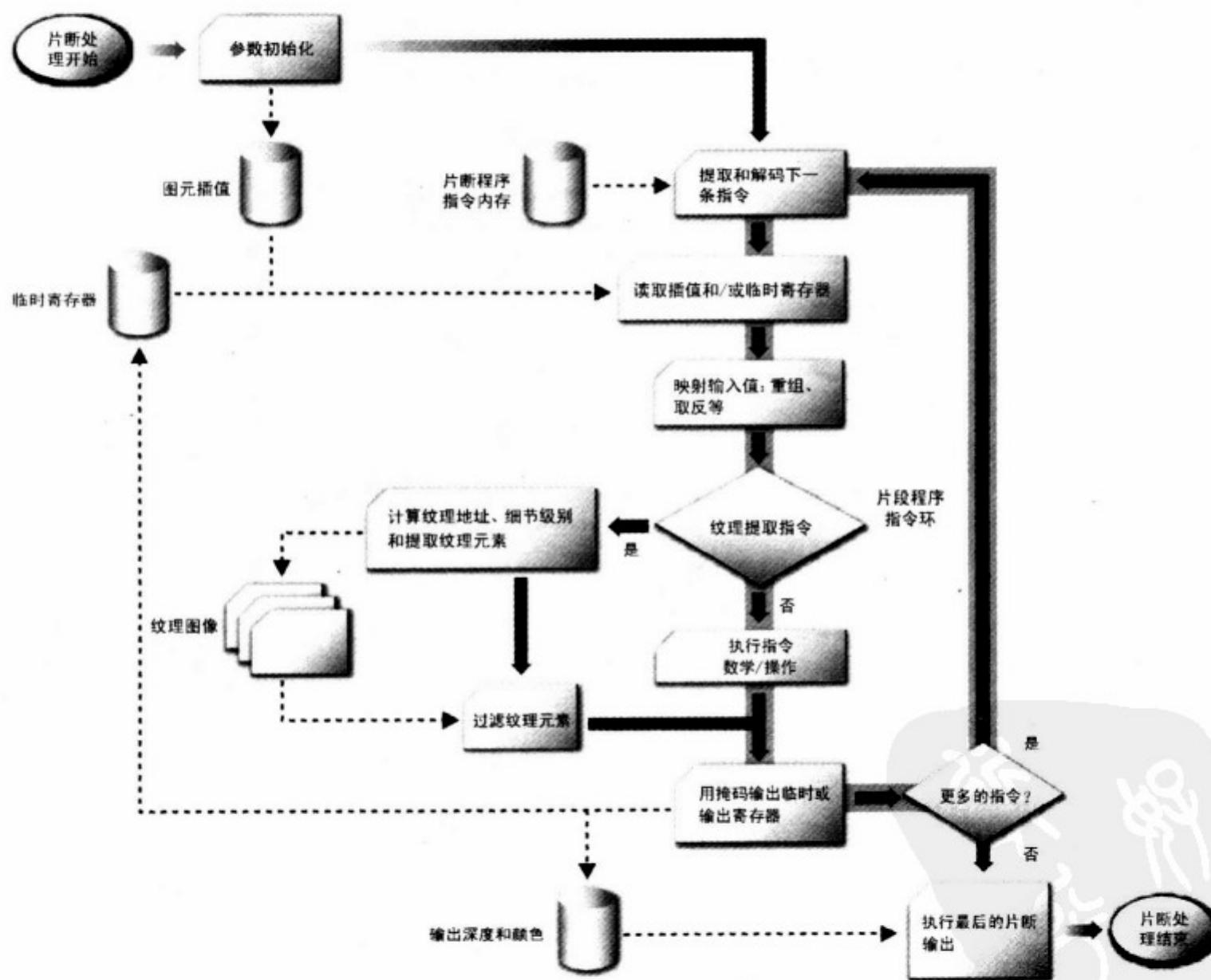


图 1-9 可编程片段处理器的流程图

更新的图形处理器提供了浮点数的支持；旧一点的图形处理器只有有限的固

定点数据类型。即使有了浮点运算，片段操作在使用低精度的数据类型的时候，常常更加有效。图形处理器必须同时处理如此多的片段。当代的图形处理器还没有任意分支功能，但是随着硬件的发展这很可能会改变。**Cg** 通过使用条件赋值和循环操作模拟分支和迭代，仍然允许你编写有分支和迭代的程序。

图 1-9 显示了当代可编程片段处理器的流程图。就像可编程顶点处理器那样，数据流涉及执行一系列的指令直到程序终止。同样，它也有一组输入寄存器。但是，片段处理器的只读输入寄存器包含了从片段图元的顶点参数经过插值获得的片段参数，而不是顶点属性。可读写的临时寄存器存储了中间结果。向只写输出寄存器进行的写操作是片段的颜色和可选的新的深度值。片段程序指令包括纹理获取。

1.2.5 Cg 提供了顶点和片段的可编程能力

你的图形处理器中的这两种可编程处理器需要你——一个程序员，为每个处理器提供一个程序来执行。**Cg** 所提供的是一种语言和一个编译器，能够把你的光照算法翻译成一种你的图形处理器硬件能够执行的形式。通过使用 **Cg**，你能够使用和 C 语言类似的高级语言来编写程序，而不是如图 1-8 和 1-9 所显示的层次上编程。

1.3 Cg 的发展史

Cg 的遗产来自三个来源，如图 1-10 所示。首先，**Cg** 是以多用途的 C 编程语言的语法和语义为基础的。其次，**Cg** 结合了许多来自非实时着色语言（例如 RenderMan 的着色语言）和早期的由学术界开发的硬件着色语言的概念。第三，**Cg** 的图形功能是基于针对实时三维的 OpenGL 和 Direct3D 编程接口的。

图 1-11 显示了多用途的编程语言、三维应用程序编程接口和着色语言对 **Cg** 开发的启发。

前面，我们提到过 **Cg** 是如何借鉴 C 的语法和语义的。通读本书，你会发现 **Cg** 实现了大部分 C 程序员所期待的。**Cg** 在某些方面与 C 不同，**Cg** 是专门为图形处理器设计的，或者性能能够验证这种改变。

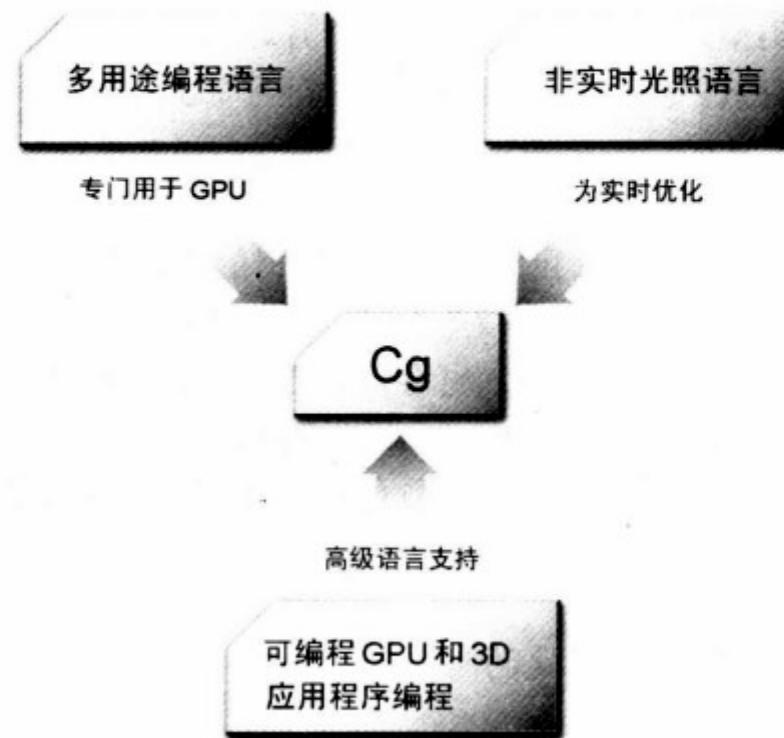


图 1-10 Cg 技术遗产的来源

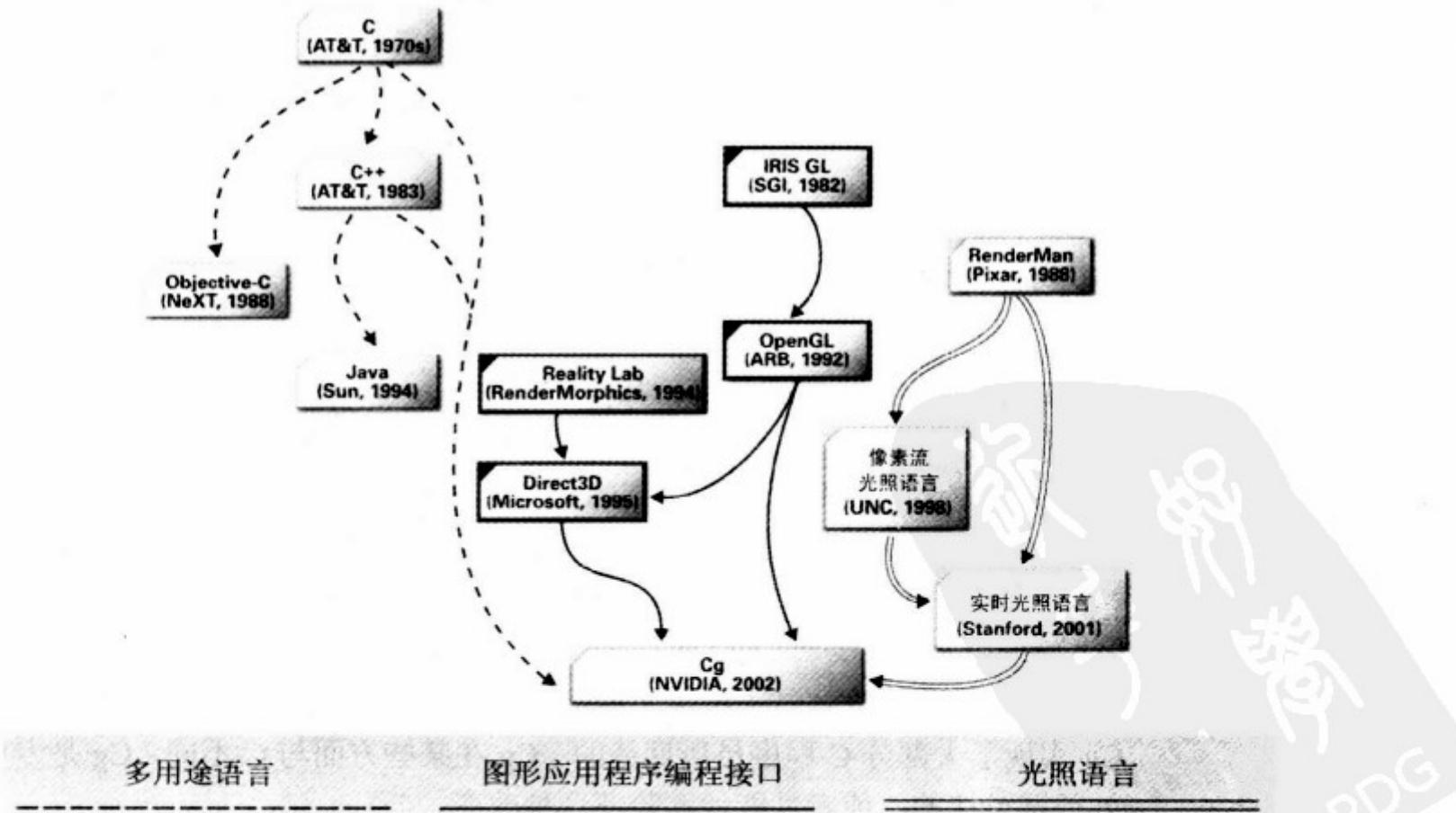


图 1-11 Cg 开发的灵感

1.3.1 Microsoft 和 NVIDIA 协作开发了 Cg 和 HLSL

NVIDIA 和 Microsoft 密切协作开发了 Cg 语言。Microsoft 把他们的实现称为高级着色语言，或简称 HLSL。HLSL 和 Cg 是同样的语言，但是反应了每个公司使用不同的名字来识别语言和它的基础技术。HLSL 是 Microsoft 的 DirectX Graphics 的一部分，DirectX Graphics 是 DirectX9 多媒体框架的一个成员。Direct3D 是 Microsoft 的 DirectX Graphics 的三维部件。Cg 是独立于三维编程接口的，完全和 Direct3D 或者 OpenGL 结合在一起。一个正确编写的 Cg 应用程序可以编写一次，然后可以工作在 OpenGL 或 Direct3D 上了。

这种灵活性意味着 NVIDIA 的 Cg 实现提供了一种方法用来编写能够同时工作在主要的三维编程接口和你选择的任何操作系统上的程序。无论你选择 Windows、Linux、Mac OS X、游戏控制台或嵌入式的三维硬件作为平台，Cg 都能很好的工作。Cg 程序能够在多个硬件厂商提供的硬件上运行，因为 Cg 层是完全在 Direct3D 或 OpenGL 之上的。Cg 程序可以工作在所有主要图形硬件厂商的可编程图形处理器上，例如 3Dlabs、ATI、Matrox 和 NVIDIA。

Cg 语言的这种多厂商、跨 API 和多平台的特征使得它成为可编程图形处理器编写程序的最好选择。

1.3.2 非交互的着色语言

RenderMan 接口标准为非交互着色描述了众所周知的着色语言。Pixar 在 20 世纪 80 年代后期开发了这个语言，使用复杂的着色来为电影和商业制作高质量的计算机动画。Pixar 已经用 RenderMan 接口标准的实现，创造了一整套渲染系统，非实时的渲染器 PRMan (PhotoRealistic RenderMan)。RenderMan 着色语言只是该系统的一部分。

一、着色树 (Shade Trees)

RenderMan 着色语言的创作灵感来之早期被称为着色树的想法。Rob Cook 在 Lucasfilm 有限公司（随后变成了 Pixar），于 1984 年发表了一篇关于着色树的 SIGGRAPH 论文。一个着色树在一个树结构的节点上组织了不同的着色操作。图 1-12 显示了渲染一个铜币表面的着色树。叶子节点是输入到着色树的数据。非叶

子节点代表了简单的着色操作。在渲染的过程中，渲染器计算与给定的表面相关联的着色树来决定表面在渲染的图像中的颜色。为了估算一个着色树，一个渲染器需要执行着色树中最高节点相关联的着色操作。但是，要估算一个给定的节点，渲染器首先要估算该节点的子节点。这个规则将被递归使用来估算整个着色树。一个着色树在一个表面上给定点的估算结果就是这个点的颜色。

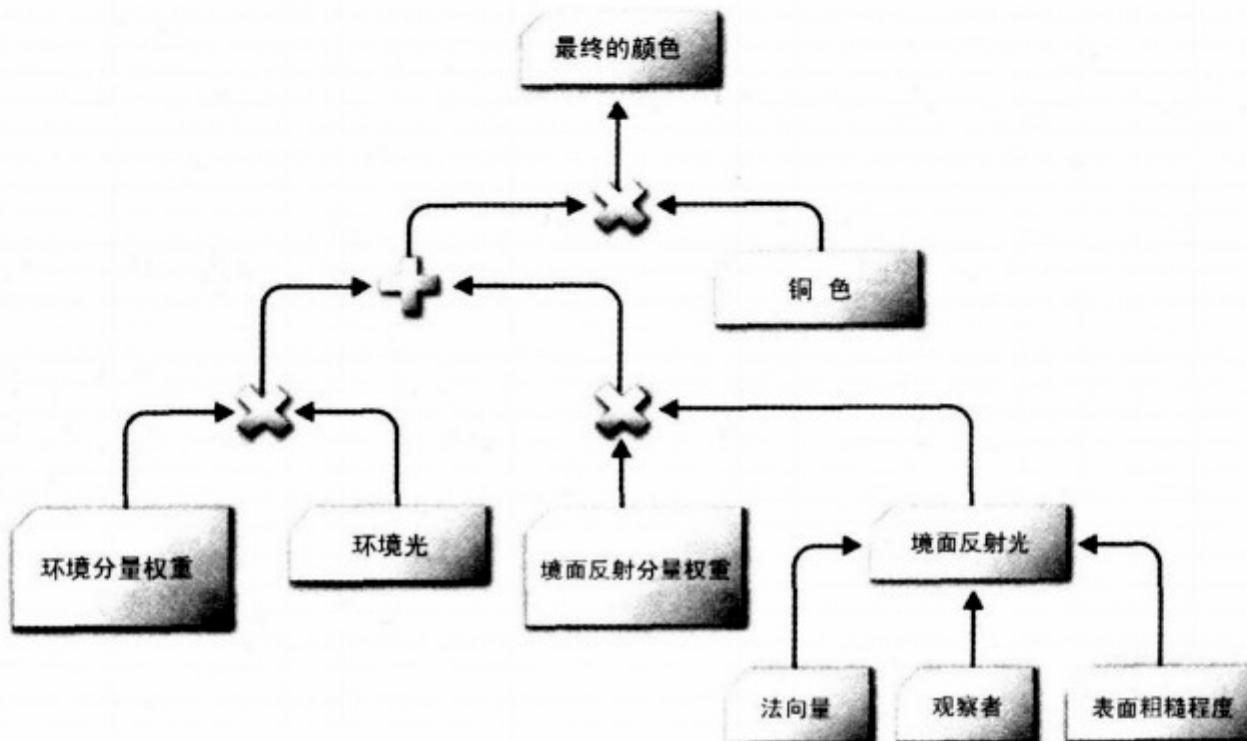


图 1-12 一个基于 Rob Cook 最初的 SIGGRAPH 论文的着色树的例子

着色树的出现是基于这样一个现实，一个预先定义的着色模型对想要渲染的所有物体和场景是远远不够的。

着色树图表对可视化一个数据流的着色操作非常重要。但是，如果着色树非常复杂，它的图表将变得非常不实用。Pixar 和其他地方的研究人员意识到每个着色树是一种受限制的程序。这个现实为一种新的编程语言提供了推动力，这种语言被称为着色语言。

二、RenderMan 着色语言

RenderMan 着色语言是由着色树和在追求照相现实主义中渲染的表面的外观进行广泛的控制需要可编程性这样一个现实而来的。

现在许多在实际生产中使用的非实时渲染器已经对着色语言有了某种类型的

支持。RenderMan 着色语言对非实时渲染来说是最完善和众所周知，它在 20 世纪 90 年代末进行了重要的检修和扩展。

三、受硬件指导的着色语言

当一个任务可以分解成很长的一系列阶段，而且每个阶段的通信被限制在它之前和之后的阶段的时候（也就是说，当它可以被流水操作的时候），一个算法的硬件实现是最有效的。

在 1.2 节中描述的基于顶点和基于片段的流水线是非常服从硬件实现的。但是，被 PhotoRealistic RenderMan 使用的 Reyes 算法不太适合有效的硬件实现，主要是因为高级的几何处理。现在的图形处理器完全依赖于基于顶点和片段的图形流水线。

在 20 世纪 90 年代中期，当北卡罗莱纳大学（UNC）开发一种新的可编程图形硬件构架（被称为 PixelFlow）的时候，UNC 的研究人员开始研究可编程的图形硬件。这个项目促进了计算机图形学研究的一个新的领域服从硬件的着色语言，Marc Olano 和 UNC 的其他工作人员参加了这项工作。不幸地是，PixelFlow 实在是太昂贵了，遭到了商业上的失败。

随后，在 Silicon Graphics 的研究人员开发了一个系统能够把着色器翻译成多过程的 OpenGL 渲染。虽然瞄准 OpenGL 硬件并不是今天的图形处理器使用的可编程方法，OpenGL 着色器系统配合多次的渲染过程来获得一个着色器预期的效果。

斯坦福大学的研究人员包括 Kekoa Proudfoot、Bill Mark、Svetoslav Tzvetkov 和 Pat Hanrahan，开始构造一种专门为第二代和第三代图形处理器设计的着色语言。这种语言被称为斯坦福实时着色语言（RTSL），它可以把用 RTSL 编写的着色器编译成一个或多个 OpenGL 渲染过程。

在斯坦福的研究启发了 NVIDIA 自身的努力来开发一个商业质量的服从于硬件的着色语言。Bill Mark 在 2001 年加入了 NVIDIA，来领导定义和实现这个我们现在称为 Cg 的着色语言。在这个时候，NVIDIA 和 Microsoft 协作就一个通用语言的语法和特征集达成了一致。

1.3.3 三维图形的编程接口

第三个对 Cg 有影响的是 OpenGL 和 Direct3D 这对标准的三维编程接口。这些编程接口对 Cg 的影响还在继续，将会在下一部分进行解释。

1.4 Cg 环境

Cg 只是利用可编程图形处理器实时渲染复杂的三维场景的全部软件和硬件基础结构的一个组成部分。本节将解释 Cg 是如何与实际的三维应用程序和游戏相互作用的。

1.4.1 标准三维编程接口：OpenGL 和 Direct3D

个人计算机上的三维图形在过去的日子（图形处理器出现以前），中央处理器处理所有的渲染一个三维场景所需要的顶点变换和各种像素驱动的任务。图形硬件只是为在屏幕上显示像素提供了一个缓存。程序员必须编写软件实现他们自己的三维图形渲染算法。在某种意义上，所有关于顶点和片段的处理都是完全可编程的。不幸的是，中央处理器实在是太慢了，以至于不能生成引人注目的三维效果。

现在，三维应用程序不再需要使用中央处理器来实现他们自己的三维渲染算法了；而是依赖于 OpenGL 或者 Direct3D，这两种标准三维编程接口，来对图形处理器传达渲染命令。

一、OpenGL

早在 20 世纪 90 年代初，Silicon Graphics 就在一个称为 OpenGL 构架评论委员会 ARB (OpenGL Architecture Review Board) 的协调下开发了 OpenGL，ARB 包括了所有主要的计算机图形系统厂商。最初，OpenGL 只能在功能强大的 UNIX 图形工作站上运行。然后，Microsoft (ARB 的创始成员之一) 实现了 OpenGL，以作为一种在它自己的 Windows NT 操作系统上支持三维图形的方法。Microsoft 随后在 Windows 95 和其他 Microsoft 的桌面操作系统上都增加了对 OpenGL 的支持。

OpenGL 并不是限制在一个单一的操作或窗式系统上。除了支持 UNIX 工作站和 Windows 个人计算机以外，OpenGL 还被 Apple 在它的 Macintosh 个人计算机上支持。Linux 用户既可以使用开放源码的 OpenGL 实现 Mesa，也可以使用硬件加速的实现，例如 NVIDIA 为 Linux 开发的 OpenGL 驱动程序。这种灵活性使得 OpenGL 是工业界里最好的跨平台的三维图形编程接口。

在过去的 10 年里，OpenGL 伴随着图形硬件一起发展。OpenGL 是可扩展的，这意味着 OpenGL 实现可以以一种逐渐增加的方式在 OpenGL 中添加新功能。现在，大量的 OpenGL 扩展功能提供了使用最新的图形处理器功能的能力。这其中包括 ARB 为可编程顶点和片段标准化的扩展功能。当一个扩展功能被确定以后，它们经常会被放入 OpenGL 的核心标准，以使得整个标准可以进步。在写这本书的时候，OpenGL 的当前版本是 1.4。发展 OpenGL 的工作正在不同的 OpenGL ARB 工作组中进行。这个工作同时包括汇编级和高级编程接口。因为 Cg 工作在这些接口以上的一层，它将会继续以一种兼容的方式和未来版本的 OpenGL 协作。

二、Direct3D

Microsoft 大约从 1995 年就开始开发 Direct3D 编程接口了，它是 DirectX 多媒体的一部分。Direct3D 是组成 DirectX 的编程接口之一。Microsoft 提出 DirectX 和 Direct3D 后，在装有 Windows 系统的个人计算机上促进了三维图形的消费市场，特别是游戏的消费。Microsoft 的 Xbox 游戏控制台也支持 Direct3D。对 Windows 上的游戏来说，Direct3D 是最流行的图形 API，由于它的历史和可得到的图形硬件功能非常匹配。

大约每年，Microsoft 都会更新 DirectX，包括 Direct3D，以跟上个人计算机硬件飞快的创新步伐。在写这本书的时候，DirectX 的当前版本是 DirectX9，它已经包括了 HLSL。它是 Microsoft 实现的与 Cg 有相同的语法和结构的一种语言。

三、三维编程接口

几年以前，OpenGL 和 Direct3D 相互竞争，想知道哪种编程接口能够占优势，特别是在使用 Windows 的个人计算机系统上。这种竞争的持续对两种编程接口都值得，每种接口都在性能、质量和功能方面得到了提高。在 Cg 所指的图形处理器可编程性方面，这两种编程接口具有相同的功能。这是因为 OpenGL 和 Direct3D 运行在相同的图形处理器硬件上，并由图形硬件决定了所需要的功能和性能。OpenGL 在功能方面拥有微小的优势，因为硬件厂商通过 OpenGL 能够更好地发布他们整个功能集，虽然厂商指定的扩展确实为开发人员增加了一些复杂度。

现在许多软件开发人员凭着程序员的喜好、历史、他们的市场目标和硬件平台选择三维编程接口，而不是基于技术基础。

Cg 支持任意一个编程接口。你可以编写 Cg 程序使得它们可以同时在 OpenGL

和 Direct3D 编程接口上运行。这对三维内容开发人员来说是巨大的实惠。他们可以把三维内容和用 Cg 编写的程序结合在一起，然后无论最后的应用程序用哪种编程接口来进行三维渲染都可以直接渲染这些内容。

1.4.2 Cg 编译器和运行库 (Runtime)

没有图形处理器能够从文本的方式直接运行 Cg 程序。一个编译过程必须先把 Cg 程序翻译成一种图形处理器可以执行的形式。Cg 编译器首先把你的 Cg 程序翻译成一种应用程序选择的三维编程接口 (OpenGL 或者 Direct3D) 所接受的形式。然后，你的应用程序使用适当的 OpenGL 或 Direct3D 命令把翻译后的 Cg 程序传给图形处理器。OpenGL 或 Direct3D 驱动程序最后把它翻译成图形处理器所需要的硬件可执行的格式。

翻译的细节要依赖图形处理器的功能和三维编程接口的结合。一个 Cg 程序如何编译它的 OpenGL 和 Direct3D 的中间形式依赖于你计算机中的图形处理器的类型和年代。因为图形处理器自身的限制，你的图形处理器也许不能支持某个特别的有效的 Cg 程序。例如，如果你的程序需要存取的纹理单元超过了目标图形处理器支持的数目，你的 Cg 片段程序将不能被编译。

一、对动态编译的支持

当你编译一个用传统编程语言（例如 C 或者 C++）编写的程序的时候，编译是一个非实时的过程。你的编译器把程序编译成可以在中央处理器上直接运行的可执行文件。一旦编译以后，你的程序不需要重新再编译，除非你改变了程序的代码。我们称这种方式为静态编译。

Cg 是不同的，因为它支持动态编译，虽然也支持静态编译。Cg 的编译器不是一个独立的程序，而是一个被称为 Cg 运行库的一部分。使用 Cg 程序的三维应用程序和游戏程序必须与 Cg 运行库链接。使用 Cg 的应用程序通过调用 Cg 运行库程序，通常以字母 cg 作为前缀，来编译和操作 Cg 程序。动态编译允许 Cg 程序为安装在你机器上的图形处理器的精确模型进行优化。

二、三维 API 指定的 Cg 函数库：CgGL 和 CgD3D

除了核心 Cg 运行库之外，Cg 还提供了两个密切相关的函数库。如果你的应用程序使用 OpenGL，则你将使用 CgGL 库来调用正确的 OpenGL 例行过程把翻译

过的 Cg 程序传递给 OpenGL 驱动程序。同样地，如果你的应用程序使用 Direct3D，你将使用 CgD3D 库来调用正确的 Direct3D 例行过程把翻译过的 Cg 程序传递给 Direct3D 驱动程序。通常，你只使用 CgGL 或者 CgD3D 库，但不能同时都使用，因为大部分应用程序使用 OpenGL 或者 Direct3D，而不是都使用。

与包含 Cg 编译器的核心 Cg 运行库相比，CgGL 和 CgD3D 库相对比较小。它们的主要任务是使用正确的 OpenGL 或 Direct3D 的调用，为 Cg 程序的运行帮你进行设置。这些调用把翻译过的 Cg 程序传递给适当的驱动程序，驱动程序将进一步把程序翻译成你的图形处理器可以执行的形式。在极大程度上，CgGL 和 CgD3D 库有许多相似的例行过程。在 CgGL 库中的例行过程以 cgGL 开头；在 CgD3D 库中的例行过程以 CgD3D 开头。

三、如何将 Cg 运行库嵌入到你的应用程序

图 1-13 展示了一个典型的三维应用程序是如何使用 Cg 函数库的。如果你是一名程序员，你可能想学习更多的有关 Cg 运行库和为你的应用程序使用渲染三维图形的三维 API 所特定的函数库。本书的大部分内容将集中在 Cg 语言本身和如何编写 Cg 程序，但是附录 B 有许多关于 Cg 运行库的信息。

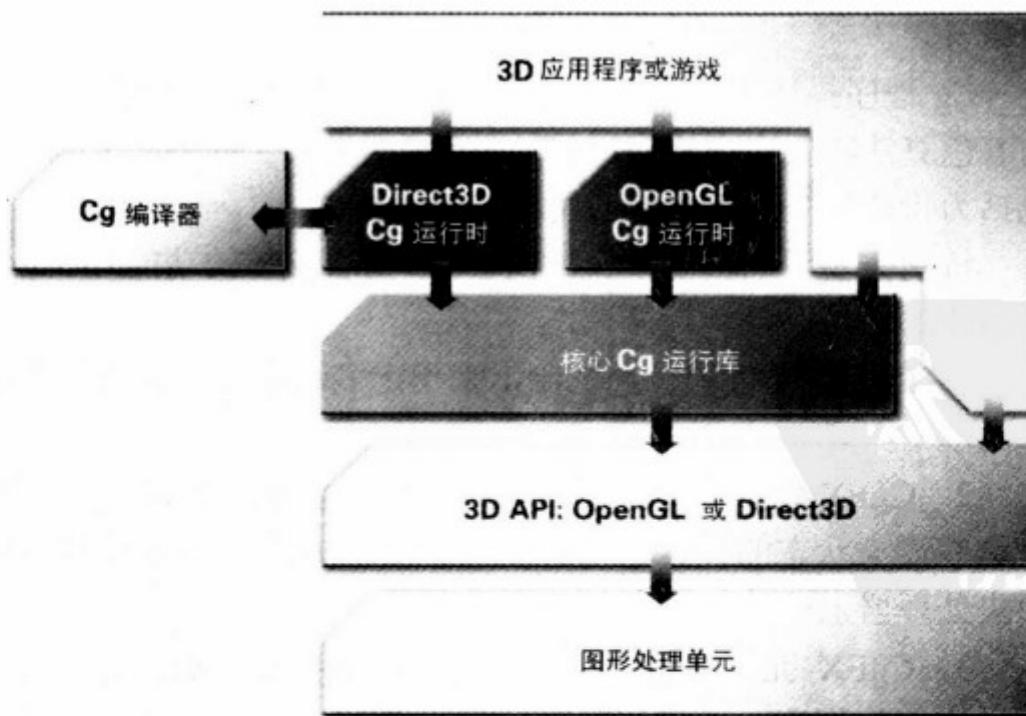


图 1-13 Cg 是如何嵌入到一个标准 Cg 应用程序中的

1.4.3 CgFX 工具箱和文件格式

Cg 程序需要三维模型，纹理和其他数据进行操作。一个 Cg 程序没有与之相关联的数据是没有用的。Cg 程序和数据也需要正确的三维编程接口配置和状态。提供一种方法把渲染一个三维模型所需的所有信息、包括与之相联系的 Cg 程序捆绑在一起是非常有用的。

一、CgFX 提供了什么

CgFX 是一种标准化的文件格式，可以用来表现全部的效果和外观。正如他们开发 Cg 那样，Microsoft 和 NVIDIA 协作开发了 CgFX 格式。CgFX 文件是基于文本的，它的语法是 Cg 的一个超集，并且可以同时包含多个 Cg 程序。**.fx** 后缀用来识别 CgFX 文件。一个 CgFX 文件描述了为一个特殊效果所需要的全部渲染状态：多过程、纹理状态和任意数目的单独的顶点和片段程序可以被用来定义创造一个完整的外观或效果。一个附随的开发工具箱被提供用来使用和分析 CgFX 文件。这个工具箱展示了用户接口到寄主应用程序的钩子，这样知道 CgFX 的应用程序就可以自动的向用户和开发人员提供有意义的控制和语义。

Cg 程序描述了发生在一个单独的渲染过程顶点或者片段处理，但是一些复杂的着色算法需要多个渲染过程。CgFX 提供了一种格式来编码复杂的多过程效果，包括为每个渲染过程指定应用的 Cg 程序。

特别需要明确指出，除了核心 Cg 语言支持的功能外，CgFX 还支持三种附加的功能：

1. CgFX 提供了一种机制，来指定多渲染过程和为某个单独的效果提供可选择的多个实现。
2. CgFX 允许你指定不可编程的渲染状态，例如 alpha 测试模式和纹理过滤。对这些渲染状态的设定采用简单表达式的方式，当效果被初始化的时候，它将在中央处理器上计算。
3. CgFX 允许在着色引擎和着色引擎参数上加注释。这些注释为应用程序包括内容创作程序，提供了附加信息。例如，一个注释可以指定一个着色引擎参数所允许的数值范围。

二、多个着色引擎实例化

CgFX 文件格式封装了对一个给定的着色引擎的多个 Cg 程序的实现。这意味着你可以拥有一个同时为第三代和第四代图形处理器编写的 Cg 着色引擎程序，同时也包含一个简单的程序只支持功能较少的第二代图形处理器。一个应用程序载入 CgFX 文件后，可以在运行的时候基于计算机拥有的图形处理器来决定最合适着色引擎实现。

使用 CgFX 的 Cg 程序多实例化是一种解决在不同时代的功能各异的图形处理器或不同硬件厂商的方法。多实例化还使你专门为一个特殊的三维 API 开发一个 Cg 程序——例如，如果 OpenGL 通过一个扩展发布了额外的功能。专门为 Direct3D、标准 OpenGL 或带扩展的 OpenGL 编写的 Cg 程序可以包含在一个单独的 CgFX 文件里。

三、CgFX 和数字内容创作 (Digital Content Creation, DCC)

CgFX 工具箱由支持全部 CgFX 语法的 CgFX 编译器、加载和操作 CgFX 文件的 CgFX 运行 API 和为主要数字内容创作的应用程序（例如 Alias|Wavefront 的 Maya 和 Discreet 的 3ds max）设计的插件模块组成。图 1-14 显示了这些应用程序是如何使用 CgFX 的。Softimage|XSI 3.0 在它的 Render Tree 中对 Cg 的编译提供了直接的支持。



图 1-14 使用 Cg 和 CgFX 的数字内容创作程序

在 CgFX 之前，没有一个标准的方法可以使数字内容创作应用程序输出包含所有相关着色信息的三维内容，以使这些内容可以被实时渲染。现在，主要的数字内容创作应用程序在它们的内容创作过程中使用 CgFX，并支持 CgFX 文件格式。这意味着 CgFX 可以显著地提高从数字内容创作应用程序到实时游戏和其他三维应用程序的工作流。使用 CgFX，美工人员可以在他们选择的数字内容创作工具中显示和调整 Cg 着色引擎和与之相关联的三维内容，以此来观察他们将如何在一个三维游戏或应用程序中工作。

四、CgFX 是如何嵌入你的应用程序的

图 1-15 显示了一个包含多个实例化的着色引擎的 CgFX 文件是如何被你的应用程序和 Cg 运行库和你选择的渲染 API 协作使用的。本书的大部分内容将集中在 Cg 语言本身和如何编写 Cg 程序，而不是 CgFX，但是在本书的附录 C 可以找到有关 CgFX 文件格式和与此相关的运行 API 的信息。

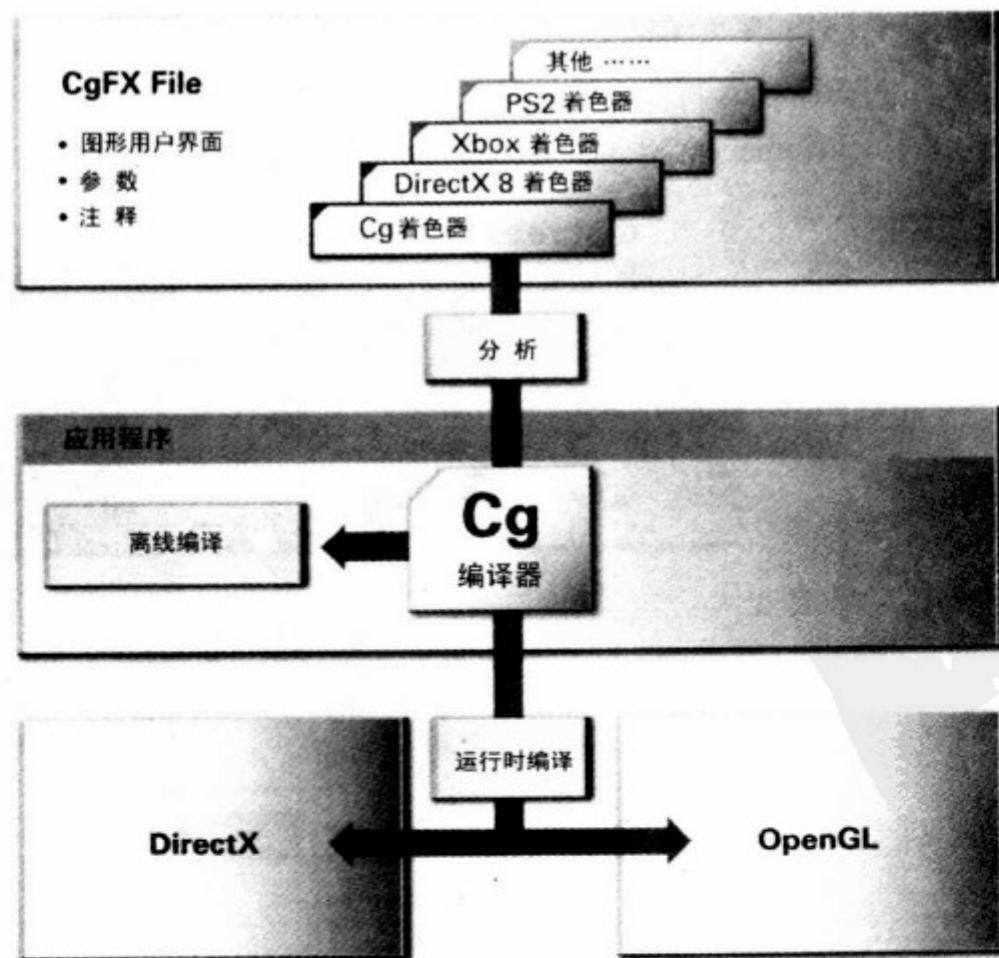


图 1-15 CgFX 是如何嵌入一个标准应用程序的

五、进一步阅读指南

结束了 Cg 编程语言的介绍后，现在你已经准备好接纳其他章节了，这些章节将教你如何编写 Cg 程序。

1.5 练习

每章结尾的练习将帮助你复习学过的知识并帮你练习实际的编程技巧。

1. 回答这个问题：说出两个你可以用来编译 Cg 程序的标准三维编程接口。每个编程接口支持什么操作系统？
2. 回答这个问题：图形流水线的主要阶段有哪些？这些阶段是以什么顺序安排的？
3. 回答这个问题：顶点和片段程序是在哪里嵌入图形流水线的？
4. 回答这个问题：什么是顶点？什么是片段？区别片段和像素。
5. 你自己尝试一下：我们还没有开始写 Cg 程序（我们将会很快在下一章开始），因此休息一下并看一部计算机图形动画长片，例如怪物公司。

1.6 补充阅读

Cg 建立在一大群计算机语言设计、计算机硬件和计算机图形学的概念上。在本指南的上下文中公平对待所有这些贡献是不现实的。我们放在每章结尾的补充阅读部分是想为你学习每个章节中的基础话题提供一些线索。

有关 C 的书有很多。由 Brian Kernighan 和 Dennis Ritchie 编写的 *The C Programming Language* 第三版 (PrenticeHall, 2000 年) 是一部经典著作，该书的作者是 C 语言的发明者。Cg 包括了从 C 和 C++ 而来的许多概念。现在也许实际上关于 C++ 的书比 C 更多了。经典的 C++ 书是由 Bjarne Stroustrup 撰写的 *The C++ Programming Language* 第三版 (Addison-Wesley, 2000 年)，他发明了 C++ 语言。

要想学习 RenderMan 着色语言，请阅读 Steve Upstill 撰写的 *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics* (Addison-Wesley, 1989 年)。Pat Hanrahan 和 Jim Lawson 在 1990 年发表了一篇关于

RenderMan 的题为 “A Language for Shading and Lighting Calculations” 的 SIGGRAPH 论文 (ACM Press)。

Robert Cook 1984 年的题为 “Shade Trees” 的 SIGGRAPH 论文 (ACM Press) 激励了 RenderMan 的开发。

可编程图形硬件和与之相关的语言的开发成为一个非常活跃和富有成效的研究领域已经快 10 年了。UNC 的 Anselmo Lastra、Steven Molnar、Marc Olano 和 Yulan Wang 在 1995 年发表一篇早期的题为 “Real-Time Programmable Shading” 的研究论文 (ACM Press)。UNC 的研究人员还发表了几篇关于他们的可编程 PixelFlow 图形构造的论文。Marc Olano 和 Anselmo Lastra 在 1998 年发表了一篇题为 “A Shading Language on Graphics Hardware: The PixelFlow Shading System” 的 SIGGRAPH 论文 (ACM Press)。

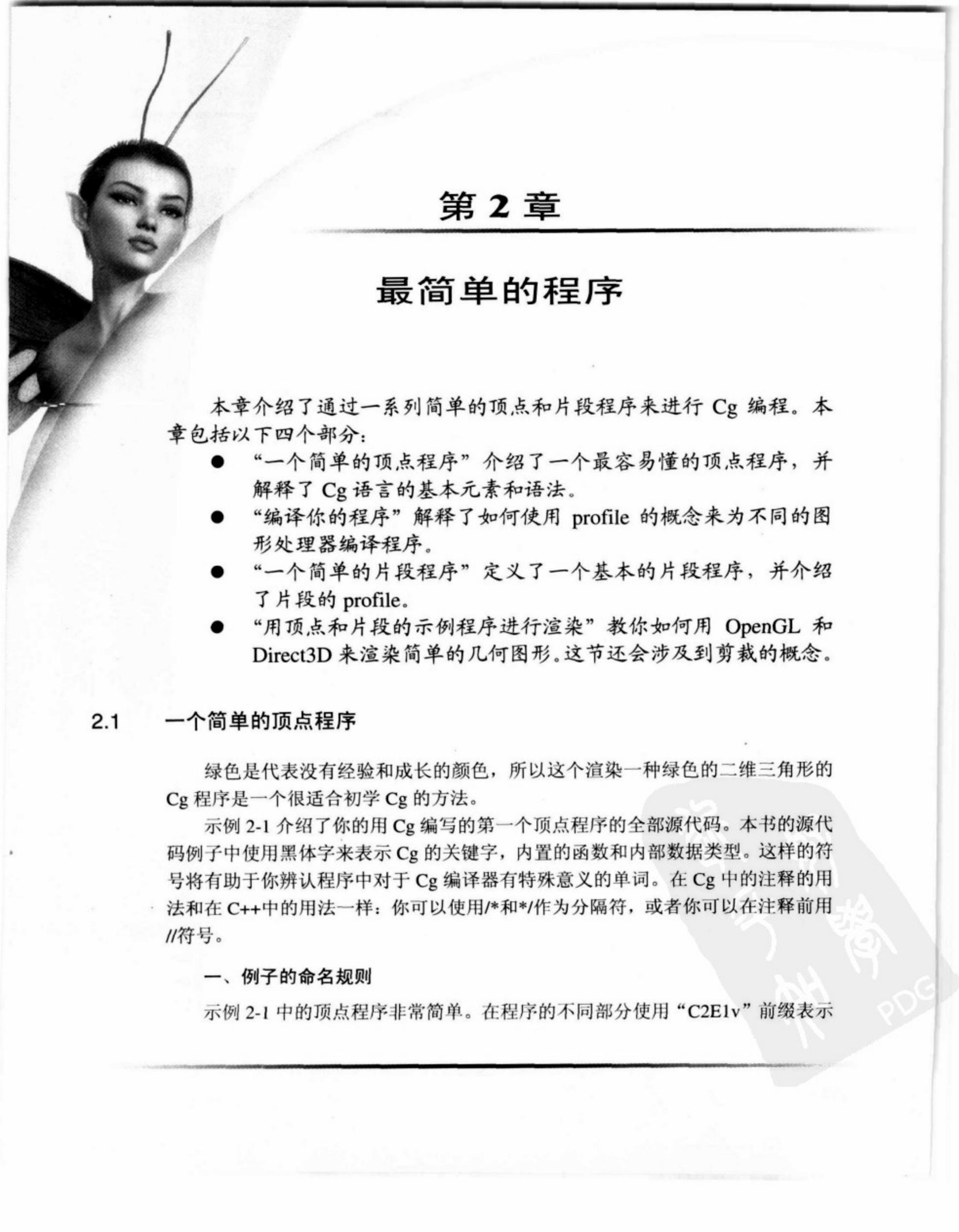
Kekoa Proudfoot、Bill Mark、Svetoslav Tzvetkov 和 Pat Hanrahan 在 2001 年发表了一篇题为 “A Real-Time Procedural Shading System for Programmable Graphics Hardware” 的 SIGGRAPH 论文 (ACM Press)，描述了在斯坦福开发的面向图形处理器的一种着色语言。

由 Eric Haines 和 Tomas Akenine-Moller 撰写的 *Real-Time Rendering* 第二版 (A.K. Peters, 2002 年) 是有关图形硬件和交互技术的非常好的信息来源。

The OpenGL Graphics System: A Specification 是 OpenGL 三维编程接口的文档。学习 OpenGL 编程最好的教材是由 Mason Woo、Jackie Neider、Tom Davis 和 Dave Shreiner 撰写的 *OpenGL Programming Guide: The Official Guide to Learning OpenGL* 第三版 (Addison-Wesley, 1999 年)。www.opengl.org 网站提供了大量关于 OpenGL 的信息。

关于 Direct3D 编程接口的文档可以从 Microsoft 的网站 msdn.microsoft.com 获得。

NVIDIA 在它的开发人员网站 developer.nvidia.com/Cg 上提供了有关 Cg 运行库、CgFX 和 Cg 本身进一步信息。



第 2 章

最简单的程序

本章介绍了通过一系列简单的顶点和片段程序来进行 Cg 编程。本章包括以下四个部分：

- “一个简单的顶点程序”介绍了一个最容易懂的顶点程序，并解释了 Cg 语言的基本元素和语法。
- “编译你的程序”解释了如何使用 profile 的概念来为不同的图形处理器编译程序。
- “一个简单的片段程序”定义了一个基本的片段程序，并介绍了片段的 profile。
- “用顶点和片段的示例程序进行渲染”教你如何用 OpenGL 和 Direct3D 来渲染简单的几何图形。这节还会涉及到剪裁的概念。

2.1 一个简单的顶点程序

绿色是代表没有经验和成长的颜色，所以这个渲染一种绿色的二维三角形的 Cg 程序是一个很适合初学 Cg 的方法。

示例 2-1 介绍了你的用 Cg 编写的一个顶点程序的全部源代码。本书的源代码例子中使用黑体字来表示 Cg 的关键字，内置的函数和内部数据类型。这样的符号将有助于你辨认程序中对于 Cg 编译器有特殊意义的单词。在 Cg 中的注释的用法和在 C++ 中的用法一样：你可以使用/*和*/作为分隔符，或者你可以在注释前用 // 符号。

一、例子的命名规则

示例 2-1 中的顶点程序非常简单。在程序的不同部分使用“C2E1v”前缀表示

“第 2 章，例子 1 顶点程序”。我们使用这种标志方法是为了在书中各章和配套的软件中更容易找到所需要的例子，这种命名规则使得读者更容易找到这本书中的各种不同的例子，但 Cg 语言本身并没有要求这种命名规则，而且在你自己的程序中也可以不使用这种命名规则。

示例 2-1 C2E1v_green 顶点程序

```
struct C2E1v_Output {
    float4 position : POSITION;
    float4 color     : COLOR;
};

C2E1v_Output C2E1v_green(float2 position : POSITION)
{
    C2E1v_Output OUT;

    OUT.position = float4(position, 0, 1);
    OUT.color     = float4(0, 1, 0, 1); // RGBA green

    return OUT;
}
```

如果你很熟悉 C 或者 C++，你很有可能推断出这个程序所做的事情。该程序是将输入的二维顶点的坐标位置赋值到输出的二维顶点的坐标位置，而且将代表绿色的 RGBA（红，绿，蓝，alpha）常量赋给了顶点的输出颜色。

2.1.1 输出结构 (Output Structure)

C2E1v_green 程序是从以下这个声明开始的：

```
struct C2E1v_Output {
    float4 position : POSITION;
    float4 color     : COLOR;
};
```

这个声明是为了一个专用的结构，这个结构被称为输出结构。这个结构包含了一组值代表了一个给定的 Cg 程序的输出（或结果）。

一个用多用途中央处理器语言（例如 C）编写的程序可以执行很多种任务，例如读写文件、向用户请求输入、打印文本、显示图形和网络通信。与之相反，Cg

//在本章剩余部分的输出将被忽略

程序被限制只能输出一组值。对一个给定的 Cg 程序，Cg 程序的输出结构封装了输出值的潜在范围。

Cg 使用与 C 和 C++一样的语法来声明结构。一个结构声明是以关键字 struct 开始的，紧跟着的是结构的名字。大括号里的是结构的定义，它包含了一列结构成员，每个成员都有一个名字和类型。

输出结构与传统的 C 或者 C++结构不同，因为它的每个成员还包括了一个语义项。我们很快将会在第 2.1.6 小节介绍语义的概念。

2.1.2 标识符

当你声明一个结构的时候，在 struct 关键字之后，你需要提供一个标识符或名字；你对声明的每个结构都需要做这项工作。Cg 中的标识符与 C 和 C++中的标识符有相同的形式。标识符包括一系列的一个或多个大写或小写字母，0 到 9 的数字和下划线（_）。例如，Matrix_B 和 pi2 是有效的标识符。一个标识符不能以数字开头，也不能是一个关键字。

标识符不仅可以命名结构，也可以命名类型声明、结构的成员、变量、函数和语义（你很快会学到更多关于这些内容的知识）。示例 2-1 中的其他标识符如下所示：

- C2E1v_green——入口函数名字。
- position——一个函数参数。
- OUT——一个局部变量。
- float4——一个向量数据类型，是 Cg 标准库的一部分。
- color 和 position——结构成员。
- POSITION 和 COLOR——语义。

Cg 采用了与 C 和 C++一样的方式，基于一个标识符的上下文维护了不同的命名空间。例如，标识符 position 标示了一个函数参数和 C2E1v_Output 结构的一个成员。

Cg 中的关键字

许多 Cg 的关键字也是 C 和 C++的关键字，但是 Cg 使用了 C 和 C++中没有出现的其他关键字。随着本书的进展，我们将解释这些关键字中的大部分。附录 D 包含了全部 Cg 关键字的一个列表。与 C 和 C++一样，Cg 的关键字也不能够作为

//在本章剩余部分的输出将被忽略

标识符使用。

2.1.3 结构成员

在一个结构声明的大括号里，你可以发现一个或多个结构成员。每个成员都是一个数据类型，并有一个与之关联的成员名字。

在 C2E1v_Output 结构中，有两个成员：position 和 color。这两个成员都是四元的浮点向量，如它们的类型 float4 所指示的。

2.1.4 向量

C 和 C++ 中最基本的数据类型是标量类型，例如 int 或 float。在 C 和 C++ 中没有原带的向量类型，因为向量只不过是一组标量。因为向量对顶点和片段处理是不可缺少的，并且图形处理器内置了对向量数据类型的支持，Cg 具有向量数据类型。

成员（position 和 color）使用了 float4 数据类型来声明。这个名字并不是 Cg 的保留字，它是定义在 Cg 标准库中的标准类型。不像 C 和 C++，包含 Cg 标准库的声明不需要指定预处理指令（例如 #include）。Cg 自动地包含了大部分 Cg 程序所需要的声明。

你需要依赖于预先定义的向量数据类型，它由 Cg 标准库提供，例如 float2、float3、float4 和其他一些类似地数据类型，以确保你的程序可以最有效地使用你的可编程图形处理器的向量处理能力。

Advanced

Cg 中的向量类型，例如 float3 和 float4，并不是百分之百地和一组 float 相等。例如，float x[4] 和 float4 x 是不同的声明。实际上，这些向量类型是压缩数组（packed arrays）。压缩数组，通常被称为向量，告诉编译器分配压缩数组的元素使得这些变量的向量操作最为有效。如果两个向量以压缩方式存储，可编程图形硬件通常能够在一个单独指令里执行三元或四元的操作，例如乘法、加法和内积。

在传统的编程语言像 C 和 C++ 中压缩数组并不存在。最近的中央处理器指令集例如 Intel 的 SSE2、SSE 和 MMX，AMD 的 3DNow 和 Motorola 的 AltiVec 都有附加的向量指令，但是压缩数组并没有被大部分多用途编程语言

天生支持。然而，Cg 对压缩数组提供了特定的支持，因为向量使得顶点和片段处理更加完整。压缩数组能够帮助 Cg 编译器充分利用可编程图形处理器提供的快速向量操作。

和许多 Cg 的其他方面一样，你如何使用压缩数组将依赖于你所挑选的 Cg profile。例如，压缩数组通常只限于四元或更少。它们通常对向量操作非常有效，例如赋值、取反、绝对值、乘法、加法、线性插值、最大值和最小值。压缩数组的内积和外积运算也非常有效。

另一方面，使用非常量的数组索引存取压缩数组将使效率很低或根本不支持，这将由 profile 决定。例如：

```
float4 data={0.5, -2, 3, 3.14159}; // Initializer,
                                         // as in C
int index = 3;
float scalar;
scalar = data[3];      // Efficient
scalar = data[index]; // Inefficient or unsupported
```

最好的规则就是通过 Cg 内置的向量类型把所有二元、三元或四元的向量（例如颜色、位置、纹理坐标集和方向）声明为压缩数组类型。

2.1.5 矩阵

除了向量类型以外，Cg 天生还支持矩阵类型。这里有一些在 Cg 中声明的矩阵的例子：

```
float4x4 matrix1;    // Four-by-four matrix with 16 elements
half3x2 matrix2;    // Three-by-two matrix with 6 elements
fixed2x4 matrix3;   // Two-by-four matrix with 8 elements
```

你可以用与 C 和 C++ 中使用的初始化数组一样的方法来声明和初始化矩阵。

```
fixed2x4 matrix4;   // { 1.0, 2.0
                      3.0, 4.0,
                      5.0, 6.0}
```

和向量一样，矩阵在 Cg 中也是压缩数据类型，因此使用标准矩阵类型的操作在图形处理器上非常有效地执行。

2.1.6 语义

一个冒号和一个被称为语义的特定词跟随在 C2E1v_Output 结构的 position 和 color 成员后面。在某种意义上，语义是一种黏合剂，它把一个 Cg 程序和图形流水线的其他部分绑定在一起。当 Cg 程序返回它的输出结构的时候，POSITION 和 COLOR 语义指明了各个成员的硬件资源。它们指明了在它们之前的这些变量是如何与图形流水线的其他部分相连接的。

POSITION 语义（在这种情况下，在一个被 Cg 顶点程序使用的输出结构里）是被变换的顶点在剪裁空间的位置。以后的图形流水阶段将使用和这个语义相关联的输出向量作为顶点的变换后的剪裁空间的位置来进行图元装配、剪裁和光栅化。剪裁空间将会在本章稍后的部分介绍，正式地解释将放在第 4 章。目前，你可以把一个二维顶点的剪裁空间位置简单地看成是它在一个窗口中的位置。

在这段上下文中的 COLOR 语义是 Direct3D 所谓的“漫射顶点颜色”和 OpenGL 所谓的“主要顶点颜色”。一个三角形或其他几何图元在光栅化期间的颜色插值依赖于图元每个顶点的颜色。

 不要把一个成员名字和它的语义搞混。在示例 2-1 中，position 成员与 POSITION 语义相关联。但是，正是使用成员名之后的 POSITION 语义而不是成员名本身，使光栅器把 position 成员当成一个位置信息。在下面的输出结构中，成员名 density 和 position 选择得不好，尽管这些名字有误导性，但是 Cg 仍然坚持所有语义的内容。

```
Struct misleadingButLegal {
    Float4 density : POSITION; // Works, but confusing
    Float4 position : COLOR;   // Also confusing
};
```

后面的示例中将会介绍其他的输出语义名字。并不是所有的语义在所有 profile 里都存在，但是在我们的示例中，我们将使用被现有 profile 广泛支持的语义。

你还可以创建自己得语义名，但是在本书中，我们把示例限制在只使用标准的语义集。有关如何使用你自己的语义名的信息可以阅读文档 *Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics*。

2.1.7 函数

在 Cg 中声明函数与在 C 和 C++ 中采用的方式一样。你需要为函数指定一个返回类型（或者如果不返回任何东西，则用 `void`）、它的名字和一个放在括号里用逗号分开的参数列表。在函数声明之后，函数体描述了该函数所要执行的计算。

函数既可以是入口函数，也可以是内部函数。

一、入口函数

一个入口函数定义了一个顶点程序或片段程序，它类似于 C 或 C++ 中的 `main` 函数。一个程序的执行从它的入口函数开始。在示例 2-1 中，入口函数被命名为 `C2Elv_green` 是按如下方式定义的：

```
C2Elv_Output C2Elv_green(float2 position : POSITION)
```

这个函数返回的输出结构 `C2Elv_Output` 在前面已描述。这意味着该函数同时输出一个位置和一种颜色。这些输出拥有结构所定义的语义。

这个函数还接受命名为 `position` 的输入参数。该参数是 `float2` 类型，因此是一个二元的浮点向量。当一个冒号和语义名跟随在一个输入参数名后面的时候，这指明了和输入参数相关联的语义。当 `POSITION` 用作一个输入语义的时候，这就告诉顶点处理器用应用程序指定的函数将要处理的每个顶点的位置来初始化这个参数。

二、内部函数

内部函数是帮助函数，由入口函数或其他内部函数调用。你可以使用 Cg 标准库提供的内部函数，也可以使用自己定义的内部函数。

内部函数忽略任何应用于它们的输入或输出参数或返回值上的语义。只有入口函数使用语义。

2.1.8 输入和输出语义是不同的

图 2-1 显示了 `C2Elv_green` 顶点程序的输入和输出语义流程。

输入和输出语义是不同的，虽然某些语义有相同的名字。例如，对一个顶点程序的输入参数，`POSITION` 指应用程序指定的位置，当应用程序传递一个顶点给

图形处理器的时候,由应用程序赋值。然而,一个输出结构的成员使用了 POSITION 语义就表示要输入给硬件光栅器的剪裁空间位置。

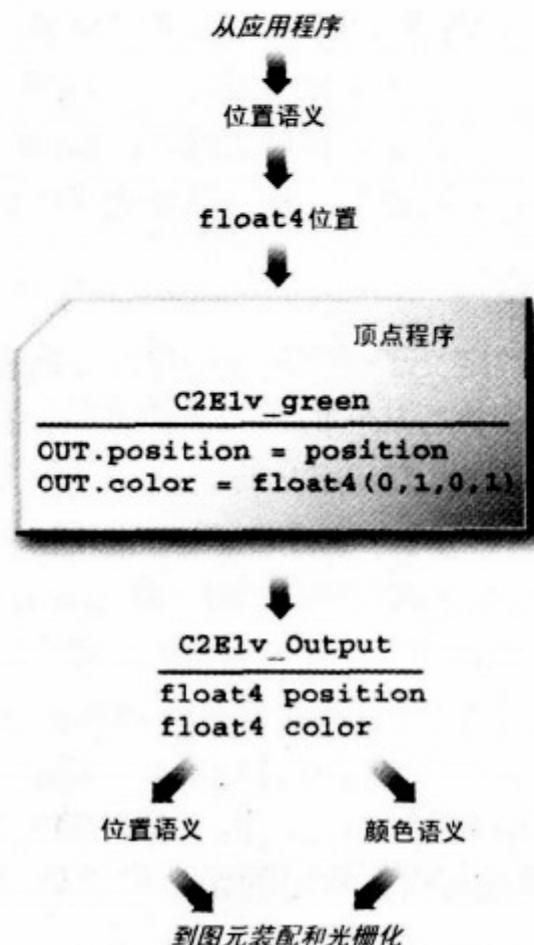


图 2-1 `C2E1v_green` 的输入和输出语义流程

两个语义都命名为 POSITION, 而且实际上每个都是一个位置。但是, 每个位置语义所代表的位置是图形流水线上不同阶段的位置。你的 Cg 顶点程序把应用程序提供的顶点位置变换为适合图元装配、剪裁和光栅化的位置信息。在 `C2E1v_green` 程序里, 这种变换是微不足道的 (顶点位置没有经过改变就被传递了), 但是在本书的以后部分特别是第 4 章和第 5 章, 你将会学到更有用和有趣的方法。

2.1.9 函数体

`C2E1v_green` 函数的主要内容包括在它的函数体内:

```

{
    C2Elv_Output OUT;

    OUT.position = float4 (position, 0, 1);
    OUT.color     = float4 (0, 1, 0, 1); // RGBA green

    return OUT
}

```

因为函数的返回类型是 `C2Elv_Output`, 你必须定义一个这种类型的变量来存储函数所返回的值。我们通常把一个入口函数返回的结构称为输出结构。函数体设置这个结构变量的每个成员并返回这个结构(请注意入口函数的返回类型不需要与入口函数有相同的前缀, 虽然我们在所有的示例中选择使它们有相同的前缀)。

`OUT` 和 `position` 以及 `OUT` 和 `color` 之间的圆点是成员操作符, 它使你能够存取结构中的一个成员。这与 C 和 C++ 存取结构成员采用的方法是一样的。你可以把结构想成是一个容纳多个值的容器。成员操作符使你能够获得包含在这个结构里的值:

```

    OUT.position = float4 (position, 0, 1);
    OUT.color     = float4 (0, 1, 0, 1); // RGBA green

```

首先, 程序要把输入参数 `position` 赋给 `OUT.position`。但是, 输出结构成员 `OUT.position` 是 `float4` 类型(第4章将解释为什么)。表达式 `float4 (position, 0, 1)` 通过分别把第三和第四个成员设成 0 和 1, 把一个二元的位置向量转化为一个四元向量。

其次, 程序要把代表绿色的 RGBA 颜色值赋给 `OUT.color`。要为绿色提供数值信息, 需要构造一个正确的四元颜色向量。`color` 成员的类型是 `float4`, 因为这个颜色是一个 RGBA 颜色。RGBA 中的“`A`”代表“`alpha`”, 它通常表示一个颜色值是如何的不透明或透明。`alpha` 值为 1 表示这个颜色是完全不透明的。

当你像函数那样使用 `float4` 或类似的向量类型名时(例如, `float4 (0, 1, 0, 1)`), 被称为一个构造函数(`constructor`)。构造函数将使用列在括号里的值创建一个指定类型的值。C++ 有构造的概念, 但是 C 没有。在 Cg 中, 构造是提供给向量和矩阵的。

语法 `float4 (0, 1, 0, 1)` 创建一个向量 $\langle 0, 1, 0, 1 \rangle$, 这个向量将赋给类型为 `float4`

的 OUT 中的 color 成员。该向量 $<0, 1, 0, 1>$ 代表绿色，因为按照红、绿、蓝和 alpha (RGBA) 顺序的颜色成员里只有绿色和 alpha 有贡献，而红和蓝没有任何贡献。

```
return OUT
```

最后，`return` 语句返回你初始化的输出结构。`OUT` 中包括的值将按照每个成员所赋予的语义，传给图形流水线的下一个阶段。

2.2 编译你的例子

你使用 Cg 运行库来载入和编译 Cg 程序。当你编译一个程序的时候，除了程序文本之外，你还需要指定两件事情：

- 需要编译的入口函数的名字。
- 编译入口函数所需要的 profile 的名字。

示例 2-1 的入口函数的名字是 `C2E1v_green`。

`C2E1v_green` 是一个顶点程序，因此你需要一个顶点 profile 来进行编译。你需要根据你的应用程序为三维渲染所使用的编程接口和你的图形处理器的硬件能力来选择合适的顶点 profile。

2.2.1 顶点程序 Profile

如表 2-1 所列，有好几个适合的顶点 profile 可以用来编译我们的示例。未来的图形处理器勿庸置疑将支持功能更强的 profile。

表 2-1

Cg 顶点 profile

Profile 名字	编程接口	描 述
arbvp1	OpenGL	基本的多厂商顶点可编程能力（对应 ARB_vertex_program 功能）
vs_1_1	DirectX8	基本的多厂商顶点可编程能力
vp20	OpenGL	基本的 NVIDIA 顶点可编程能力（对应 NV_vertex_program 功能）
vs_2_0 vs_2_x	DirectX9	高级的多厂商顶点可编程能力
vp30	OpenGL	高级的 NVIDIA 顶点可编程能力（对应 NV_vertex_program2 功能）

你的第一个例子非常简单，因此使用表 2-1 中的任何一个 profile 或任何未来

的顶点 profile 编译它都不会有什么问题。当你深入阅读本书后，你将遇到复杂的 Cg 程序需要高级的顶点或片段 profile 来编译。当一个高级的 profile 被使用的时候，我们会非常仔细地指出。本书中的大部分示例是为了能够在大部分 Cg profile 上编译而编写的。

因为你将希望你的 C2E1v_green 示例能够在大量的图形处理器上编译，对你的示例来说 arbvp1 是使用 OpenGL 最好的 profile，vs_1_1 是使用 DirectX8 最好的 profile。存在选择其他 profile 的理由吗？是的，如果你想要使用高级的顶点可编程功能，例如复杂的流控制或快速的硬件指令，这些在基本的 profile 中都不存在。例如，如果你选择 vp30 profile，你可以编写一个 Cg 顶点程序进行一个任意次数的循环。



如果存在一个基本的 profile 足够编译你的 Cg 程序，使用它可以得到广泛的硬件支持。但是，如果你选择一个更高级的 profile，你的 Cg 程序也许可以更加有效地执行，并且你可以使用更加通用的编程习惯来写程序。

那么，什么是更加高级的 profile 的缺点呢？它将会限制你的程序只能在更新的图形处理器上运行。为了在两方面都取得最好的效果——广泛的硬件支持和对最新的硬件支持——你需要为基本的 profile 提供一个可靠的 Cg 程序，为高级的 profile 提供一个更加高级的 Cg 程序。CgFX 格式通过提供一个统一的方法在一个单独源文件中封装对一个给定的渲染效果的多个 Cg 实现来简化这种方法。附录 C 解释了更多关于 CgFX 的知识。

参考附录 B 可以学到一个应用程序如何使用 Cg 运行库来载入和编译 Cg 程序。一般而言，你需要调用一组 Cg 运行例程，需要你提供程序文本、你的入口函数名和你选择的 profile。如果在你的程序中存在错误，编译过程将会失败。你可以获得一个编译错误列表来帮助你改正程序代码。当你的程序成功编译以后，其他的 Cg 运行例程可以帮助你设置所选择的三维编程接口（OpenGL 或 Direct3D），以使用你的程序进行渲染。

2.2.2 Cg 编译错误类别

Cg 程序有两个类别的编译错误：传统的和依赖 profile 的。

传统错误是指由不正确的语法引起的，通常是由于输入错误引起的，或者不

正确的语义引起的，例如使用了错误的参数数目来调用一个函数。这些类型的错误和现在的 C 和 C++ 程序员处理的编译错误基本没有区别。

依赖 profile 的错误是由于在使用 Cg 的时候虽然在语法和语义上是正确地，但是却不被你所指定的 profile 支持。你也许编写了有效的 Cg 程序，但因为你所指定的 profile 它却不能编译。多用途的编程语言没有这种类型的错误。

2.2.3 依赖 Profile 的错误

依赖 Profile 的错误通常是由三维编程接口和你试图用来编译你程序的下层的图形处理器硬件的限制造成的。Cg 中有三类依赖 profile 的错误：能力 (capability)、上下文环境 (context) 和容量 (capacity)。

一、能力

所有当前的片段程序的 profile 都允许纹理存取，但是没有一个当前的顶点 profile 允许这样做。之所以如此原因是非常简单的。目前大部分的图形处理器中的可编程顶点处理器不支持纹理存取。未来的顶点 profile 很可能会允许纹理存取。

根据给定的用来编译的 profile，Cg 不允许你编译一个不可能执行的程序。如果一个顶点 profile 不支持纹理存取，一个依赖于 profile 的能力错误将会产生。硬件或三维编程接口缺乏能力来实现 Cg 允许你表达的任务。

二、上下文环境

虽然非常罕见，一个依赖 profile 的上下文错误是非常基本的。例如，编写的顶点程序没有返回一个绑定到 POSITION 语义的参数。这是因为图形流水线的其余部分假设所有的顶点都有位置信息。

同样地，一个片段 profile 不能像顶点 profile 那样必须返回一个 POSITION 语义。这样的错误是由于在使用 Cg 的时候用了与图形流水线数据流不一致的方式所引起的。

三、容量

容量错误是由于图形处理器能力限制而产生的。有些图形处理器只能在一个渲染过程中执行 4 个纹理存取。其他的图形处理器在一个渲染过程中可以执行任意数目的纹理存取，只是受硬件所支持的片段程序指令数目的限制。如果你在一

个不允许存取超过 4 个纹理的 profile 下存取了超过 4 个纹理，你就会接到一个容量错误。

容量错误可能是最让人感到灰心的，因为从程序的角度来看哪部分的容量超过了是非常不明显的。例如，你也许超过了图形处理器所允许的一个顶点程序的最大指令数目，但是这个事实并不是那么明显。

四、防止错误

有两种方法可以避免令人灰心的错误的发生。一种是使用更高级的 profile。一个 profile 越高级，你碰到 profile 能力和容量限制的可能性越小。随着可编程图形硬件功能的提高，你将越来越不用担心能力和容量限制。

另一种解决办法是培训你自己关于在你的三维应用程序里使用的 profile 在能力、上下文环境和容量限制方面的知识。参考 Cg 工具箱配套的文档可以学习到有关这些限制的知识。

通过了解你所使用的最基本的三维编程接口的限制，可以增强你对依赖 profile 限制的判断能力。参考你的 OpenGL 和 Direct3D 文档，能够帮助你确认 Cg 可能受到依赖 profile 限制的构造。

2.2.4 标准：多重入口函数

当操作系统激活一个程序的进程并调用程序的 main 过程（或者对 Windows 程序来说是 WinMain）的时候，C 和 C++ 程序开始被执行。示例 2-1 是完整的，但是它没有一个过程命名为 main。为什么呢？因为我们命名 C2E1v_green 为入口函数。在这本书中，我们坚持我们的命名规则来区别我们的示例程序，使得它们很容易被找到。然而在你自己的三维应用程序里，你可以用任何你选择的名字来命名入口函数，只要这个名字是一个有效的标识符就行了。

你的三维应用程序通常将会使用一堆 Cg 程序，而不是一个。最少情况下，你也会使用一个顶点程序和一个片段程序，虽然你可以使用固定功能的流水线来处理顶点、处理片段或者如果你愿意可以两者都处理。一个复杂的应用程序也许会有上百个 Cg 程序。因为 Cg 程序可以在运行时编译，你甚至可以在你的应用程序运行的时候，通过程序上的格式化 Cg 程序文本生成新的 Cg 程序。

当然，你仍然可以使用 main 这个名字，当你编译 Cg 源代码的时候如果没有明确指出入口函数名，它是 Cg 程序缺省的入口函数名。

010
101

为了避免混淆，为你的入口程序找一个描述性的名字。如果你所有的入口函数都命名为 main 的话，在一个单独的 Cg 源文件里放置多个入口函数将变得非常困难，即使它是被运行库假设为缺省的入口函数名。

2.2.5 下载和配置顶点和片段程序

在多用途语言里，操作系统调用 main（或者 WinMain）例程，然后程序执行包含在 main 例程里的代码。如果 main 例程返回，程序就会终止。

Cg 程序文本：

```
struct C2Elv_Output {
    float4 position : POSITION;
    float4 color : COLOR;
};

C2Elv_Output C2Elv_green(float2 position : POSITION)
{
    C2Elv_Output OUT;
    OUT.position = float4(position, 0, 1);
    OUT.color = float4(0, 1, 0, 1); // RGBA green
    return OUT;
}
```

Cg Profile: arbvp1 Cg 入口函数: C2Elv_green

ARB_vertex_program 文本：

```
!!ARBvp1.0
PARAM c0 = {0,1,0,1};
ATTRIB v16 = vertex.position;
MOV result.position.xy, v16;
MOV result.color.front.primary.xyz, c0;
END
```

Cg 运行时 API

Cg 编译器

CgGL 运行时 API

OpenGL 驱动支持 ARB_vertex_program

顶点程序硬件微码：

```
01010000101101010101010101
10010010001010111000110110
1011100110100010101010...
```

带可编程顶点处理器的 GPU

图 2-2 编译和加载一个 Cg 程序到图形处理器中

但是，在Cg里你不需要像在C和C++中那样，调用一个程序运行直到它终止。相反，Cg编译器把你的程序翻译成可以下载到硬件的三维编程接口的格式。这就是说将由你的应用程序来调用需要的Cg运行库和三维编程接口例程来下载和设置你的程序，使它可以被图形处理器使用。

图2-2显示了一个应用程序是如何编译一个Cg程序，并把它转换成一种二进制微代码，使得图形处理器的顶点处理器在变换顶点的时候可以直接执行。

一旦加载了一个顶点程序，在你的图形处理器中的顶点处理器每当应用程序提供一个顶点给图形处理器的时候就运行一次。当渲染一个由上千个顶点组成的复杂模型的时候，当前的顶点程序将处理模型中的每一个顶点。这个顶点程序为每个顶点都运行一次。一个顶点程序被加载到可编程顶点处理器中后，可以在任何时间执行。但是，你的应用程序可以根据需要改变当前的顶点程序。

当一个片段程序应用到你图形处理器中的可编程片段处理器的时候，这个概念同样成立。编译你的Cg片段程序，然后使用Cg运行库和你的三维编程接口来下载你的程序并把它绑定为当前的片段程序，来处理由光栅器产生的片段。当你的片段程序绑定以后，三维图元被光栅化为片段，然后当前的片段程序将处理每个生成的片段。片段程序为每个片段运行一次。

通常，三维应用程序同时对图形处理器中的可编程顶点和片段处理器进行编程，以达到某种特殊的渲染效果。因为图形处理器中的可编程顶点和片段处理器的并行性和高度可流水性，这个方法是非常有效的。

2.3 一个简单的片段程序

到目前为止，我们的示例仅涉及了一个顶点程序，C2E1v_green。本部分将介绍一个简单的片段程序，你可以使用它来处理顶点程序的输出。

示例2-2显示了我们第一个片段程序的全部Cg源代码。

示例2-2 C2E2f_passthrough片段程序

```
struct C2E2f_Output {
    float4 color : COLOR;
};

C2E2f_Output C2E2f_passthrough(float4 color : COLOR)
{
```

```

    C2E2f_Output OUT;
    OUT.color = color;
    return OUT;
}

```

这个程序甚至比示例顶点程序 C2E1v_green 更加简单。实际上，这个程序什么也没有做。这个程序只是把光栅器赋给每个片段的插值颜色不经改变直接输出。如果片段通过了各种光栅操作，例如剪切和深度测试，图形处理器的光栅操作硬件就使用这个颜色来更新帧缓存。

这是 C2E2f_passthrough 返回的输出结构：

```

struct C2E2f_Output {
    float4 color : COLOR;
};

```

片段程序与顶点程序相比有一个简单的输出结构。一个顶点程序必须输出一个位置和返回一种或多种颜色、纹理坐标集、和其他每个顶点的输出。而一个片段程序必须简化这些输出到一个单独的颜色用来更新帧缓存(在某些高级的 profile 中，片段程序可也以输出其他数据，例如深度值)。在一个片段程序中，赋给 color 成员的 COLOR 语义指明这个成员是用来更新帧缓存的颜色。

C2E2f_passthrough 声明的入口函数是：

```
C2E2f_Output C2E2f_passthrough(float4 color : COLOR)
```

这个函数返回只有一种颜色的 C2E2f_Output 输出结构。这个函数接受一个被绑定到 COLOR 输入语义并命名为“color”的四元向量。一个片段程序的 COLOR 输入语义是片段的经过光栅器基于图元的顶点的颜色插值的颜色。

C2E2f_passthrough 的函数体如下所示：

```

{
    C2E2f_Output OUT;
    OUT.color = color;
    return OUT;
}

```

在用 C2E2f_Output 输出结构类型定义了一个 OUT 变量以后，这个程序在输出结构中把片段的插值颜色(唯一的输入参数)赋给最后的片段颜色。最后，这个程序返回 OUT 结构。

片段程序 profile

正如你需要一个 profile 来编译 C2E1v_green 示例那样，你也需要一个 profile 来编译 C2E2f_passthrough 示例。但是，用来编译 C2E1v_green 示例的 profile 是给顶点程序的。要想编译 C2E2f_passthrough，你必须选择一个适合的片段 profile。

表 2-2 列出了各种用来编译片段程序的 profile。

表 2-2

Cg 片段 profile

Profile 名字	编程接口	描述
ps_1_1		
ps_1_2	DirectX8	基本的多厂商片段可编程能力
ps_1_3		
fp20	OpenGL	基本的 NVIDIA 片段可编程能力（对应 NV_texture_shader 和 NV_register_combiners 功能）
arbfpl	OpenGL	高级的多厂商片段可编程能力（对应 ARB_fragment_program 功能）
ps_2_0	DirectX9	高级的多厂商片段可编程能力
ps_2_x		
fp30	OpenGL	高级的 NVIDIA 片段可编程能力（对应 NV_fragment_program 功能）

和早些的顶点示例程序一样，第一个片段示例程序是如此简单，以至于你可以使用表 2-2 中任何一种 profile 来编译 C2E2f_passthrough 示例。Cg 有一个命令行编译器，叫做 cgc，是 Cg 编译器的简称。在运行时的动态编译功能极为强大是我们极力推荐的。但是，当你编写 Cg 程序的时候，你通常想要不运行你的三维应用程序，就检验编译结果是否正确。想要在编写程序的时候查找 Cg 编译错误，试着运行一下 cgc。它将试着编译被你的应用程序使用的 Cg 程序文件，作为你正常的应用程序创建过程的一部分。在一个集成开发环境（Integrated Development Environment, IDE，例如 Microsoft 的 Visual C++）中使用 cgc 将使得发现编译错误更加迅速和方便。一个好的集成开发环境甚至可以基于错误信息中的行号，帮助你迅速而正确地定位错误代码所在的行，就像你编译 C 和 C++ 程序那样。图 2-3 显示了在 Microsoft Visual Studio 中的一个编译错误的例子。



图 2-3 在一个集成开发环境中定位错误所在的代码行



Cg 的开发人员经常编写一个单独的 Cg 程序，可以同时在 OpenGL 和 Direct3D 上运行，即使他们主要使用其中一种编程接口。但是对这些三维编程接口而言，基于 profile 的区别可以在对应的 profile 中有效的地方存在。因此，你需要使用 cgc 编译你的 Cg 程序两次：一次用适当的 OpenGL profile，一次用适当的 Direct3D profile。

2.4 用顶点和片段示例程序渲染

现在是看你的两个简单 Cg 程序运行的时候了。不要期望太多，因为它们都是非常简单的程序。但是，通过检查程序是如何与图形流水线的其他部分一起工作

来画一个绿色三角形的，你仍然能够学到许多知识。

看一下图 2-4 中的二维三角形。这是你的顶点和片段程序在这个例子中将会操作的几何图形。

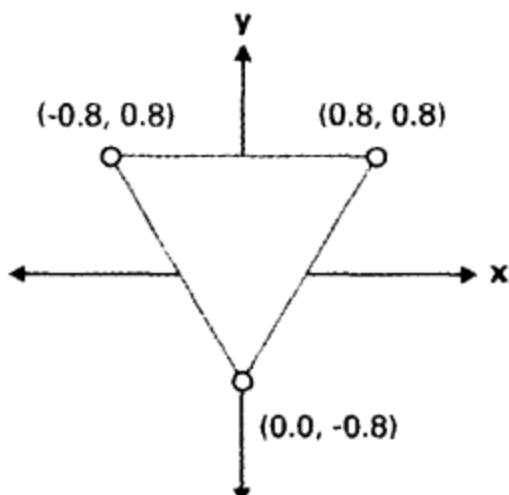


图 2-4 要渲染的一个二维三角形

2.4.1 用 OpenGL 渲染一个三角形

在 OpenGL 中，你可以用下面所示的命令渲染一个二维三角形：

```
glBegin(GL_TRIANGLES);
    glVertex2f(-0.8, 0.8);
    glVertex2f(0.8, 0.8);
    glVertex2f(0.0, -0.8);
glEnd();
```

2.4.2 用 Direct3D 渲染一个三角形

在 Direct3D 中，你可以用如下代码渲染同样的三角形：

```
D3DXVECTOR4 vertices[3] =
{
    D3DXVECTOR4(-0.8F, 0.8, 0.F, 1.f),
    D3DXVECTOR4(0.8F, 0.8, 0.F, 1.f),
    D3DXVECTOR4(0.0F, -0.8, 0.F, 1.f),
};
```

```
m_pD3DDEVICE->DrawPrimitiveUP(D3DPT_TRIANGLELIST, 1,
vertices, sizeof(D3DXVECTOR4));
```

在 OpenGL 和 Direct3D 中，有许多有效的方法来把顶点传递给图形处理器。当你使用 Cg 程序来处理顶点的时候，应用程序如何把顶点传给图形处理器是不重要的。

2.4.3 获得同样的结果

图 2-5 显示了使用 C2E1v_green 顶点程序和 C2E2f_passthrough 片段程序渲染三角形的结果。无论是使用 OpenGL 还是 Direct3D 渲染，这个结果都是一样的。无可否认的，这个结果不是很令人激动，但是这个三角形是纯绿色的。

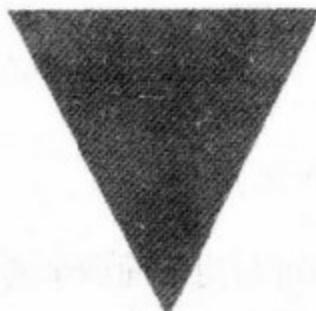


图 2-5 用 C2E1v_green 和 C2E2f_passthrough 渲染三角形

顶点程序把每个顶点指定的二维位置传给光栅器。光栅器认为这些指定的位置坐标都是在剪裁空间以内的。剪裁空间定义了一个从当前视点可见的一个空间。如果顶点程序提供的二维坐标，如图 2-5 所示的情况，被光栅化的图元部分是图元 x 和 y 坐标在 -1 和 +1 之间的部分。整个三角形都在剪裁区域以内，所以整个三角形都被光栅化了。

图 2-6 显示了二维剪裁空间光栅化的区域。

如果一个图元全部落在灰色的区域里（这个区域是剪裁空间的 x 和 y 坐标在 -1 到 +1 之间的区域），这个图元将被渲染到帧缓存中。你将在第 4 章中学习到更多的有关剪裁空间的知识。

图 2-7 显示了当你使用 C2E1v_green 顶点程序和 C2E2f_passthrough 片段程序渲染不是完全在二维剪裁空间的可见区域里的二维几何图形时候的结果。当这些星形图形包含了 x 或 y 坐标在 -1 和 +1 区域以外的顶点的时候，光栅器将用可见区

域的边界去剪裁这些绿色的星形。如果一个图元没有一个部分是在剪裁空间的可见区域里的，光栅器将丢弃这个图元。

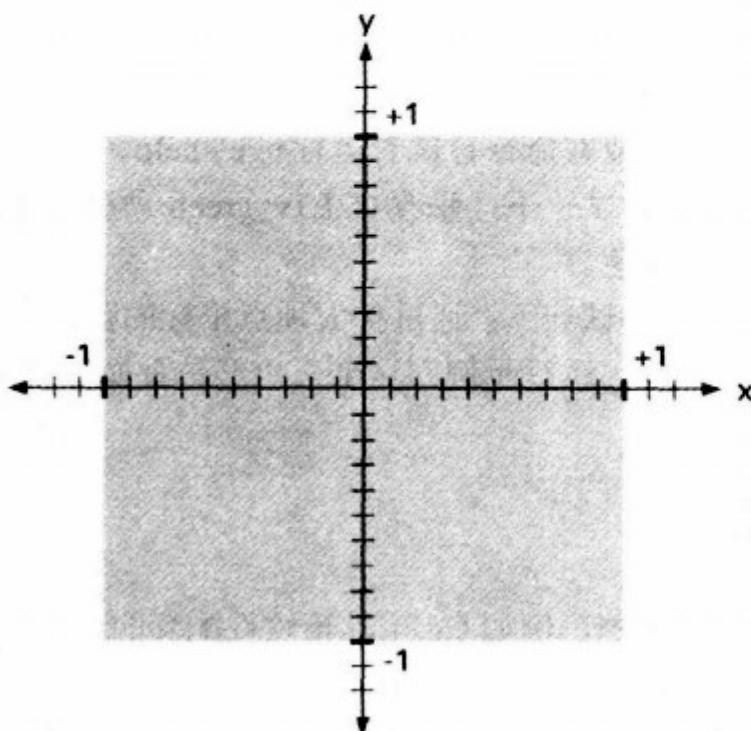


图 2-6 一个二维剪裁空间的视图



图 2-7 用 C2E1v_green 和 C2E2f_passthrough 渲染的图元需要二维剪裁

图形处理器自动对剪裁空间进行变换（通过一个简单地缩放和偏移）到光栅器使用的窗口空间坐标。图形处理器在光栅化之前根据需要伸缩剪裁空间的坐标，以有效地渲染视图和窗口位置。

第 4 章将把二维剪裁空间的概念归纳到三维，并将介绍透视图。但是在本章和第 3 章，所有的示例程序都将使用二维渲染，以使情况变得简单一点。

2.5 练习

1. 回答这个问题：你的 Cg 编译器支持什么样的图形处理器 profile？请教你编译器的文档，或在命令行试着运行 `cgc -help`。
2. 你自己尝试一下：修改 `C2E1v_green` 顶点程序，使输出的顶点颜色是红色而不是原来的绿色。
3. 你自己尝试一下：通过扩大顶点坐标的范围到 $[-1,+1]$ 以外，来改变一个三角形顶点的位置。运行示例。这个三角形是否如你预期的那样被剪裁了？

2.6 补充阅读

查阅一些文档，例如 Cg 工具箱软件配套的文档 *Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics*。

还可以经常访问 www.cgshaders.org 网站，在哪里可以获得最新的有关 Cg 的信息。这个网站有文章，有可以询问问题的论坛，以及免费的着色器示例代码。

第 3 章

参数、纹理和表达式

本章将通过一系列的简单顶点和片段程序继续介绍 Cg 的基本概念。本章有以下三个部分：

- “参数”解释了 Cg 程序是如何处理参数的。
- “纹理样本”解释了片段程序是如何存取纹理的。
- “数学表达式”展示数学表达式是如何计算新的顶点和片段值的。

3.1 参数

第 2 章的 C2E1v_green 和 C2E2f_passthrough 示例程序是非常基本的。我们将增加这些示例的复杂性来介绍附加参数。

3.1.1 Uniform 参数

C2E1v_green（请参见 2.1 节的内容）总是把绿色赋给顶点颜色。如果你给 C2E1v_green 程序改名，并改变赋值给 OUT.color 的那行程序，你可以使用你喜欢的任何颜色生成一个不同的顶点程序。

例如，改变适当的代码产生一个热情的粉红色的着色器：

```
OUT.color = float4(1.0, 0.41, 0.70, 1.0); // RGBA hot pink
```

这个世界是如此的色彩缤纷，因此你不可能想要为阳光下的每种颜色都编写一个不同的 Cg 程序。相反，你可以通过传递一个指示当前被请求颜色的参数来概括这个程序。示例 3-1 中的 C3E1v_anyColor 顶点程序提供了一个 constantColor 参数，使你的应用程序可以指派任何颜色，而不是只用某种不变的颜色。

示例 3-1 C3E1v_anyColor 顶点程序

```

struct C3E1v_Output {
    float4 position : POSITION;
    float4 color     : COLOR;
};

C3E1v_Output C3E1v_anyColor(float2 position : POSITION,
                           uniform float4 constantColor)
{
    C3E1v_Output OUT;

    OUT.position = float4(position, 0, 1);
    OUT.color = constantColor; // Some RGBA color

    return OUT;
}

```

C3E1v_anyColor 和 C2E1v_green 的区别是函数接口的定义和每个程序赋给 OUT.color 的值。

更新过的函数定义是这样的：

```

C3E1v_Output C3E1v_anyColor(float2 position : POSITION,
                           uniform float4 constantColor)

```

除了位置参数以外，新的函数定义有一个名为 constantColor 的参数，并被定义为 uniform float4 类型。正如我们早些时候讨论的，float4 是四元的浮点向量，在这个程序中被认为是一个 RGBA 颜色。我们没有讨论过的是 uniform 类型限制符。

一、uniform 类型限制符

uniform 限制符指明了一个变量初始值的来源。当一个 Cg 程序声明了一个变量为 uniform 的时候，它指明了这个变量的初始值来自指定的 Cg 程序的外部环境。这个外部环境包括你的三维编程接口状态和其他通过 Cg 运行库建立起来的名字/值对。

在 C3E1v_anyColor 示例中的 constantColor 变量的情况下，Cg 编译器生成一个顶点程序从图形处理器的顶点处理器常量寄存器取回变量的初始值。

使用 Cg 运行库，你的三维应用程序可以在一个 Cg 程序里为一个 uniform 参数名查询一个参数句柄——在这个情况下是 constantColor——并使用句柄来把

`uniform` 变量的正确值载入到图形处理器里。`uniform` 参数的值是如何指定和载入的细节随 profile 的不同而改变，但是 Cg 运行库使这个过程变得非常简单。附录 B 解释了这是如何实现的。

我们的 C3E1v_anyColor 顶点程序把它的 `uniform` 变量 `constantColor` 的值赋给顶点输出颜色，如下所示：

```
OUT.color = constantColor; // Some RGBA color
```

无论应用程序为 `uniform` 变量 `constantColor` 指定了什么颜色，当 C3E1v_anyColor 变换一个顶点的时候，这种颜色就是 Cg 程序赋给输出顶点的颜色。

增加了一个 `uniform` 参数，使我们能够推广最初的示例程序来渲染任何颜色，最初的程序只用渲染绿色。

二、当没有 `uniform` 限制符的时候

当一个 Cg 程序没有包含一个 `uniform` 限制符来指定一个变量的时候，你可以用下面的方法之一把初始值赋给变量：

- 使用明确的初始赋值：

```
float4 green = float4(0, 1, 0, 1);
```

- 使用一个语义：

```
float4 position : POSITION;
```

- 由 profile 决定留下不定义或使它等于 0：

```
float4 whatever; //May be initially undefined or zero
```

三、`uniform` 在 RenderMan 和 Cg 中意味着什么

 在 RenderMan 里编写过着色器的程序员是很熟悉 `uniform` 保留字的。但是，在 Cg 中 `uniform` 的意思与它在 RenderMan 中是不一样的。

在 RenderMan 中，`uniform` 存储修饰符指明变量的值在着色表面上是不变的，而 `varying` 变量的值可以在表面上改变。

Cg 没有这种区别。在 Cg 中，一个合格的 `uniform` 变量从一个外部环境获得它的初始值。除了初始化不同以外，`uniform` 变量和其他变量是完全一样的。Cg 允许所有的变量被改变，除非变量被指定了一个 `const` 类型限制符。不像 RenderMan，Cg 没有 `varying` 保留字。

尽管 RenderMan 的 `uniform` 概念和 Cg 的概念有语义上的区别，但在 RenderMan

中声明的 `uniform` 变量与 Cg 中定义的 `uniform` 变量是相对应的，反之亦然。

3.1.2 const 类型限制符

Cg 也提供了 `const` 限制符。`const` 限定符与 C 和 C++ 中的限制符影响变量的方式一样。它限制了你程序中的变量是如何使用的。你不能给一个指定为常量的变量赋值，或改变它。使用 `const` 限制符指明某个值永远不能改变。Cg 编译器会产生一个错误，如果它发现了一个声明为 `const` 的变量将要被修改。

当一个程序用 `const` 限制了一个变量的时候，这里有一些不允许的使用方式的例子：

```
const float pi = 3.14159;
pi = 0.4;           //An error because pi is specified const
float a = pi++;    //Implicit modification is also an error
```

`const` 和 `uniform` 类型限制符是互相独立的，因此一个变量可以使用 `const` 或 `uniform` 来限定，或同时使用 `const` 和 `uniform`，或都不。

3.1.3 Varying 参数

你已经在 `C2E1v_green` 和 `C3E1v_anyColor` 中看到了每个顶点都改变的参数的例子。跟在 `C2E1v_green` 和 `C3E1v_anyColor` 的 `position` 参数后面的 `POSITION` 输入语义，指明图形处理器要用各个程序处理的每个顶点的输入位置来初始化每个 `position` 参数。

语义提供了一种使用随顶点变化而变化（在顶点程序中）或随片段变化而变化（在片段程序中）的值来初始化 Cg 程序参数的方法。

在示例 3-2 中，我们对 `C3E1v_anyColor` 做了微小的修改，新的程序被称为 `C3E2v_varying`。让该程序不仅仅输出一个单一的颜色，而是可以随顶点改变的颜色或纹理坐标集（用来存取纹理）。

示例 3-2 C3E2v_varying 顶点程序

```
struct C3E2v_Output {
    float4 position : POSITION;
    float4 color    : COLOR;
    float2 texCoord : TEXCOORD0;
};
```

```
C3E2v_Output C3E2v_varying(float2 position : POSITION,
                           float4 color      : COLOR,
                           float2 texCoord   : TEXCOORD0)
{
    C3E2v_Output OUT;

    OUT.position = float4(position, 0, 1);
    OUT.color    = color;
    OUT.texCoord = texCoord;

    return OUT;
}
```

C3E2v_varying 示例声明它的顶点程序如下所示：

```
C3E2v_Output C3E2v_varying(float2 position : POSITION,
                           float4 color      : COLOR,
                           float2 texCoord   : TEXCOORD0)
```

C3E2_varying 示例用两个 nonuniform 参数 color 和 texCoord 替换了在 C3E1v_anyColor 示例中声明的 uniform 参数 constantColor 参数。程序对这两个参数分别赋予了 COLOR 和 TEXCOORD0 语义。这两个语义分别对应着程序指定的顶点颜色和纹理坐标集 0。

这个新程序用如下代码通过输出每个顶点的位置、颜色和一个单独的纹理坐标集来变换顶点，而不是只输出每个顶点的位置和一个单一的颜色。

```
OUT.position = float4(position, 0, 1);
OUT.color    = color;
OUT.texCoord = texCoord;
```

图 3-1 显示了使用 C3E2v_varying 顶点程序和 C2E2f_passthrough 片段程序渲染原来的三角形的结果。在这里，我们假设你已经使用了 OpenGL 或 Direct3D 来给三角形的每个顶点赋颜色：上面的两个顶点用亮兰色而下面的顶点则用浅兰色。颜色插值由光栅硬件执行，平滑地对三角形内的片段着色。虽然每个顶点的纹理坐标是由 C3E2v_varying 顶点程序输入和输出的，随后的 C2E2f_passthrough 片段程序忽略了这些纹理坐标。



图 3-1 使用 C3E2v_varying 和 C2E2f_passthrough 渲染的一个二维梯度变化的三角形

3.2 纹理样本

C3E2v_varying 示例通过顶点程序传递了每个顶点的纹理坐标。虽然 C2E2f_passthrough 片段程序忽略了纹理坐标，在示例 3-3 中显示的名为 C3E3f_texture 的下一个片段程序，使用了纹理坐标对一个纹理图像进行采样。

示例 3-3 C3E3f_texture 片段程序

```
struct C3E3f_Output {
    float4 color : COLOR;
};

C3E3f_Output C3E3f_texture(float2 texCoord : TEXCOORD0,
                           uniform sampler2D decal)
{
    C3E3f_Output OUT;
    OUT.color = tex2D(decal, texCoord);
    return OUT;
}
```

C3E3f_Output 结构本质上和我们早先的片段示例程序 C2E2f_passthrough 使用的 C2E2f_Output 结构是一样的。C3E3f_texture 示例中的新概念显示在它的声明中：

```
C3E3f_Output C3E3f_texture(float2 texCoord : TEXCOORD0,
                           uniform sampler2D decal)
```

C3E3f_texture 片段程序接受了一个经过插值的纹理坐标，但忽略了经过插值的颜色。这个程序还收到了一个名为 `decal` 的 uniform sampler2D 类型的参数。

3.2.1 样本对象

在 Cg 中一个样本指一个外部对象，Cg 可以对它进行采样，例如一个纹理。Sampler2D 类型的 2D 后缀指明这个纹理是一个传统的二维纹理。表 3-1 列出了 Cg 支持的对应不同纹理类型的其他样本类型。在以后的章节中你将会遇到一些这样的样本对象。

表 3-1

Cg 样本类型

样本类型	纹理类型	应用程序
sampler1D	一维纹理	一维函数
sampler2D	二维纹理	贴图、法向量贴图、光泽贴图、阴影贴图和其他
sampler3D	三维纹理	容积数据、三维衰减函数
sampleCUBE	立方图纹理	环境贴图、标准立方贴图
samplerRECT	二维非二的指数的 non-mipmap 纹理	视频图像、照片、临时的缓存

在存取一个纹理的时候，纹理坐标指定了在哪里查找。图 3-2 显示了一个二维纹理和一个基于纹理坐标 (0.6, 0.4) 的查询。通常，纹理坐标的范围是从 0 到 1 的，但是你也可以超出这个范围的值。在这里我们不对这个问题进行详细地解释，因为结果取决于在 OpenGL 和 Direct3D 中如何设置你的纹理。

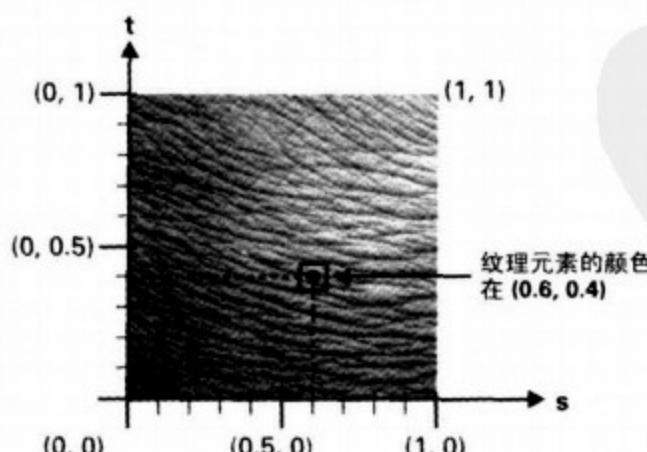


图 3-2 查询一个纹理

在示例 3-3 中命名为 `texCoord` 的纹理坐标集的语义是 `TEXCOORD0`，它对应了给纹理单元 0 的纹理坐标。正如样本参数 `decal` 的名字暗示地，这个片段程序的目的是使用片段的经过插值的纹理坐标集来存取一个纹理。

3.2.2 纹理采样

`C3E3f_texture` 的下一个有趣的代码行使用插过值的纹理坐标存取 `decal` 纹理：

```
OUT.color = tex2D(decal, texCoord);
```

`tex2D` 例程属于 `Cg` 标准库。它是一族可以用指定的纹理坐标集存取不同类型的样本并返回一个向量结果的例程之一。这个结果是在样本对象上由纹理坐标指定的位置采样的数据。

实际上，这等同于纹理查找。一个纹理是如何采样和过滤的由纹理类型和与 `Cg` 样本变量相关联的纹理对象的纹理参数决定。你可以使用 `OpenGL` 和 `Direct3D` 的纹理指定命令来决定一个给定纹理的纹理属性，这取决于你选择哪个三维编程接口。你的应用程序很可能要用 `Cg` 运行库来建立这种关联。

后缀 `2D` 表明 `tex2D` 必须采样一个类型为 `sampler2D` 的样本对象。同样地，`texCUBE` 例程返回一个向量，接受一个 `samplerCUBE` 类型的样本作为它的第一个参数，并需要一个三元纹理坐标集作为它的第二个参数。

基本片段 `profile`（例如 `ps_1_1` 和 `fp20`）把纹理采样例程，例如 `tex2D` 和 `texCUBE`，限制到对应于样本的纹理单元的纹理坐标集。为了尽可能地简单和支持所有的片段 `profile`，`C3E3f_texture` 示例遵守了这个限制（请看第 2.3.1 小节对 `profile` 的简单介绍）。

高级的片段 `profile`（例如 `ps_2_x`、`arbfp1` 和 `fp30`）允许使用从其他纹理单元取得的纹理坐标集，或者甚至是由你的 `Cg` 程序计算的纹理坐标集对一个样本进行采样。

3.2.3 在对一个纹理采样的时候，发送纹理坐标

`C3E2v_varying` 顶点程序把每个顶点的位置、颜色和纹理坐标集传给光栅器。`C3E3f_texture` 片段程序忽略插过值的颜色，但用插过值的纹理坐标对一个纹理图像采样。图 3-3 显示了当你首先把一个包含着可怕的鬼脸的纹理和你的 `Cg` 程序绑定在一起，然后用额外赋予的每个顶点的纹理坐标来渲染我们的简单的三角形的结果。



图 3-3 用 C3E2v_varying 和 C3E3f_texture 渲染的一个有纹理的二维三角形

3.3 数学表达式

到目前为止，我们所示的所有 Cg 示例仅仅做了传递参数，或使用参数采样一个纹理的简单工作。传统的非可编程三维编程接口能实现的也就这么多了。这些示例的意图是把 Cg 介绍给你，并显示简单 Cg 程序的结构。

许多有趣的 Cg 程序可以通过操作符和 Cg 标准库提供的内置函数根据输入的参数进行计算。

3.3.1 操作符

Cg 支持的数学、关系和其他操作符与 C 和 C++ 提供的一样。这意味着加法用一个 + 号来表示，乘法用 * 号，大于或等于用 \geq 操作符。你已经在以前的示例中看到了赋值用 = 来完成。

这里有一些 Cg 表达式的例子：

```
float total = 0.333 * (red + green + blue);
total += 0.333 * alpha;
float smaller = (a < b)? a : b;
float eitherOption = optionA || optionB;
float allTrue = v[0] && v[1] && v[2]
```

Cg 不同于 C 和 C++, 因为它对向量的数学操作提供了内置的支持。你可以在 C++ 中通过编写你自己的使用操作符重载的类来实现这点，但向量数学操作在 Cg 中是语言的一个标注部分。

下面这些操作符以位元方式工作在向量上：

*	乘法
/	除法
-	取反
+	加法
-	减法

当一个标量和一个向量被用作这些位元操作符之一的操作数的时候，标量被复制到一个大小匹配的向量中。

这里有一些向量 Cg 表达式的例子：

```
float3 modulatedColor = color * float3(0.2, 0.4, 0.5);
modulatedColor*=0.5;
float3 specular = float3(0.1, 0.0, 0.2);
modulatedColor +=specular;
megatedColor = -modulatedColor;
float3 = direction = positionA - positionB
```

表 3-2 呈现了全部操作符的列表，并伴随了它们的优先级、结合率和使用方法。被反向加亮的操作符是被保留的。但是，不存在 Cg profile 支持这些保留操作符，因为当前的图硬件不支持位元的整数操作。

表 3-2 操作符的优先级、结合率和使用方法

操作符	结合率	使用方法
() [] -> *	从左到右	函数调用，数组引用，结构引用和成员选择
! ~ ++ - + - *	从右到左	一元操作符：取反、增加、减少、正号、负号、间接、地址、转换
& (type) sizeof		
* / %	从左到右	乘法、除法、余数
+ -	从左到右	加法、减法
<< >>	从左到右	移位操作符
< <= > >=	从左到右	关系操作符
== !=	从左到右	相等、不相等

续表

操作符	结合率	使用方法
&	从左到右	位操作与 (AND)
^	从左到右	位操作异或 (XOR)
	从左到右	位操作或 (OR)
&&	从左到右	逻辑与 (AND)
	从左到右	逻辑或 (OR)
?:	从右到左	条件表达式
= += -= *= /= %=& ^= = <<= >>=	从右到左	赋值、赋值表达式
,	从左到右	逗号操作符

注意

- 操作符是从上到下，从高级到低级的优先级。
- 在同一行的操作符有相同的优先级。
- 被反向加亮的操作符是当前保留为以后使用的。

3.3.2 依赖于 profile 的数值数据类型

当你用 C 和 C++ 编程并声明变量的时候，你可以从一些不同大小的整型数据 (int, long, short, char) 和两个浮点数据类型 (float, double) 中选择。

你的中央处理器对所有这些基本的数据类型提供了硬件支持。但是，图形处理器通常不支持这么多的数据类型，虽然随着图形处理器的发展，它们保证会提供更多的数据类型。例如，现存的图形处理器在顶点和片段处理器中不支持指针类型。

表示连续的数据类型

Cg 提供了 float、half 和 double 浮点类型。Cg 定义这些类型的方法与 C 的相似——该语言不要求特别精确。half 的范围和精度小于或等于 float 的范围和精度，而 float 的范围和精度小于或等于 double 的范围和精度是可以理解的。

half 数据类型在 C 和 C++ 中并不存在。由 Cg 介绍的这个新的数据类型可以保

留一半精度的浮点值（通常是 16 位），它在存储和性能方面比标准的浮点类型（通常是 32 位）更有效。

CineFX

NVIDIA CineFX 图形处理器构架在片段程序支持 half 精度的值。half 数据类型在片段程序中通常适合被用作中间值，例如颜色和标准化的向量。通过在可能的时候使用 half 值而不是 float，你可以加速你的片段程序的性能。

图形处理器被设计用来提供可以表示连续量的数据类型，例如颜色和向量。图形处理器现在还不支持表示固有的离散量（例如文字数字和位屏蔽）的数据类型，因为图形处理器通常不操作这种类型的数据。

连续量并不限制于整型值。当用中央处理器编程的时候，程序员通常使用浮点类型的数据来表示连续值，因为浮点类型可以表示分数值。图形处理器处理的连续值，特别是在片段级别上，已经被限制在一个很窄的范围内，例如[0, 1]和[-1, +1]，而不支持浮点所提供的昂贵的范围。例如，颜色通常被限制在[0, 1]的范围内，而标准化的向量按照定义被限制在[-1, +1]的范围内。这些范围受限制的数据类型被认为是“定点”，而不是浮点。

虽然定点数据类型使用了有限的精度，但是它们可以表示连续量。然而，它们缺乏浮点数据类型的范围，它的编码方式与它的科学符号相似。一个浮点值除了尾数以外还编码了一个可变的指数（这与数字用科学符号写出来的方式一样，例如 2.99×10^8 ），而一个顶点值则假定了一个固定的指数。例如，一个没有标准化的向量或者足够大的纹理坐标也许会需要浮点类型的值来避免一个给定的定点范围的溢出。

当前的图形处理器在执行顶点和片段程序的时候，处理浮点一样好。但是，早期的可编程图形处理器只为顶点处理提供了浮点类型数据，而给片段处理提供了定点数据类型。

Cg 必须能够操作定点数据类型来支持缺乏浮点片段编程能力的图形处理器的可编程性。这意味着某些片段 profile 使用定点值。表 3-3 列出了各种 Cg profile 并描述了它们是如何表示不同的数据类型的。这暗示 Cg 程序员 float 在所有的 profile 中在各种环境中也许并不真正地意味浮点。

表 3-3

各种 profile 的数据类型

profile 名	类型	数值
arbfp1	float	浮点
arbvp1	double	
vs_1_1	half	
vs_2_0	fixed	
vp20	int	浮点转化的整数
vp30		
fp20	float double half int fixed	纹理贴图是浮点；片段着色是范围为[-1, +1]的定点
ps_1_1	float	纹理贴图是浮点；片段着色是范围依赖图形处理器的定点；范围依赖底层的 Direct3D 的能力
ps_1_2	double	
ps_1_3	half int fixed	
ps_2_0	float	24 位的浮点（最小值）
ps_2_x	double	
	int	浮点转化的整数
	half	16 位的浮点（最小值）
	fixed	依赖编译器设置
fp30	float	浮点
	double	
	int	浮点转化的整数
	half	16 位浮点
fp30	fixed	范围为[-2, +2]的定点



`fp20` 和 `ps_1_1 profile` 把在片段着色中的变量当作范围在 $[-1, +1]$ 内的定点值。通过片段着色，我们想要数学操作在纹理贴图结果之后执行。如果你想要用真正的浮点类型数据，需要使用 `arbftp1`、`fp30` 或 `vp_2_0 profile`，但请注意这些高级的 `profile` 不被较老的图形处理器支持。



CineFX 构架在片段程序中还支持一种特殊的称为 `fixed` 的高性能连续数据类型。`fixed` 数据类型在 `fp30 profile` 中的范围是 $[-2, +2]$ （意味着从 -2 没有恰好到 $+2$ ）。在其他 `profile` 中，`fixed` 数据类型和可用的最小的连续数据类型是同义的。虽然 Cg 的编译器（`cgc`）和运行库支持 `fixed` 数据类型（和向量类型例如 `fixed3` 和 `fixed4`），但是 Microsoft 的 HLSL 编译器（`fxc`）不支持。

3.3.3 标准库内置的函数

Cg 标准库包含许多内置的函数可以简化图形处理器编程。在许多情况下，这些函数被映射为一个单独的本地图形处理器指令，因此它们非常有效。

这些内置的函数和 C 的标准库函数很相似。Cg 标准库提供了一套实际的三角几何、指数、向量、矩阵和纹理函数。但是，Cg 标注库函数没有提供输入/输出、字符串处理或内存分配，因为 Cg 不支持这些操作（虽然你的 C 或 C++ 应用程序肯定能够）。

我们在示例 3-3 中已经使用了 Cg 标准库函数 `tex2D`。参考表 3-4 选择性地列出了 Cg 标准库提供的其他函数。你可以在附录 E 中找到 Cg 标准库函数的完整列表。

表 3-4

选择过的 Cg 标准库函数

函数原型	profile 用法	描述
<code>abs(x)</code>	所有	绝对值
<code>cos(x)</code>	顶点和高级片段	以弧度为单位的角的余弦
<code>cross(v1, v2)</code>	顶点和高级片段	两个向量的外积
<code>ddx(a)</code>	高级片段	分别相对于窗口空间的 x 和 y 坐标，近似 a 的偏导数
<code>ddy(a)</code>		
<code>determinant(M)</code>	顶点和高级片段	矩阵的行列式值
<code>dot(a, b)</code>	所有，但在低级片段受限制	两个向量的内积

续表

函数原型	profile 用法	描述
<code>floor(x)</code>	顶点和高级片段	不超过 x 的最大整数
<code>isnan(x)</code>	高级顶点和片段	如果 x 不是一个数，则真
<code>lerp(a, b, f)</code>	所有	基于 f 的在 a 和 b 之间的线性插值
<code>log2(x)</code>	顶点和高级片段	基于 2 的 x 的对数
<code>max(a, b)</code>	所有	a 和 b 中最大值
<code>mul(M, N)</code>	顶点和高级片段	矩阵和矩阵乘法
<code>mul(M, v)</code>		矩阵和向量乘法
<code>mul(v, M)</code>		向量和矩阵乘法
<code>pow(x, y)</code>	顶点和高级片段	x 的 y 次方
<code>radians(x)</code>	顶点和高级片段	度转化为弧度
<code>reflect(v, n)</code>	顶点和高级片段	入射光线 v 和法向量 n 的反射向量
<code>round(x)</code>	顶点和高级片段	取最靠近 x 的整数
<code>rsqrt(x)</code>	顶点和高级片段	平方根的倒数
<code>tex2D(sampler, x)</code>	片段，但基本片段受限制	二维纹理查找
<code>tex3Dproj(sampler, x)</code>	片段，但基本片段受限制	投影三维纹理查找
<code>texCUBE(sampler, x)</code>	片段，但基本片段受限制	立方贴图纹理查找

一、函数重载

Cg 标准库重载了大部分自己的例程，以使同样的例程能够用于多种数据类型。与 C++一样，函数重载为使用同一个名字和不同类型参数的例程提供了多种实现方法。

重载是非常方便的。这意味着你能够用一个标量参数、一个二元参数、一个三元参数或一个四元参数来使用一个函数，例如 `abs`。在每种情况下，Cg 都会调用正确版本的绝对值函数：

```
float4 a4 = float4 (0.4, -1.2, 0.3, 0.2);
float2 b2 = float2 (-0.3, 0.9);
float4 a4abs = abs (a4);
float2 b2abs = abs (b2);
```

这段代码片段调用了 `abs` 例程两次。在第一个实例中，`abs` 接受了一个四元向

量。在第二个实例中，`abs` 接受了一个二元向量。根据传递给函数的参数，编译器能够自动调用正确版本的 `abs`。在 Cg 标准库中广泛使用重载意味着你不需要为某个给定大小的向量或其他参数考虑调用哪个例程。Cg 能够自动选择你制定函数的正确的实现。

函数重载并不只限于 Cg 标准库。另外，你也可以用函数重载编写你自己的内部函数。

函数重载在 Cg 中甚至可以应用于不同 profile 的同一个函数的不同实现。例如，一个新 GPU 的高级顶点 profile 也许有专门的指令来计算三角正弦和余弦函数。一个老一点的 GPU 的基本顶点 profile 则不会有这样专门的指令。但是，你也许能够用一系列被支持的顶点指令来近似正弦和余弦函数，虽然这样会损失一些精度。你可以编写两个函数，并为每个都制定一个专门的 profile。

Cg 对依赖于 profile 的重载的支持能够帮助你把 Cg 程序中依赖于 profile 的限制分离到帮助函数中去。*Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics* 有关于依赖 profile 重载的更多信息。

二、Cg 标准函数库的有效性和精确性

只要可能，尽量使用 Cg 标准库来进行数学计算和其他被支持的操作。Cg 标准库函数和你编写的类似函数一样有效和精确，或比你的函数更加有效和更加精确。

例如，`dot` 函数计算两个向量的点积。你自己也可以编写一个点积函数，就像下面这样：

```
float myDot ( float3 a, float3 b )
{
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

这与 `dot` 函数所实现的数学是一样的。但是，`dot` 函数会映射到一个专门的 GPU 指令，因此 Cg 标准库提供的点积计算很可能会比 `myDot` 函数要更快和更精确。

**010
01** 通过尽可能的使用 Cg 标准库，你能够引导 Cg 编译器为你的专门的 GPU 生成最有效和精确的程序。

3.3.4 二维扭曲

在下一个例子中，你将把表达式、操作符和 Cg 标准库结合在一起。这个例子将演示如何扭曲一个 2D 几何形状。一个顶点离窗口的中心越远，顶点程序把这个顶点以窗口为中心旋转的也就越多。

在示例 3-4 中的 C3E4v_twist 程序示范了标量与向量乘法，标量的加法和乘法，标量取反，length 标准库函数和 sincos 标准库函数。

示例 3-4 C3E4v_twist 顶点程序

```
struct C3E4_Output {
    float4 position : POSITION;
    float4 color     : COLOR;
};

C3E4_Output C3E4v_twist(float2 position : POSITION,
                         float4 color      : COLOR,
                         uniform float twisting)
{
    C3E4_Output OUT;
    float angle = twisting * length(position);
    float cosLength, sinLength;
    sincos(angle, sinLength, cosLength);
    OUT.position[0] = cosLength * position[0] +
                      -sinLength * position[1];
    OUT.position[1] = sinLength * position[0] +
                      cosLength * position[1];
    OUT.position[2] = 0;
    OUT.position[3] = 1;
    OUT.color = color;
    return OUT;
}
```

C3E4v_twist 程序以变化参数输入顶点的位置和颜色，而以统一标量方式输入缩放因子。图 3-4 显示了使用不同扭曲量的例子。

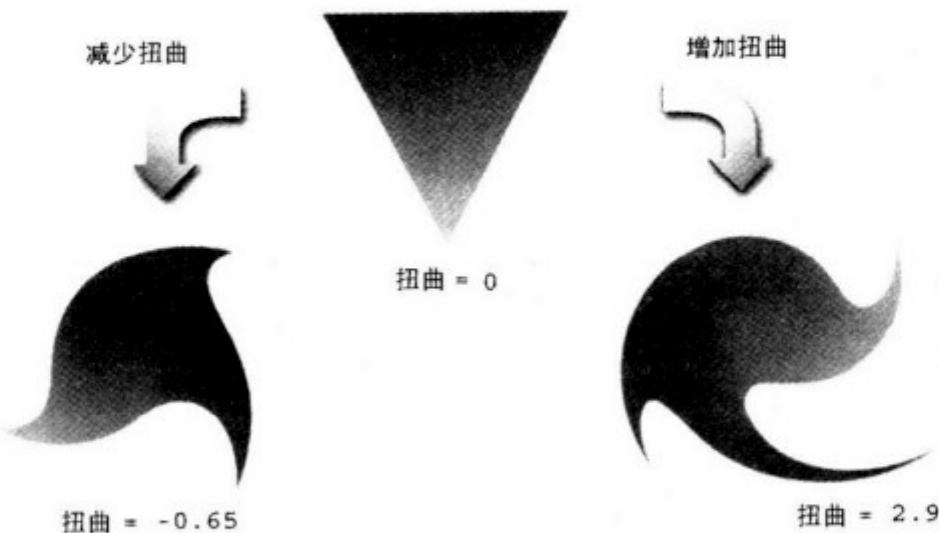


图 3-4 C3E4v_twist 使用不同 twisting 参数设置的结果

一、length 和 sincos 标准库函数

length 函数有一个重载的原型，其中 SCALAR 是任何的标量数据类型，而 VECTOR 是同一标量类型的一个向量，以 SCALAR 作为一元、二元、三元或四元分量：

```
SCALAR length ( VECTOR x );
```

Cg 标准库函数 length 返回它唯一参数的标量长度：

```
float angle = twisting * length ( position );
```

这个程序计算了弧度角，代表了 twisting 参数乘以输入位置的长度的结果。然后 sincos 标准库函数计算这个角的正弦和余弦。

sincos 函数有如下重载类型，其中 SCALAR 是任何标量类型：

```
void sincos ( SCALAR angle, out SCALAR s, out SCALAR c );
```

当 sincos 返回的时候，Cg 用 angle 参数（假定为弧度）的正弦和余弦更新调用参数 s 和 c。

Advanced

二、传结果 (call-by-result) 参数传递

限定词 out 制定了当函数返回的时候，Cg 必须把由 out 限定的形式参数的最终值传递给它对应的调用参数。初始的时候，out 参数的值是没有定义的。这个规则被称为传结果参数传递。

C 没有类似的参数传递规则。C++ 允许传递一个引用参数给函数（在形式参数前用&标明），但是这被称为引用参数传递，而不是 Cg 的传结果参数传递规则。

Cg 还提供了 `in` 和 `inout` 关键字。`in` 类型限定符指明了 Cg 通过值来传递参数 (`call-by-value`)。调用函数的参数值被用来初始化函数的形式参数。当一个函数有 `in` 限定符参数返回的时候, Cg 会丢弃这些参数的值除非这些参数也被 `out` 限定了。

C 对所有的参数使用了参数值拷贝 (`copy-by-value`) 的参数传递方式。C++对传递引用以外的所有参数使用参数值拷贝的方式。

`inout` 类型限定符 (或者 `in` 和 `out` 类型限定符制定在同一个参数上) 结合了传值和传结果参数传递 (也被称为 `call-by-value-result` 或者 `copy-in-copy-out`)。

`in` 限定符是可选择的, 因为如果你不指定一个 `in`, `out` 或 `inout` 限定符, `in` 限定符将被制定。

你可以在使用 `out` 和 `inout` 参数的同时, 仍然返回一个常规的返回值。

三、旋转顶点

一旦程序为顶点计算了旋转角度的正弦和余弦, 它将使用一个旋转变换。公式 3-1 表达了 2D 旋转。

公式 3-1 2D 旋转

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

下面这段程序代码实现了这个公式。在第 4 章, 你将学习如何更加简洁和有效地表达矩阵类型的数学, 但是现在我们将用简单易懂的方法来实现这个公式:

```
OUT.position[0] = cosLength * position[0] +
                  -sinLength * position[1];
OUT.position[0] = sinLength * position[0] +
                  cosLength * position[1];
```

四、镶嵌对顶点程序的重要性

C3E4v_twist 程序通过围绕图像中心旋转顶点来实现效果。当扭曲旋转的程度增加, 一个物体也许需要更多的顶点来产生有效合理的扭曲效果, 因此需要更高程度的镶嵌。

通常, 当一个顶点程序涉及了非线性计算的时候, 例如在这个例子中的三角函数, 获得可接受的结果将需要足够的镶嵌。这是因为在光栅器生成片段的时候, 顶点的值将被光栅器线性插值。如果没有足够的镶嵌, 顶点程序将显示出底层几

何形状的镶嵌情况。图 3-5 显示了如何增加镶嵌程度来提高 C3E4v_twist 示例的扭曲的结果。

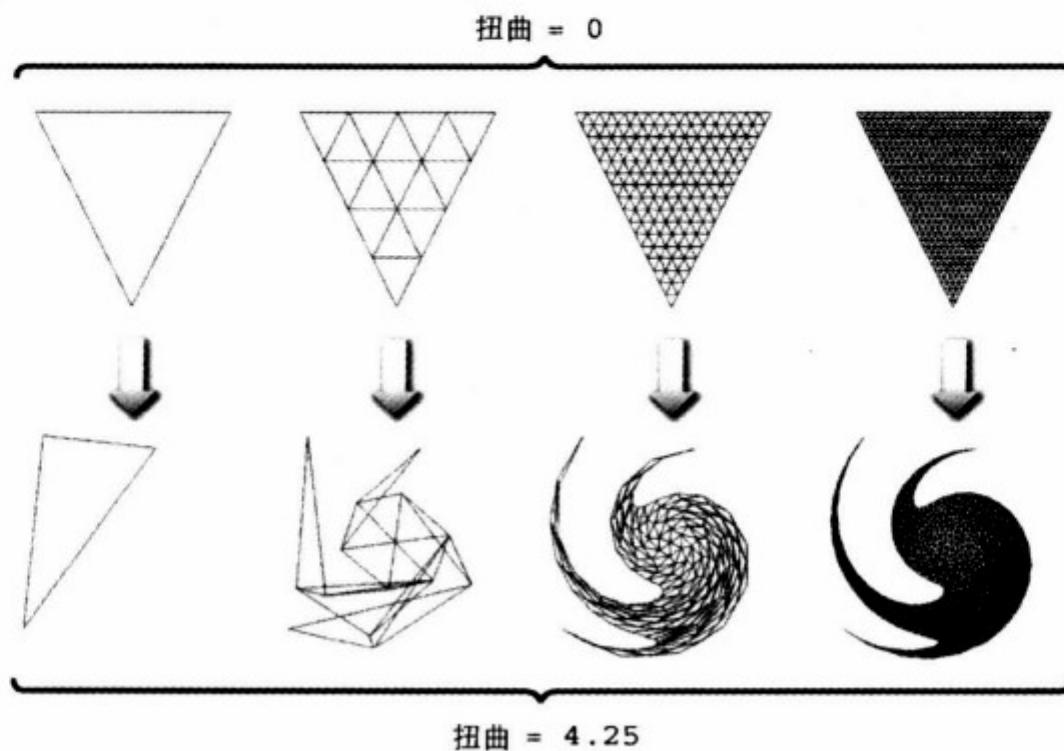


图 3-5 通过增加镶嵌来提高 C3E4v_twist 的真实程度

3.3.5 重影效果

现在，我们示范如何把一个顶点程序和一个片段程序结合在一起来获得一种纹理的重影效果。最主要的概念是在稍微移动纹理坐标的基础上对同一个纹理采样两次，然后把采样值平等地混合。

在示例 3-5 中显示的 C3E5v_twoTextures 顶点程序通过使用两个不同的偏移量来生成两个稍微分开的纹理坐标集，从而偏移了一个纹理坐标两次。然后，片段程序在两个偏移位置对一个纹理图像存取两次，并把纹理结果平等地混合。图 3-6 显示了渲染的结果和所需要的输入。

示例 3-5 C3E5v_twoTextures 顶点程序

```
void C3E5v_twoTextures(float2 position : POSITION,
                        float2 texCoord : TEXCOORD0,
                        out float4 oPosition : POSITION,
                        out float2 leftTexCoord : TEXCOORD0,
                        out float2 rightTexCoord : TEXCOORD1,
```

```

uniform float2 leftSeparation,
uniform float2 rightSeparation)

{
    C3E5v_Output OUT;

    oPosition      = float4(position, 0, 1);
    leftTexCoord  = texCoord + leftSeparation;
    rightTexCoord = texCoord + rightSeparation;

    return OUT;
}

```

一、重影顶点程序

在示例 3-5 中显示的 C3E5v_twoTextures 程序只是简单地传递了顶点位置。另外，这个程序输出了同一输入纹理坐标两次，一次用统一参数 leftSeparation 平移，另外一次用统一参数 rightSeparation 平移。

```

oPosition      = float4(position, 0, 1);
leftTexCoord  = texCoord + leftSeparation;
rightTexCoord = texCoord + rightSeparation;

```

每个片断两个纹理采样
 $leftSeparation = (-0.5, 0)$
 $rightSeparation = (+0.5, 0)$ 带一个纹理坐标集的三角形



图 3-6 用 C3E5v_twoTextures 和 C3E6f_twoTextures 创建重影效果

二、Out 参数对比输出结构

C3E5v_twoTextures 示例还显示一种不同的方法来输出参数。不像我们以前的例子那样返回一个输出结构，C3E5v_twoTextures 示例什么也不返回，函数的返回类型是 **void**。改为用 **out** 参数和与之相关联的语义（入口函数原型的一部分）来表明哪个参数是输出参数。选择使用 **out** 参数或一个输出返回结构来从一个入口函数输出参数由你来决定。这两种方法没有功能上的区别。你甚至能够混合使用它们。

本书剩余的部分将使用 **out** 参数的方法，因为这样能够避免指定输出结构。我们给 **out** 参数增加了一个“o”作为前缀，以区别输入和输出参数，否则它们将有相同的名字，例如 **position** 和 **oPosition** 参数。

示例 3-6 C3E6f_twoTextures 片段程序

```
void C3E6f_twoTextures(float2 leftTexCoord : TEXCOORD0,
                      float2 rightTexCoord : TEXCOORD1,
out float4 color : COLOR,
uniform sampler2D decal)
{
    float4 leftColor = tex2D(decal, leftTexCoord);
    float4 rightColor = tex2D(decal, rightTexCoord);
    color = lerp(leftColor, rightColor, 0.5);
}
```

在示例 3-5 和随后的例子中，我们还把入口函数的参数按照输入、输出和统一参数排列分组。这种方式需要额外的工作量来格式化代码，但是我们在这本书中使用这个方法以使示例代码更加容易阅读，特别是当示例代码有很多参数的时候。

三、用于高级片段 profile 的重影片段程序

在示例 3-6 中的 C3E6f_twoTextures 片段程序用 C3E5v_twoTextures 程序计算的两个平移和插值的纹理坐标集来对同一纹理图像采样两次，如图 3-6 所示。

```
float4 leftColor = tex2D(decal, leftTexCoord);
float4 rightColor = tex2D(decal, rightTexCoord);
然后，程序计算两个采样颜色的平均值。
color = lerp(leftColor, rightColor, 0.5);
```

lerp 函数计算两个相同大小的向量的线性插值。帮助记忆的 **lerp** 代表线性插值 (linear interpolation)。这个函数有一个重载原型，其中 **VECTOR** 是一个一元、二元、三元或四元的向量，而 **TYPE** 是一个标量或者是一个与 **VECTOR** 有相同维数和元素类型的向量。

```
VECTOR lerp (VECTOR a, VECTOR b, TYPE weight);
```

lerp 函数计算：

结果 = (1-weight) × a + weight × b

0.5 的 **weight** 将计算一个平均值。这个函数不需要 **weight** 在 0 和 1 的范围之间。

不幸地是，C3E6f_twoTextures 片段程序将无法用基本的片段 **profile** 编译，例如 **fp20** 和 **ps_1_1**（你将马上学习为什么不行）。这个程序用高级片段 **profile** 编译得很好，例如 **fp30** 和 **ps_2_0**。

四、用于基本片段 **profile** 的重影片段程序

C3E6f_twoTextures 示例使用了两个纹理坐标集 0 和 1，来存取纹理单元 0。因为这样，这段程序无法用基本的片段程序 **profile** 编译。由于第三代和早期的 GPU 的限制，这样的 **profile** 只能用一个给定的纹理坐标集存取它对应的纹理单元。

你可以稍微修改一下 C3E6f_twoTextures 程序，让它可以同时工作在基本和高级片段 **profile** 上。在示例 3-7 中的 C3E7f_twoTextures 版本包含了必需的修改。

示例 3-7 C3E7f_twoTextures 片段程序

```
void C3E7f_twoTextures(float2 leftTexCoord : TEXCOORD0,
                      float2 rightTexCoord : TEXCOORD1,
                      out float4 color : COLOR;

uniform sampler2D decal0,
uniform sampler2D decal1)
{
    float4 leftColor = tex2D(decal0, leftTexCoord);
    float4 rightColor = tex2D(decal1, rightTexCoord);
    color = lerp(leftColor, rightColor, 0.5);
}
```

被修改后的程序需要两个纹理单元:

```
uniform sampler2D decal0,  
uniform sampler2D decal1)
```

因为两个纹理单元需要对同一个纹理图像采样, C3E7f_twoTextures 片段程序需要应用程序为两个不同的纹理单元绑定同一个纹理。原来的 C3E6f_twoTextures 程序不需要应用程序绑定纹理两次。

当程序对两个纹理采样的时候, 由于基本片段程序 profile 的需要, 它用对应的纹理坐标集对每个纹理采样:

```
float4 leftColor = tex2D(decal0, leftTexCoord);  
float4 rightColor = tex2D(decal1, rightTexCoord);
```

这两种方法的性能是同等的。这个示例证明了简单的 Cg 程序(那些不是非常复杂的程序)通常能够通过一点额外的努力编写, 就能够在较早的只支持基本的顶点和片段 profile 的 GPU 上与支持高级 profile 的最新的 GPU 一样运行。

3.4 练习

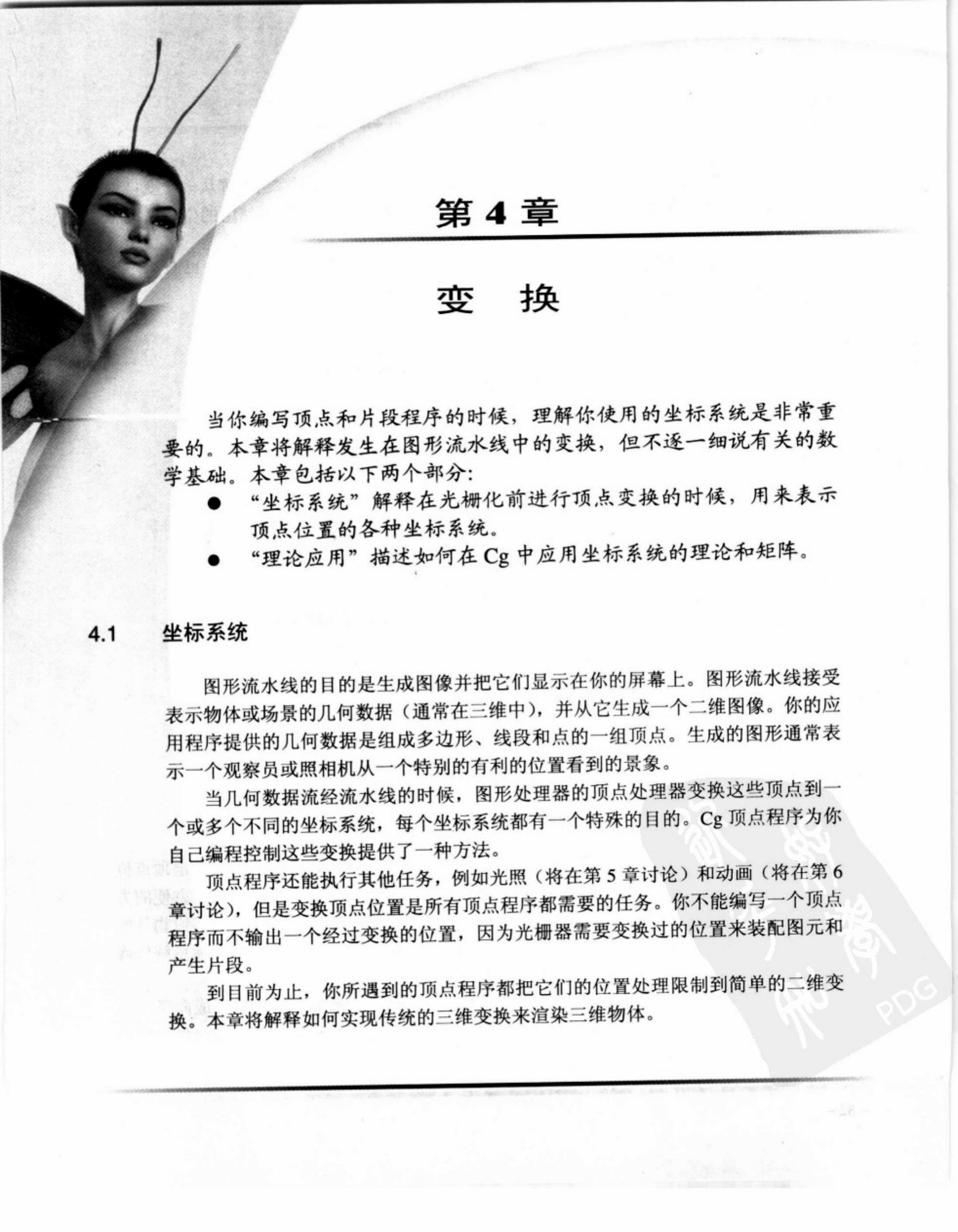
- 回答这个问题: 除了方便以外, 你认为为什么 sincos 标准库函数同时返回一个角的 sine 和 cosine? 提示: 考虑一下三角恒等式。
- 回答这个问题: 用你自己的话解释为什么增加图 3-5 显示的 tessellation 是使扭曲的三角形看起来更好所需要的。
- 你自己尝试一下: 修改 C3E4v_twist 示例使得扭曲中心为指定为 uniform float2 参数的任意二维点, 而不是在原点 (0, 0)。
- 你自己尝试一下: 修改 C3E5v_twoTextures 和 C3E7f_twoTextures 程序来提供“四重视觉”效果。证实你的新程序同时可以在基本和高级 profile 上工作。假设你的图形处理器支持 4 个纹理单元。
- 你自己尝试一下: 修改 C3E5v_twoTextures 示例使它返回一个输出结构而不是使用 out 参数。同样, 修改一个早期的例子, 例如 C3E4v_twist, 只使用 out 参数而不是返回一个输出结构。你喜欢哪种方法呢?

3.5 补充阅读

你能从 Gerald Farin 和 Dianne Hansford 撰写的 *The Geometry Toolbox for Graphics and Modeling* (A.K. Peters, 1998 年) 学到许多 2×2 矩阵的知识，例如 twist 示例中的旋转矩阵。







第 4 章

变 换

当你编写顶点和片段程序的时候，理解你使用的坐标系统是非常重要的。本章将解释发生在图形流水线中的变换，但不逐一细说有关的数学基础。本章包括以下两个部分：

- “坐标系统”解释在光栅化前进行顶点变换的时候，用来表示顶点位置的各种坐标系统。
- “理论应用”描述如何在 Cg 中应用坐标系统的理论和矩阵。

4.1 坐标系统

图形流水线的目的是生成图像并把它们显示在你的屏幕上。图形流水线接受表示物体或场景的几何数据（通常在三维中），并从它生成一个二维图像。你的应用程序提供的几何数据是组成多边形、线段和点的一组顶点。生成的图形通常表示一个观察员或照相机从一个特别的有利的位置看到的景象。

当几何数据流经流水线的时候，图形处理器的顶点处理器变换这些顶点到一个或多个不同的坐标系统，每个坐标系统都有一个特殊的目的。Cg 顶点程序为你自己编程控制这些变换提供了一种方法。

顶点程序还能执行其他任务，例如光照（将在第 5 章讨论）和动画（将在第 6 章讨论），但是变换顶点位置是所有顶点程序都需要的任务。你不能编写一个顶点程序而不输出一个经过变换的位置，因为光栅器需要变换过的位置来装配图元和产生片段。

到目前为止，你所遇到的顶点程序都把它们的位置处理限制到简单的二维变换。本章将解释如何实现传统的三维变换来渲染三维物体。

图 4-1 举例说明了传统的用来处理顶点位置的变换安排。当位置从一个变换进入到下一个的时候，这个图表在每个变换之间用被顶点位置所使用的坐标空间来注解变换的转变。

下面的部分将按照顺序描述每个坐标系统和变换。我们假设你已经具备了矩阵和变换的基础知识，因此我们将概括地解释流水线的每一个阶段。

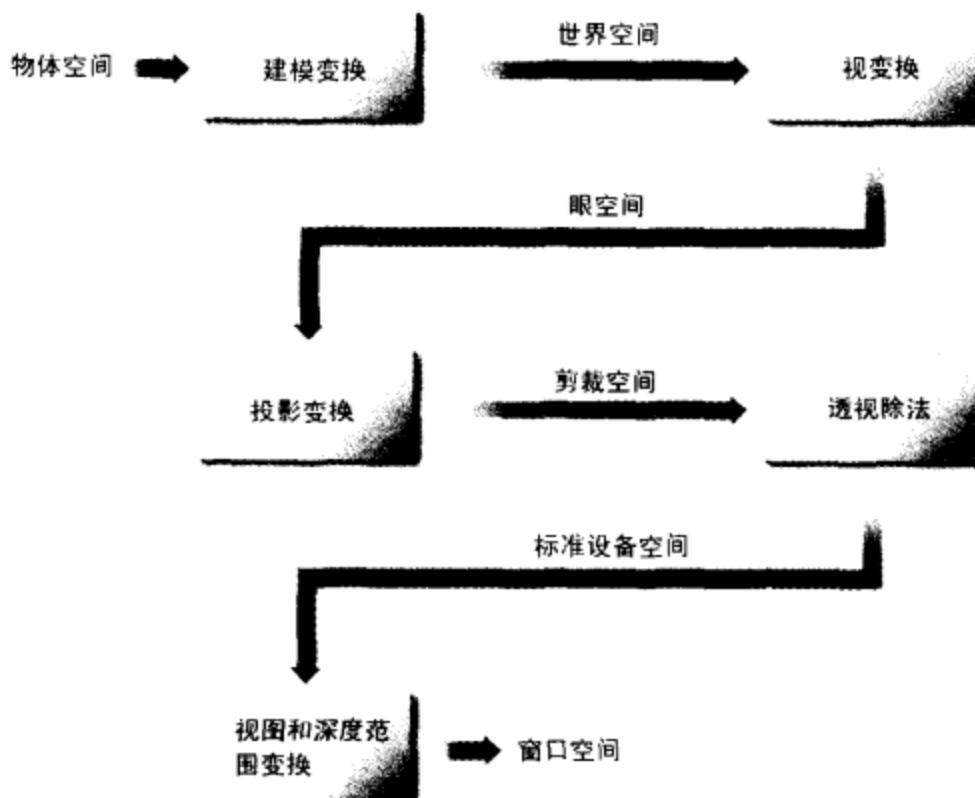


图 4-1 用于顶点处理的坐标系统和变换

4.1.1 物体空间

应用程序在一个被称为物体空间（也叫模型空间）的坐标系统里指定顶点位置。当一个美工人员创建了一个物体的三维模型的时候，他选择了一个方便的方向、比例和位置来放置模型的组成顶点。一个物体的物体空间可以与其他物体的物体空间没有任何关系。例如，一个圆柱体可以创建一个物体空间坐标系统以底面的中心为原点， z 方向沿着对称轴的方向。

无论在物体空间还是在以后的空间之一，你可以把顶点位置表示成向量。通常，你的应用程序维护每个物体空间的三维顶点位置为一个 $\langle x, y, z \rangle$ 向量。每个

顶点也许还有一个附随的在物体空间的表面的法向量，也被存储为一个 $\langle x, y, z \rangle$ 向量。

4.1.2 齐次坐标

通常，我们认为 $\langle x, y, z \rangle$ 位置向量只不过是四元 $\langle x, y, z, w \rangle$ 形式的一种特殊情况。这种四元位置向量类型被称为一个齐次位置（homogeneous position）。当我们表达一个向量位置为一个 $\langle x, y, z \rangle$ 量的时候，我们假设有一个隐含的 1 作为它的 w 成员。

数学上，w 值是你用来除 x, y 和 z 成员，以获得一个传统的三维（非齐次）位置，如公式 4-1 所示。

公式 4-1 在非齐次和齐次位置之间转换

$$\left\langle \frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1 \right\rangle = \langle x, y, z, w \rangle$$

用这种齐次形式表示位置有许多优点。其中一个优点是多变换，包括三维投影透视所需要的投影变换，可以被有效地组合成一个单独的 4×4 矩阵。这个技术将在第 4.2 节解释。同样，使用齐次位置不必执行昂贵的中间除法和创建涉及投影透视的特殊情况了。齐次位置用于表示方向和有理多项式描述的曲面也非常方便。

当讨论投影变换的时候我们将继续讨论 w 成员。

4.1.3 世界空间

一个物体的物体空间和其他对象没有空间上的关系。世界空间的目的是为在你的场景中的所有物体提供一个绝对的参考。一个世界空间坐标系如何建立可以任意选择。例如，你可以决定世界空间的原点是你房间的中心。然后，房间里的物体就可以相对房间的中心和某个比例（一个距离单位是一尺还是一米？）和某个方向（y 轴的正方向是朝“上”吗？北是在 x 轴的正方向上吗？）来放置了。

4.1.4 建模变换

在物体空间中指定的物体被放置到世界空间的方法要依靠建模变换。例如，你也许需要旋转、平移和缩放一个椅子的三维模型，以使椅子可以正确地放置在

你的房间的世界坐标系统里。在同一个房间中的两把椅子可以使用同样的三维椅子模型，但使用不同的建模变换，以使每把椅子放在房间中不同的位置。

在数学上，你能用一个 4×4 矩阵表示本章中的所有变换。使用矩阵的性质通过把几个矩阵乘在一起，你能够把几个平移、旋转、缩放和投影组合在一个单独的 4×4 矩阵。当你用这种方法连接矩阵，矩阵的组合也代表了各个变换的组合。正如你将要看到的，这个结果将非常有用。

如果你用表示的建模变换的 4×4 的矩阵乘以齐次形式的物体空间的位置（如果没有明确的 w 成员，则假设 w 成员为 1），这个结果和变换到世界空间的位置是一样的。同样的矩阵数学原理可以应用到所有以后在本章讨论的变换。

图 4-2 说明了几个不同的建模变换的效果。左边的图显示了一个机器人没有应用任何建模变换以一个基本的姿势放置。当你在机器人的各个身体部件使用了一系列的建模变换后，右边的图显示了发生在机器人身上的结果。例如，你必需选择和平移右臂到如图所示的位置。要在世界空间中把机器人放置在合适的位置和方向需要进一步地进行平移和旋转变换。

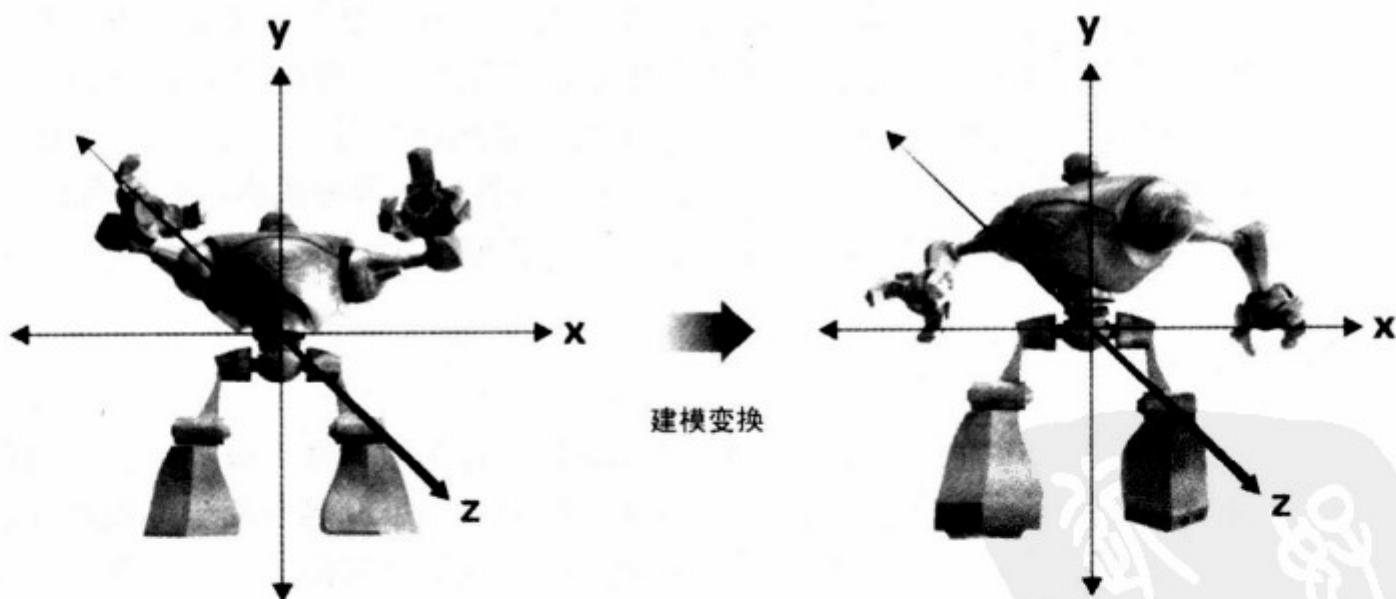


图 4-2 建模变换的效果

4.1.5 眼睛空间

最后，你要从一个特殊的视点（“眼睛”）观看你的场景。在称为眼睛空间（或视觉空间）的坐标系统里，眼睛位于坐标系统的原点。朝“上”的方向通常是 y

轴正方向。遵循标准惯例，你可以确定场景的方向使眼睛是从 z 轴向下看。

眼空间对光照特别有用，将会在第 5 章讨论。

4.1.6 视变换

从世界空间位置到眼空间位置的变换是视变换。再一次，你可以用一个 4×4 的矩阵表达视变换。典型的视变换结合了一个平移把眼睛在世界空间的位置移到眼空间的原点，然后适当地旋转眼睛。通过这样做，视变换定义了视点的位置和方向。

图 4-3 举例说明了视变换。左边的图显示了从图 4-2 来的机器人和被放置在世界坐标系 $<0, 0, 5>$ 位置的眼睛。右边的图则在眼空间显示它们。你会观察到眼空间把原点放在眼睛的位置上。在这个例子中，视变换平移机器人以使它移动到在眼空间里的正确位置。经过平移之后，机器人站在眼空间 $<0, 0, -5>$ 的位置，而眼睛在原点。在这个例子里，眼空间和世界空间都把 y 轴正方向当成它们“朝上”的方向，并且平移完全在 z 轴上进行。否则，会需要一个像平移那样的旋转。

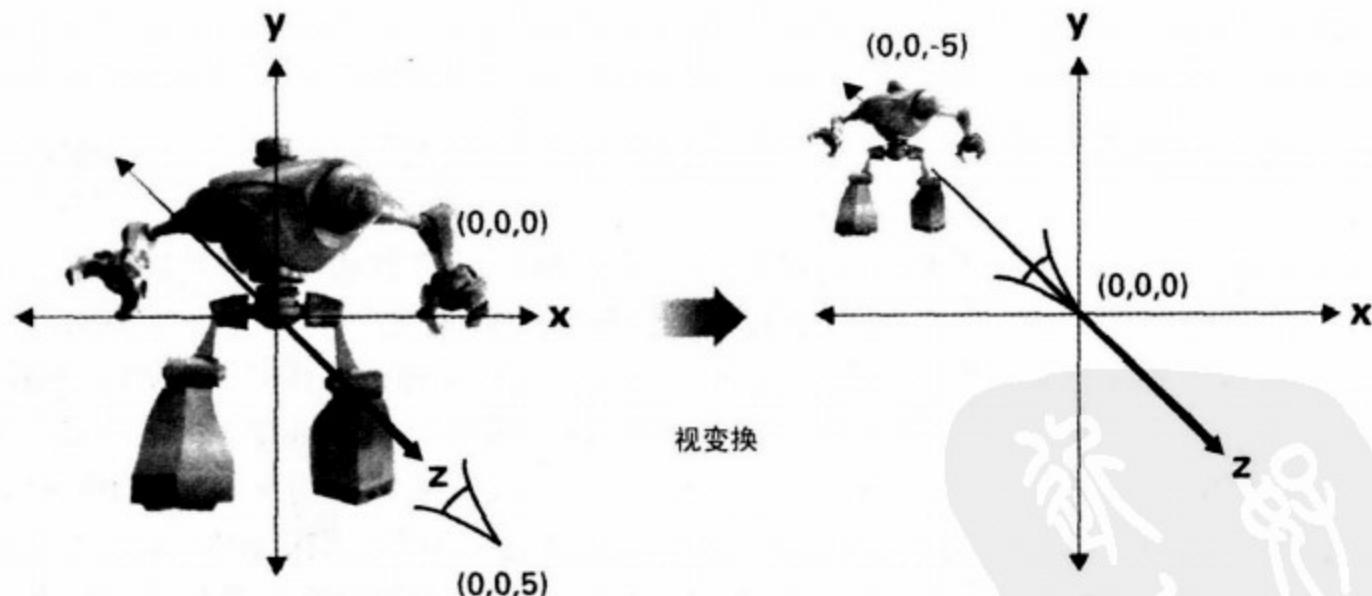


图 4-3 视变换的效果

Modelview 矩阵

大部分光照和其他着色计算都会涉及位置和表面法向量。通常，当它们在眼空间或物体空间中执行时，这些计算往往更加有效。世界空间在你的应用程序中

为场景里的物体建立总的空间关系时非常有用，但是它对光照和其他着色计算并不特别有效。

因为这个原因，我们通常把分别代表建模和视变换的两个矩阵结合在一起，组成一个单独的被称为 `modelview` 的矩阵。你可以通过简单地用建模矩阵乘以视矩阵把它们结合在一起。

4.1.7 剪裁空间

当位置在眼空间以后，下一步是决定什么位置是在你最终要渲染的图像中可见的。在眼空间之后的坐标系统被称为剪裁空间（Clip Space），在这个空间中的坐标系统被称为剪裁坐标。

一个 Cg 顶点程序输出的顶点的位置是在剪裁空间中的。每个顶点程序可选择的输出一些参数，例如纹理坐标和颜色，但是一个顶点程序总是要输出一个剪裁空间的位置。正如你在较早的例子中看到的，`POSITION` 语义是用来指明一个特别的顶点程序输出是剪裁空间的位置。

4.1.8 投影变换

从眼空间坐标到剪裁空间坐标的变换被称为投影变换（Projection Transform）。

投影变换定义了一个视线平截体（view frustum），代表了眼空间中物体的可见区域。只有在视线平截体中的多边形、线段和点被光栅化到一幅图像中时，才潜在的有可能被看得见。OpenGL 和 Direct3D 对剪裁空间有些微不同的规则。在 OpenGL 中，任何可见的东西必须在一个轴对齐的立方体中，使得它的剪裁空间位置的 x 、 y 和 z 成员小于或等于它对应的 w 成员。这暗示着 $-w \leq x \leq w$ ， $-w \leq y \leq w$ 和 $-w \leq z \leq w$ 。Direct3D 对 x 和 y 有相同的剪裁要求，而 z 必须是 $0 \leq z \leq w$ 。这些剪裁规则假设剪裁空间的位置是齐次形式的，因为它们依赖于 w 。

投影变换提供了从眼空间的可视区域（也就是视线平截体）到包含剪裁空间的可视区域的剪裁空间轴对称立方体的映射。你可以把这个映射表示成一个 4×4 的矩阵。

投影矩阵

对应于投影变换的这个 4×4 矩阵被称为投影矩阵。

图 4-4 说明了投影矩阵是如何把在眼空间的机器人从图 4-3 变换到剪裁空间的。整个机器人都落在剪裁空间中，因此生成的图像应该显示整个机器人没有任何部分被剪裁掉。

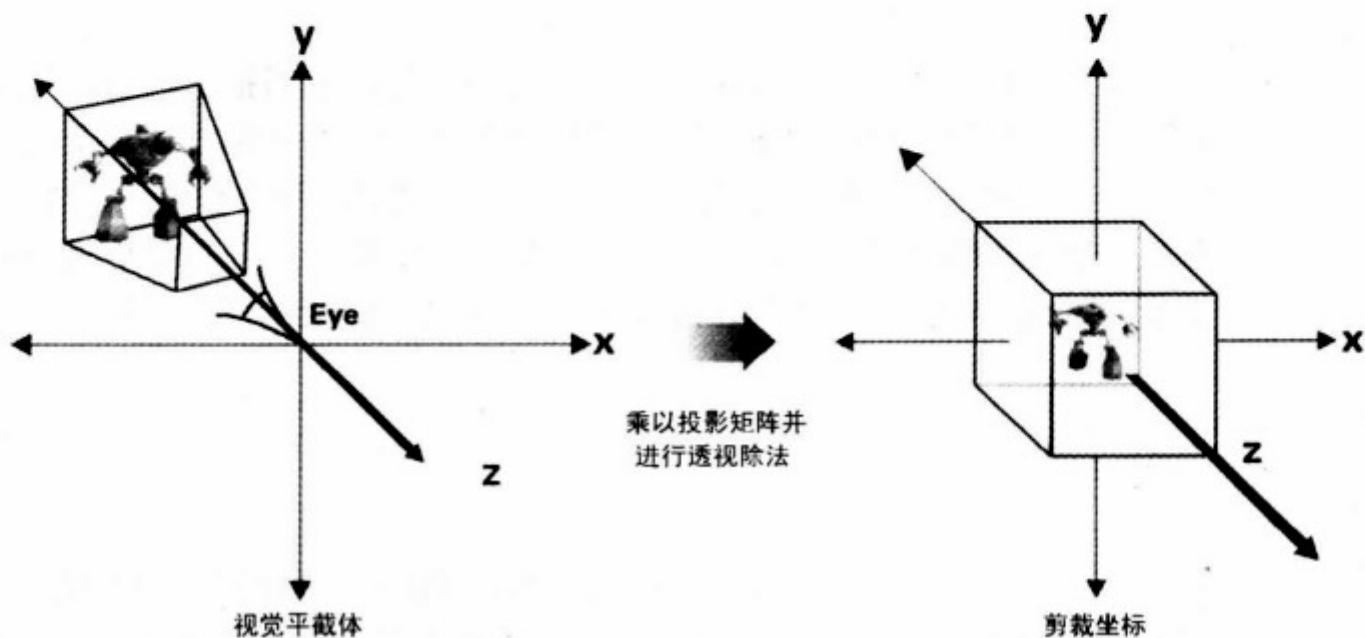


图 4-4 投影矩阵的效果

OpenGL 和 Direct3D 的剪裁空间的规则是不同的，并分别被设置在一个投影矩阵里。结果，如果 Cg 程序员依靠了他们选择的三维编程接口的正确投影矩阵，这两种剪裁空间定义的区别并不明显。通常，应用程序负责提供合适的投影矩阵给 Cg 程序。

4.1.9 标准化的设备坐标

剪裁坐标是齐次形式 $\langle x, y, z, w \rangle$ 的，但我们需要计算一个二维位置（一对 x 和 y）和一个深度值（深度值是为了进行深度缓冲（depth buffering），一种硬件加速的渲染可见表面的方法）。

透视除法（perspective division）

用 w 除 x、y 和 z 能完成这项工作。生成的结果坐标被称为标准化的设备坐标（normalized device coordinates）。现在所有的可见几何数据在 OpenGL 中都位于坐标 $\langle -1, -1, -1 \rangle$ 和 $\langle 1, 1, 1 \rangle$ 的立方体中，在 Direct3D 中则为 $\langle -1, -1, 0 \rangle$ 和 $\langle 1, 1, 1 \rangle$ 。

在第 2 和第 3 章中的二维顶点程序输出的就是你现在所知的标准化的设备坐标。在这些例子中输出的二维位置假设隐含的 z 值为 0 以及一个值为 1 的 w。

4.1.10 窗口坐标

最后一步是取每个顶点的标准化的设备坐标，然后把它们转换为使用像素度量 x 和 y 的最后的坐标系统。这一步骤被命名为视图变换（viewport transform），它为图形处理器的光栅器提供数据。然后，光栅器从顶点组成点、线段或多边形，并生成决定最后图像的片段。另一个被称为深度范围变换（depth range transform）的变换，缩放顶点的 z 值到在深度缓冲中使用的深度缓存的范围内。

4.2 理论应用

尽管所有的讨论都是关于坐标空间的，但你需要的正确进行顶点变换的 Cg 代码是非常繁琐的。通常，顶点程序接收在物体空间的顶点位置。这个程序然后用 modelview 和 projection 矩阵乘以每个顶点，以得到在剪裁空间的顶点。实际上，你需要连接这两个矩阵，这样只需要进行一次乘法，而不是两次。图 4-5 通过展示从对象坐标到剪裁坐标的两个方法，来举例说明了这个原理。

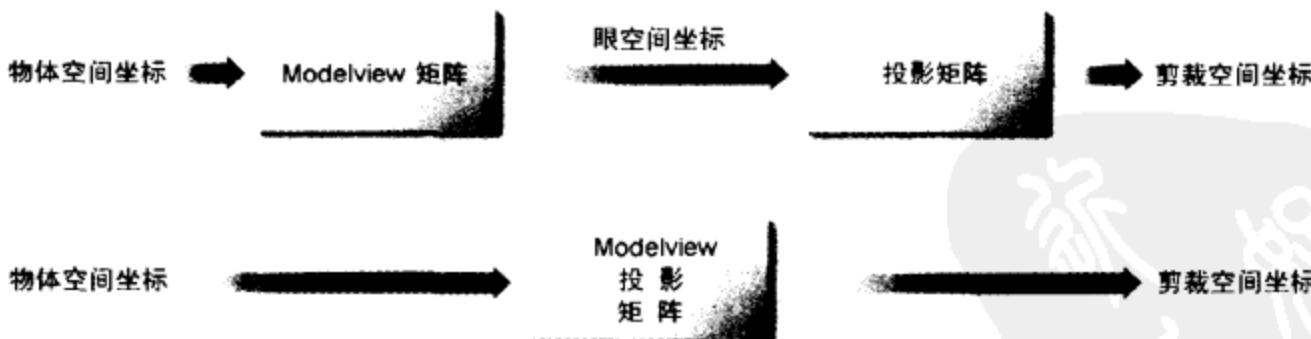


图 4-5 到剪裁空间的变换的优化

例 4-1 显示了一个典型的 Cg 程序是如何有效地从物体空间直接到剪裁空间的三维顶点变换。

示例 4-1 C4E1v_transform 顶点程序

```

void C4E1v_transform(float4 position : POSITION,
                    out float4 oPosition : POSITION,
                    uniform float4x4 modelViewProj)
{
    // Transform position from object space to clip space
    oPosition = mul(modelViewProj, position);
}

```

这个程序用物体空间的位置（position）和连接好的 modelview 和投影矩阵（modelViewProj）作为输入参数。你的 OpenGL 或 Direct3D 应用程序将负责提供这些数据。有几个 Cg 运行例程能够帮你基于当前 OpenGL 或 Direct3D 的变换状态来载入正确的矩阵。然后，使用一个矩阵乘法对 position 参数进行变换，并且把结果写到 oPosition：

```

// Transform position from object space to clip space
oPosition = mul(modelViewProj, position);

```

在本书中，我们明确地给所有的输出参数赋值，即使它们只是被简单地传递。我们用“o”前缀来区分有相同名字的输入和输出参数。

4.3 练习

- 回答这个问题：列出各种坐标空间和从一空间变换到另一个空间所使用的变换顺序。
- 回答这个问题：如果你对变换理论感兴趣，可以列举一些只使用 3×3 的 modelview 和 projection 矩阵代替 4×4 的矩阵。
- 你自己尝试一下：使用 cgc 输出 C4E1v_transform 示例的顶点程序汇编代码。DP4 指令计算了一个四元内积。被程序的 mul 例程生成的这样的指令有多少？

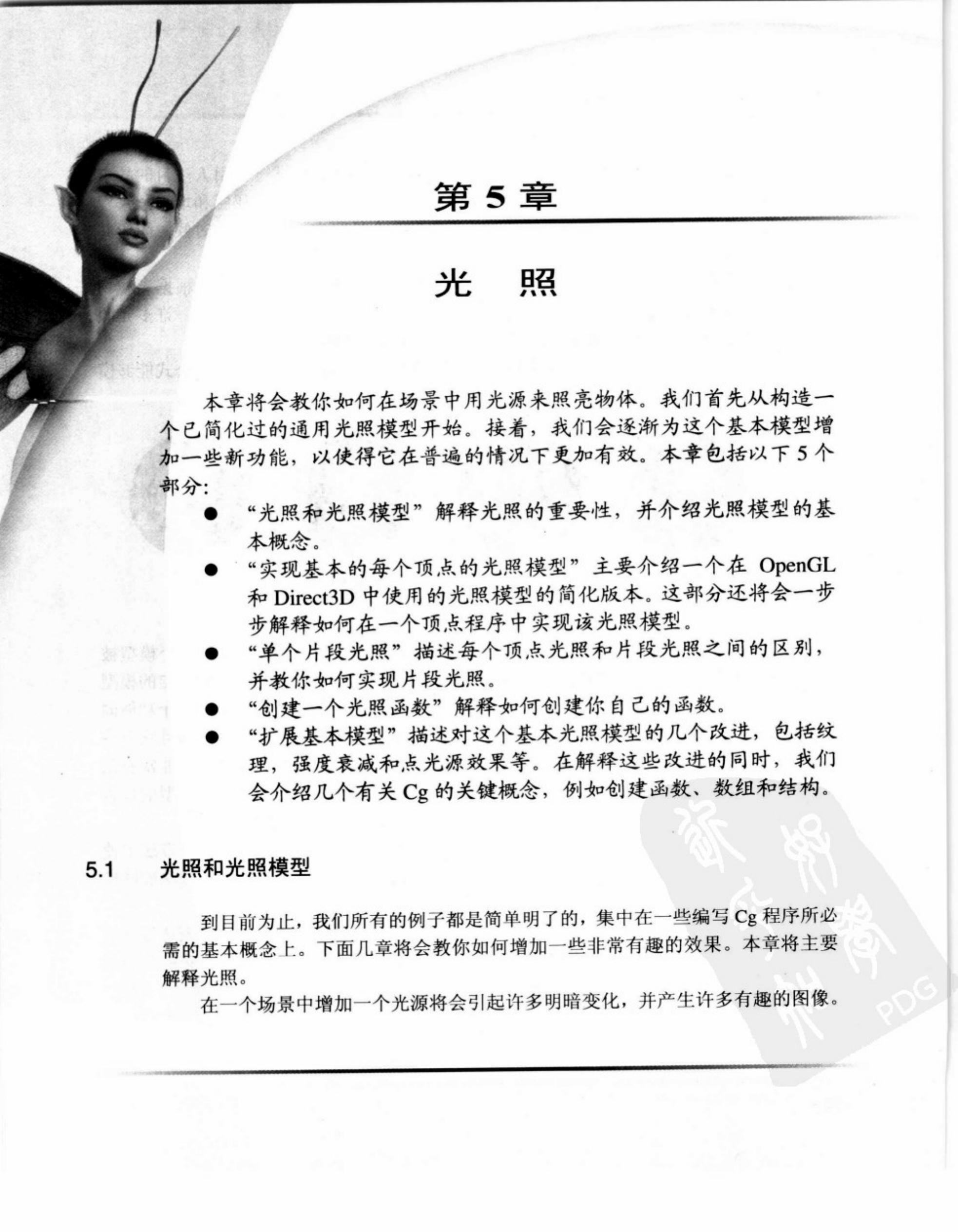
4.4 补充阅读

计算机图形学教科书解释了顶点变换，并逐步地展开了有关矩阵数学基础这一主题的许多知识。我们推荐 Edward Angel 撰写的 *Interactive Computer Graphics:*

A Top-Down Approach with OpenGL 第三版 (Addison-Wesley, 2002 年)。

如果你想发展对作为顶点变换基础的投影变换的兴趣，并同时得到一些娱乐，请阅读 *Jim Blinn's Corner: A Trip Down the Graphics Pipeline* (Morgan Kaufmann, 1996 年)。

由 Andrew Glassner 编辑的 *Graphics Gems* (Academic Press, 1994 年) 包含了许多有用的关于建模、变换和矩阵技术的简短文章。



第 5 章

光 照

本章将会教你如何在场景中用光源来照亮物体。我们首先从构造一个已简化过的通用光照模型开始。接着，我们会逐渐为这个基本模型增加一些新功能，以使得它在普遍的情况下更加有效。本章包括以下 5 个部分：

- “光照和光照模型”解释光照的重要性，并介绍光照模型的基本概念。
- “实现基本的每个顶点的光照模型”主要介绍一个在 OpenGL 和 Direct3D 中使用的光照模型的简化版本。这部分还将会一步步解释如何在一个顶点程序中实现该光照模型。
- “单个片段光照”描述每个顶点光照和片段光照之间的区别，并教你如何实现片段光照。
- “创建一个光照函数”解释如何创建你自己的函数。
- “扩展基本模型”描述对这个基本光照模型的几个改进，包括纹理，强度衰减和点光源效果等。在解释这些改进的同时，我们会介绍几个有关 Cg 的关键概念，例如创建函数、数组和结构。

5.1 光照和光照模型

到目前为止，我们所有的例子都是简单明了的，集中在一些编写 Cg 程序所必需的基本概念上。下面几章将会教你如何增加一些非常有趣的效果。本章将主要解释光照。

在一个场景中增加一个光源将会引起许多明暗变化，并产生许多有趣的图像。

这就是为什么许多电影导演会非常重视光照：因为它在讲述一个引人注目的故事中起到了非常重要的作用。一个场景中阴暗的区域可以引起神秘感和增强紧张程度（不幸地是，在计算机图形学中，当你在一个场景中增加光源的时候，阴影并不能够很轻易的生成）。第 9 章将讨论有关生成阴影的主题。

光照和一个物体的材质特性一起决定了它的外观。一个光照模型根据光和物体的特征描述了光和物体之间的相互作用和影响。在过去的十几年里，许多光照模型被开发和使用，从简单的近似一直到及其精确的模拟。

图 5.1 显示了一组使用不同的光照模型所渲染的物体。注意不同的公式能够模拟多种多样的真实世界中的材质。

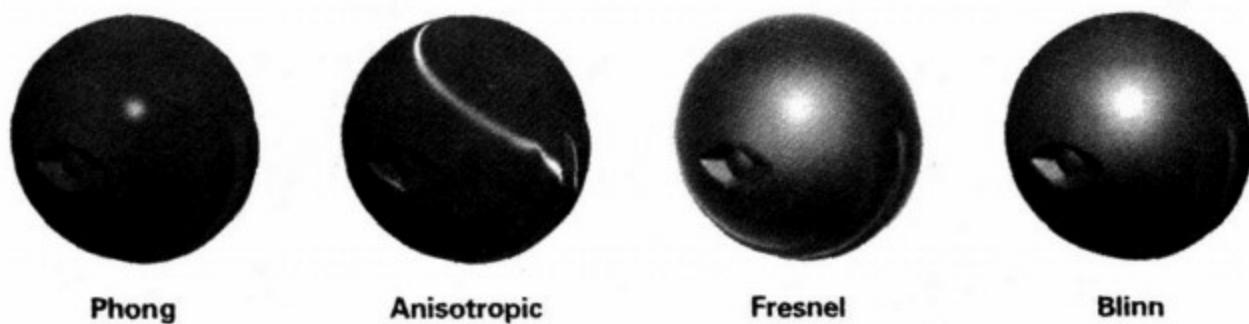


图 5-1 不同的光照模型

在过去，固定功能的图形流水线被限制只能使用一个光照模型，这个模型被我们称为功能固定光照模型（fixed-function lighting model）。这个功能固定的模型是基于众所周知的 Phong 光照模型的，但是经过了一些调整和增强。这个功能固定的光照模型有几个优点：它看上去已经足够了，它的计算量很少，而且它有许多直观的参数可以用来调整和控制物体的外观。但是，它的问题是只对非常有限的一些材质工作的很好。一个塑料或橡胶的外表是使用功能固定光照模型的最普遍的特征，这解释了为什么许多计算机渲染的图像看起来并不太真实。

为了解除功能固定光照模型的限制，图形程序员发现了许多创新的方法来使用渲染流水线的其他一些特性。例如，聪明的程序员使用基于纹理的方法能够模仿一组范围很广的材质的表面特性。

随着 Cg 和可编程硬件的到来，你现在能够用一种高级语言简洁地表达复杂的光照模型。你不再需要设置有限的一组图形流水线状态或编写冗长乏味的汇编语言例程。并且，你不需要限制你的光照模型来适应固定功能流水线的能力。相反，你能够表达你自定义的光照模型为一个在可编程图形处理器中执行的 Cg 程序。

5.2 实现基本的每个顶点的光照模型

本节将解释如何用一个顶点程序实现一个固定功能的光照模型的简化版本。大家对这个光照模型的熟悉程度和这个模型的简单性使它成为了一个很好的起点。首先，我们将介绍一下固定功能的光照模型的背景知识。如果你已经对光照模型很熟悉了，你可以自由地跳过下一节直接到有关实现细节的第 5.2.2 节。

5.2.1 基本的光照模型

OpenGL 和 Direct3D 提供了几乎相同的固定功能光照模型。在我们的例子中，我们将使用一个简化的版本，我们将称之为“基本”模型。这个基本模型和 OpenGL 和 Direct3D 模型一样，对经典的 Phone 模型进行了修改和扩展。在基本模型里，一个物体的表面颜色是放射 (emissive)、环境反射 (ambient)、漫反射 (diffuse) 和镜面反射 (specular) 等光照作用的总和。每种光照作用取决于表面材质的性质（例如亮度和材质颜色）和光源的性质（例如光的颜色和位置）的共同作用。我们用一个包含红、绿和蓝三元色的 float3 向量来表示每种光照作用。

从数学上描述基本模型的高级公式如下所示：

$$\text{surfaceColor} = \text{emissive} + \text{ambient} + \text{diffuse} + \text{specular}$$

一、放射项

放射项表示了由表面所发出的光。这种光照作用是独立于所有光源的。放射项是一个 RGB 值，指明了表面所发出光的颜色。如果你在一个完全黑暗的房间里观察一种放射性材质，它将呈现出这种颜色。放射项可以模拟炽热的物体。图 5-2 举例从概念上说明了放射项，图 5-3 显示了一个纯放射性物体的渲染。这个渲染比较单调沉闷是可以理解的，因为放射颜色在物体表面的所有地方都是一样的。不像在真实世界里，一个物体的放射性发光在场景中并不照亮附近的其他物体。一个放射性物体本身并不是一个光源——它不照亮其他物体或投下阴影。另一个解释放射项的方法是，它是一种在计算完其他所有光照项后添加的颜色。更加高级的全局照明模型将模拟发射光是如何影响场景的其余部分的，但是这些模型超出

了本书的范畴。

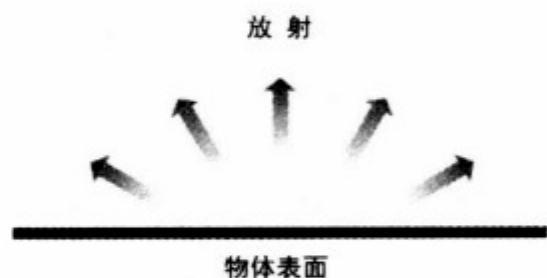


图 5-2 放射项



图 5-3 放射项的渲染

下面是我们用于放射项的数学公式：

$$\text{emissive} = K_e$$

其中：

- K_e 代表材质的放射光颜色。

二、环境反射项

环境反射项代表了光在一个场景里经过多次折射后看起来就像来自四面八方一样。环境反射光看起来并不是来自某个方向的，相反它看起来像是来自所有方向。因为这个原因，环境反射光项并不依赖于光源的位置。图 5-4 举例说明了这个概念，而图 5-5 则显示了一个只受到环境反射光照的物体的一个渲染。环境反射项依赖于一个材质的环境反射能力，以及照射到材质上的环境光的颜色。和放射项一样，环境反射项依赖于它本身，是一种固定的颜色。但是不像放射颜色，环境反射项受全局环境光照的影响。

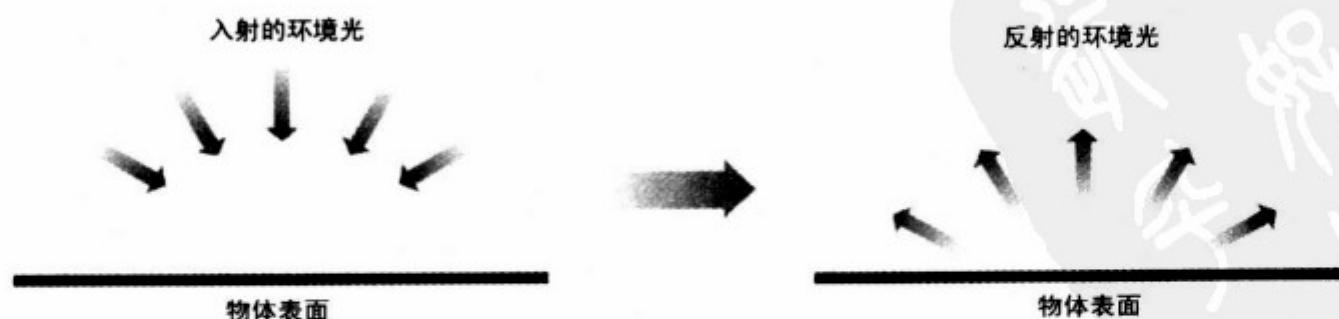


图 5-4 环境反射项



图 5-5 渲染环境反射项

下面是我们用于环境反射项的数学公式：

$$\text{ambient} = K_a \times \text{globalAmbient}$$

其中：

- K_a 是材质的环境反射系数。
- `globalAmbient` 是入射环境光的颜色。

三、漫反射项

漫反射项代表了从一个表面相等地向所有方向反射出去的方向光。通常，漫反射表面在微观尺寸上是非常粗糙的，有许多向很多方向反射光线的凹坑和裂缝。当入射光线到达这些凹坑和裂缝的时候，光线会向各个方向反射，如图 5-6 所示。

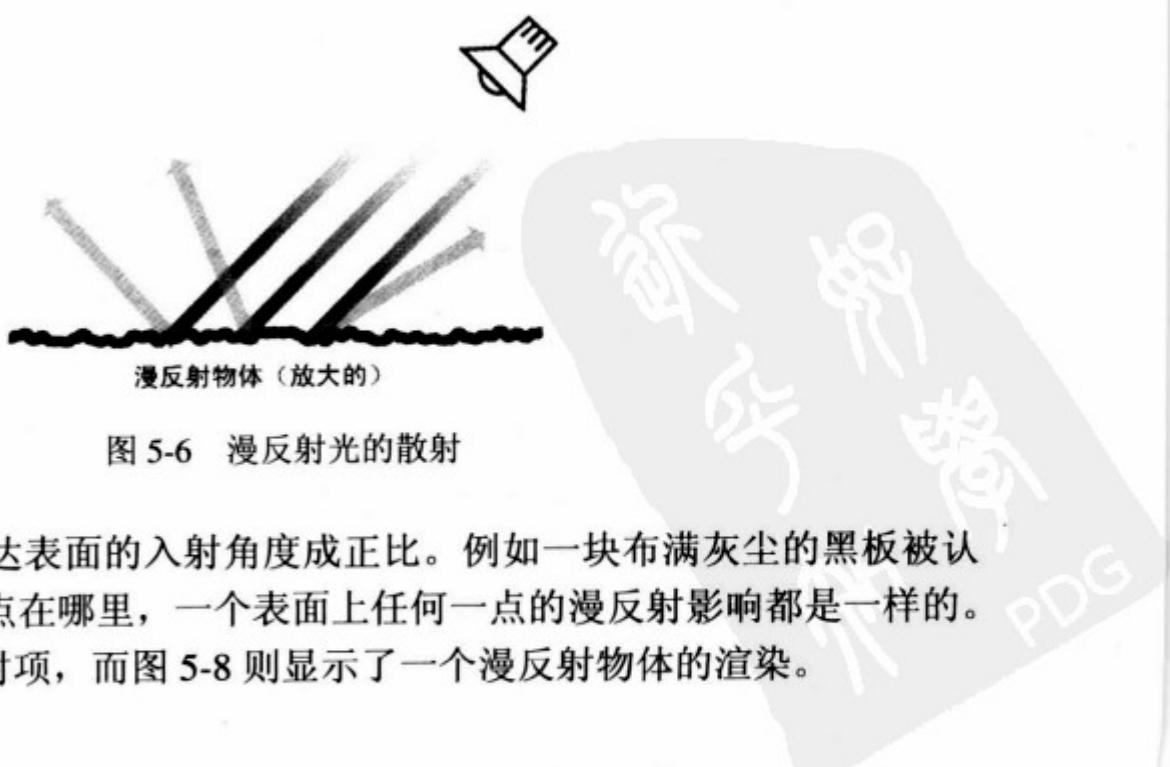


图 5-6 漫反射光的散射

光的反射量与光到达表面的入射角度成正比。例如一块布满灰尘的黑板被认为是漫反射的。无论视点在哪里，一个表面上任何一点的漫反射影响都是一样的。图 5-7 举例说明了漫反射项，而图 5-8 则显示了一个漫反射物体的渲染。

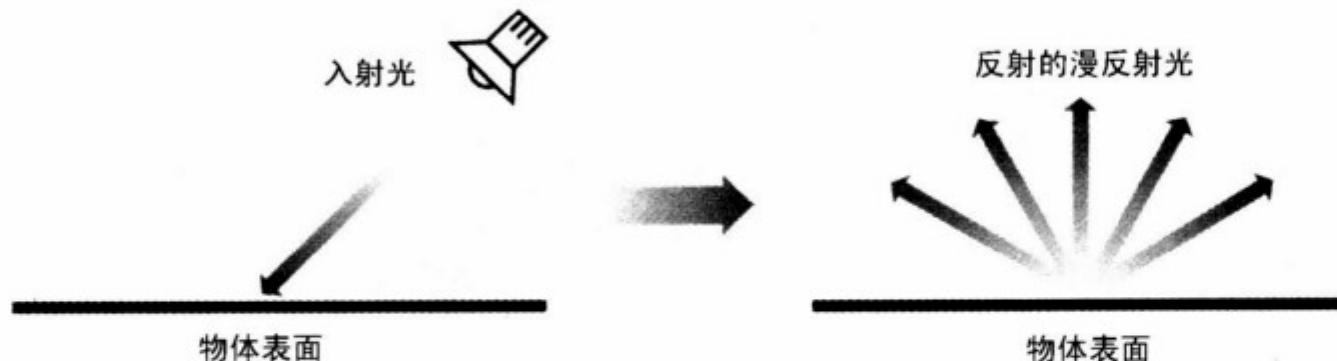


图 5-7 漫反射项



图 5-8 漫反射项的渲染

下面是我们用来计算漫反射项的数学公式（在图 5-9 中显示了）：

$$\text{diffuse} = K_d \times \text{lightColor} \times \max(N \cdot L, 0)$$

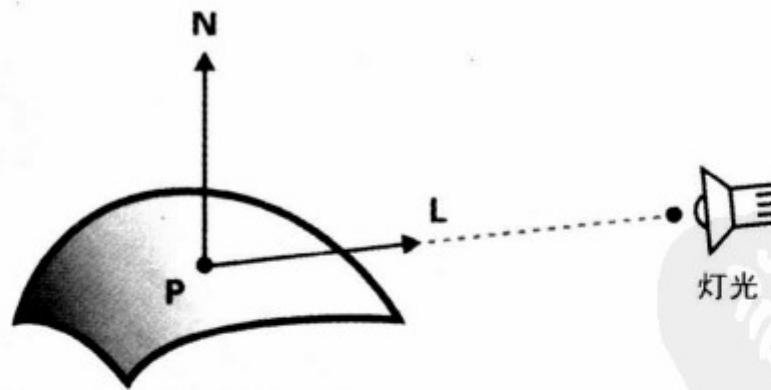


图 5-9 漫反射光照的计算

其中：

- K_d 是材质的漫反射颜色。
- lightColor 是入射漫反射光的颜色。
- N 是规范化的表面法向量。
- L 是规范化的指向光源的向量。

- P 是被着色的点。

规范化的向量 N 和 L 的点积（或内积）是两个向量之间夹角的一个度量。两个向量之间的夹角越小，点积的值越大，而表面会受到更多的入射光照。背向光源的表面将产生负的点积值，因此在公式中的 $\max(N \cdot L, 0)$ 项确保了这样的表面不会显示漫反射光照。

四、镜面反射项

镜面反射项代表了从一个表面主要的反射方向附近被反射的光。镜面反射项在非常光滑和光泽的表面上是最显著的，例如被磨光的金属。图 5-10 举例说明了镜面反射的概念，而图 5-11 则显示了一个完全镜面反射的物体的渲染。

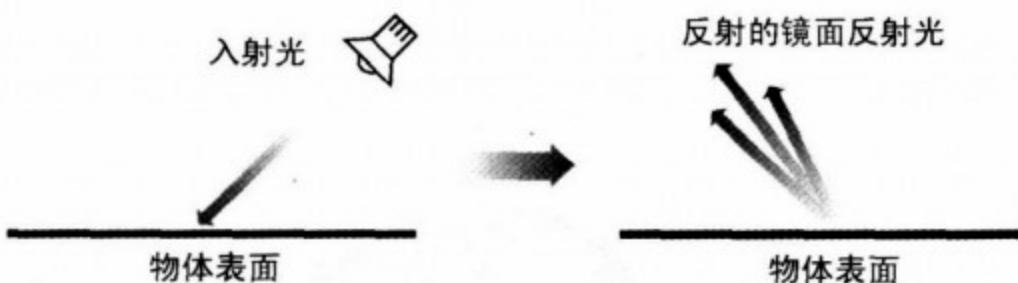


图 5-10 镜面反射项



图 5-11 镜面反射项的渲染

不像放射、环境反射和漫反射光照项，镜面反射的作用依赖于观察者的位置。如果观察者不在一个能够接收到反射光线的位置，观察者将不可能在表面上看到一个镜面反射强光。镜面反射项不仅受光源和材质的镜面反射颜色性质的影响，而且受表面的光泽度的影响。越有光泽的材质的高光区越小，而较少光泽的材质的高光区则分散的很开。图 5-12 显示了光泽的一些例子，光泽度指数从左到右增加。

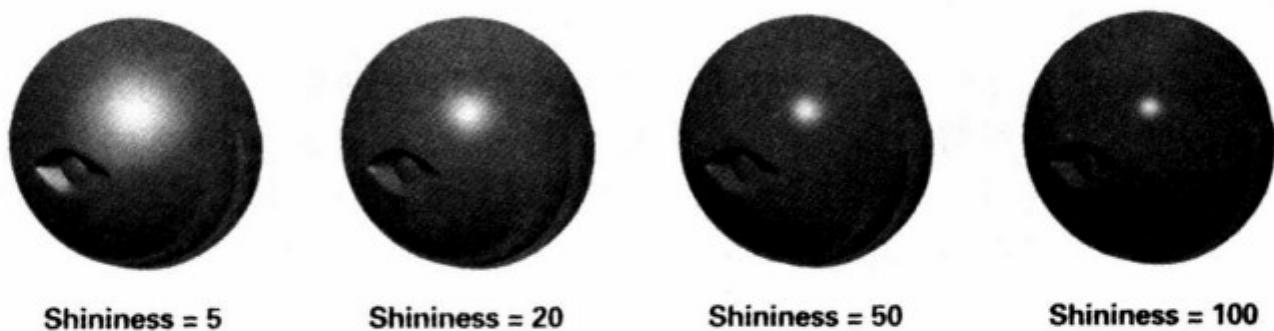


图 5-12 不同光泽指数的例子

下面是我们用来计算镜面反射项的数学公式（如图 5-13 所示）：

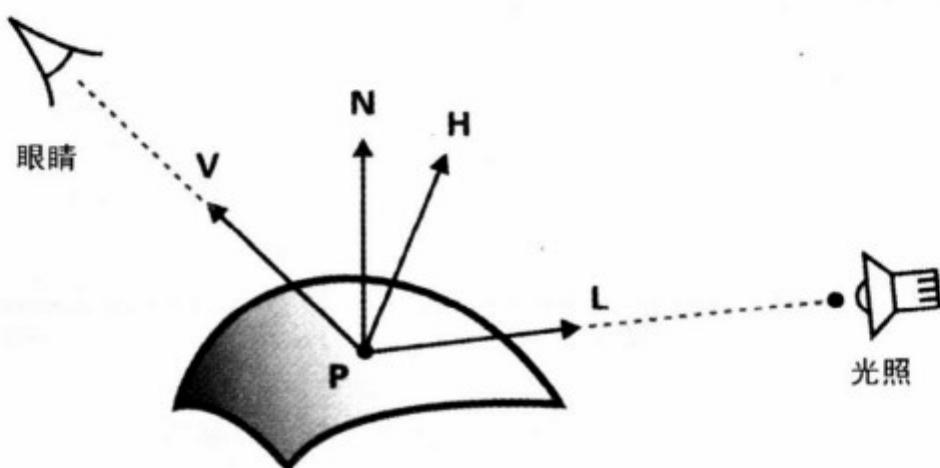


图 5-13 计算镜面反射项

$$\text{specular} = K_s \times \text{lightColor} \times \text{facing} \times (\max(N \cdot H), 0)^{\text{shininess}}$$

其中：

- K_s 是材质的镜面反射颜色。
- lightColor 是入射镜面反射光的颜色。
- N 是规范化的表面法向量。
- V 是指向视点的规范化的向量。
- L 是指向光源的规范化的向量。
- H 是 V 和 L 的中间向量的规范化向量。
- P 是要被着色的点。
- facing 是 1 如果 $N \cdot L$ 是大于 0 的，否则为 0。

当视向量 V 和半角向量 H 之间的夹角很小的时候，材质的镜面反射外表将变得很明显。 N 和 H 的点积的幂确保了镜面反射外表当 H 和 V 分开的时候能够迅速

减弱。

另外，如果因为 $\mathbf{N} \cdot \mathbf{L}$ （来自漫反射光照）是负值，漫反射项为 0 的话，镜面反射项将被强制设为 0。这能确保镜面反射高光不出现在背向灯源的几何表面上。

五、把所有项加在一起

把环境反射、漫反射和镜面反射项结合在一起将给出最后的光照，如图 5-14 所示。在这幅图中，我们故意地把放射项排除在外，因为它通常被用来实现特殊效果而不是为了照亮普通的物体。

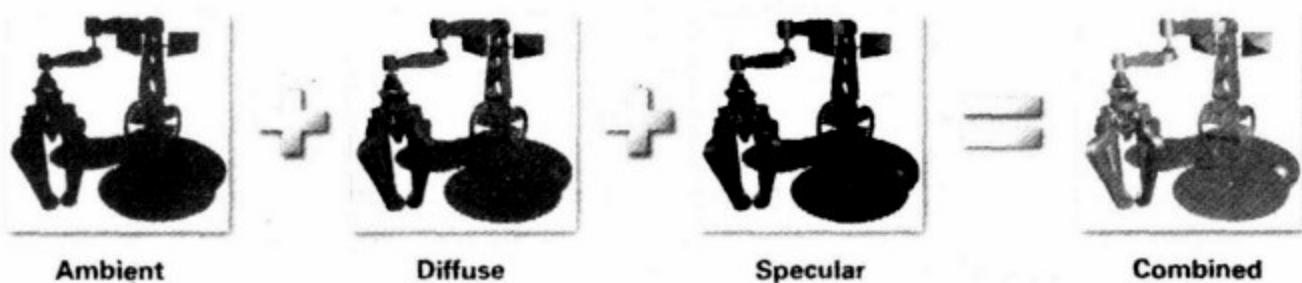


图 5-14 把所有项综合起来

六、简化

如果你有使用 OpenGL 或 Direct3D 的经验，你也许已经注意到了对基本光照模型的许多简化。我们使用了一个全局的环境反射颜色，而不是每个光源一个环境反射颜色。我们还为光的漫反射和镜面反射使用了同样的颜色，而不允许它们每个取不同的值。另外，我们没有引入衰减或聚光灯效果。

5.2.2 一个基本的每个顶点光照的顶点程序

在本节我们将详细地解释一个 Cg 顶点程序，这个程序实现了在第 5.2.1 小节中描述的基本光照模型。

在示例 5-1 中的 `C5E1v_basicLight` 顶点程序做了以下工作：

- 把顶点的位置从物体空间变换到剪裁空间。
- 计算顶点的光照颜色，包括来自一个单独光源的放射、环境反射、漫反射和镜面反射光照作用。

示例 5-1 C5E1v_basicLight 顶点程序

```

void C5E1v_basicLight(float4 position : POSITION,
                      float3 normal : NORMAL,
out float4 oPosition : POSITION,
out float4 color : COLOR,
uniform float4x4 modelViewProj,
uniform float3 globalAmbient,
uniform float3 lightColor,
uniform float3 lightPosition,
uniform float3 eyePosition,
uniform float3 Ke,
uniform float3 Ka,
uniform float3 Kd,
uniform float3 Ks,
uniform float shininess)
{
    oPosition = mul(modelViewProj, position);

    float3 P = position.xyz;
    float3 N = normal;

    // Compute the emissive term
    float3 emissive = Ke;

    // Compute the ambient term
    float3 ambient = Ka * globalAmbient;

    // Compute the diffuse term
    float3 L = normalize(lightPosition - P);
    float diffuseLight = max(dot(N, L), 0);
    float3 diffuse = Kd * lightColor * diffuseLight;

    // Compute the specular term
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), 0),
                           shininess);
    if (diffuseLight <= 0) specularLight = 0;
    float3 specular = Ks * lightColor * specularLight;

    color.xyz = emissive + ambient + diffuse + specular;
    color.w = 1;
}

```

在这个例子中，我们在物体空间执行光照计算。假如你把所有需要的向量变换到正确的坐标系统，你也能够使用其他空间。例如，OpenGL 和 Direct3D 在眼空间执行它们的光照计算而不是物体空间。当场景中有多个光源的时候，眼空间要比物体空间更加有效，但是物体空间更容易实现。

在本章结尾的练习部分探讨了眼空间和物体空间光照的性能问题。

一、由应用程序指定的数据

表 5-1 列出了应用程序需要传递给图形流水线的各种不同类型的数据。我们把每一项分类成变化 (varying) (如果这项随着每个顶点改变)，或者统一 (uniform) (如果这项几乎不变) (例如在一个每个物体的基础上)。

表 5-1 应用程序为图形流水线指定的数据

参数	变量名	类型	分类
几何参数			
物体空间的顶点位置	Position	float4	变化
物体空间的顶点法向量	normal	float3	变化
连接在一起的 modelview 和 projection 矩阵	modelViewProj	float4×4	统一
物体空间灯源位置	lightPosition	float3	统一
物体空间的眼睛位置	eyePosition	float3	统一
灯光参数			
光的颜色	lightColor	float3	统一
全局环境反射颜色	globalAmbient	float3	统一
材质参数			
放射系数	Ke	float3	统一
环境反射系数	Ka	float3	统一
漫反射系数	Kd	float3	统一
镜面反射系数	Ks	float3	统一
光泽度	shininess	float	统一

二、一个调试技巧

正如你所看到的，用于光照的代码比你迄今为止见到的任何 Cg 代码都要复杂得多。当你正在工作的程序并非简单的时候，慢慢地创建这个程序是一个好主意。每当你增加了一些代码，你最好运行一下程序并察看一下结果，以确保输出的结果是你所预期的。这比为程序编写完全部的代码，然后希望它能生成正确的结果要明智许多。如果你犯了一个微小的错误，如果你知道也许是最近的修改引起的错误，那么，跟踪一个问题将变得很容易。

这个技巧特别适用于光照代码，因为光照可以被分解成不同的作用（放射、环境反射、漫反射和镜面反射）。因此，一个好办法是计算 emissive，然后只把 color 设为 emissive。随后计算 ambient，并把 color 设为 emissive+ambient。通过用这种方法逐渐创建你的 Cg 程序，你将能够少受很多挫折。

三、顶点程序体

1. 计算剪裁空间位置

我们从计算给光栅器的通常的剪裁空间位置开始，如在第 4 章所解释的那样：

```
oPosition = mul(modelViewProj, position);
```

下一步，我们将实例化一个变量来存储物体空间的顶点位置，因为我们在以后需要这个信息。我们将使用一个 float3 临时变量，因为其他在光照中被使用的向量（例如表面法向量、光源位置和眼睛的位置）也都是 float3 类型。

```
float3 P = position.xyz;
```

在这里我们看到了一个有趣的新语法：position.xyz。这是我们第一次看到被称为重组（swizzling）的 Cg 的一个功能。

通过忽略物体空间的 w 分量，我们可以有效地假设 w 为 1。

2. 重组（Swizzling）

重组允许你重新安排一个向量的分量来创建一个新的向量——使用任何你选择的方法。重组使用与被用来存取结构成员的相同的句号操作符，再加上一个后缀用来指出你想要如何重新安排你正在工作的一个向量的分量。这个后缀是字母 x、y、z 和 w 的一些组合。字母 r、g、b 和 a——适用于 RGBA 颜色——也能够被使用。但是，这两组后缀字母不能被混合使用。当创建一个新的向量的时候，这些字母指出了原来向量中的哪些分量将被使用。字母 x 和 r 对应于一个向量的第一个分量，y 和 g 对应于第二个分量，以此类推。在前面的例子中，position 是一个

`float4` 类型的变量。`.xyz` 后缀提取了 `position` 的 `x`、`y` 和 `z` 分量，并从这三个值创建了一个新的三元向量。这个新的向量则被赋给了 `float3` 类型的向量 `P`。

C 和 C++都不支持重组，因为这两种语言都没有内置的对向量数据的支持。但是，重组对在 Cg 中有效地操作向量是非常有用的。

下面是一些重组的例子：

```
float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);
float2 vec2 = vec1.yx;                                // vec2 = (-2.0, 4.0)
float scalar = vec1.w;                                // scalar = 3.0
float3 vec3 = scalar.xxx                            // vec3 = (3.0, 3.0, 3.0);
```

仔细地看一下这 4 行代码。第一行声明了一个命名为 `vec1` 的 `float4` 类型的变量。第二行提取了 `vec1` 的 `y` 和 `x` 分量，并从它们创建了一个新的交换了 `x` 和 `y` 的 `float2` 类型，并把这个向量赋给 `vec2`。在第三行，`vec1` 的 `w` 分量被赋给了一个单独的 `float`，命名为 `scalar`。最后，在最后一行，一个 `float3` 类型的向量是通过复制 `scalar` 3 次创建的。这被称为拖影 (smearing)，它证明了 Cg 对待标量值就像一个分量的向量（也就是 `.x` 后缀被用来存取标量的值）。

你还能够重组矩阵，以基于一系列的矩阵元素来创建向量。为了实现这点，使用 `.m<行><列>` 的符号。你能够把一系列的矩阵重组连在一起，而结果将是一个大小合适的向量。例如：

```
float4x4 myMatrix;
float    myFloatScalar;
float4   myFloatVec4;

// 把 myFloatScalar 设成 myMatrix[3][2]
myFloatScalar = myMatrix._m32;

// 把 myMatrix 的主对角线赋给 myFloatVec4
myFloatVec4 = myMatrix._m00_m11_m22_m33;
```

另外，你能够使用 `[]` 数组操作符来存取矩阵的单独的一行。使用在前面代码示例中声明的变量：

```
// 把 myFloatVec4 设成 myMatrix 的第一行
myFloatVec4 = myMatrix[0];
```

3. 写入掩码

Cg 支持的另一个与重组相关的操作是写入掩码 (write masking)，它只允许指定的向量分量通过一个赋值语句被更新。例如，你只能够通过使用一个 `float2` 类型

的向量写入一个 float4 类型向量的 x 和 w 分量:

```
//假设初始的时候 vec1=(4.0, -2.0, 5.0, 3.0)
//                                vec2=(-2.0, 4.0)
vec1.xw = vec2; //现在 vec1=(-2.0, -2.0, 5.0, 4.0)
```

写入掩码后缀可以用任何顺序列出 x、y、z 和 w (或 r、g、b 和 a) 分量。每个字母在一个给定的写入掩码后缀中最多能够出现一次，而且你不能够在一个写入掩码后缀中混合 xyzw 和 rgba 字母。



在最先进的图形处理器中，重组和写入掩码是没有任何性能损失的操作。因此，使用这两个功能能够帮助你提高代码的清晰性或有效性。

4. 放射光照作用

对放射项没有什么可以做的。为了使代码清楚，我们为放射光照作用定义了一个变量，命名为 emissive:

```
//计算放射项
float3 emissive = Ke;
```



当 Cg 编译器把你的程序翻译成可执行代码的时候，它还对被翻译的代码进行优化，因此在上面的代码片段中创建中间变量（例如 emissive 变量），是没有任何性能损失的。因为实例化这样的变量使得你的代码更加可读，你被鼓励为中间结果使用实例命名的变量来提高你的代码的可读性。

5. 环境反射光照作用

对环境反射光照，还记得我们不得不用材质的环境反射颜色 Ka，并用全局环境反射光的颜色乘以它。这是一个每个分量的乘法，意味着我们必须提取 Ka 的每个分量，并用全局环境反射光的颜色的对应的颜色乘以它。下面的代码同时使用了重组和写入掩码，来完成这项工作：

```
//一个效率很差的计算环境反射项的方法
float3 ambient;
ambient.x = Ka.x * globalAmbient.x;
ambient.y = Ka.y * globalAmbient.y;
ambient.z = Ka.z * globalAmbient.z;
```

这段代码能够正常工作，但是它看起来确实很费事，而且形式上也不够优雅。

因为 Cg 对向量有本地支持，它允许你更简洁地表达这种类型的操作。下面是一个更简洁地用一个向量缩放另一个向量的方法：

```
//计算环境反射项
float3 ambient = Ka * globalAmbient;
```

非常简单，不是吗？在一个对向量和矩阵以及对它们执行的普通操作内置支持的语言下工作是非常方便的。

6. 漫反射光照作用

现在，我们进入光照模型更加有趣的部分。为了漫反射计算，我们需要从顶点到光源的向量。要定义向量，你可以用目标点减去起始点。在这种情况下，向量结束于 lightPosition 而起始于 P：

```
//计算光向量
float3 L = normalize(lightPosition-P);
```

我们只对方向感兴趣，而不是长度，因此我们需要对向量进行规范化。幸运地是，在 Cg 标准函数库中声明了一个 normalize 函数，能够返回一个向量的规范化版本。如果一个向量没有正确地规范化，光照将会太暗或者太亮。

normalize (v)	返回向量 v 的一个规范化的版本
---------------	------------------

下一步，我们需要进行实际的光照计算。这是一个稍微复杂的表达式，因此我们一段段地考虑它。首先，公式里有一个内积。回忆一下，内积是一个基本的计算单独一个值的数学函数，它代表了两个单位向量夹角的余弦值。在 Cg 中，你能够使用 dot 函数来计算两个向量间的点积：

dot (a, b)	返回向量 a 和 b 的点积
------------	----------------

因此，计算 N 和 L 间的点积的代码片段是：

```
dot (N, L);
```

但是，这里有一个问题。背向光源的表面被用“负”的光照亮，因为当法向量背向光源的时候点积的值是负的。负的光照值没有任何物理意义，而且在光照公式中当和其他项加在一起的时候会引起错误。为了处理这个问题，你必须把这样的结果映射为 0。这意味着如果点积的值小于 0，则它将被设成 0。这个映射操作使用 Cg 的 max 函数非常容易实现：

max (a, b)	返回 a 和 b 中的最大值
------------	----------------

把映射增加到前面的表达式中得到：

```
max (dot (N, L), 0);
```

因此，最后的代码片段看起来是这样的：

```
float diffuseLight = max (dot (N, L), 0);
```

最后，你需要加入漫反射材质颜色 (Kd) 和光的颜色 (lightColor) 等因素。你刚刚计算的 diffuseLight 值是一个标量。记住在 Cg 中你能够用一个标量乘以一个向量，这么做将会用这个标量缩放向量中的每个分量。因此，你能够用两个乘法把这些颜色很容易地结合起来：

```
float3 diffuse = Kd * lightColor * diffuseLight;
```

7. 镜面反射光照作用

镜面反射计算需要更多一点地工作。返回去看一下图 5-13，这幅图显示了你需要的各种向量。你已经从漫反射计算中获得了 L 向量，但是 V 和 H 向量仍然需要计算。考虑到你已经有了眼睛位置 (eyePosition) 和顶点位置 (P)，这并不是很难。

我们从计算从顶点到眼睛的向量开始。这个向量通常被称为视向量 (view vector)，或者如在示例代码中那样简单地命名为 V。因为我们正在试图定义一个方向，我们应该规范化这个向量。下面的代码是所得的结果：

```
float3 V = normalize(eyePosition - P);
```

下一步，你需要 H，这个向量是光向量 L 和视向量 V 的中间向量。因为这个原因，H 被称为半角向量 (half-angle vector)。和 V 一样，H 需要被规范化，因为它代表了一个方向。要计算 H，你可以使用下面的表达式：

```
//一个效率很低的计算 H 的方法
```

```
float3 H = normalize(0.5*L+0.5*V);
```

但是，因为你正在做一个规范化操作，用 0.5 来缩放 L 和 V 没有任何作用，因为在规范化过程中缩放因子将被抵消。因此，实际的代码应该是这样的：

```
float3 H = normalize(L+V);
```

在这点上，你已经计算了镜面反射项。与漫反射项一样，我们一段一段地为镜面反射项构造表达式。我们从 H 和 N 的点积开始：

```
dot(N, H)
```

和漫反射光照一样，结果也需要被限制成大于 0：

```
max (dot (N, H), 0)
```

这个结果需要使用 shininess 指定的指数求幂。当 shininess 的值增加的时候，这将产生缩小镜面反射高光的效果。要求一个量的幂，可以使用 Cg 的 pow 函数：

pow (x, y)	返回 x^y
------------	----------

增加 pow 函数到镜面反射光照表达式得到:

```
pow(max(dot(N, H), 0), shininess);
```

把所有综合在一起, 你将得到镜面反射光照:

```
float specularLight = pow(max(dot(N, H), 0), shininess);
```

最后, 你必须确保当漫反射光照是 0 的时候 (因为表面背向光源), 镜面反射高光不会出现。换句话说, 也就是如果漫反射光照是 0, 则把镜面反射光照也设成 0。否则, 则使用计算的镜面反射光照值。这是一个很好的使用 Cg 的条件表达式功能的机会。

8. 条件表达式

和在 C 中一样, Cg 允许你使用关键字 if 和 else 来估算条件表达式。例如:

```
if (value == 1) {
    color = float4(1.0, 0.0, 0.0, 1.0); //颜色为红色
} else {
    color = float4(0.0, 1.0, 0.0, 1.0); //颜色为绿色
}
```

还和 C 一样, 你可以使用? : 符号来非常简洁地实现条件表达式。?: 符号的使用如下所示:

```
(test expression) ? (statements if true)
                  : (statements if false)
```

因此, 上面的示例可以被表达成:

```
color = (value==1) ? float4(1.0, 0.0, 0.0, 1.0)
                  : float4(0.0, 1.0, 0.0, 1.0);
```

回到我们原来的例子中, 下面是包括了条件测试的镜面反射光照代码:

```
float specularLight = pow(max(dot(N, H), 0), shininess);
if (diffuseLight<=0) specularLight=0;
```

和漫反射光照计算一样, 你需要加入材质的镜面反射颜色 (Ks) 和光的颜色 (lightColor) 等因素。起先, 使用两个单独的颜色来控制镜面反射高光看起来有些奇怪。但是, 这是非常有用的, 因为有些材质 (例如金属) 有与材质颜色相似的镜面反射高光, 但是其他材质 (例如塑料) 的镜面反射高光则是白色的。然后, 两种高光都经过光的颜色的调制。Ks 和 lightColor 变量是一种方便的调整光照模型来获得某种特别效果的方法。

镜面反射分量可以按如下方式计算:

```
float3 specular = Ks * lightColor * specularLight;
```

9. 集成在一起

最后一步是把放射、环境反射、漫反射和镜面反射作用综合起来以获得最后的顶点颜色。你还需要把这个颜色赋给名目为 color 的输出参数：

```
Color.xyz = emissive + ambient + diffuse + specular;
```

5.2.3 每个顶点光照的片段程序

因为顶点程序已经执行了光照计算，你的片段程序只需要接受插值的颜色并把它传给帧缓冲。我们重新使用 C2E2f_passthrough 片段程序来完成这项任务。现在，我们完成了所有的工作。

5.2.4 单个顶点光照结果

图 5-15 显示了每个顶点光照程序的一个渲染示例。



图 5-15 每个顶点光照的结果

5.3 单个片段光照

你也许已经注意到了每个顶点光照的结果看上去有些粗糙。着色趋向于有一点“三角形”的外表，意味着如果模型足够简单，你可以分辨出底层的网格结构。

如果模型只有很少的顶点，每个顶点的光照将不够充分。但是，当你的模型获得了越来越多的顶点，你将发现结果开始得到相当的改善，如图 5-16 所示。这幅图显示了 3 个不同镶嵌等级的被镶嵌的圆柱体。在每个被照亮的圆柱体下面是模型的线框图版本，显示了每个圆柱体的镶嵌情况。镶嵌的量从左向右增加，可以看到光照效果改善的很明显。

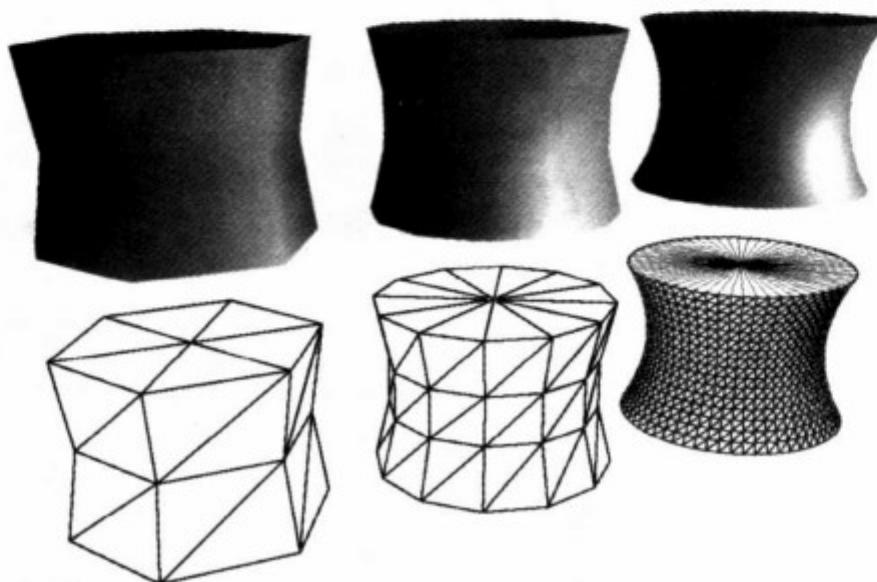


图 5-16 镶嵌在光照下的效果

由于数据被插值和使用的方式，镶嵌少的模型使用每个顶点的光照看起来效果很差。使用每个顶点的光照，光照只在每个三角形的每个顶点计算。然后，光照为每个三角形所生成的片段进行插值。这个方法被称为光滑颜色插值或 Gouraud 着色，它会丢失一些细节因为光照公式并不是真正地为每个片段估算的。例如，一个镜面反射高光没有被任何一个三角形的顶点捕捉到，则不会显示在三角形上，即使它本来应该显示在三角形中。

使用前面的每个顶点光照示例所发生的情况是这样的：你的顶点程序计算了光照，然后光栅器为每个片段对颜色进行插值。

为了获得一个更加精确的结果，你需要为每个片段估算整个光照模型，而不是仅仅只为每个顶点计算。因此，表面法向量将被插值，而不是插值最后的光照颜色。然后，片段程序使用插值的表面法向量在每个像素计算光照。这种技术被称为 Phong 着色（不要与 Phone 光照模型相混淆，Phone 光照模型指的是在基本光照模型中使用的镜面反射的近似）或更普通一点每个像素光照（per-pixel lighting）或每个片段光照（per-fragment lighting）。正如你所预期的，每个片段的

光照给出了更好的结果，因为整个光照公式是为每个三角形的每个片段计算的，如图 5-17 所示。图的左边显示了一个用每个顶点光照渲染的一个被镶嵌的圆柱体，而右边则显示了用每个片段光照渲染的同样的圆柱体。每个圆柱体有着同样粗糙程度的镶嵌。注意一下，高光在左边的圆柱体很粗糙，而在右边的圆柱体则比较明显。

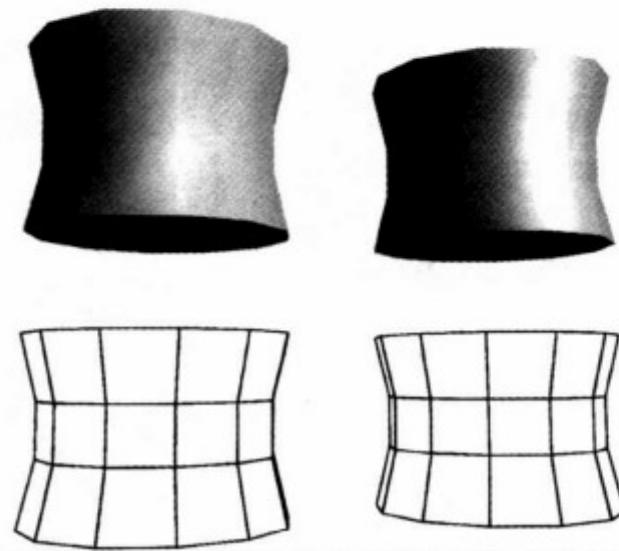


图 5-17 每个顶点光照和每个片段光照的比较

5.3.1 实现每个片段的光照

CineFX

在这个例子中的片段程序需要第四代的图形处理器，例如 NVIDIA 的 GeForce FX 或者 ATI 的 Radeon 9700。

在这个例子中，顶点和片段程序的计算负担将会被交换。这次，片段程序将会用来做有趣的工作。而顶点程序将只是帮助打基础，为片段程序传递一些参数。许多高级技术也会按照这个模式进行，这并不让人很吃惊，因为片段程序与顶点程序相比能够让你对最后的图像有更详细的控制。使用一个顶点或片段程序还有其他一些牵连，例如性能。第 10 章将会进一步讨论这个主题。

你将发现用于每个片段光照的片段程序与用于每个顶点光照的顶点程序看上去非常地相似。这还是因为 Cg 允许你使用共同的语言来进行顶点和片段编程。这种能力证明对每个片段的光照非常有用，因为你已经对所需要的代码很熟悉了。和每个顶点的光照一样，每个片段的光照也是在物体空间中实现的。

5.3.2 用于每个片段光照的顶点程序

这个示例的顶点程序只是一个传输管道：它执行最小限度的计算和传输必要的数据给流水线，因此片段程序能够用来做一些有趣的工作。在输出齐次位置以后，顶点程序还传递物体空间的位置和物体空间法向量，它们被输出到纹理坐标集 0 和 1 中。

示例 5-2 显示了顶点程序的完整代码。在代码中没有任何新概念，因此抓住这个机会确保你已经完全理解了程序中的每一行代码。

示例 5-2 C5E2v_fragmentLighting 顶点程序

```
void C5E2v_fragmentLighting(float4 position : POSITION,
                           float3 normal : NORMAL,
                           out float4 oPosition : POSITION,
                           out float3 objectPos : TEXCOORD0,
                           out float3 oNormal : TEXCOORD1,
                           uniform float4x4 modelViewProj)
{
    oPosition = mul(modelViewProj, position);
    objectPos = position.xyz;
    oNormal = normal;
}
```

5.3.3 用于每个片段光照的片段程序

C5E3f_basicLight 程序与用于每个顶点光照的顶点程序几乎完全一样，因此我们将不会对它进行详细地讨论。示例 5-3 显示了用于每个片段光照程序的源代码。

假设物体空间的每个顶点法向量已经被规范化了是很平常的。在这种情况下，C5E3f_basicLight 执行了一个操作，而对应的顶点程序不需要的是对插值的每个片段的法向量重新进行规范化：

```
float3 N = normalize(normal);
```

这个 normalize 是必须的，因为一个纹理坐标集的线性插值会使得每个片段的法向量不再规范化。

示例 5-3 C5E3f_basicLight 片段程序

```

void C5E3f_basicLight(float4 position : TEXCOORD0,
                      float3 normal : TEXCOORD1,
out float4 color : COLOR,
uniform float3 globalAmbient,
uniform float3 lightColor,
uniform float3 lightPosition,
uniform float3 eyePosition,
uniform float3 Ke,
uniform float3 Ka,
uniform float3 Kd,
uniform float3 Ks,
uniform float shininess)
{
    float3 P = position.xyz;
    float3 N = normalize(normal);

    // Compute the emissive term
    float3 emissive = Ke;

    // Compute the ambient term
    float3 ambient = Ka * globalAmbient;

    // Compute the diffuse term
    float3 L = normalize(lightPosition - P);
    float diffuseLight = max(dot(N, L), 0);
    float3 diffuse = Kd * lightColor * diffuseLight;

    // Compute the specular term
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), 0),
                           shininess);
    if (diffuseLight <= 0) specularLight = 0;
    float3 specular = Ks * lightColor * specularLight;

    color.xyz = emissive + ambient + diffuse + specular;
    color.w = 1;
}

```

C5E3f_basicLight 片段程序显示了 Cg 真的允许你在顶点和片段程序中用同样的方法表示你的想法（只要你的图形处理器足够强大——这个程序在编写的时候需要第四代或更好的图形处理器）。但是每个片段的计算并不是免费的。在大多数情况下，在一帧中的片段的数目将比顶点的数目多很多，这意味着片段程序需要比顶点程序多运行很多次。因此，较长的片段程序将比较长的顶点程序对性能造成更严重的影响。第 10 章将更详细地讨论使用顶点和片段程序之间的平衡问题。

在本章的其他部分和本书的其余部分，在可能的情况下我们将避免复杂的片段程序，以使得这些例子能够运行于一个范围更广泛的图形处理器。但是，如果你希望的话，你通常能够把一个每个顶点的计算移到片段程序中。

5.4 创建一个光照函数

在前面的小节中，我们只是简单地从每个顶点示例中拷贝了大部分的代码到每个片段的示例中，然而我们有一个更好的解决方案：把光照的关键部分封装在一个函数里。

在一个复杂的 Cg 程序中，光照也许只是程序所执行的几个计算中的一个。在每个顶点的光照例子中，你看到了计算光照所需要的各个步骤。但是，无论何时你要计算光照的时候，你都不会想要重写所有这些代码。幸运的是，你不需要这样做。正如我们在第 2 章中提到过的，你能够编写一个内部函数来封装光照功能，然后可以在不同的入口函数中重用相同的光照函数。

不像在 C 或 C++ 中的函数，在 Cg 中的函数通常都是内嵌的（虽然这需要依赖于 profile——高级的 profile，例如 vp30 除了内嵌以外还能够支持函数调用）。内嵌函数意味着它们没有与之相关联的函数调用的开销。因此，当可能的时候你应该使用函数，因为它们能够提高可读性、简化调试、鼓励重用，并且能够使得未来的优化更加容易。

Cg 和 C 一样需要你在使用一个函数之前必须先声明这个函数。

5.4.1 声明一个函数

在 Cg 中，函数就像在 C 中那样被声明。你可以随意地指定传递给函数的参数，以及将被函数返回的值。下面是一个简单的函数声明：

```
float getX (float3 v)
{
    return v.x;
}
```

这个函数采用了一个三元向量 v 作为一个参数，并且将 v 的 x 分量作为返回值，其类型为 float。关键字 return 被用来返回函数的结果。你可以像调用任何其他 Cg 函数那样调用 getX 函数：

```
// 声明一个临时使用的向量
float3 myVector = float3(0.5, 1.0, -1.0);
```

```
// 取得 myVector 的 x 分量
float x = getX(myVector);
// 现在 x=0.5
```

有些时候，你想要一个函数返回几个结果而不仅仅是一个结果。在这种情况下，你能够使用 out 修饰符（如 3.3.4 节中解释的）来指定一个程序的某个特定的参数只用于输出。下面的例子用一个向量作为输入，然后返回它的 x、y 和 z 分量。

```
void getComponents(float3 vector,
                  out float x,
                  out float y,
                  out float z)
{
    x = vector.x;
    y = vector.y;
    z = vector.z;
}
```

注意这个函数被声明为 void 类型，因为它通过参数来返回所有的值。下面的代码示例显示了 getComponents 是如何被使用的：

```
// 声明一个临时使用的向量
float3 myVector = float3(0.5, 1.0, -1.0);

// 声明临时使用的变量
float x, y, z;

// 获得 myVector 的 x、y 和 z 分量
getComponents(myVector, x, y, z);
// 现在 x=0.5, y=1.0, z=-1.0
```

5.4.2 一个光照函数

因为光照是一个复杂的过程，你能够编写许多不同类型的光照函数，每个函数都能够接受不同的参数。现在，你只需要采用你实现的简单模型，并为它创建一个函数。下面是这个函数的最基本的样子：

```
float3 lighting ( float3 Ke,
                  float3 Ka,
                  float3 Kd,
                  float3 Ks,
                  float shininess,
                  float3 lightPosition,
                  float3 lightColor,
                  float3 globalAmbient,
                  float3 P,
                  float3 N,
                  float3 eyePosition)

{
    //在这里计算光照
}
```

这个方法的一个主要问题是这个函数需要很多参数。把这些参数组成“材质参数”和“光参数”，然后把每个参数集当成一个单独变量来传递，这将使得整个函数整洁许多。幸运的是 Cg 支持结构，恰好能够提供这种功能。

5.4.3 结构

正如我们在第 2 章提到过的，Cg 的结构使用与 C 和 C++同样的方法来声明。**struct** 关键字被用来声明结构，它后面跟随的是结构的成员。下面是一个结构的例子，它封装了基于基本光照模型的某个材质的所有性质：

```
struct Material {
    float3 Ke;
    float3 Ka;
    float3 Kd;
    float3 Ks;
    float shininess;
};
```

结构的成员可以通过逗号操作符来进行存取。下面的代码片段显示了如何声明和存取一个结构：

```
Material shiny;
shiny.Ke = float3 (0.0, 0.0, 0.0);
shiny.Ka = float3 (0.1, 0.1, 0.1);
shiny.Kd = float3 (0.2, 0.4, 0.8);
shiny.Ks = float3 (0.8, 0.8, 0.8);
shiny.shininess = 90.0;
```

你还能够创建第二个结构来保存光的性质：

```
struct Light {
    float4 position;
    float3 color;
};
```

现在，你可以使用结构作为参数来改进一下光照函数：

```
float3 lighting (Material material,
    Light light,
    float3 globalAmbient,
    float3 P,
    float3 N,
    float3 eyePosition)
{
    //在这里计算光照
}
```

使用这个方法能够在以后使你的光或材质模型更加复杂，而不用在光照函数本身增加更多的参数。另外一个优点是，你能够通过使用一个 Light 结构的数组而不仅仅是一个结构来计算多个光源的效果。

5.4.4 数组

Cg 像 C 那样支持数组。因为 Cg 现在不支持指针，所以，当处理数组的时候，你必须使用数组语法，而不能使用指针语法。下面是一个数组在 Cg 中声明和存取的例子：

```
//声明一个 4 个元素的数组
float3 myArray[4];
int index = 2;
```

```
//把一个向量赋给数组的第 2 个元素
myArray[index] = float3 (0.1, 0.2, 0.3);
```

Advanced

一个与 C 的重要区别是数组在 Cg 中是第一类的数据类型 (first-class type)。这意味着数组赋值实际上拷贝了整个数组，而且作为参数传递的数组是通过值来传递的（在做任何修改之前整个数组都拷贝了）而不是通过引用来传递。

你也能把数组作为参数传递给函数。我们将使用这个功能来创建一个函数用来从两个不同的光源计算光照，就像在示例 5-4 中展示的那样。

当你浏览 C5E4v_twoLights 的代码的时候，你将会注意到这个程序从计算 emissive 和环境项开始，这两项都是独立于光源的。然后，这个函数用一个 for 循环来从这两个光源累积漫反射和镜面反射作用。这些作用通过使用 C5E5_computeLighting 帮助函数来计算，我们马上就会定义这个函数。首先，让我们学习一些有关 for 循环和其他用来控制 Cg 程序流的模式的一些概念。

示例 5-4 C5E4v_twoLights 顶点程序

```
void C5E4v_twoLights(float4 position : POSITION,
                      float3 normal   : NORMAL,
                      out float4 oPosition : POSITION,
                      out float4 color     : COLOR,
                      uniform float4x4 modelViewProj,
                      uniform float3 eyePosition,
                      uniform float3 globalAmbient,
                      uniform Light lights[2],
                      uniform float shininess,
                      uniform Material material)
{
    oPosition = mul(modelViewProj, position);

    // Calculate emissive and ambient terms
    float3 emissive = material.Ke;
    float3 ambient = material.Ka * globalAmbient;
```

```

// Loop over diffuse and specular contributions
// for each light
float3 diffuseLight;
float3 specularLight;
float3 diffuseSum = 0;
float3 specularSum = 0;
for (int i = 0; i < 2; i++) {
    CSE5_computeLighting(lights[i], position.xyz, normal,
                          eyePosition, shininess, diffuseLight,
                          specularLight);
    diffuseSum += diffuseLight;
    specularSum += specularLight;
}

// Now modulate diffuse and specular by material color
float3 diffuse = material.Kd * diffuseSum;
float3 specular = material.Ks * specularSum;

color.xyz = emissive + ambient + diffuse + specular;
color.w = 1;
}

```

5.4.5 流控制

Cg 提供了 C 的流控制能力的一个子集。特别地是，Cg 支持：

- 函数和 return 指令
- if-else
- for
- while 和 do-while

这些指令与它们在 C 中的对应指令完全相同，除了有一些 profile 指定的限制存在。例如，一些 profile 允许 for 或者 while 循环，仅当循环迭代的次数能够事先被 Cg 编译器决定的情况。

Cg 保留了用于其他 C 的流控制模式的关键字，例如 goto 和 switch。但是，这些模式现在还没有被支持。

5.4.6 计算漫反射和镜面反射光照

这个难题的最后一部分是 C5E5_computeLighting 函数，它负责为某个特定的光源计算漫反射和镜面反射作用。示例 5-5 重新实现了我们以前编写的漫反射和镜面反射光照代码。

示例 5-5 C5E5_computeLighting 内部函数

```
void C5E5_computeLighting(Light light,
                           float3 P,
                           float3 N,
                           float3 eyePosition,
                           float shininess,

                           out float3 diffuseResult,
                           out float3 specularResult)
{
    // Compute the diffuse lighting
    float3 L = normalize(light.position - P);
    float diffuseLight = max(dot(N, L), 0);
    diffuseResult = light.color * diffuseLight;

    // Compute the specular lighting
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), 0),
                               shininess);
    if (diffuseLight <= 0) specularLight = 0;
    specularResult = light.color * specularLight;
}
```

5.5 扩展基本模型

现在你已经实现了一个基本的光照模型，现在让我们看看如何能够使它变得更加有用。下面几个小节将介绍 3 种增强：距离衰减、聚光灯效果和方向光。每种增加都将同时在顶点级和片段级工作。

光照是一种非常复杂的问题，其中有许多技术需要探索。而我们的目标只是使你入门。

5.5.1 距离衰减

无论光源离被着色的表面有多远，基本的光照模型都假设光源的强度总是一样的。虽然对某些光源这是一个合理的近似（例如，当太阳光照射地球上的物体的时候），我们通常想要创建强度随距离减少的光源。这个性质被称为距离衰减。在 OpenGL 或 Direct3D 中，在任意给定点的衰减使用下面的公式进行模拟：

$$\text{attenuationFactor} = \frac{1}{K_C + K_L d + K_Q d^2}$$

其中：

- d 是到光源的距离。
- K_C 、 K_L 和 K_Q 是控制衰减量的常量。

在这个衰减的公式中， K_C 、 K_L 和 K_Q 分别是衰减的常数项、一次系数和二次系数。在真实世界中，来自一个点光源的光照强度以 $1/d^2$ 衰减，但是有时候这可能不会使你获得想要的效果。使用 3 个衰减参数的想法是使你对场景光照的外表能够有更多的控制。

这个衰减因子被用来调整光照公式的漫反射和镜面反射项。因此，整个公式变成为：

$$\text{lighting} = \text{emissive} + \text{ambient} + \text{attenuationFactor} \times (\text{diffuse} + \text{specular})$$

示例 6-5 描述了在给定表面的位置和 Light 结构（我们在 Light 结构中增加了 K_C 、 K_L 和 K_Q ）之后，一个计算衰减的 Cg 函数。

示例 6-5 C5E6_attenuation 内部函数

```
float C5E6_attenuation(float3 P,
                        Light light)
{
    float d = distance(P, light.position);
    return 1 / (light.kC + light.kL * d +
                light.kQ * d * d);
}
```

我们将利用 `distance` 函数，这个函数是另一个 Cg 标准库函数之一。这是 `distance` 的正式定义：

distance (pt1, pt2) 点 pt1 和 pt2 之间的欧式距离

衰减计算需要加入到 `C5E5_computeLighting` 函数中，因为它同时影响了来自光源的漫反射作用和镜面反射作用。你应该首先计算衰减，这样你就能够用它来调整漫反射和镜面反射的作用。在示例 5-7 中的 `C5E7_attenuateLighting` 内部函数显示了必要的修改。

示例 5-7 C5E7_attenuateLighting 内部函数

```

void C5E7_attenuateLighting(Light light,
                           float3 P,
                           float3 N,
                           float3 eyePosition,
                           float shininess,

                           out float diffuseResult,
                           out float specularResult)
{
    // Compute attenuation
    float attenuation = C5E6_attenuation(P, light);

    // Compute the diffuse lighting
    float3 L = normalize(light.position - P);
    float diffuseLight = max(dot(N, L), 0);
    diffuseResult = attenuation *
                    light.color * diffuseLight;

    // Compute the specular lighting
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), 0),
                               shininess);
    if (diffuseLight <= 0) specularLight = 0;
    specularResult = attenuation *
                     light.color * specularLight;
}

```

5.5.2 增加一个聚光灯效果

另一个经常被使用的对基本光照模型的扩展是，使得灯成为聚光灯来代替全方向的灯光。一个聚光灯的取舍角（cut-off angle）控制了聚光灯圆锥体的传播，如图 5-18 所示。只有在聚光灯圆锥体内的物体才能受到光照。

为了创建聚光灯的圆锥体，你需要知道聚光灯的位置、聚光灯的方向和将要试图进行着色的点的位置。使用这些信息你就能够计算向量 V （从聚光灯到顶点的向量）和向量 D （聚光灯的方向），如在图 5-19 中所显示的那样。

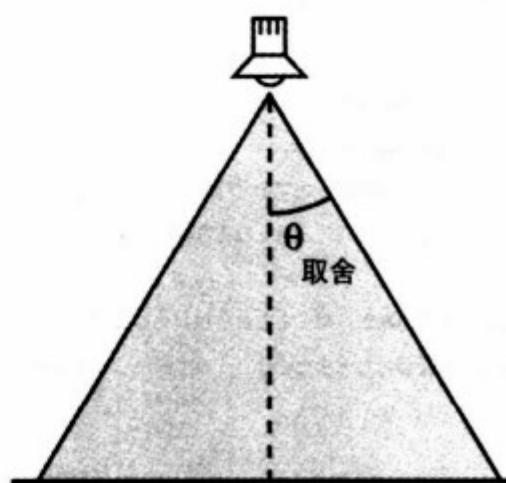


图 5-18 指定一个聚光灯的取舍角

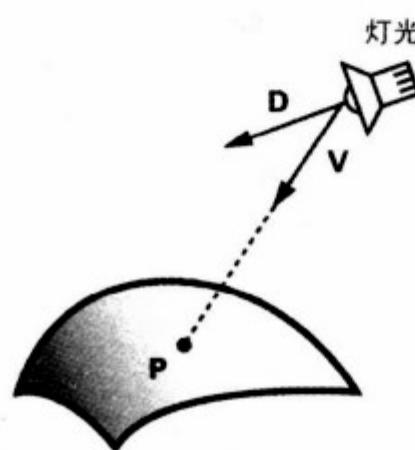


图 5-19 用来计算聚光灯效果的向量

通过计算这两个规范化向量的点积，你能够得到它们之间夹角的余弦，然后用它来找出 P 是否在聚光灯的圆锥体内。 P 只有当 $\text{dot}(V, D)$ 大于聚光灯取舍角的余弦值时才受聚光灯的影响。

基于这个数学原理，我们能够为聚光灯计算创建一个函数，如在示例 5-8 中显示的那样。如果 P 在聚光灯的圆锥体内，`C5E8_spotlight` 函数返回 1，否则返回 0。注意我们已经在来自示例 5-6 的 `Light` 结构中增加了 `direction`（聚光灯方向——假设已经被规范化了）和 `cosLightAngle`（聚光灯的取舍角的余弦值）。

示例 5-8 C5E7_spotlight 内部函数

```
float C5E8_spotlight(float3 P,
                      Light light)
{
    float3 V = normalize(P - light.position);
    float cosCone = light.cosLightAngle;
```

```

float cosDirection = dot(v, light.direction);
if (cosCone <= cosDirection)
    return 1;
else
    return 0;
}

```

强度变化

迄今为止，我们假设由聚光灯发出的光的强度在聚光灯的圆锥体内是均匀不变的。实际上很少有真实的聚光灯是这样均匀聚焦的。为了使事情更加有趣，我们将把圆锥体分成两部分：一个内部圆锥和一个外部圆锥。内部圆锥（或“热区”）发出固定强度的光，而在内部圆锥以外强度平滑地逐渐减少，如图 5-20 所示。这个常用的方法创造出了更加复杂的效果，在图 5-21 的右边演示了这种效果。

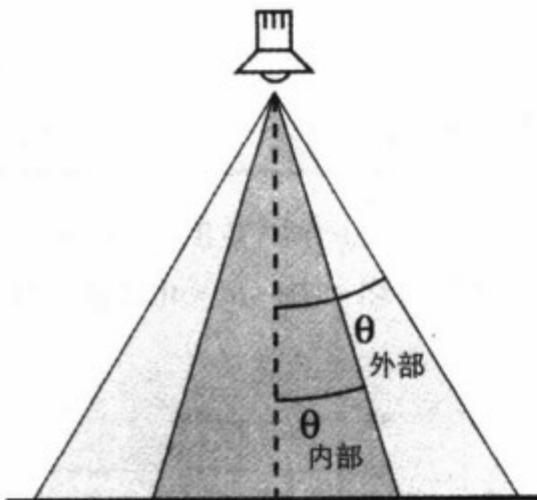


图 5-20 为一个聚光灯指定内部和外部圆锥

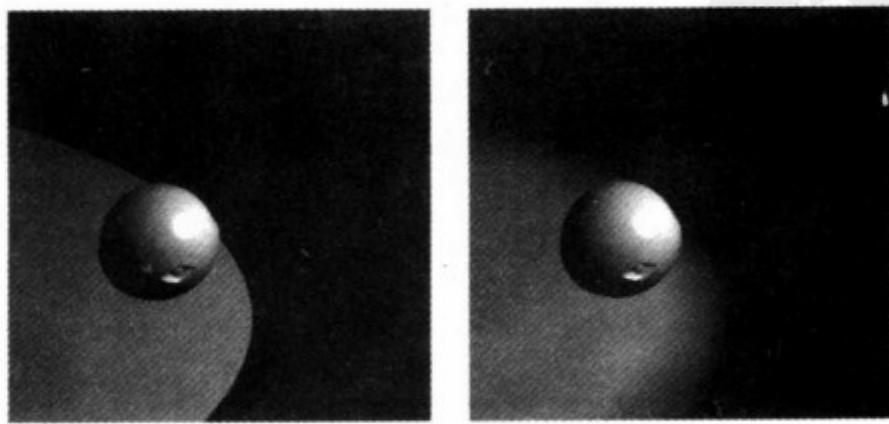


图 5-21 增加内部和外部聚光灯圆锥体的效果

找出一个特定的点 P 是在内部圆锥中还是在外部圆锥中并不是很难。与基本聚光灯的惟一区别是，你需要根据 P 所在的圆锥体来变化光强度的计算。

如果 P 在内部圆锥中，它将接受聚光灯的全部光强度。如果 P 在内部圆锥和外部圆锥之间，你需要根据 P 离内部圆锥有多远来逐渐降低光强度。 Cg 的 `lerp` 函数非常适合这种类型的过渡，但是使用高级的 `profile` 你能够做得更好。

Cg 有一个 `smoothstep` 函数可以用来生成视觉上比一个简单的 `lerp` 更加吸引人的结果。不幸地是，`smoothstep` 函数也许在一些基本的 `profile` 上无法工作，因为它们的能力有限。我们将在这个例子中使用 `smoothstep`，虽然你能够用你选择的另一个函数来代替它。`smoothstep` 函数使用一个平滑的多项式在两个值之间插值：

<code>smoothstep (min, max, x)</code>	如果 $x < \min$ 返回 0; 如果 $x \geq \max$ 返回 1; 否则返回一个光滑的 Hermite 插值在 0 和 1 之间，由以下公式给定： $-2 * ((x - \min) / (\max - \min))^3 + 3 * ((x - \min) / (\max - \min))^2$
---	--

图 5-22 给出了 `smoothstep` 函数的曲线图。当你想要在两个值之间创建一个视觉上令人满意的过渡，可以使用 `smoothstep`。`smoothstep` 的另外一个方便的特点是

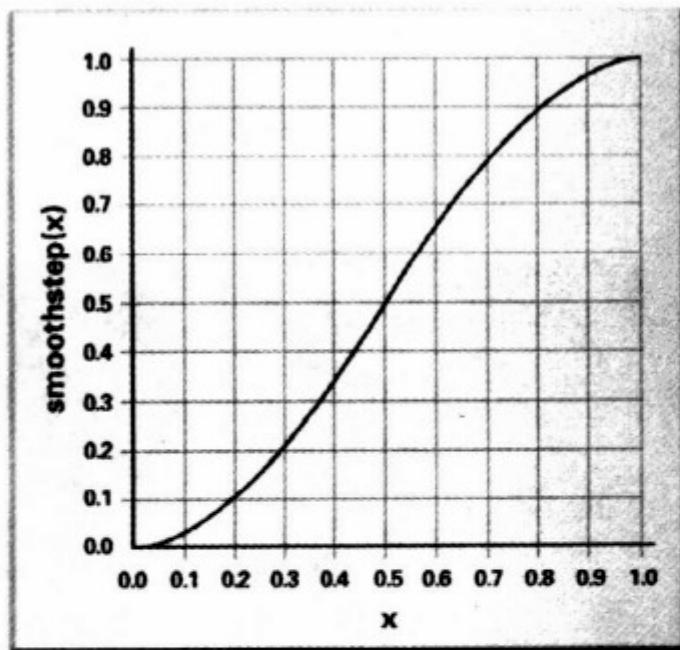


图 5-22 `smoothstep` 函数的曲线图

它把值映射到[0, 1]范围内。如果你为 smoothstep 正确地设置参数，当 P 在内部圆锥的时候返回 1.0，当 P 在外部圆锥以外的时候返回 0.0。

我们可以再次扩展 Light 结构来包括新的聚光灯参数。单独的取舍角现在被内角余弦和外角余弦所代替了。

下面是 Light 结构进一步更新的版本的内容：

```
struct Light{
    float4 position;
    float3 color;
    float kC;
    float kL;
    float kQ;
    float direction;
    float cosInnerCone; // New member
    float cosOuterCone; // New member
};
```

显示在示例 5-9 中的 C5E9_dualConeSpotlight 内部函数结合了所有这些创建了一个带热区的聚光灯。

示例 5-9 C5E9_dualConeSpotlight 内部函数

```
float C5E9_dualConeSpotlight(float3 P,
                               Light light)
{
    float3 V = normalize(P - light.position);
    float cosOuterCone = light.cosOuterCone;
    float cosInnerCone = light.cosInnerCone;
    float cosDirection = dot(V, light.direction);
    return smoothstep(cosOuterCone,
                      cosInnerCone,
                      cosDirection);
}
```

如示例 5-10 中所显示的，C5E10_spotAttenLighting 内部函数在漫反射和镜面反射中结合了衰减和聚光灯项。

示例 5-10 C5E10_spotAttenLighting 内部函数

```

void C5E10_spotAttenLighting(Light light,
                           float3 P,
                           float3 N,
                           float3 eyePosition,
                           float shininess,

                           out float diffuseResult,
                           out float specularResult)
{
    // Compute attenuation
    float attenuationFactor = C5E6_attenuation(P, light);

    // Compute spotlight effect
    float spotEffect = C5E9_dualConeSpotlight(P, light);

    // Compute the diffuse lighting
    float3 L = normalize(light.position - P);
    float diffuseLight = max(dot(N, L), 0);
    diffuseResult = attenuationFactor * spotEffect *
                    light.color * diffuseLight;

    // Compute the specular lighting
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), 0),
                                shininess);
    if (diffuseLight <= 0) specularLight = 0;
    specularResult = attenuationFactor * spotEffect *
                    light.color * specularLight;
}

```

5.5.3 平行光

虽然像聚光灯效果和衰减等计算给你的场景增加了视觉复杂度，但这些结果通常并不显著。考虑一下来自太阳照射地球上物体的光线。所有这些光线看上去像是来自同一个方向，因为太阳是离得如此的远。在这样一种情况下，计算聚光灯效果或增加衰减是没有意义的，因为所有的物体基本上接受同样多的光。拥有该性质的一种光被称为平行光 (directional light)。平行光在实际中是不存在的，但

是在计算机图形学中，确定哪些平行光已经足够了的情况是非常值得的。这将允许你避免那些不会产生可察觉的结果的计算。

5.6 练习

1. 回答这个问题：区别用一个单个顶点光照模型渲染的表面和用单个片段光照模型渲染的表面的视觉差异是什么样的。特别地，讨论 `specular highlight` 和 `spotlight`。

2. 你自己尝试一下：修改 `C5E5_computeLighting` 函数，假设使用一个方向光源，如在 5.5.3 节中描述的那样。你通过消除一个向量的差别和标准化，应该能够提高函数的性能。

3. 你自己尝试一下：在本章中的光照示例都假设你的应用程序在物体空间中指定光的位置。使用眼空间来指定光的位置是非常方便的，因为它避免了应用程序从世界或眼空间把光的位置变换到每个物体的不同的物体空间。同样地，用来计算镜面反射作用的视向量只不过是眼空间顶点位置，因为眼睛位于原点。但是，在眼空间的光照对每个顶点都需要变换物体空间位置和法向量到眼空间。变换一个物体空间位置和法向量到眼空间需要对它们分别乘以 `modelview` 矩阵和 `modelview` 矩阵转置的逆矩阵。如果 `modelview` 矩阵伸缩了向量，你必须标准化眼空间的法向量，因为 `modelview` 矩阵会破坏法向量的标准化。修改 `C5E5_computeLighting` 函数，假设位置 `P` 和法向量 `N` 都在眼空间。同样地，在把这些向量传给新的 `C5E5_computeLighting` 函数之前，修改 `C5E4v_twoLights` 来变换物体空间的位置和法向量到眼空间。

4. 你自己尝试一下：修改你为练习 3 所实现的 `C5E5_computeLighting` 函数的眼空间版本，假设眼空间向量 `V` 的方向总是 $(0, 0, 1)$ 。这被称为无限远观察者镜面反射优化，它取消了计算 `H` 之前对 `V` 进行标准化的需要。这样对你的渲染结果改变多少呢？当实现物体空间光照的时候，这个优化仍然可行吗？

5. 回答这个问题：物体空间和眼空间哪一个更有效？同样，哪一个对应用程
序员更方便？

6. 你自己尝试一下：编写一对顶点和片段程序来混合单个顶点和单个片段光
照。在顶点程序中计算发射、环境和漫反射的作用。同样，在顶点程序中计算半
角和法向量，然后把这些向量传给你的片段程序。在片段程序中，只使用插值半

角和法向量计算镜面反射作用。这个划分意味着大部分的光照数学发生在顶点级，但镜面反射作用常受单个顶点人为痕迹的影响，所以为了更好的质量，镜面反射在单个片段级计算。把这种方法的质量和性能与纯粹的单个顶点和单个片段光照实现相比较。

7. 你自己尝试一下：起绒或有凹槽的表面，例如毛发、唱片、绸缎的圣诞装饰物和起毛的金属表面反光不同于常规的材质，因为这些表面的细微的凹凸结构产生了各向异性的光散射。有关各向异性的光照的研究在补充阅读部分讨论。创建一个顶点或片段程序来实现各向异性的光照。

5.7 补充阅读

Mason Woo、Jackie Neider、Tom Davis 和 Dave Shreiner 撰写的 *OpenGL Programming Guide: The Official Guide to Learning OpenGL* 第三版 (Addison-Wesley, 1999 年) 的有关光照的章节介绍了完整的 OpenGL 固定功能光照模型。虽然介绍是以 OpenGL 为中心的，相应的 Direct3D 固定功能光照模型几乎是完全一样的。想知道更多有关 Direct3D 的光照模型，可以参考 Microsoft 的在线 DirectX 文档 “Mathematics of Lighting”。

Eric Lengyel 撰写的 *Mathematics for 3D Game Programming and Computer Graphics* (Charles River Media, 2001 年) 一书中有一个题为 “Illumination” 的章节写得非常好，详细讨论了实现各种光照模型所应用的数学知识。

David Banks 在 1994 年发表的题为 “Illumination in Diverse Codimensions” 的 SIGGRAPH 论文 (ACM Press) 讨论了各向异性的光照。Wolfgang Heidrich 和 Hans-Peter Seidel 在 1998 年发表了一篇题为 “Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware” 的论文。这篇论文把各向异性的光照实时地应用到一个表面上，并提供了数学公式，你将在练习 7 中用到这些公式。

Robert Cook 和 Kenneth Torrance 1982 年在 *ACM Transactions on Graphics* 上在题为 “A Reflectance Model for Computer Graphics” 的论文中发表了一个详细的合乎物理的光照模型。

Andrew Glassner 撰写的两卷 *Principles of Digital Image Synthesis* (Morgan Kaufmann, 1995 年) 从计算机图形的角度详细地研究了光是如何与材质相互作用的。

第 6 章

动画

本章将描述使用 Cg 驱动物体的各种方法。它包括以下 5 个部分：

- “随时间运动”介绍了动画的基本概念。
- “一个有规律搏动的物体”显示了一个顶点程序，可以周期性地朝外沿着它的法向量的方向移动一个物体的表面。
- “粒子系统”描述了如何用一个基于物理的模拟来创建一个粒子系统，用到的所有计算都是由图形处理器完成的。
- “关键帧插值”解释了关键帧动画，一个顶点程序通过在物体的不同姿态之间插值来使物体运动。
- “顶点混合”解释了在人物动画中如何基于多权重的控制矩阵移动顶点来实现更多的动态控制。

6.1 随时间运动

动画是随时间发生的一个动作的结果——例如，一个物体不停的搏动，一个光的淡出，或一个跑动的人物。你的应用程序可以使用用 Cg 编写的顶点程序来实现这些类型的动画。动画的来源是在你的应用程序中的一个或多个随时间变化的程序参数。

为了创建动画的渲染，你的应用程序必须在 Cg 以上或甚至在 OpenGL 或 Direct3D 以上的层次记录时间。应用程序通常用一个全局变量表示时间，这个变量将随着应用程序的时间向前推进而有规律的增加。然后，应用程序用一个时间函数来更新其他变量。

你可以在中央处理器上计算动画更新，然后把这些动画数据传给图形处理器。

但是，更有效的方法是用一个顶点程序在图形处理器上执行尽可能多的动画计算，而不需要中央处理器来做所有的数字处理。把动画工作从中央处理器上卸载下来可以帮助平衡中央处理器和图形处理器的资源，可以释放出中央处理器来进行更多的有关计算，例如碰撞检测、人工智能和博弈。

6.2 一个有规律搏动的物体

在该第一个示例中，你将学习如何使得一个物体周期性的变形，以至于它看起来有些膨胀。我们的目的是用时间参数作为输入，然后基于时间修改几何物体的顶点的位置。更明确地说就是，你需要沿着物体表面法向量的方向移动物体表面的位置，如图 6-1 所示。

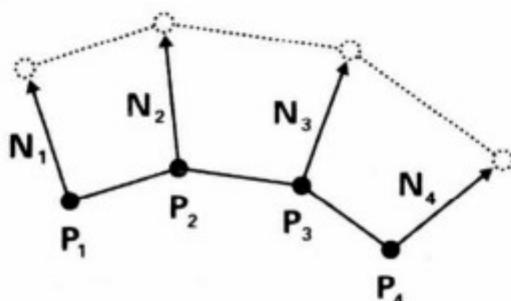


图 6-1 使一个物体膨胀

通过随时间改变位移的大小，你可以创建一个膨胀或搏动的效果。当它被应用到一个人物的时候，图 6-2 显示了这种效果的渲染。这种搏动动画发生在一个顶点程序里。

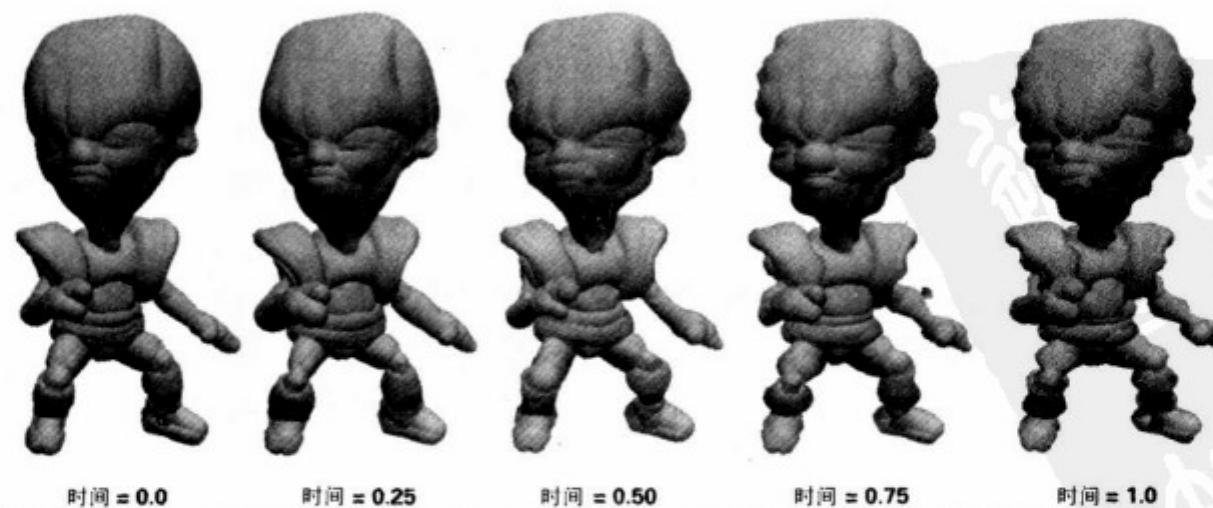


图 6-2 一个搏动的异形

6.2.1 顶点程序

示例 6-1 显示了 C6E1v_bulge 顶点程序的全部源代码，它计划与第 2 章的 C2E2f_passthrough 片段程序一起使用。只有顶点位置和法向量是膨胀效果真正需要的。但是，光照将使这个效果看起来更有效，因此我们同样包含了材质和光照信息。一个叫 computeLighting 的帮助函数只是计算漫射和镜面反射光照（为了简单镜面反射材质被假定为白色）。

示例 6-1 C6E1v_bulge 顶点程序

```

float3 computeLighting(float3 lightPosition,
                      float3 lightColor,
                      float3 Kd,
                      float shininess,
                      float3 P,
                      float3 N,
                      float3 eyePosition)
{
    // Compute the diffuse lighting
    float3 L = normalize(lightPosition - P);
    float diffuseLight = max(dot(N, L), 0);
    float3 diffuseResult = Kd * lightColor * diffuseLight;

    // Compute the specular lighting
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float3 specularLight = lightColor * pow(max(dot(N, H), 0),
                                              shininess);
    if (diffuseLight <= 0) specularLight = 0;
    float3 specularResult = lightColor * specularLight;
    return diffuseResult + specularResult;
}

void C6E1v_bulge(float4 position : POSITION,
                   float3 normal : NORMAL,
                   out float4 oPosition : POSITION,
                   out float4 color : COLOR,

```

```

uniform float4x4 modelViewProj,
uniform float time,
uniform float frequency,
uniform float scaleFactor,
uniform float3 Kd,
uniform float shininess,
uniform float3 eyePosition,
uniform float3 lightPosition,
uniform float3 lightColor)

{
    float displacement = scaleFactor * 0.5 *
        sin(position.y * frequency * time) + 1;
    float4 displacementDirection = float4(normal.x, normal.y,
                                           normal.z, 0);
    float4 newPosition = position +
        displacement * displacementDirection;
    oPosition = mul(modelViewProj, newPosition);
    color.xyz = computeLighting(lightPosition, lightColor,
                                Kd, shininess,
                                newPosition.xyz, normal,
                                eyePosition);
    color.w = 1;
}

```

6.2.2 位移计算

一、创建一个基于时间的函数

这个想法是计算一个被称为 `displacement` 的变量，它沿着表面法向量的方向向上或向下移动顶点的位置。要使程序的效果运动起来，`displacement` 不得不随着时间改变。你可以为它选择任何你喜欢的函数。例如，你可以选择如下的函数：

```
float displacement = time;
```

当然，这个行为没有什么意义，因为 `displacement` 总是在增加，使得物体随着时间不断地变得越来越大。相反，我们想要一种搏动的效果，使物体在膨胀变大和返回它原来的正常形状之间振荡。正弦函数提供了这样一种平滑的振荡行为。

正弦函数的一个有用的性质是它的结果总是在 -1 和 1 之间。在某些情况下，例如在这个例子里，你不想要负数，因此你可以缩放和偏移结果到一个更方便的范围，例如 0 到 1：

```
float displacement = 0.5 * (sin(time)+1);
```



你是否知道正弦函数(**sin**)在CineFX构架中与加法和乘法一样有效？实际上，**cos**函数计算余弦的函数一样快。利用这些性质可以给你的程序增加视觉复杂度而不减慢它们的执行。

二、为程序增加控制

为了更好地控制你的程序，你可以增加一个**uniform**参数来控制正弦波的频率。把这个**uniform**参数**frequency**放入位移公式得到：

```
float displacement = 0.5 * (sin(frequency*time)+1);
```

你也许还想控制膨胀的幅度，因此为它增加一个**uniform**参数是非常有用的。把这个要素加入公式，我们得到：

```
float displacement = scaleFactor * 0.5 *  
    (sin(frequency * time) + 1 );
```

到目前为止，这个公式在整个模型上产生的突起的量是完全一样的。你也许想用它来显示一个人物在一段很长的追逐之后不停地喘气。要实现这点，你需要把这个程序应用到人物的胸部。另一种可选择的方法是，你能够提供附加的统一参数来指明人物的呼吸是怎样迅速的，而且经过一段时间以后，呼吸会恢复到正常的样子。这些动画效果在一个游戏中实现起来是很容易的，并且能够帮助玩家沉浸到这个游戏的世界里。

三、改变膨胀的大小

但是如果你想要膨胀的大小在模型上的不同的部位发生变化应该怎么办呢？要实现这点，你不得不增加一个依赖于每个顶点的变化参数。一个办法是传递一个**scaleFactor**作为变化参数，而不是作为一个统一参数。在这里我们向你显示一个更加简单的基于顶点位置的方法来给膨胀增加一些变化：

```
float displacement = scaleFactor * 0.5 *  
    sin (position.y * frequency * time) + 1;
```

这行代码使用了位置的y坐标来对膨胀进行变化，但是，如果你喜欢你可以使用几个坐标的组合。这将取决于你想要的效果类型。

四、更新顶点位置

在我们的例子中，我们先用位移量缩放物体空间的表面法向量。然后，通过

把结果加到物体空间的顶点位置，你可以获得一个被移位的物体空间的顶点位置：

```
float4 displacementDirection = float4 (normal.x, normal.y,
                                         normal.z, 0);
float4 newPosition = position +
                         displacement + displacementDirection;
```

五、当可能的时候预先计算统一参数

前面的例子证明了很重要的一点。再仔细看看下面这一行来自示例 6-1 的代码：

```
float displacement = scaleFactor * 0.5 *
                         sin(position.y * frequency * time) + 1;
```

如果你想使用这个公式来计算位移，所有的项对每个顶点都将是一样的，因为它们都依赖于统一参数。这意味着你将在图形处理器上为每个顶点计算这个位移，但实际上你能够在中央处理器上为整个网格只计算位移一次，然后把位移当作一个统一参数传递给顶点程序。但是，当顶点的位置是位移公式的一部分的时候，`sine` 函数必须为每个顶点求值。并且如你所预期的那样，如果位移的值像这样在每个顶点都发生变化，类似这样的一个每个顶点的计算在图形处理器上执行要比在中央处理器上执行效率要高很多。



如果一个被计算的值对整个物体是个常数，你可以通过用中央处理器在每个物体的基础上预先计算这个值来优化你的程序。然后把这个预先计算的值当作一个统一参数传递给 Cg 程序。这个方法比为被处理的每个片段或顶点预先计算这个值要有效许多。

6.3 粒子系统

有时候，你需要把每个顶点当作一个小物体或粒子（particle），而不是动画一个网格中的顶点。根据某个特定规律运动的大量的粒子被称为一个粒子系统（particle system）。下面这个示例在一个顶点程序中实现了一个简单的粒子系统。现在，我们只关注这个系统是如何工作的，而不必担心它简单的外表。在本节的最后，我们将提到一个简单的方法来增强你的粒子系统的外表。图 6-3 显示了一个随时间变化的粒子系统的例子。

这个示例的粒子系统根据一个来自物理学的简单的向量运动公式运转。这个公式给出了任意时间的每个粒子的x、y和z位置。你将使用的这个简单公式被显示在公式 6-1 中。

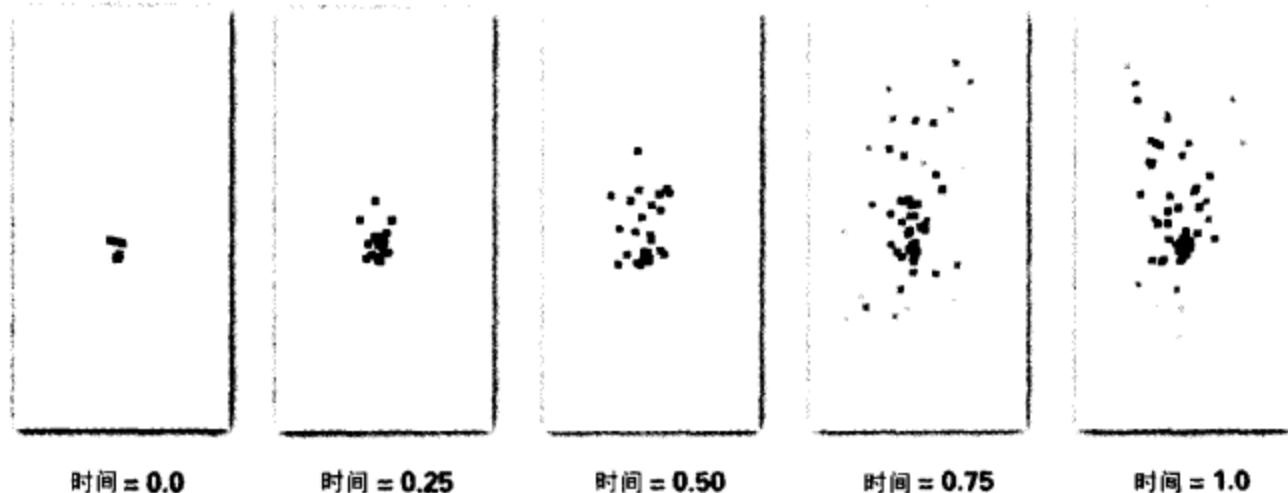


图 6-3 一个粒子系统

公式 6-1：粒子轨道

$$P_{\text{final}} = P_{\text{initial}} + vt + 1/2at^2$$

其中：

- P_{final} 是粒子的最后位置。
- P_{initial} 是粒子的初始位置。
- v 是粒子的初始速度。
- a 是粒子的加速度。
- t 是时间。

这个公式建模了一个粒子集合在初始速度和重力影响下的轨道，而没有与其他粒子的相互作用。假如你提供了粒子的初始位置、初始速度和不变的加速度例如重力加速度，这个公式能够给出任何给定时间下一个粒子的位置。

6.3.1 初始条件

应用程序必须用变化参数的形式提供每个粒子的初始位置和初始速度。这两个参数值被称为初始条件 (initial condition)，因为它们描述了粒子在模拟开始时的情况。

在这个简单的模拟中，由于重力而产生的加速度对每一个粒子都是相同的。

因此，重力加速度是一个统一参数。

为了使这个模拟更加精确，你能够增加一些效果例如拖拽或者甚至是旋转——我们把这个当成一个练习留给你。

6.3.2 向量化计算

现代的图形处理器拥有强大的向量处理能力，特别是对加法和乘法，它们特别适合处理最多到四元的向量。因此，使用向量计算与使用标量计算通常是一样有效的。

公式 6-1 是一个向量公式，因为初始位置、初始速度、常数加速度，以及被计算的位置都是三元向量。在写 Cg 顶点程序的时候通过用向量表示来实现这个粒子系统的公式的时候，你能够帮助编译器把你的程序翻译成能够在你的图形处理器上有效执行的形式。



尽可能使你的计算向量化，来充分利用图形处理器强大的向量处理能力。

6.3.3 粒子系统的参数

表 6-1 列出了在下一节中将介绍的顶点程序所使用的变量。

表 6-1 在粒子公式中的变量

变 量	类 型	描 述	来 源 (类型)
pInitial	float3	初始位置	应用程序 (变化)
vInitial	float3	初始速度	应用程序 (变化)
tInitial	float3	粒子创建的时间	应用程序 (变化)
acceleration	float3	加速度 (0.0, -9.8, 0.0)	应用程序 (统一)
globalTime	float	全局时间	应用程序 (统一)
pFinal	float3	当前位置	内部
t	float	相对时间	内部

除了在顶点程序内部计算的相对时间 (t) 和最后的位置 (pFinal) 以外，每个变量都是顶点程序的一个参数。注意加速度的 y 分量是负的——因为重力的作用是向下的，也就是 y 轴负方向。常量 9.8 平方米/秒是地球的重力加速度。初始位

置、初始速度和统一的加速度都是物体空间向量。

6.3.4 顶点程序

示例 6-2 显示了 C6E2v_particle 顶点程序的源代码。这个程序打算与 C2E2f_passthrough 片段程序一起使用。

示例 6-2 C6E2v_particle 顶点程序

```
void C6E2v_particle(float4 pInitial : POSITION,
                     float4 vInitial : TEXCOORD0,
                     float tInitial : TEXCOORD1,

                     out float4 oPosition : POSITION,
                     out float4 color : TEXCOORD0,
                     out float pointSize : PSIZE,

                     uniform float globalTime,
                     uniform float4 acceleration,
                     uniform float4x4 modelViewProj)
{
    float t = globalTime - tInitial;
    float4 pFinal = pInitial +
                    vInitial * t +
                    0.5 * acceleration * t * t;

    oPosition = mul(modelViewProj, pFinal);

    color = float4(t, t, t, 1);

    pointSize = -8.0 * t * t +
                8.0 * t +
                0.1 * pFinal.y + 1;
}
```

一、计算粒子的位置

在该程序中，应用程序记录了一个全局时间并把它用作统一参数 globalTime 传递给顶点程序。这个全局时间从 0 开始，当应用程序初始化以后，不断地增加。当每个粒子被创建的时候，该粒子的创建时间使用变化参数 tInitial 被传递给顶点程序。要想知道一个粒子已经活动了多久，你只需要用 tInitial 减去 globalTime：

```
float t = globalTime - iInitial;
```

现在，你可以把 **t** 放到公式 6-1 中来计算粒子的当前位置：

```
float4 pFinal = pInitial +
    vInitial * t +
    0.5 * acceleration * t * t;
```

这个位置在物体空间中，因此它需要被变换到剪裁空间，像以前一样：

```
oPosition = mul (modelViewProj, pFinal);
```

二、计算粒子的颜色

在这个例子中，时间控制了粒子的颜色：

```
color = float4 (t, t, t, 1);
```

这是一个简单的方法，但是它能够产生一个有趣的视觉变化。颜色将随着时间线性地增加。注意颜色增加到纯白色 (1, 1, 1, 1) 后就饱和了。你可以尝试你自己的选择，例如根据粒子的位置改变颜色，或者根据位置和时间的一个组合来改变颜色。

三、计算粒子的大小

C6E2v_particle 使用了一个新的被命名为 PSIZE 的顶点程序输出语义。当你渲染一个点到屏幕上的时候，一个带有该语义的输出指定了这个点以像素为单位的宽度（和高度）。这使你的顶点程序能够程序化控制被光栅器使用的点的大小。

每个粒子的点的大小随着时间的流逝而变化。粒子开始的时候很小，然后大小逐渐增加，最后再逐渐缩小。这种变化增加了像焰火 (firework-like) 那样的效果。作为一个额外的修饰，我们增加了一点对粒子高度的依赖，因此，当粒子向上运动的时候会变大一点。为了实现所有这些，我们使用了如下的函数来计算点的大小：

```
pointSize = -8.0 * t * t +
    8.0 * t +
    0.1 * pFinal.y + 1;
```

图 6-4 显示了这个函数看起来是什么样子的。

这个函数没有什么特殊的——我们只不过是创建了一个公式来实现我们想要的效果。换句话说，除了试图模拟我们想象的效果，这个公式没有任何实际的物理意义。

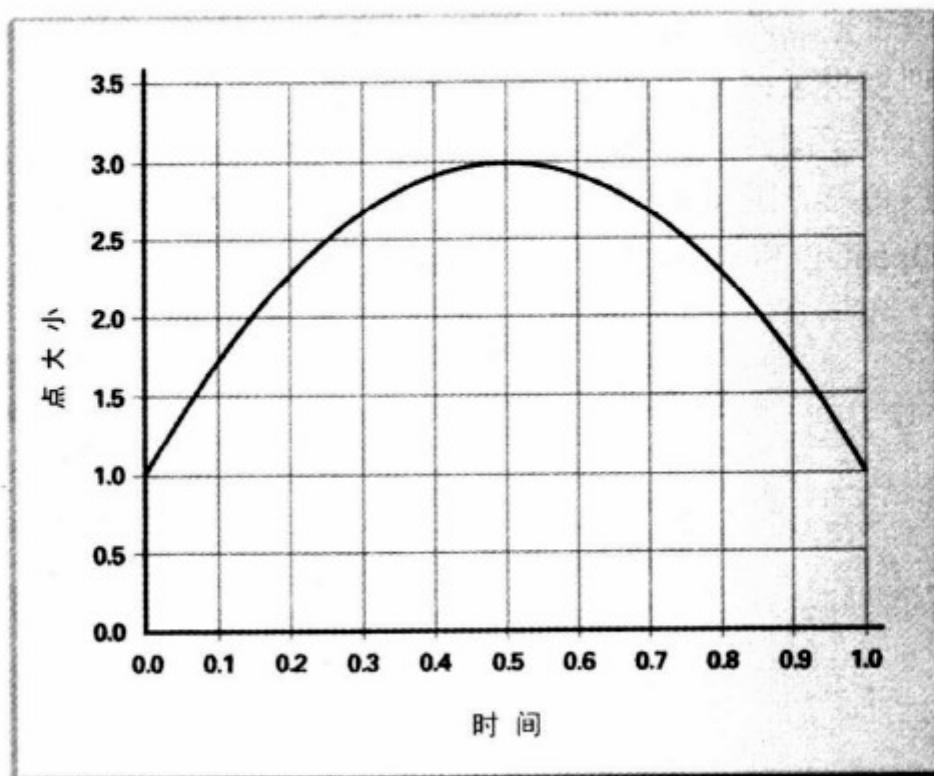


图 6-4 一个点的大小的函数

6.3.5 修饰你的粒子系统

虽然 C6E2v_particle 程序生成了有趣的粒子运动，粒子本身看上去并不吸引人，它们只不过是不同大小的纯色的正方形。

但是，你能够通过使用粒子图（point sprites）来改善粒子的外表。通过使用粒子图，硬件能够取得每个被渲染的点，并画一个如图 6-5 所示的由 4 个顶点组成的正方形，而不是只画一个单独的顶点。粒子图自动地给每个角的顶点赋予一个纹理坐标。这使得你能够从一个正方形到任何你想要的纹理图像来改变粒子的外表。

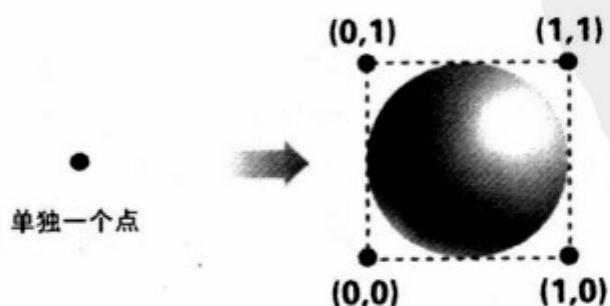


图 6-5 把点转换为粒子图

通过把点当作粒子图来渲染，你能够使用被赋予的纹理坐标来采样一个纹理，该纹理提供了每个顶点的形状和外表，而不是简单地把每个顶点渲染成一个方形的点。粒子图能够提供一种增加了几何复杂性的印象，而实际上没有多画一个额外的三角形。图 6-6 显示了一个视觉上更加有趣的使用了粒子图的粒子系统的例子。OpenGL 和 Direct3D 都有渲染粒子图的标准接口。

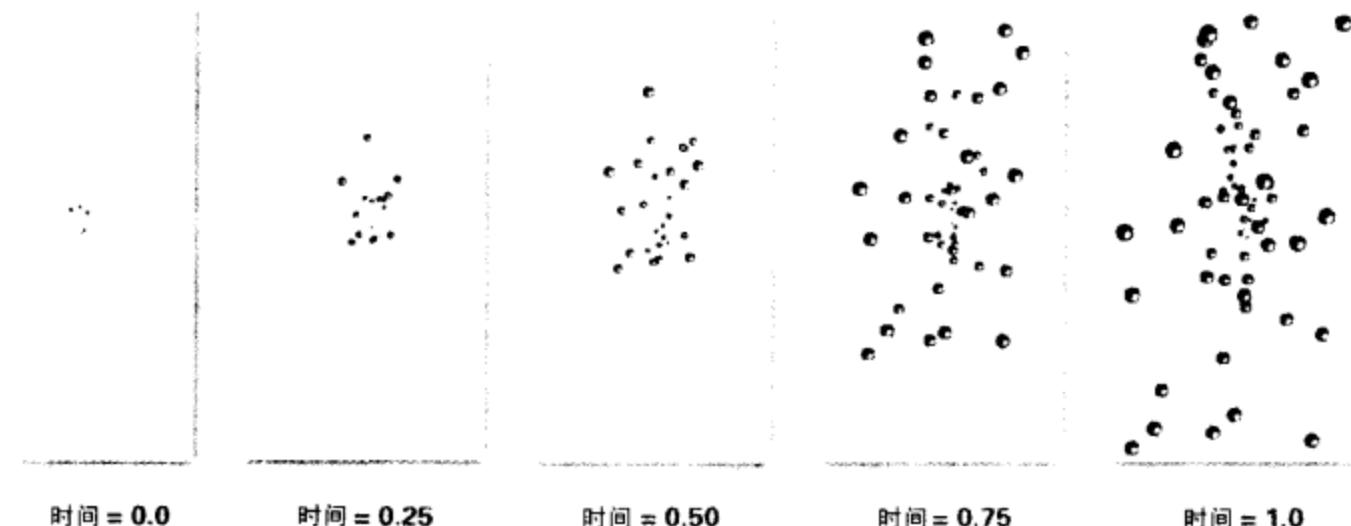


图 6-6 一个使用粒子图的粒子系统

6.4 关键帧插值

3D 游戏通常使用一系列的关键帧来表示一个动画的人物或动物的不同姿态。例如，一个动物也许拥有站着、奔跑、跪下、躲避、攻击和死亡的动画序列。美工人员将他们为一个给定的 3D 模型创建的每个特定的姿态称为一个关键帧。

6.4.1 关键帧的背景知识

关键帧这个术语来自卡通动画。为了生成一个卡通片，一名美工人员首先迅速地描绘一个粗略的帧序列来动画一个人物。而不是画最后动画所需要的每一帧，美工人员只是画最重要的或“关键”的帧。随后，美工人员回过头来再添补缺少的帧。然后，一些在中间的帧就比较容易画了，因为在它之前和之后的关键帧能够充当前后的参考。

计算机动画工作者使用了一种类似的技术。一名 3D 美工人员为一个动画人物的每种姿势创建一个关键帧。即使是一个站着的人物可能也需要一系列的关键帧

来显示人物的重心从一只脚移到另一只脚。一个模型的所有关键帧必须使用同样数目的顶点，并且每个关键帧必须共享同样的顶点连接性。在一个给定的关键帧中使用的顶点，在模型的所有其他关键帧中对应于模型上的同一个点。整个的动画序列都维护了这种对应性。但是，由于模型的运动，一个特定顶点的位置可以从一帧到另一帧发生变化。

给定了这样一个关键帧模型以后，一种游戏就可以通过选取两个关键帧，然后把每一组对应的顶点位置混合在一起动画这个模型。混合是一个加权平均的过程，所有权重加起来之和为 100%。图 6-7 显示了一个异形模型的一些关键帧。这幅图包括了两个关键帧，分别被标示为 A 和 B，这两个关键帧被一个 Cg 程序混合为了一个中间的姿态。

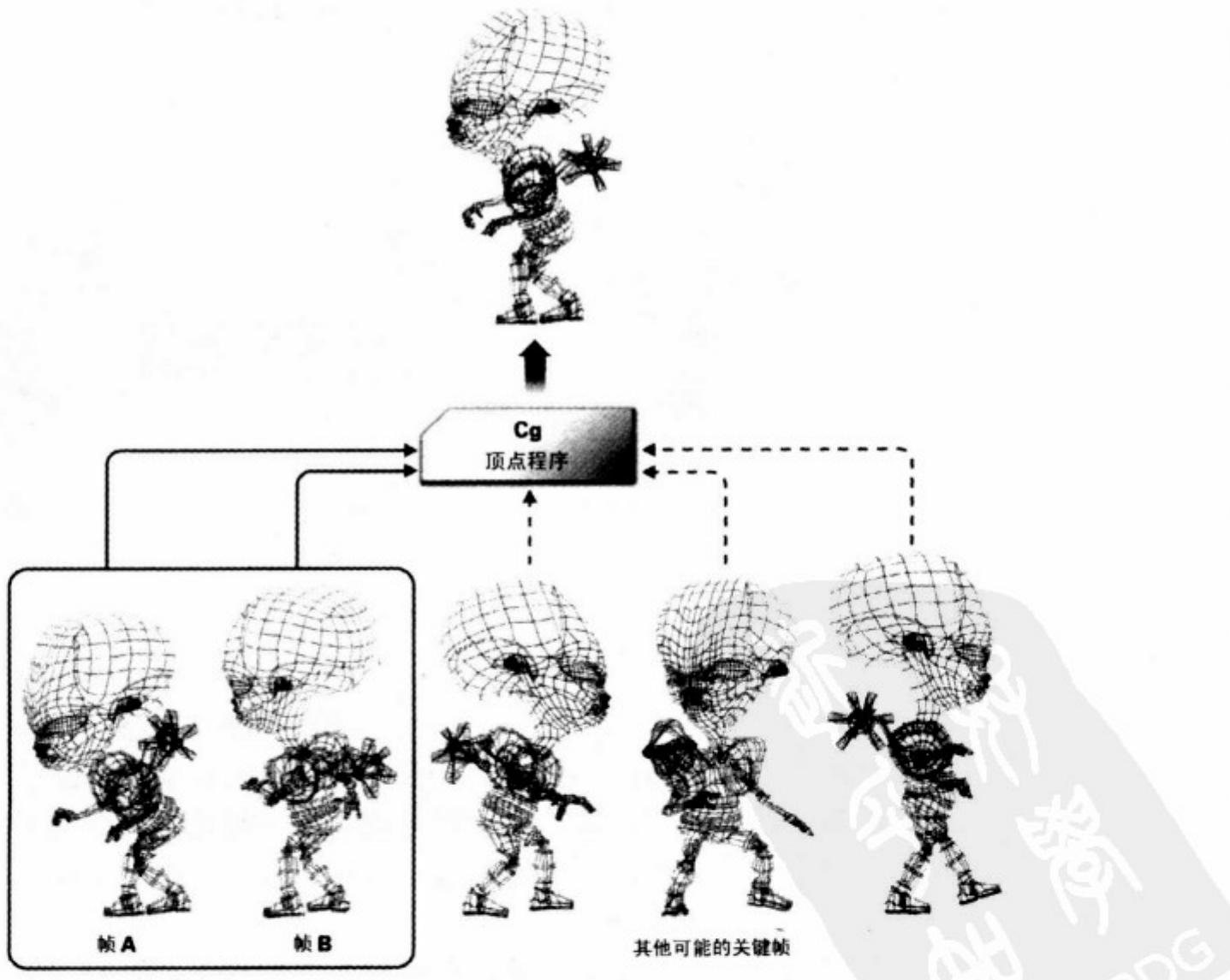


图 6-7 一个异性模型的关键帧

一个应用程序能够使用一个 Cg 顶点程序来把两个顶点混合在一起。为了更加真实，这个混合可能包括进一步的操作来正确地照亮被混合的人物。通常，一个应用程序为每个顶点指定了一个单独的位置，但是为了关键帧混合，每个顶点需要有两个位置，这两个位置将被用一个统一的权重因子混合在一起。

关键帧插值假设对一个给定的模型顶点的数目和顺序在所有的关键帧中都是相同的。这个假设确保了顶点程序总是能够混合正确的一对顶点。下面的 Cg 代码片段被用来混合关键帧的位置：

```
blendedPosition = ( 1 - weight ) * keyFrameA + weight * keyFrameB;
```

keyFrameA 和 keyFrameB 变量分别包含了关键帧 A 和 B 将被处理的顶点的(x, y, z) 位置。请注意 weight 和 (1-weight) 之和为 1。如果 weight 是 0.53，Cg 程序把关键帧 A 的位置的 47% 加到 53% 的关键帧 B 的位置上。图 6-8 显示了该类型动画的一个例子。

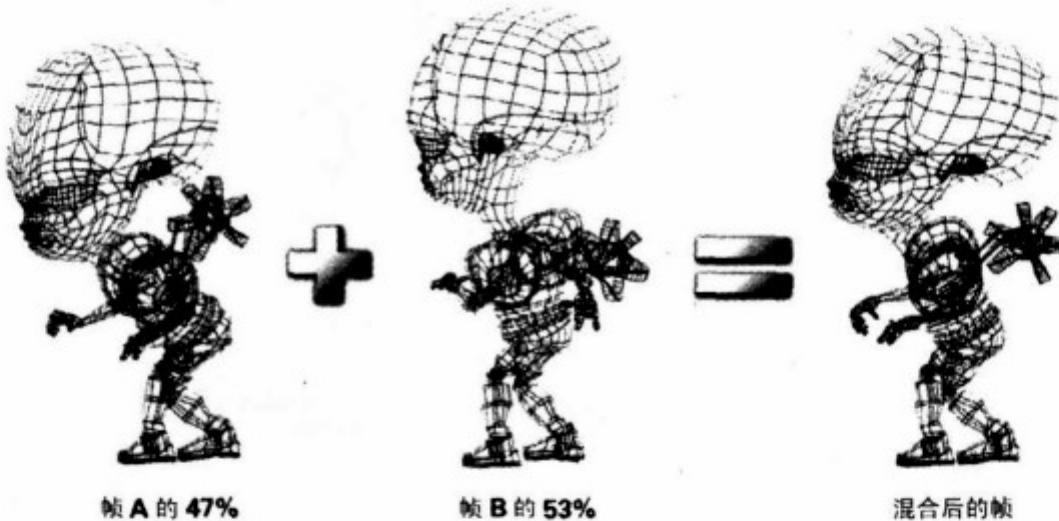


图 6-8 关键帧混合的一个例子

为了维持动画的连续性，关键帧的权重在每个被渲染的帧都逐渐增加直到它达到 1.0 (100%)。在这个点上，已经存在的关键帧 B 变成了新的关键帧 A，而权重被重新设为 0，然后游戏选择了一个新的关键帧 B 来继续进行动画。动画序列，例如行走，可能会在定义了人物行走动作的一组关键帧之间不断循环。游戏可以只通过选取定义奔跑动作的关键帧序列，而从一个行走动画切换到一个奔跑动画。

使用关键帧来生成令人满意的栩栩如生的动画是游戏引擎的任务。许多现有的 3D 游戏使用这种类型的关键帧动画。当一个应用程序使用 Cg 顶点程序来

执行关键帧的混合操作时，中央处理器可以把时间花在提高游戏性而不是不断地混合关键帧。通过使用一个 Cg 顶点程序，图形处理器承担了关键帧混合的任务。

6.4.2 插值方法

插值的类型有许多种。关键帧插值的两种最普通的形式是线性插值（linear interpolation）和二次插值（quadratic interpolation）。

一、线性插值

使用线性插值，两个位置的转换是以一种固定速率发生的。公式 6-2 显示了线性插值的定义：

公式 6-2 线性插值

$$\text{blendedPosition} = \text{positionA} \times (1-f) + \text{positionB} \times f$$

在这个公式中当 f 在 0 到 1 之间变化的时候，中间的位置则在 positionA 和 positionB 之间变化。当 f 等于 0 的时候，中间的位置正好是起始位置 positionA。当 f 等于 1 的时候，中间的位置恰好是结束位置 positionB。你能够再一次使用 Cg 的 lerp 函数来完成插值。

使用 lerp，在两个位置之间的插值可以简洁地写成如下形式：

```
intermediatePosition = lerp(positionA, positionB, f);
```

二、二次插值

线性插值在很多情况下都工作的很好，但是有些时候你需要变化率随着时间改变。例如，你也许想要从 positionA 转换到 positionB，开始的时候很慢，而随着时间的流逝逐渐加快。要实现这种效果，你必须使用二次插值，如下面的代码片段所示：

```
intermediatePosition = position * (1 - f * f) +
                      position * f * f;
```

其他你可以使用的函数是阶梯函数（step function）、样条函数（spline function）和指数函数。图 6-9 显示了几种常用的插值类型。

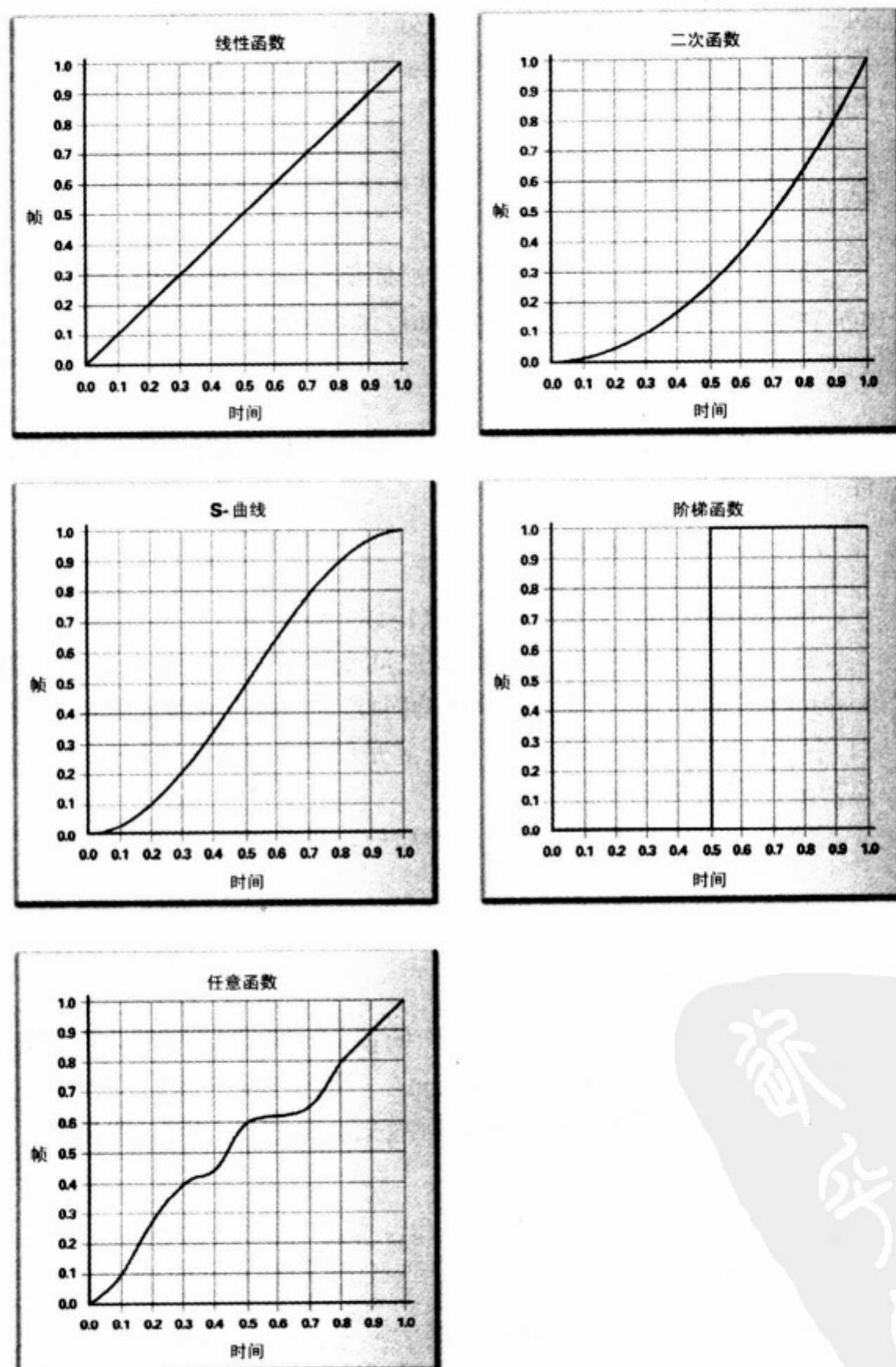


图 6-9 各种插值函数

6.4.3 基本的关键帧插值

示例 6-3 显示了 C6E3v_keyFrame 顶点程序。这个程序执行了物体空间的两个位置的混合，每个位置分别来自不同的关键帧。lerp 标准库函数线性插值了两个位置，然后程序把混合后的位置变换到剪裁空间。这个顶点程序还传递一个纹理坐标集和一种颜色。

由 positionA 和 positionB 的输入语义指明，应用程序负责把关键帧 A 的位置设置成传统的位置（POSITION），而关键帧 B 的位置则被设置成纹理坐标集 1（TEXCOORD1）。

示例 6-3 C6E3v_keyFrame 顶点程序

```
void C6E3v_keyFrame(float3 positionA : POSITION,
                     float3 positionB : TEXCOORD1,
                     float4 color      : COLOR,
                     float2 texCoord   : TEXCOORD0,

                     out float4 oPosition : POSITION,
                     out float2 oTexCoord : TEXCOORD0,
                     out float4 oColor    : COLOR,

                     uniform float     keyFrameBlend,
                     uniform float4x4 modelViewProj)
{
    float3 position = lerp(positionA, positionB,
                           keyFrameBlend);
    oPosition = mul(modelViewProj, float4(position, 1));
    oTexCoord = texCoord;
    oColor = color;
}
```

应用程序还负责通过统一参数 keyFrameBlend 来决定关键帧混合因子。keyFrameBlend 的值应该在 0 到 1 之间变化。一旦达到了 1，应用程序需要从动画序列中选择其他的关键帧，旧的关键帧 B 的位置输入则被设置成关键帧 A 的位置输入，然后再用新的关键帧位置提供给关键帧 B 的位置输入。

6.4.4 带光照的关键帧插值

你通常需要对一个关键帧模型进行光照。这不仅仅涉及混合两个位置（在两个不同关键帧中的顶点），而且还需要混合两个对应的表面法向量。然后，你就能通过混合的法向量进行光照计算。混合两个法向量也许会改变结果法向量的长度，因此你必须在光照之前规范化被混合的法向量。

示例 6-4 C6E4v_litKeyFrame 顶点程序

```

struct Light {
    float3 eyePosition; // In object space
    float3 lightPosition; // In object space
    float4 lightColor;
    float specularExponent;
    float ambient;
};

float4 computeLighting(Light light,
                        float3 position, // In object space
                        float3 normal) // In object space
{
    float3 lightDirection = light.lightPosition - position;
    float3 lightDirNorm = normalize(lightDirection);
    float3 eyeDirection = light.eyePosition - position;
    float3 eyeDirNorm = normalize(eyeDirection);
    float3 halfAngle = normalize(lightDirNorm + eyeDirNorm);
    float diffuse = max(0, dot(lightDirNorm, normal));
    float specular = pow(max(0, dot(halfAngle, normal)),
                           light.specularExponent);
    return light.lightColor * (light.ambient +
                               diffuse + specular);
}

void C6E4v_litKeyFrame(float3 positionA : POSITION,
                      float3 normalA : NORMAL,
                      float3 positionB : TEXCOORD1,
                      float3 normalB : TEXCOORD2,
                      float2 texCoord : TEXCOORD0,
```

```

        out float4 oPosition : POSITION,
        out float2 oTexCoord : TEXCOORD0,
        out float4 color      : COLOR,

uniform float      keyFrameBlend,
uniform Light     light,
uniform float4x4   modelViewProj)

{
    float3 position = lerp(positionA, positionB,
                           keyFrameBlend);
    float3 blendNormal = lerp(normalA, normalB,
                               keyFrameBlend);
    float3 normal = normalize(blendNormal);
    oPosition = mul(modelViewProj, float4(position, 1));
    oTexCoord = texCoord;
    color = computeLighting(light, position, normal);
}

```

示例 6-4 显示了在 C6E3v_keyFrame 示例程序的基础上增加了每个顶点光照的 C6E4v_litKeyFrame 顶点程序。在这个更新的示例中，每个关键帧还提供了自己对应的每个顶点的表面法向量。

computeLighting 内部函数使用了一个传统光照模型计算了物体空间的光照。

6.5 顶点混合

另一个人物动画的方法是顶点混合（vertex skinning）。许多 3D 建模软件包创作的 3D 内容很适合进行顶点混合。这种技术也被称为矩阵调色板混合（matrix palette blending）。

6.5.1 顶点混合理论

顶点混合维护了一个单独缺省姿势和许多用来正确旋转和平移缺省姿势的多边形网格的各个分部的矩阵，而不是为一个人物的每个姿势保存一个关键帧。由于将要变得明显的原因，这些不同的矩阵变换通常被称为“骨头”。

一个或多个这样的矩阵控制了在这个缺省姿势的多边形网格中的每个顶点。

每个矩阵被赋予了一个权重因子（从 0 到 100%），指明了这个矩阵对每个顶点的影响有多少。通常只有一小部分的矩阵控制每个顶点，意味着只有这一小部分矩阵对一个给定的顶点有正的和重要的权重因子。我们把这一小部分矩阵称为每个顶点的骨头集（bone set）。我们假设在骨头集中所有矩阵的权重因子加起来的和为 100%。

当渲染这种类型的模型的时候，你首先用在顶点的骨头集中的每个矩阵变换每个顶点，然后根据矩阵对应的权重因子对每个矩阵变换的结果进行加权求和。这个新的位置是被混合的顶点位置。

当所有的矩阵是单位矩阵的时候（没有旋转，没有平移），这个网格总是保持缺省的姿势。3D 美工人员通常选取一个缺省的姿势，使得人物总是面向前站着而且手脚分开。

一、从矩阵构造姿势

通过控制这些矩阵，你能够创建新的姿势。例如，一个位于人物前臂靠近肘部的顶点可以使用 67% 的前臂的矩阵、21% 的肘部矩阵和 12% 的上臂的矩阵。为顶点混合创建模型的动画制作者必须正确的局部化这个矩阵，例如，控制左肩的矩阵应该对靠近脚踝的顶点没有影响。通常，影响任何一个给定顶点的矩阵的数目被限制在不超过 4 个。对 3D 美工人员而言，一旦所有的权重和矩阵都被赋予了模型的缺省姿势，创建一个新的姿势只是简单地正确地操纵这些矩阵，而不是试图放置每个单独的顶点。当一个模型为顶点混合创建以后，摆姿势和动画是非常简单的。

对一个字符的模型，最重要的矩阵表示了在人物身体中的刚性骨头的移动和旋转的方法。因此，顶点混合的矩阵又被称为骨头。顶点代表了皮肤上的点。顶点混合模拟了用矩阵表示的骨头是如何拖拉和重新放置用顶点表示的在人物皮肤上的点。

二、光照

为了正确的光照，你需要为每个位置计算相同的被变换的和加权平均的法向量，除了你需要用每个矩阵的转置的逆来变换法向量而不是用矩阵本身。加权平均的法向量也许不再是单位长度了，因此需要规范化。

假设骨头矩阵只是旋转和平移将简化为光照进行的法向量变换，因为没有缩放和投影的矩阵的转置的逆就是这个矩阵本身。

三、与关键帧比较存储需求

使用关键帧的方法，每个姿势都需要一组不同的顶点位置和法向量。如果大量的姿势被需要，这个方法将变得不实用。

但是，使用顶点混合，每个姿势所需要的只是被所有姿势共享的缺省姿势和给定姿势的矩阵值。通常，每个人物所需要的矩阵要比顶点少很多，因此用一组骨头矩阵来表示一个姿势要比用一个关键帧表示的姿势简洁。使用顶点混合，通过混合现有的不同姿势的矩阵或者直接控制矩阵，你还能够动态地创建新的姿势。例如，如果你知道什么矩阵控制了手臂，你能够通过控制这些矩阵来挥动手臂。

除了需要每个姿势的矩阵以外，模型的缺省姿势需要每个顶点都有一个缺省的位置，一个缺省的法向量，一些矩阵索引来确定哪个矩阵的子集是控制这个顶点的，以及同样数目的分别对应于每个矩阵的权重因子。

缺省姿势的这些数据对所有其他的姿势都是固定不变的。产生一个新的姿势只需要新的矩阵，而不需要对缺省姿势的数据做任何修改。如果图形处理器能够执行所有的顶点混合的计算，这意味着中央处理器只需要为每个新姿势更新骨头矩阵，但是不需要操作或存取缺省姿势的数据。

顶点混合对存储和回放运动捕捉序列是禁得起检验的。你能够用一组骨头矩阵来表示每个运动捕捉帧，这些矩阵可以应用到共享同一个缺省姿势和相关联的矩阵的不同模型。逆向运动求解器还能够程序化地生成骨头矩阵。一个逆向运动求解器试图以一个真实和自然的方式找到从一个给定姿势过渡到另一个给定姿势的一个逐渐增加的骨头矩阵序列。

6.5.2 在顶点程序中的顶点混合

在示例 6-5 中的顶点程序 C6E5v_skin4m 实现了顶点混合，这个程序假设最多有 4 个骨头矩阵影响每个顶点（一个共同的假设）。

一个由 24 个骨头矩阵组成的数组表示了每个姿势，每个矩阵是一个 3×4 矩阵。整个数组是给程序的统一参数。这个程序假设每个骨头矩阵只包含一个旋转和一个平移（没有缩放或投影）。

每个顶点的输入向量 matrixIndex 提供了一组 4 个存取 boneMatrix 数组的骨头矩阵索引。每个顶点的输入向量 weight 分别为每个骨头矩阵提供了 4 个权重因子。这个程序假设每个顶点的权重因子总和为 100%。

示例 6-5 C6E5v_skin4m 顶点程序

```

void C6E5v_skin4m(float3 position : POSITION,
                    float3 normal : NORMAL,
                    float2 texCoord : TEXCOORD0,
                    float4 weight : TEXCOORD1,
                    float4 matrixIndex : TEXCOORD2,

                    out float4 oPosition : POSITION,
                    out float2 oTexCoord : TEXCOORD0,
                    out float4 color : COLOR,

uniform Light light,
uniform float4 boneMatrix[72], // 24 matrices
uniform float4x4 modelViewProj)
{
    float3 netPosition = 0, netNormal = 0;

    for (int i = 0; i < 4; i++) {
        float index = matrixIndex[i];
        float3x4 model = float3x4(boneMatrix[index + 0],
                                boneMatrix[index + 1],
                                boneMatrix[index + 2]);
        float3 bonePosition = mul(model, float4(position, 1));
        // Assume no scaling in matrix, just rotate & translate
        float3x3 rotate = float3x3(model[0].xyz,
                                model[1].xyz,
                                model[2].xyz);
        float3 boneNormal = mul(rotate, normal);
        netPosition += weight[i] * bonePosition;
        netNormal += weight[i] * boneNormal;
    }
    netNormal = normalize(netNormal);

    oPosition = mul(modelViewProj, float4(netPosition, 1));
    oTexCoord = texCoord;
    color = computeLighting(light, netPosition, netNormal);
}

```



由于性能的原因，这个程序把 boneMatrix 当作 float4 向量类型的数组而不是一个 $\text{float3} \times 4$ 类型的矩阵数组。matrixIndex 数组包含的是浮点类型的值而不是整数类型，因此一个向量的数组的寻址要比一个矩阵的数组的寻址更加有效。这暗示着在 matrixIndex 向量中的索引应该是实际矩阵索引的 3 倍。因此，程序假设 0 是数组中的第一个矩阵，3 是第二个矩阵，等等。每个顶点的索引是固定的，因此你能够通过把“乘以 3”移到顶点程序以外来提高性能。

一个 for 循环，循环 4 次，用每个骨头矩阵变换缺省姿势的位置和法向量。每个结果都是加权平均的。

这个程序为每个姿势计算加权的位置和法向量。来自示例 6-4 的同一个内部函数 computeLighting，用加权的位置和法向量计算每个顶点物体空间的光照。

虽然这个例子是非常有限的，但你能够把它推广到处理更多的骨头矩阵，更普通的骨头矩阵（例如，允许缩放），以及影响每个顶点的矩阵——和计算一个更好的光照模型。

6.6 练习

- 回答这个问题：对关键帧插值来说，什么是插值函数的同样类型？描述你在哪种情况下，会使用它们。
- 你自己尝试一下：通过增加一个随时间变化的放射项，使得膨胀程序也能够周期性的发光。作为进一步的改进，使得每个顶点的放射项基于顶点的漫反射材质颜色。
- 你自己尝试一下：通过把 t 、 $0.5*t*t$ 和 $-0.8*t*t+0.8t$ 当作 uniform 参数传递来优化 C6E2v_particle 程序。
- 你自己尝试一下：你所实现的粒子系统为粒子大小和颜色使用了特定的函数。尝试用你自己的函数对此进行代替。一个有趣的关于粒子大小的想法是，使粒子的大小依靠于它到眼睛的距离。要做到这一点，修改应用程序，把眼睛的位置当作 uniform 参数传给顶点程序。然后，你就可以计算从任何粒子到眼睛的距离，并基于 $1/(距离)^2$ 或者其他你喜欢的方法进行缩放。
- 你自己尝试一下：如果你临时改变扭曲角度，第 3 章中的 C3E4v_twist 示

例将变为一个程序上的二维动画。编写一个动画扭曲的三维版本，并把它应用到三维模型上。

6. 你自己尝试一下：修改 C6E5v_skin4m 示例来处理包括伸缩的骨矩阵。如果矩阵包含了伸缩，记住需要变换法向量的转置矩阵的逆与变换位置所需要的矩阵是完全不同的。

7. 你自己尝试一下：修改 C6E5v_skin4m 示例在每个顶点处理 6 种骨矩阵，而不是原来的 4 个矩阵。这意味着你需要拆分重量和矩阵索引到多个输入参数。

6.7 补充阅读

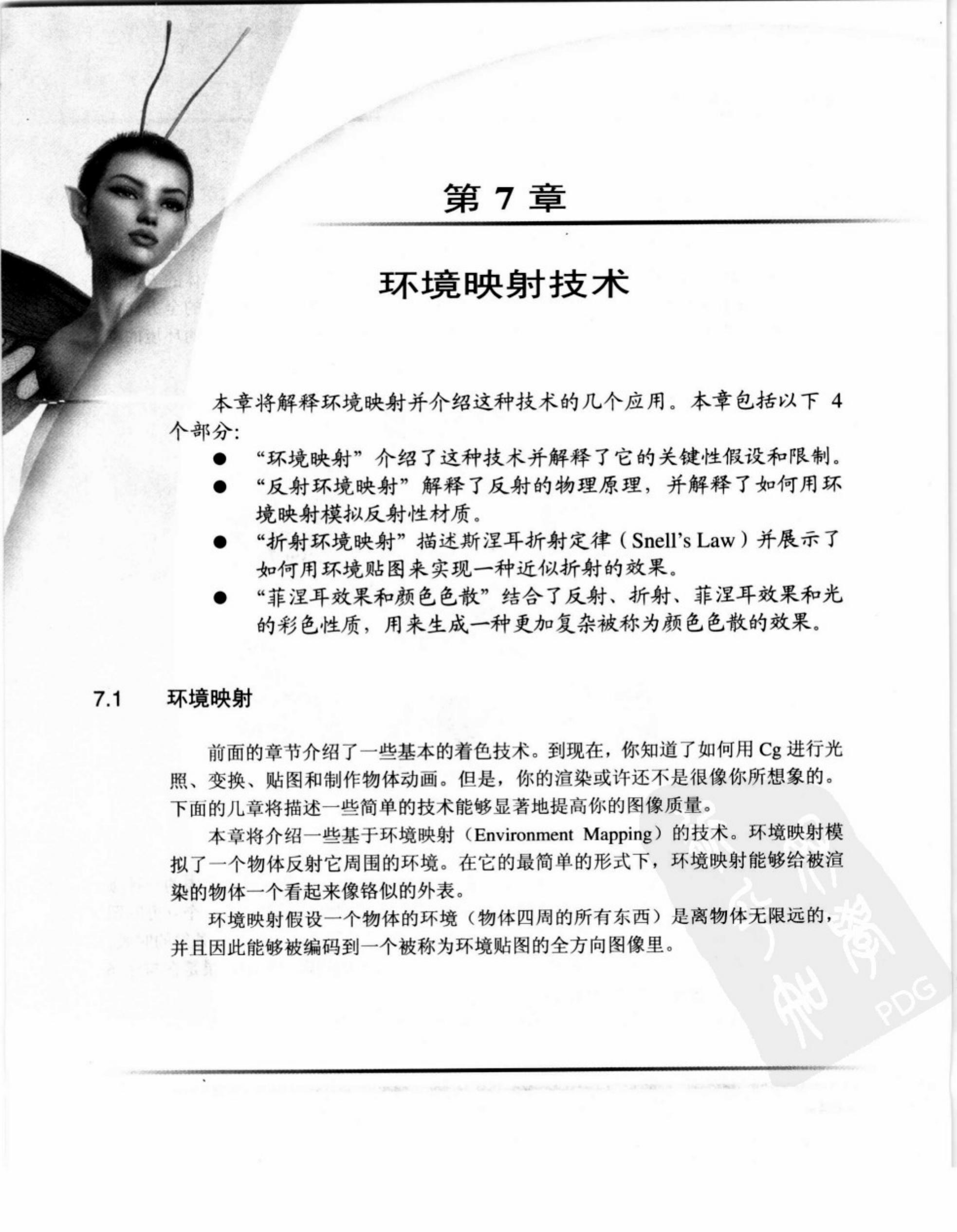
如果你对所创建的粒子系统背后的物理学感兴趣，你可以通过浏览运动学的高中或大学物理课本。

Jeff Lander 在 1998 年和 1999 年为 *Game Developer Magazine* 写了一系列关于各种动画技术的文章。你可以在 www.drawin3d.com 网站上发现这些文章。关于粒子系统，可以阅读“The Ocean Spray in Your Face”。关于顶点混合，可以查阅“Skin Them Bones: Game Programming for the Web Generation”。

由 Mark DeLoura 编辑的 *Game Programming Gems* (Charles River Media, 2000 年) 的最初的一卷包含了一些关于关键帧动画和顶点混合的宝典。请查阅以下这些文章：Herbert Mareslas 撰写的“Interpolated 3D Keyframe Animation”；Torgeir Hagland 撰写的“A Fast and Simple Skinning Technique”和 Ryan Woodland 撰写的“Filling the Gaps—Advanced Animation Using Stitching and Skinning”。

John Vince 的书 *3D Computer Animation* (Addison-Wesley, 1992 年) 覆盖了本章描述的许多技术和其他一些技术，例如自由变形 (FFD)。

DirectX 8 增加了 point sprite 到 Direct3D 中。多个硬件供应商的 OpenGL 实现支持 NV_point_sprite 扩展。OpenGL 扩展的说明可以从 www.opengl.org 网站上获得。



第 7 章

环境映射技术

本章将解释环境映射并介绍这种技术的几个应用。本章包括以下 4 个部分：

- “环境映射”介绍了这种技术并解释了它的关键性假设和限制。
- “反射环境映射”解释了反射的物理原理，并解释了如何用环境映射模拟反射性材质。
- “折射环境映射”描述斯涅耳折射定律 (Snell's Law) 并展示了如何用环境贴图来实现一种近似折射的效果。
- “菲涅耳效果和颜色色散”结合了反射、折射、菲涅耳效果和光的彩色性质，用来生成一种更加复杂被称为颜色色散的效果。

7.1 环境映射

前面的章节介绍了一些基本的着色技术。到现在，你知道了如何用 Cg 进行光照、变换、贴图和制作物体动画。但是，你的渲染或许还不是很像你所想象的。下面的几章将描述一些简单的技术能够显著地提高你的图像质量。

本章将介绍一些基于环境映射 (Environment Mapping) 的技术。环境映射模拟了一个物体反射它周围的环境。在它的最简单的形式下，环境映射能够给被渲染的物体一个看起来像铬似的外表。

环境映射假设一个物体的环境（物体四周的所有东西）是离物体无限远的，并且因此能够被编码到一个被称为环境贴图的全方向图像里。

7.1.1 立方贴图纹理

所有近来的图形处理器都支持一种被称为立方贴图的纹理。一个立方贴图不是由一幅，而是由 6 幅正方的刚好能够组合在一起形成一个立方体的表面的纹理图像组成。这 6 幅图像一起形成了一个我们用来编码环境贴图的全方向图像。图 7-1 显示了一个记录了由多云的天空和模糊的山脉地形构成的环境的立方贴图。

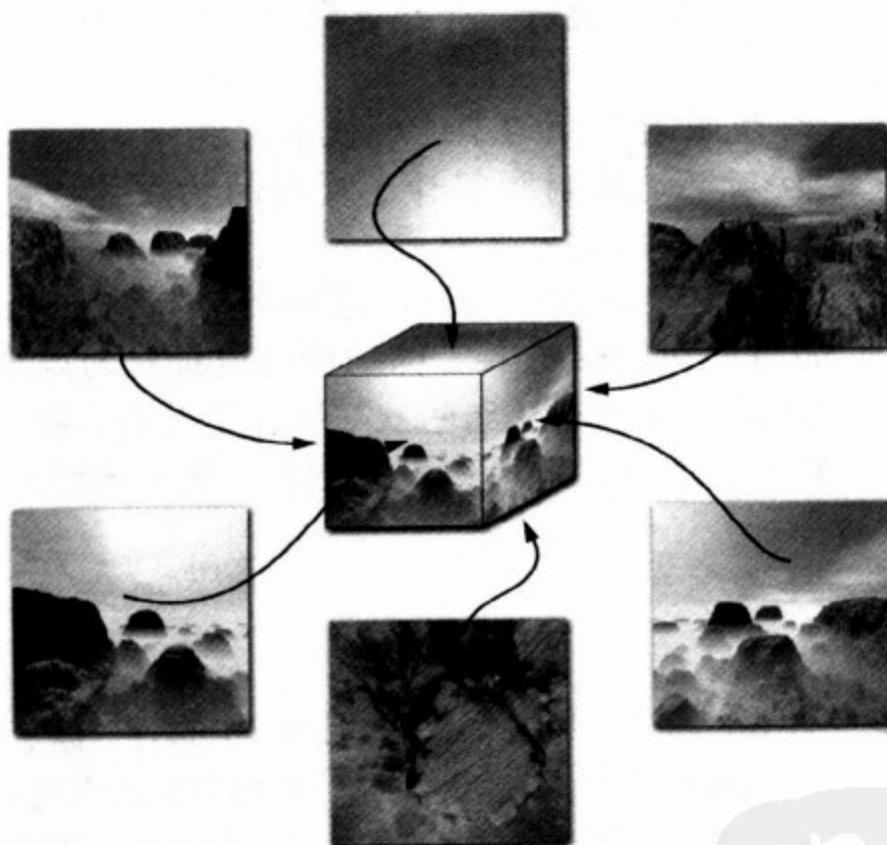


图 7-1 一个立方贴图的纹理图像

一个 2D 的纹理映射一个 2D 的纹理坐标集到一个单独纹理图像中的一种颜色。相反，你需要用一个表示 3D 方向向量的三元纹理坐标集来存取一个立方贴图纹理。你可以把这个向量看成是从立方体中心射出的光线。当光线向外射的时候，它会与立方贴图的 6 个表面之一相交。一个立方贴图纹理存取的结果是在与这 6 个纹理图像相交的点的过滤颜色。

立方贴图纹理用于环境映射是非常理想的。立方贴图的每个表面编码了围绕

一个物体的全景环境的 1/6。一个立方贴图纹理为决定位于环境中央的物体在任何一个特定方向所“看”到的颜色提供了一个非常快速的方法。

7.1.2 生成立方贴图

为了生成一个立方贴图，你需要用一个照相机替换你想要把反射加在它表面的物体，把照相机放在物体所在位置并在 6 个方向（X 正轴、X 负轴、Y 正轴、Y 负轴、Z 正轴和 Z 负轴）上照相。每张相片都应该有一个 90 度的视角和一个正方的长宽比例，这样 6 个立方体表面就可以紧密地结合在一起了——没有任何缝隙或重叠——创造了一个全方向的全景图。可以使用这些图像作为你的立方体图的 6 个表面。

你可以用一台计算机渲染 6 个视图，或者用一组照片来记录一个真实的环境，并将它们一起变形来创建一个环境贴图。

7.1.3 环境映射的概念

当你看一个高度反射的物体例如铬金球体的时候，你看到的不是物体的本身，而是物体反射的它周围的环境。当你凝视一个高度反射的表面上的某一点的时候，在那个点的表面反射视线——也就是光线从你的眼睛传播到表面上的这个点——到环境中去。反射光线的特征是基于初始的视线和在视线到达表面上那一点的法向量的。你所看到的不是表面本身，而是环境在反射光线的方向上看上去的样子。

当你使用一个立方贴图来编码环境从各个方向看上去的样子的时候，渲染反射表面上的一个点大概只需要为表面上的那个点计算反射的视线方向。然后你就可以基于反射的视线方向存取立方贴图，来为表面上的这个点决定环境的颜色。

7.1.4 计算反射向量

图 7-2 显示了一个物体、一个眼睛的位置和一个记录的围绕物体环境的立方贴图纹理。当然因为图 7-2 是在 2D 中描绘 3D 场景，所以物体只用一个梯形来显示，而环境只用一个环绕的正方形来显示，而不是一个正真的立方体。

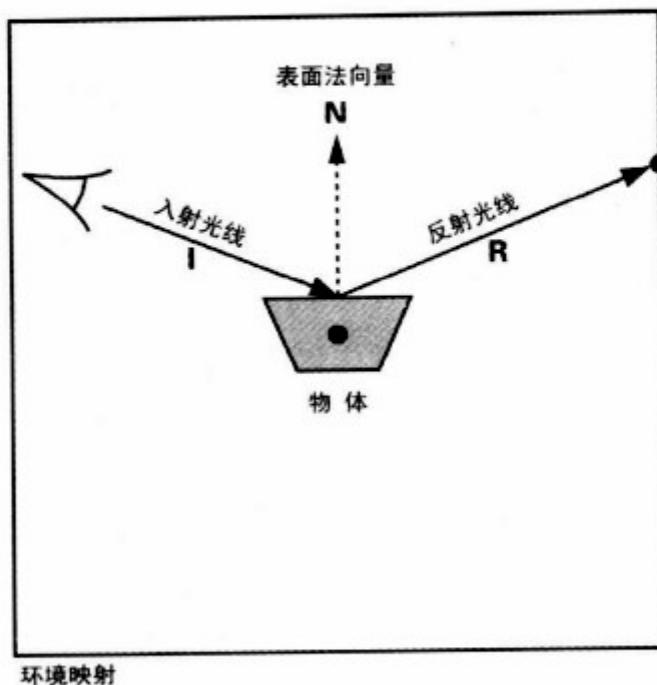


图 7-2 环境映射

向量 I——被称为入射光线 (incident ray) ——从眼睛射到物体的表面。当 I 到达表面的时候，它会根据表面的法向量 N 被从 R 方向反射出去。第二个光线是被反射光线 (reflected ray)。图 7-3 显示了这种情况的几何排列。

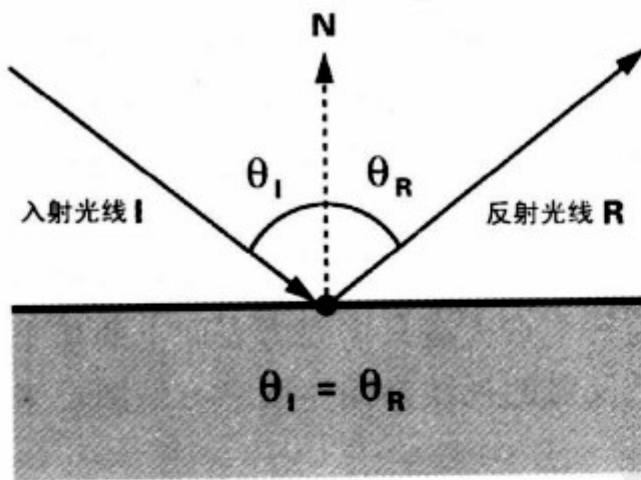


图 7-3 计算被反射的光线

对一个完美的反射体例如一面镜子，入射角度 (θ_I) 与反射角度 (θ_R) 是一样的。你能够根据向量 I 和 N 用公式 7-1 计算被反射的向量 R。

公式 7-1 向量反射

$$\mathbf{R} = \mathbf{I} - 2\mathbf{N}(\mathbf{N} \cdot \mathbf{I})$$

计算一个反射向量在计算机图形学中是一种非常普通的操作，因此 Cg 提供了 reflect 标准库函数。这个函数接受入射向量和表面的法向量，然后返回反射向量。

reflect (I, N) 为入射光线 I 和表面法向量 N 返回反射向量。向量 N 必须规范化。反射向量的长度等于 I 的长度。这个函数只对三元向量有效。

虽然你最好使用 Cg 标准库函数因为它们效率很高，最直接的 reflect 的实现方法如下所示：

```
float3 reflect(float3 I, float3 N)
{
    return I - 2.0 * N * dot(N, I);
}
```

我们将稍后使 reflect 函数投入工作。

7.1.5 环境映射的一些假设

在前面的讨论中我们提过环境映射假设环境离物体无限远。现在，我们将探索这个假设的含义。

进行这种假设的原因是，环境贴图只是单独地基于一个 3D 方向进行存取。环境映射不允许位置上的变化来影响表面的反射外观。如果环境中的所有东西都离表面足够远，那么这个假设就近似地成立了。

实际上，当环境离物体没有足够远的时候所造成的视觉上的人为现象完全不会被察觉。特别是曲面上的反射已经足够精细了，当反射在物理上不是完全精确的时候，大部分人都不会注意到。只要反射与环境颜色大致匹配并随着表面的曲率适当变化，用环境映射的渲染的表面就会看上比较可信。

你将会非常惊奇地看到你所能获得的效果。

在理想情况下，场景中每个环境映射的物体都应该有它自己的环境贴图。实际上，物体通常能够共享环境贴图而不让人注意到。

在理论上，当物体在环境中移动的时候，或者当使用环境贴图的反射物体相对环境移动了很多的时候，你应该重新生成一个环境贴图。而实际上，使用静态的环境贴图获得令人信服的反射是可能的。

使用一个环境贴图，一个物体只能反射环境；它不能反射它本身。类似地，

不要期望多个反射，例如当两个发亮的物体互相反射的情况。因为一个环境映射的物体只能反射它自己的环境而不能反射它本身，环境映射在凸的或大部分凸的物体上工作的最好——而在有很多凹陷表面的物体上不是很好。

因为环境映射只依赖于方向而不依赖于位置，它在平的反射表面上表现得很差，例如镜子，在镜面上反射主要依赖于位置。相反，环境映射在曲面上表现得很好。

7.2 反射环境映射

让我们从最普通的使用环境映射的情况开始：创建一个像铬的反射物体。这是该技术的梗概，但是它已经能够生成很好的结果了，如图 7-4 所示。

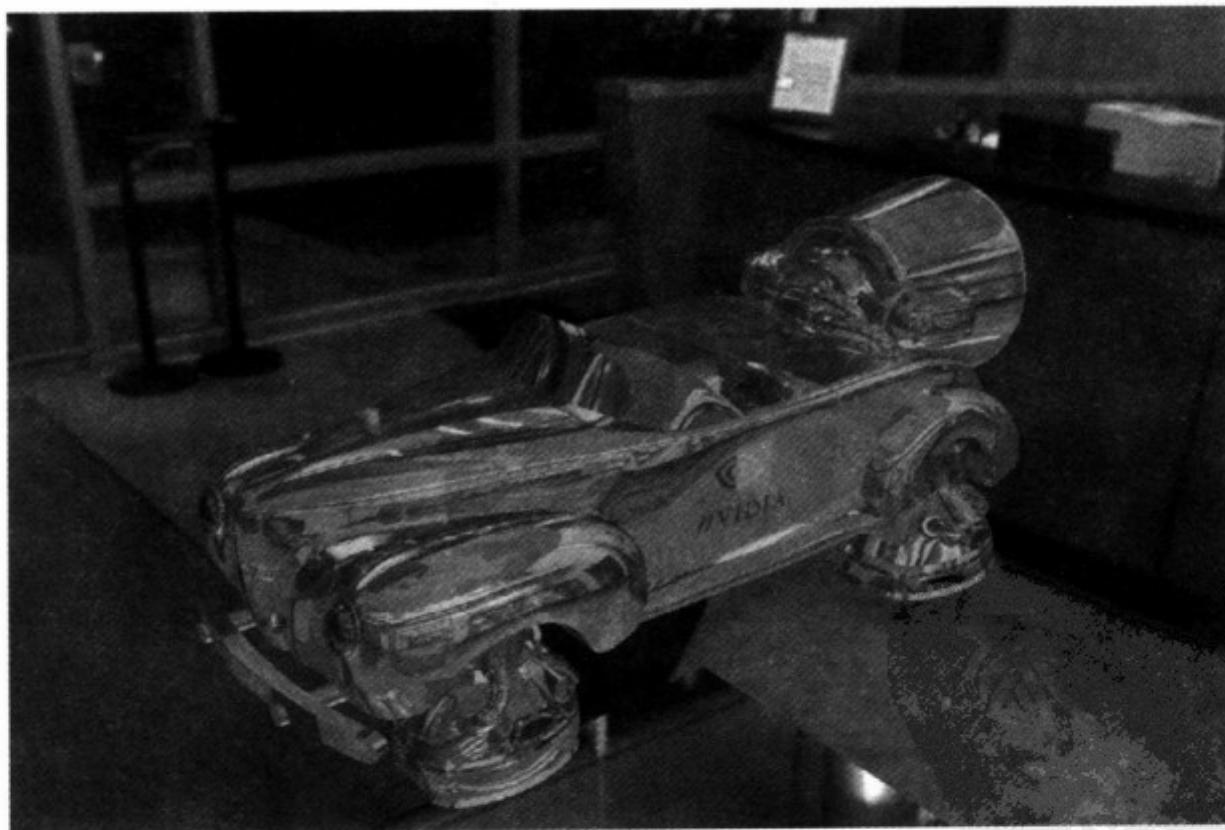


图 7-4 反射性环境映射

在这个例子中，顶点程序计算入射和反射光线。然后，它把反射光线传给片段程序，片段程序再查找环境贴图，并用它来给片段的最后颜色增加一个反射项。为了使例子更加有趣，并使我们的例子更像一个真正的应用程序，我们把反射和

一个贴图纹理混合。一个被称为 reflectivity 的统一参数允许应用程序来控制材质的反射性。

你也许想知道我们为什么不用片段程序来计算反射向量。一个由片段程序在每个片段计算的反射向量将实现更高的图像质量，但是它在基本的片段 profile 上不能运行。因此，我们把每个片段的实现当作练习留给你来实现。在本章的后面，我们将讨论使用顶点程序和使用片段程序的平衡和关联。

7.2.1 应用程序指定的参数

表 7-1 列出了应用程序需要提供给图形流水线的数据。

表 7-1 应用程序为每个顶点环境映射指定的参数

参 数	变 量 名	类 型
顶点程序变化参数		
物体空间的顶点位置	position	float4
物体空间的顶点法向量	normal	float3
纹理坐标	texCoord	float2
顶点程序的统一参数		
连接好的 modelview 和投影矩阵	modelViewProj	float4×4
物体空间到世界空间的变换	modelToWorld	float4×4
片断程序的统一参数		
贴图纹理	decalMap	sampler2D
环境贴图	environmentMap	samplerCUBE
眼睛的位置（在世界空间）	eyePosition	float3
反射性	reflectivity	float

7.2.2 顶点程序

示例 7-1 给出了顶点程序为环境映射执行每个顶点的反射向量的计算。

一、基本操作

顶点程序从一些普通的操作开始：把顶点的位置变换到剪裁空间以及为贴图纹理传递纹理坐标集。

```

oPosition = mul ( modelViewProj, position);
oTexCoord = texCoord;

```

示例 7-1 C7E1v_reflection 顶点程序

```

void C7E1v_reflection(float4 position : POSITION,
                      float2 texCoord : TEXCOORD0,
                      float3 normal : NORMAL,

                      out float4 oPosition : POSITION,
                      out float2 oTexCoord : TEXCOORD0,
                      out float3 R : TEXCOORD1,

                      uniform float3 eyePositionW,
                      uniform float4x4 modelViewProj,
                      uniform float4x4 modelToWorld)
{
    oPosition = mul(modelViewProj, position);
    oTexCoord = texCoord;

    // Compute position and normal in world space
    float3 positionW = mul(modelToWorld, position).xyz;
    float3 N = mul((float3x3)modelToWorld, normal);
    N = normalize(N);

    // Compute the incident and reflected vectors
    float3 I = positionW - eyePositionW;
    R = reflect(I, N);
}

```

二、把向量变换到世界空间中

环境贴图通常是基于世界空间来确定方向的，因此你需要在世界空间中计算反射向量（或者其他任何用来给环境贴图定向的坐标系统）。要实现这一点，你必须把其余顶点的数据变换到世界空间。特别地是，你需要通过用 `modelToWorld` 矩阵乘以顶点的位置和法向量来把它们变换到世界空间：

```

float3 positionW = mul(modelToWorld, position).xyz;
float3 N = mul ( ( float3x3 )modelToWorld, normal );

```

`modelToWorld` 矩阵是一个 `float4x4` 类型，当我们变换一个法向量的时候，我们只需要矩阵左上角的 3×3 部分。Cg 允许你把较大的矩阵转换成较小的矩阵，就

像在上面的代码中那样。当你把一个较大的矩阵转换成一个较小的矩阵类型的时候，例如一个 $\text{float}4 \times 4$ 的矩阵转换成一个 $\text{float}3 \times 3$ 的矩阵，较大矩阵的左上角部分会填充到较小类型的矩阵中去。例如，如果你有一个 $\text{float}4 \times 4$ 的矩阵 M：

$$M = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & 7.0 & 8.0 \\ 9.0 & 10.0 & 11.0 & 12.0 \\ 13.0 & 14.0 & 15.0 & 16.0 \end{bmatrix}$$

而你要把它转换成一个 $\text{float}3 \times 3$ 矩阵，你最后将得到矩阵 N：

$$N = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 5.0 & 6.0 & 7.0 \\ 9.0 & 10.0 & 11.0 \end{bmatrix}$$

Advanced 回忆一下在第 4 章（4.1.3 小节）中建模变换把物体空间坐标转换成为世界空间坐标。在这个例子中，我们假设建模变换是仿射的（而不是投影的）并且在缩放的时候是统一的（而不是不一致地缩放 x、y 和 z）。我们还假设 position 的 w 分量是 1，即使在 C7E1v_reflection 中 position 被定义成一个 $\text{float}4$ 类型。

这些假设通常是成立的，但是如果它们在你遇到的情况下不成立，这时你就需要做如下的工作。

如果建模变换对位置的缩放不一致，你必须用建模矩阵的逆矩阵 (`modelToWorldInvTrans`) 来乘以 `normal`，而不是简单地用 `modelToWorld`。这也就是：

```
float3 N = mul( (float3x3)modelToWorldInvTrans, normal);
```

如果建模变换是投影的或者物体空间中的 position 的 w 分量不为 1，你必须用 `positionW` 的 w 分量除以它本身。也就是：

```
positionW /= positionW.w;
```

`/=` 操作符是一个赋值操作符，就像它在 C 和 C++ 中一样，在这种情况下，它用 `positionW.w` 除以 `positionW` 然后把结果赋给 `positionW`。

三、规范化法向量

顶点的法向量需要被规范化：

```
N = normalize(N);
```

Advanced

在某些情况下，我们能够跳过这个 `normalize` 函数调用。如果我们知道 `modelToWorld` 矩阵左上的 3×3 部分没有引起不一致的缩放，并且物体空间法向量参数已经被确保规范化了，`normalize` 调用就不再需要了。

四、计算入射向量

入射向量与在第 5 章中为镜面反射光照使用的视向量是相反的。入射向量是从眼睛到顶点的向量（而视向量是从顶点到眼睛）。有了世界空间中的眼睛位置 (`eyePositionW`) 作为统一参数，并且从前一步获得的世界空间中的顶点位置 (`positionW`)，计算入射向量只需要一个简单的减法：

```
float3 I = positionW - eyePositionW;
```

五、计算反射向量

现在已经有了你需要的向量——位置和法向量，并且都在世界空间中——因此你能够计算反射向量了：

```
float3 R = reflect(I, N);
```

下一步，程序将用一个三元纹理坐标集来输出被反射的世界空间中的向量 `R`。随后的片段程序示例将使用这个纹理坐标集来存取一个包含了环境贴图的立方贴图纹理。

六、规范化向量

你也许想知道为什么我们没有规范化 `I` 或 `R`。规范化在这里并不需要，因为反射向量是被用来查询一个立方贴图的。当存取一个立方贴图的时候，反射向量的方向就是所有需要的东西了。不管它的长度是多少，反射光线都将会在完全一样的位置与立方贴图相交。

而且因为只要 `N` 是规范化的，`reflect` 函数就会输出一个与入射向量有着相同长度的反射向量，而入射向量的长度在这种情况下也不起什么作用。

这里还有一个不对 `R` 进行规范化的原因。光栅器将在片段程序使用 `R` 之前对它进行插值。如果每个顶点的反射向量没有被规范化，插值将更加精确。

7.2.3 片段程序

示例 7-2 显示了一个非常简短的片段程序，因为 C7E1v_reflection 顶点程序已经完成了大部分的计算。所剩下的工作只是立方贴图查询和最后的颜色计算。

示例 7-2 C7E2f_reflection 片段程序

```
void C7E2f_reflection(float2 texCoord : TEXCOORD0,
                      float3 R : TEXCOORD1,
out float4 color : COLOR,
uniform float reflectivity,
uniform sampler2D decalMap,
uniform samplerCUBE environmentMap)
{
    // Fetch reflected environment color
    float4 reflectedColor = texCUBE(environmentMap, R);

    // Fetch the decal base color
    float4 decalColor = tex2D(decalMap, texCoord);

    color = lerp(decalColor, reflectedColor, reflectivity);
}
```

片段程序接收到被用来从环境贴图获得反射颜色的经过插值的反射向量：

```
float4 reflectedColor = texCUBE(environmentMap, R);
```

请注意一下新的纹理查询函数 **texCUBE**。这个函数被特定地用来存取立方贴图，并且因此它的第二个参数（是一个三元纹理坐标集）代表了一个方向。

目前，你能够把 **reflectedColor** 赋给 **color**，使得被渲染的物体完全反射。但是，没有真实的材质是一个完美反射体，因此要使得结果更加有趣，这个程序增加了一个贴图查找，并把贴图颜色和反射颜色相混合：

```
float4 decalColor = tex2D(decalMap, texCoord);
color = lerp(decalColor, reflectedColor, reflectivity);
```

正如你在第 3.3.5 节看到的那样，**lerp** 函数执行了一个线性插值。传给 **lerp** 的参数是 **decalColor**、**reflectedColor** 和 **reflectivity**。因此，当 **reflectivity** 是 0 的时候，你的程序将只输出贴图颜色而没有任何反射。相反，当 **reflectivity** 是 1 的时候，这

个程序将只输出反射颜色，生成一个完全反射的有着铬金属样子的外表。reflectivity 的中间值将导致一个有部分反射性质的贴图模型。

7.2.4 控制贴图

Advanced

在这个例子中，reflectivity 是一个统一参数。这个假设认为在场景中的每一个几何体在它的整个表面上都拥有相同的反射率。但是并不需要都是这种情况。通过用一个纹理对反射率进行编码，你能够创建一个更加有趣的效果。这个方法允许你在每个片段改变反射的量，这使得创建一个既有反射部分也有不反射部分的物体变得很容易了。

因为使用一个纹理来控制着色参数的想法是那么的有用，我们把这样的一个纹理称为控制贴图。控制贴图是非常重要的，因为它们能够有效地使用图形处理器的高效的纹理操作能力。另外，控制贴图给了美工人员对效果的更多的控制，而不用对底层的应用程序有深刻的理解。例如，一个不知道环境映射是如何工作的美工人员就能够描绘一个“反射率贴图”。

控制贴图是一个给差不多所有程序增加细节和复杂度的非常好的方法。

7.2.5 顶点程序与片段程序

我们在前面提到过通过使用片段程序（代替顶点程序）来计算反射向量，你能够获得更高的图像质量。为什么会这样呢？这和每个片段的光照看起来比每个顶点的光照效果好的原因相同。

正如在镜面反射光照中那样，用于环境映射的反射向量从一个片段到另一片段以非线性的方式变化。这意味着进行线性插值的每个顶点的值将不能够准确地记录反射向量的变化。特别是，微小的每个顶点的人为现象往往出现在物体的轮廓附近，在这些地方反射向量在每个三角形中都变化得非常快。为了获得更精确的反射，可以把反射向量的计算移到片段程序。用这种方法，你明确地为每个片段计算反射向量而不是对它们进行插值。

虽然获得了额外的精确度，但是每个片段的环境映射也许不能对图像质量改善得足够多来证明额外的开销是必要的。正如在本章前面解释的，大部分人从一个粗略的角度不可能注意或欣赏更加正确的反射。记住环境映射不能生成物理上正确的反射。

7.3 折射环境映射

既然，你已经学会了如何实现基本的环境映射，你就能够使用它模拟一些相关的物理现象。你在下面几个小节中学习的技术将举例说明，当你使用一种像 Cg 的高级语言的时候，把理论应用于实践是多么的容易。同样的技术不使用 Cg 也能够实现，但是它们需要更多的汇编级编码的技巧。结果，这些技术和所能生成的图像质量将不能被大部分开发人员获得，即使这些效果已经被底层的图形硬件所支持。

在本小节，你将学习如何用一点物理学和一点环境映射的知识来实现折射。图 7-5 举例说明了你将尝试获得的效果。



图 7-5 折射环境映射

7.3.1 折射的物理原理

当光通过不同密度（例如空气和水）的两种材质之间的界面的时候，光的方向会改变。在方向上的改变是因为光在较浓的材质（或媒体，当材质已经在折射环境中被使用了）中传播比较慢。例如，光在空气中传播的比较快，但在水中慢很多。折射的最经典的例子是当你把吸管放在一杯水中的时候，吸管看起来就像着“折”了一样。

一、斯涅耳定律

斯涅耳定律描述了两种媒体之间的分界面（或接触面，当分界面已经在折射环境中被使用了）光发生了什么，如图 7-6 所示。折射向量被表示为 T ，代表了“transmitted”。斯涅耳定律在数学上可以用公式 7-2 表示。这个公式有 4 个变量：入射角 θ_I ，折射角 θ_T 和给每个媒体的折射索引 η_1 和 η_2 。

公式 7-2 斯涅耳定律

$$\eta_1 \sin \theta_I = \eta_2 \sin \theta_T$$

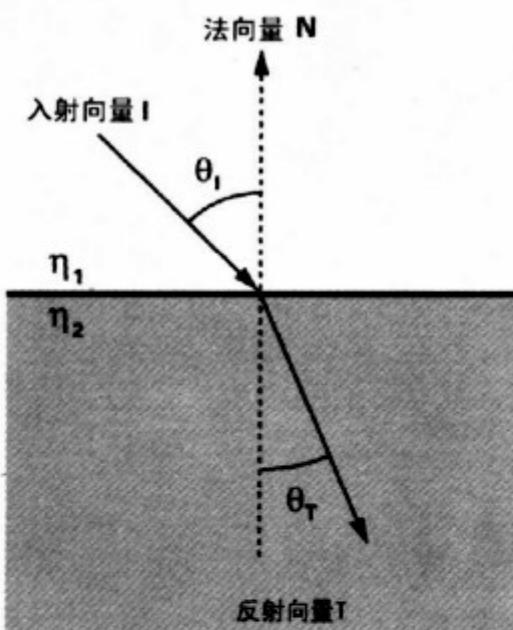


图 7-6 斯涅耳定律

一个媒体的折射指数度量了媒体是如何影响光的速度的。一个媒体的折射指

数越大，光在其中的传播速度越慢。表 7-2 列出了一些常用的材质和它们近似的折射指数（一种材质的折射指数实际上不仅依赖于材质，而且还依赖于入射光的波长，但是我们现在忽略了这种复杂性）。

表 7-2

折射指数

常用材质	折射指数
真空	1.0
空气	1.0003
水	1.3333
玻璃	1.5
塑料	1.5
钻石	2.417

注意：

不同类型的玻璃有不同的折射指数，但是 1.5 表示的是普通窗户玻璃的一个合理的值。它也是大部分塑料的合理近似值。

在这个例子里，你将模拟折射，如图 7-7 所示。每个从眼睛的入射光线被折射，并且每个折射光线被用来查找环境贴图（就像在反射映射例子中，反射光线被用来查找环境贴图）。

注意我们只模拟第一个折射光线。图 7-8 显示了我们的方法和一个更精确的方法对一个简单物体的不同之处。入射光线实际上应该被折射两次——第一次当它进入物体时，而第二次发生在它离开的时候（记做向量 T' ）。但是，我们不模拟第二次折射，因此我们用 T 作为传播光线而不用 T' 。这两个光线最后与环境在不同的位置相交（在图 7-8 中被标为 A 和 B）。幸运地是，折射实在是太复杂了，以至于生成的图像在大部分情况下很难区别。特别是对一个不经意的观察者，很难注意生成的折射不是完全正确的。

这种类型的简化通常会发生在实时计算机图形学中。要记住的是结果是最主要的。如果你的图像看起来令人信服，那么它们在物理上不精确通常没有什么关系。在许多情况下，如果你要计算一个完全的物理模拟，你的帧率将会急剧下降。这就是为什么，在它早期的日子里，实时计算机图形学已关注发现新的有效的技术使图像看起来更好。当然，我们的目标仍然是发明既精确又快速的方法，但是

在大部分情况下，程序员必须在精确和性能之间找到一个适当的平衡。

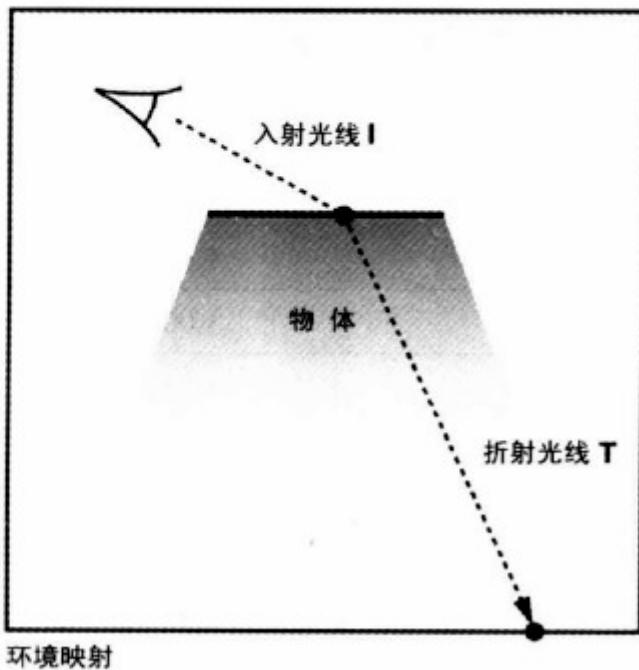


图 7-7 折射到一个环境贴图

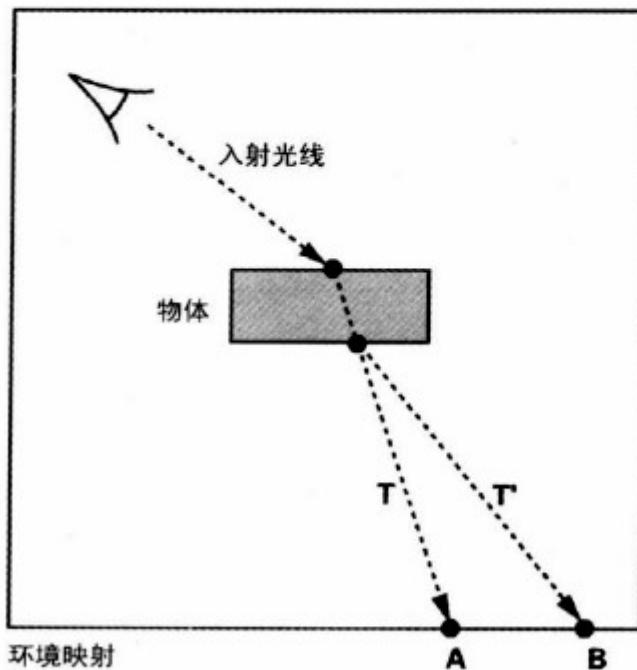


图 7-8 多折射与一个反射

二、折射指数的比率

为了计算折射，你所需要的一个关键的值是每个媒体的折射指数间的比率。在下一个例子中，应用程序需要传递 etaRatio 两个媒体折射指数间的比率给顶点程序。按照惯例，希腊字母 η (“eta”) 被用来表示单个材质的折射指数。但是在实践中折射指数的比率更加有效，因为它能够把程序从为每个顶点计算比率中节省出来（当它本来只需要为整个网格计算一次）。

7.3.2 顶点程序

折射在许多方面都类似反射。在这两种情况下，一个入射光线到达一个表面，然后某些现象会发生（在折射情况下光会被弹回，并且在折射情况下光进入表面后会改变方向）。这些相似之处预示了折射的 Cg 代码与反射的代码也是相似的。而且事实上，它们的 Cg 代码确实很相似。

在示例 7-3 中的顶点程序 C7E3v_refraction 为折射需要计算和输出折射光线，而不是像在 C7E1v_reflection 中的反射光线。你自己不需要应用斯涅耳定律，因为 Cg 有一个 refract 函数会帮你计算。下面是函数的定义：

`refract(I, N, etaRatio)` 给了入射光线的方向 I 、表面法向量 N 和相对折射指数 etaRatio ，如图 7-6 所示的那样，这个函数计算折射向量 T 。向量 N 应该被规范化。这样折射向量的长度等于 I 的长度。 etaRatio 是包含入射光线的媒体的折射指数与被进入的媒体的折射指数的比率。这个函数只对三元向量有效。

下面是 `refract` 标准库函数的一个简单的实现：

```
float3 refract ( float3 I, float3 N, float etaRatio)
{
    float cosI = dot (-I, N);
    float cost2 = 1.0f - etaRatio * etaRatio * ( 1.0f - cosI * cosI );
    float3 T = etaRatio * I +
               ((etaRatio * cosI - sqrt(abs(cost2))) * N);
    return T * (float3) (cost2 > 0);
}
```

示例 7-3 C7E3v_refraction 顶点程序

```
void C7E3v_refraction(float4 position : POSITION,
                      float2 texCoord : TEXCOORD0,
                      float3 normal : NORMAL,

                      out float4 oPosition : POSITION,
                      out float2 oTexCoord : TEXCOORD0,
                      out float3 T : TEXCOORD1,

                      uniform float etaRatio,
                      uniform float3 eyePositionW,
                      uniform float4x4 modelViewProj,
                      uniform float4x4 modelToWorld)
{
    oPosition = mul(modelViewProj, position);
    oTexCoord = texCoord;

    // Compute position and normal in world space
    float3 positionW = mul(modelToWorld, position).xyz;
    float3 N = mul((float3x3)modelToWorld, normal);
    N = normalize(N);

    // Compute the incident and refracted vectors
    float3 I = positionW - eyePositionW;
```

```

    T = refract(I, N, etaRatio);
}

```



当光从一个稠密的媒体传播到一个不太稠密的媒体时，光能够折射到发生全反射的程度。例如，如果你在一个游泳池的水下面，并且水面足够平静。当从掠射角看过去的时候，水的表面看起来就像一面镜子。在这种情况下， $\cos T_2$ 是小于或等于 0 的并且 **refract** 函数返回一个零向量。

在早期的 C7E1v_reflection 示例和 C7E3v_refraction 示例的主要不同之处是使用 **refract** 函数（而不是 **reflect** 函数）来计算折射向量 T。

7.3.3 片段程序

片段程序不需要做任何修改，因为它的角色保持不变：它将基于输入的向量来查找环境贴图。现在的输入向量是折射向量而不是反射向量，但是片段程序的运行与它在反射映射示例中所做的完全一样。片段程序查找环境贴图，把查找结果与贴图纹理颜色混合，然后返回结果。为了确保一致性，在示例 7-4 中的片段程序 C7E4f_refraction 把 **reflected-Color** 改名为 **refractedColor**，把 **reflectivity** 改名为 **transmittance**，但是这些只是对较早的 C7E2f_reflection 程序的装饰性修改。

示例 7-4 C7E4f_refraction 片段程序

```

void C7E4f_refraction(float2 texCoord : TEXCOORD0,
                      float3 T : TEXCOORD1,

                      out float4 color : COLOR,

                      uniform float transmittance,
                      uniform sampler2D decalMap,
                      uniform samplerCUBE environmentMap)
{
    // Fetch the decal base color
    float4 decalColor = tex2D(decalMap, texCoord);

    // Fetch refracted environment color
    float4 refractedColor = texCUBE(environmentMap, T);

    // Compute the final color
    color = lerp(decalColor, refractedColor, transmittance);
}

```

7.4 菲涅耳效果和颜色色散

现在你知道如何实现反射和折射了。下一个例子将把它们结合在一起，并给出一些其他扩展。你将会学到两个新的效果：菲涅耳效果和颜色色散。

7.4.1 菲涅耳效果

一般而言，当光到达两种材质的接触面的时候，一些光在接触面的表面被反射出去，而另一部分光将发生折射穿过接触面。这个现象被称为菲涅耳效果（Fresnel effect，发“freh-'nell”的音）。菲涅耳公式描述了多少光被反射和多少光被折射。如果你曾经想过为什么只有在几乎垂直向下看的时候，才能看到池塘中的鱼，这就是因为菲涅耳效果。在一个较小的角度，将会有许多反射而几乎没有折射，因此很难透过水的表面看到水下。

菲涅耳效果为你的图像增加了真实性，因为它允许你创建物体展示了反射和折射的混合，使得物体更像真实世界的物体。

量化了菲涅耳效果的菲涅耳公式是非常复杂的（你可以从大部分光学课本中学到更多的相关知识）。再一次声明，我们的想法是使创建的图像看上去真实，而不需要精确地描述底层的物理。因此，我们不使用菲涅耳公式本身，而使用公式7-3所示的经验公式，它能够用非常少的计算获得很好的结果：

公式 7-3 菲涅耳公式的一个近似

$$\text{reflectionCoefficient} = \max(0, \min(1, \text{bias} + \text{scale} \times (1 + \text{I} \cdot \text{N})^{\text{power}}))$$

这个公式的基本概念是当 I 和 N 几乎重合的时候，反射系数应该为 0 或几乎为 0，表明大部分的光应该被折射。当 I 和 N 分开的时候，反射系数应该逐渐增加并最终突然（由于指数的原因）增加到 1。当 I 和 N 分开的足够多的时候，几乎所有的光都应该被反射，仅有一点或完全没有被折射。

反射系数的范围被限制在 [0, 1] 之间，因为我们根据下面的公式使用反射系数来混合反射和折射分量（其中 C 代表颜色）：

$$\text{C}_{\text{Final}} = \text{reflectionCoefficient} \times \text{C}_{\text{Reflected}} + (1 - \text{reflectionCoefficient}) \times \text{C}_{\text{Refracted}}$$

7.4.2 颜色色散

前面对折射的讨论是有些被简化了的。我们提到了折射是基于表面法向量、入射角和折射指数的比率。除了这些因素以外，折射量还决定于入射光的波长。例如，红光要比蓝光折射得多。这个现象被称为颜色色散（chromatic dispersion），这就是当白光进入棱镜并形成一个彩虹的原因。

图 7-9 举例从概念上说明了颜色色散。入射光（假设为白色）被分离成几个折射光线。你将模拟发生在光的红色、绿色和蓝色分量的情况，因为在计算机图形学中这些是颜色的标准分量。你将使用红色、绿色和蓝色光线来查找环境贴图，正如你在折射的例子中对一个单独的光线所做的那样。

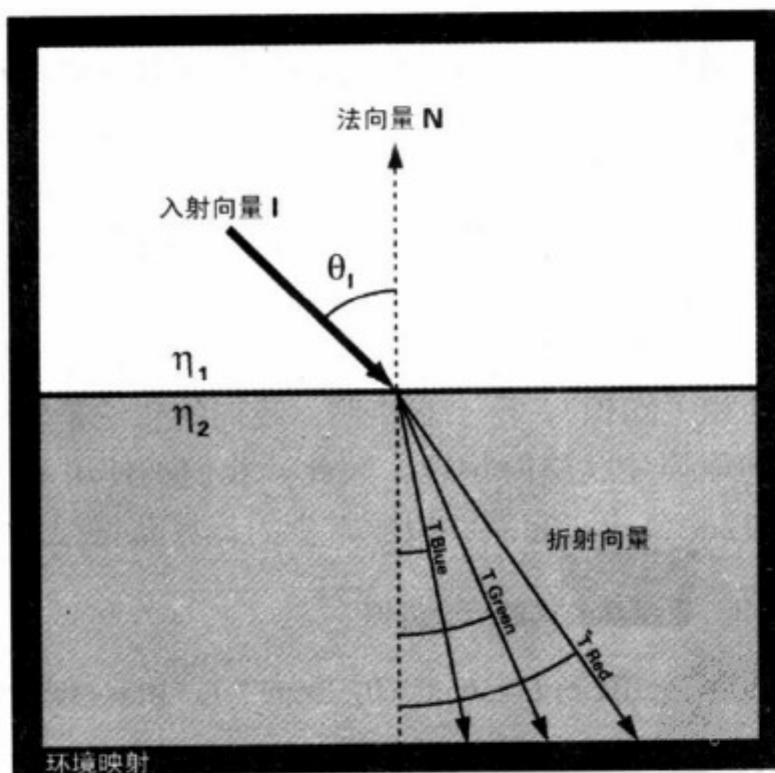


图 7-9 理解颜色色散

记住真正的光有一个波段的波长，而不是仅仅只有 3 个特殊的离散波长。尽管如此，这个近似已经足够有效来被使用了。

结合了颜色色散的菲涅耳效果将创建一种彩虹效果，就好像被渲染的物体是由水晶做成的，如图 7-10 所示。在本书开始部分的彩图 11 用彩色显示了这副图像。



图 7-10 菲涅耳效果和颜色色散

7.4.3 应用程序指定的参数

因为我们现在为物体表面使用了一个更加复杂的光照模型，应用程序需要传递额外的统一参数给顶点和片段程序。这些额外的参数被列在表 7-3 中了。

表 7-3 C7E5v_dispersion 程序参数

参数	变量名	类型
给红光、绿光和蓝光的折射指数的比率(被压缩在一个 float3 类型中)	etaRatio	float3
菲涅耳指数	fresnelPower	float
菲涅耳缩放系数	fresnelScale	float
菲涅耳偏移系数	fresnelBias	float

在 etaRatio 的 x、y 和 z 分量中分别存储了给红光、绿光和蓝光的折射指数的比率。fresnelPower、fresnelScale 和 fresnelBias 变量提供了一种方法来形成我们将用来近似菲涅耳公式的函数。总而言之，所有应用程序指定的参数定义了你的物体的材质特性。

7.4.4 顶点程序

在示例 7-5 中的 C7E5v_dispersion 顶点程序计算了反射向量，以及红光、绿光和蓝光的折射向量。另外，你将使用菲涅耳公式的近似公式来计算反射系数。所有这些信息将被插值并被片段程序接收。

示例 7-5 C7E5v_dispersion 顶点程序

```
void C7E5v_dispersion(float4 position: POSITION,
                      float3 normal : NORMAL,
                      out float4 oPosition : POSITION,
                      out float reflctionFactor: COLOR,
                      out float3 R : TEXCOORD0,
                      out float3 TRed : TEXCOORD1,
                      out float3 TGreen : TEXCOORD2,
                      out float3 TBlue : TEXCOORD3,

                      uniform float fresnelBias,
                      uniform float fresnelScale,
                      uniform float fresnelPower,
                      uniform float3 etaRatio,
                      uniform float3 eyePositionW,
                      uniform floatx4 modelViewProj,
                      uniform floatx4 modelToWorld)
{
    oPosition = mul (modelViewProj, position);

    //Compute position and normal in world space
    float3 positionW = mul(modelToWorld, position).xyz;
    float3 N = mul (float3x3)modelToWorld, normal);
    N = normalize (N);

    //Compute the incident, reflected, and refracted vectore
    float3 I = positionW-eyePositionW;
    R = reflect(I,N);
    I = normalize (I);
    Tred = refract ( I, N, etaRatio.x );
```

```

Tgreen = refract ( I, N, etaRatio.y );
Tblue = refract ( I, N, etaRatio.z );
//Compute the reflection factor
reflectionFactor = fresnelBias +
                   fresnelScale * pow(1+dot(I,N), fresnelPower);
}

```

一、计算反射向量

反射向量的计算仍然不变：

```
R = reflect ( I, N );
```

二、计算折射向量

你将使用一种类似于你在前面的折射例子中使用的方法来计算折射向量。区别是你现在需要为每种颜色分量计算一个折射向量，而不是将一个向量平均地应用到红光、绿光和蓝光上：

```

Tred = refract ( I, N, etaRatio.x );
Tgreen = refract ( I, N, etaRatio.y );
Tblue = refract ( I, N, etaRatio.z );

```

回想一下在 eatRatio 中的 x、y 和 z 分量分别存储了给红光、绿光和蓝光的折射指数的比率。

三、计算反射系数

把公式 7-3 转化为 Cg 代码是直接了当的。只要使用 dot 和 pow 函数就可以了。程序输出 reflectionFactor 作为一个插值的颜色，并用对应的 COLOR 语义指明。插值的颜色自动地被限制在[0, 1]的范围内，因此不需要明确地执行一次转换操作。

```

reflectionFactor = fresnelBias +
                   fresnelScale * pow (1 + dot (I, N) ),
                   fresnelPower );

```

7.4.5 片段程序

示例 7-6 中的 C7E6f_dispersion 片段程序接收了所有的为反射向量和折射向量插值的数据，以及被映射到[0, 1]的范围的反射系数。片段程序利用不同的反射和折射向量在环境贴图中查找颜色，并把结果适当地混合在一起。注意这个程序预期提供给每个纹理单元（一共使用 4 个纹理单元）的环境贴图是相同的。应用程

序必须把环境贴图绑定到每个纹理单元，因为该程序被编写为要同时运行在基本和高级片段 profile 上的。还记得基本的片段 profile 只能用纹理单元对应的纹理坐标集对一个给定的纹理进行采样吧，因此环境贴图必须被复制。高级的片段 profile 则没有这种限制，因此一个单独的 environmentMap 立方贴图就足够了。

示例 7-6 C7E6f_dispersion 片段程序

```

void C7E6v_dispersion(float reflectionFactor: COLOR,
                      float3 R : TEXCOORD0,
                      out float3 TRed : TEXCOORD1,
                      out float3 TGreen : TEXCOORD2,
                      out float3 TBlue : TEXCOORD3,
                      out float4 color : COLOR,
                      uniform samplerCUBE environmentMap0,
                      uniform samplerCUBE environmentMap1,
                      uniform samplerCUBE environmentMap2,
                      uniform samplerCUBE environmentMap3)
{
    //取得反射的环境颜色
    float4 reflectedColor = texCUBE ( environmentMap0, R);

    //计算折射的环境颜色
    float4 refractedColor;
    refractedColor.r = texCUBE ( environmentMap1, TRed).r;
    refractedColor.g = texCUBE ( environmentMap2, TGreen).g;
    refractedColor.b = texCUBE ( environmentMap3, TBlue).b;
    refractedColor.a = 1
    color = lerp (refractedColor, reflectedColor, reflectionFactor);
}

```

一、执行纹理查找

首先，程序执行 4 次立方贴图查找——一次为反射颜色，然后为 3 个折射颜色每个执行一次：

```

//取得反射的环境颜色
float4 reflectedColor = texCUBE ( environmentMap0, R);

```

```
//计算折射的环境颜色
float4 refractedColor;
refractedColor.r = texCUBE(environmentMap1, TRed).r;
refractedColor.g = texCUBE(environmentMap2, TGreen).g;
refractedColor.b = texCUBE(environmentMap3, TBlue).b;
```

每一次对 3 次折射纹理查找，程序使用重组（swizzling）来只提取相对应的颜色分量。也就是说，你提取在 TRed 采样的纹理值的红色分量，在 TGreen 采样的纹理值的绿色分量和在 TBlue 采样的纹理值的蓝色分量。然后，程序把 refractedColor 的 r、g 和 b 分量组合起来。

二、计算最后的结果

最后，程序将根据给定的反射因素的折射把反射颜色和折射颜色混合在一起：

```
color = lerp(refractedColor, reflectedColor, reflectionFactor);
```

到此为止你实现了带有颜色色散的菲涅耳效果。

7.5 练习

- 回答这个问题：环境映射的关键假设是什么？在什么情况下它将被违背？
- 你自己尝试一下：如果在 C7E5v_dispersion 中的 etaRatio 折射指数向量为 (1, 1, 1)，图 7-10 看起来会是什么样的？
- 你自己尝试一下：尝试实现在 C7E1v_reflection 顶点程序中在物体空间执行反射向量的计算，然后把得到的物体空间的反射向量变换到世界空间。
- 回答这个问题：什么是菲涅耳效果？
- 你自己尝试一下：当 mipmap 被选择的时候，OpenGL 和 Direct3D 都支持一种被称为纹理细节级别（LOD）偏移的纹理映射特性。纹理 LOD 偏移对避免不自然的波纹反射非常有用。修改本章的一个例子来为被用作环境贴图的立方贴图纹理提供一个正偏移。这将产生模糊的反射。
- 回答这个问题：在硬件对立方贴图纹理支持之前，一个被称为球体映射的技术被用来把 3D 向量投影到一个 2D 纹理上。研究这个技术并解释为什么现在大家都使用立方贴图了。

7.6 补充阅读

Jim Blinn 和 Martin Newell 在一篇 1976 年题为 “Texture and Reflection in Computer Generated Images” 的论文中介绍了环境映射，这篇论文被收录在 *Communications of the ACM* 中。

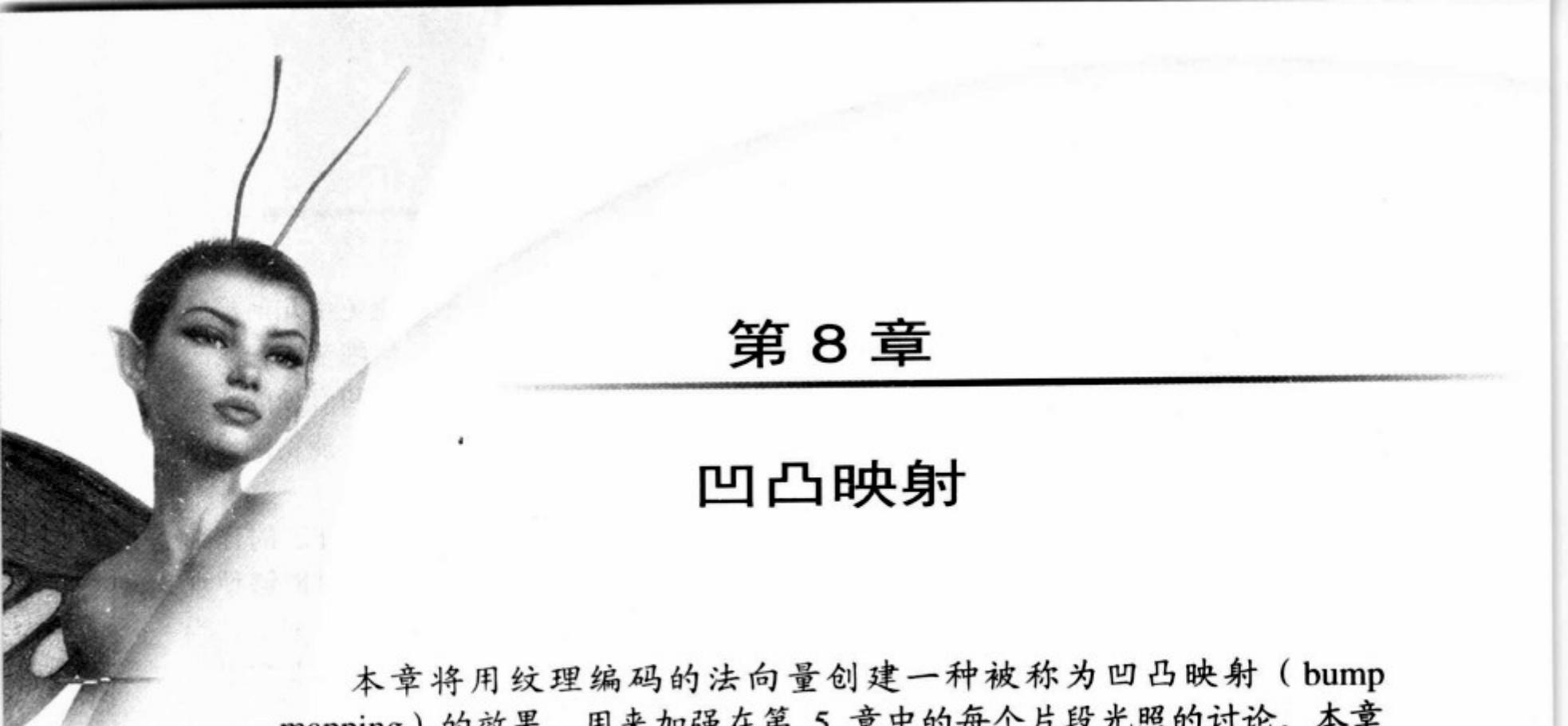
Ned Greene 发表了一篇重要的论文，题为 “Environment Mapping and Other Applications of World Projections”，这篇论文被收录在 1986 年的一期 *IEEE Computer Graphics and Applications* 中。Greene 提出了用立方贴图存储环境贴图的想法。

RenderMan 在它对环境映射的支持中使用了立方贴图纹理。查阅 Steve Upstill 撰写的 *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics* (Addision-Wesley, 1989 年) 可以获得更多的细节。

Doug Voorhies 和 Jim Foran 在 1994 年发表了一篇题为 “Reflection Vector Shading Hardware” (ACM Press) 的 SIGGRAPH 论文。这篇论文为计算用于采样存储在一个立方贴图纹理中的环境贴图的每个片段的反射向量提出了一种专门的硬件方法。

OpenGL1.3 和 DirectX 7 介绍了对立方贴图纹理的硬件支持。OpenGL1.3 或以后的规范说明提供了纹理坐标是如何映射到一个立方体表面上的数学理论。

Matthias Wloka2002 年的论文 “Fresnel Reflection” (可以从 NVIDIA 的开发人员网站上获得，网址是 developer.nvidia.com) 非常详细地讨论了菲涅耳效果。这篇论文解释了各种不同地实现方法和它们之间的比较。



第 8 章

凹凸映射

本章将用纹理编码的法向量创建一种被称为凹凸映射 (bump mapping) 的效果，用来加强在第 5 章中的每个片段光照的讨论。本章包括以下 5 个部分：

- “凹凸映射一个砖墙” 介绍单个矩形的凹凸映射。
- “凹凸映射一个砖的地板” 解释如何使两个平面的凹凸映射保持一致。
- “凹凸映射一个圆环” 描述了如何对凹凸贴图曲面，例如圆环进行数学表示。
- “凹凸映射纹理的多边形网格” 解释如何将凹凸映射应用到纹理多边形模型。
- “把凹凸映射和其他效果结合起来” 将凹凸映射技术与其他纹理，例如，更复杂的阴影印花贴图和光泽贴图结合起来。

8.1 凹凸映射一个砖墙

较早在第 5 章的光照介绍中讨论了每个顶点和每个片段的光照计算。本章将介绍一个更高级的光照方法，通常被称为凹凸映射。凹凸映射把由一个纹理提供的物体表面法向量的扰动与每个片段的光照相结合，来模拟光照与凹凸表面的相互作用。这个效果的取得不需要过多表面的几何镶嵌。

例如，你可以用凹凸映射来使得表面看上去就像有砖从它们表面突出，并且在砖之间有灰泥。

大部分真实世界的表面，例如砖墙或鹅卵石地面都有小规模的凹凸不平的性

质，这些性质太细致以至于无法用高度镶嵌的几何形状来表示。避免用几何形状来表示这种细致的细节有两个原因。第一个原因是，用足够的几何细节来记录表面的凹凸不平的性质的方法来表示模型对交互渲染来说是非常巨大和麻烦的。第二个原因是，表面的特征也许比一个像素还要小，这意味着光栅器不能精确地渲染所有的几何细节。

通过凹凸映射，你能够在一个纹理中记录影响一个物体的光照外表的详细表面特征，而不用增加物体的几何复杂度。做的非常好时，凹凸映射能够使观察者确信，一个凹凸贴图的场景比它实际展现的要有更多几何和表面细节。当灯光相对于表面移动而影响凹凸贴图表面的光照外表时，凹凸映射是非常引人注目的。

凹凸映射的好处包括：

- 在场景中提供一个级别更高的视觉复杂度，而没有增加更多的几何形状。
- 简化了内容创作，因为你可以用纹理来对表面细节进行编码，而不需要美工人员设计高度详细的 3D 模型。
- 应用不同的凹凸贴图到同一个模型的不同实例的能力，给了每个实例一种不同的表面外观。例如，一个建筑物模型能够被用一个砖凹凸贴图渲染一次，而第二次用泥灰凹凸贴图。

8.1.1 砖墙的法向量贴图

考虑一个由变化纹理的砖以一种规则模式堆在一起形成的墙。在砖之间是把砖粘在一起的泥灰。虽然一堵砖墙从远处看也许很平，但在近处观察的时候，砖墙形成的图案一点也不平坦。当墙被照亮的时候，砖之间的缝隙、裂缝和砖表面的其他特征与一个真正平坦的表面对光的散射完全不同。

一种渲染砖墙的方法是为每块砖、泥灰的缝隙或者甚至对墙上的每个裂缝都用多边形来建模，每个多边形都有用于光照的不同的表面法向量。在进行光照的时候，在每个顶点的表面法向量将恰当地改变被照亮表面的外表。但是，这个方法需要极大的数目的多边形。

在一个足够粗糙的比例下，一堵砖墙或多或少地显得比较平坦。除了所有我们提到的表面变化，一堵墙的几何形状是非常简单的。一个单独的矩形就足够表示砖墙的粗糙的矩形平坦部分。

8.1.2 把凹凸贴图存储成法向量贴图纹理

在你遇到第一个用凹凸映射对表面进行光照的 Cg 程序以前，你应该理解凹凸映射的纹理是如何创建和它们是怎样被表示的。

一、传统的颜色纹理

传统的纹理通常包含 RGB 或 RGBA 颜色值，虽然也可以使用其他的格式。你知道，一个 RGB 纹理的每一个纹理元素由三个分量组成，分别用于红色、绿色和蓝色。每个分量通常为一个无符号字节。

因为可编程的图形处理器允许在纹理查找的结果上执行任意的数学操作和其他操作，你可以用纹理来存储其他类型的数据，被当成颜色进行编码。

二、在传统的颜色纹理中存储法向量

凹凸贴图能接受多种形式。在本书中的所有凹凸映射的例子都用表面法向量来表示表面的变化。这种类型的凹凸贴图通常被称为一个法向量贴图（normal map），因为法向量而不是颜色被存储在纹理中。每个法向量是一个从表面向外指的方向向量，并且经常用一个三元向量来存储。

传统的 RGB 的纹理格式通常被用来存储法向量贴图。不像颜色是无符号的，方向向量需要有符号值。除了是无符号以外，在纹理中的颜色值通常被限制在 [0, 1] 的范围内。因为法向量是规范化的向量，每个分量的取值范围为 [-1, 1]。

为了能使针对无符号颜色的纹理过滤硬件正常操作，在 [-1, 1] 范围内的有符号的纹理值需要用一个简单的缩放和偏移，把范围压缩到无符号的 [0, 1] 范围内。

有符号的法向量被用如下方法进行范围压缩：

```
colorComponent = 0.5 * normalComponent + 0.5;
```

在传统的无符号纹理过滤以后，范围压缩的法向量被用如下方法扩展回它们有符号的范围：

```
normalComponent = 2 * (colorComponent - 0.5);
```

通过使用一个 RGB 纹理的红色、绿色和蓝色分量来存储一个法向量的 x、y 和 z 分量，并对有符号的值进行范围压缩到 [0, 1] 的无符号范围，然后法向量就可以被存储在一个 RGB 纹理中了。

最新的图形处理器支持有符号的浮点颜色格式，但是法向量贴图仍然常常被

存储在无符号的颜色纹理中，因为现存的用于无符号颜色的图像文件格式能够存储法向量贴图。最新的图形处理器在从纹理展开被范围压缩的法向量的时候没有性能损失。因此是否用一个范围压缩的形式（使用一个有符号的纹理格式）还是用一个有符号的纹理格式来存储法向量由你决定。

三、从高度域生成法向量贴图

创作法向量贴图提出了另一个问题。在计算机绘图程序中描绘方向向量是非常不直观的。但是，大部分法向量贴图可以从高度域（height field）中得到。一个高度域纹理对每个像素的高度进行编码，而不是对向量进行编码。一个高度域在每个纹理元素存储了一个单独的无符号分量，而不是使用 3 个分量来存储一个向量。图 8-1 显示了一面砖墙的高度域的一个例子（一个彩色版本的法向量贴图可以在本书开始部分的彩图 12 中看到）。高度域的较暗的区域高度较低；较亮的区域高度比较高。纯白色的砖是平滑的。有不均匀颜色的砖是凹凸不平的。泥灰是凹下去的，所以它是高度域中最暗的区域。

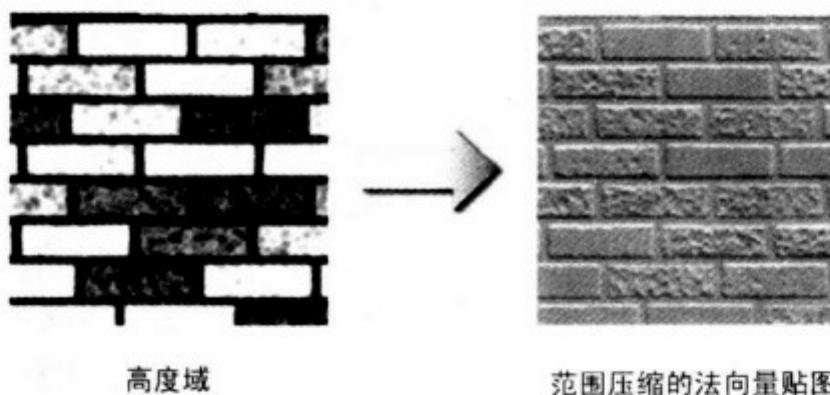


图 8-1 一堵砖墙的凹凸映射的一幅高度域图像

把一个高度域转化为一个法向量贴图是一个完全自动的过程，并且它通常与范围压缩在预处理阶段进行。对在高度域中的每个纹理元素，你需要在给定的纹理元素、以及它正上方和右方的纹理元素对高度进行采样。法向量是两个差分向量的外积的规范化版本。第一个差分向量是 $(1, 0, H_a - H_g)$ ，其中 H_g 是给定纹理元素的高度，而 H_a 是给定纹理元素正上方的纹理元素的高度。第二个差分向量是 $(0, 1, H_r - H_g)$ ，其中 H_r 是给定纹理元素右边的纹理元素的高度。

这两个向量的外积是第三个向量，它指向高度域表面的外部。规范化这个向量可以创建一个适用于凹凸映射的法向量。生成的法向量为：

$$\text{normal} = \frac{\langle H_g - H_a, H_g - H_r, 1 \rangle}{\sqrt{(H_g - H_a)^2 + (H_g - H_r)^2 + 1}}$$

这个法向量是有符号的，因此必须对它进行范围压缩才能够被存储在一个无符号的 RGB 纹理中。虽然还有其他的能够把高度域转化为法向量贴图的方法，但是这个方法已经足够了。

法向量 (0, 0, 1) 是在高度域平坦的区域中计算出来的。这个法向量可以看作是一个从表面指向表面上部的方向向量。在凹凸不平和不均匀的高度域区域里，一个朝上的法向量将适当倾斜。

正如我们已经提到过的，经过范围压缩的法向量贴图通常被存储在一个无符号的 RGB 纹理中，其中红色、绿色和蓝色分别对应 x、y 和 z。由于从高度域到法向量贴图的转化过程自身的特性，z 分量总是正的并且通常为 1 或者几乎为 1。z 分量通常被存储在蓝色分量中，而范围压缩把正的 z 值转化到 [0.5, 1] 的范围。因此，存储在一个 RGB 纹理中的经过范围压缩的法向量贴图最主要的颜色是蓝色。图 8-1 还显示了由砖墙的高度域转化成的一个法向量贴图。因为色彩是很重要的，因此对图 8-1 的彩色版本请你参看本书开始部分的彩图 12。

8.1.3 对一个砖墙的简单凹凸映射

既然你已经知道什么是一个法向量贴图了，你已经为你的第一个凹凸映射示例做好准备了。这个示例将显示如何使用在图 8-1 中的砖墙的法向量贴图来渲染一个凹凸映射的矩形，使它看起来像一堵砖墙。如图 8-2 所示，当你交互地移动一个光源的时候，由于存储在法向量贴图中的砖的模式，你将改变这堵墙的外表。图

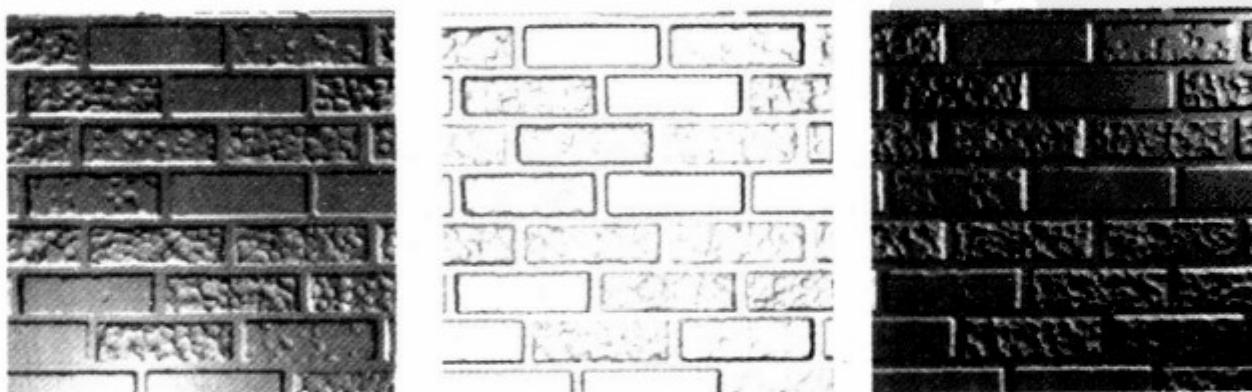


图 8-2 用不同灯的位置渲染的一堵凹凸映射的砖墙

8-2 显示了 3 个不同光源位置的效果。在左边的图像中，灯在墙的左下角。在中间的图像中，灯在墙的正前方。而在右边的图像中，灯在墙的右上角。

为了使事情简单，第一个例子受到了相当的限制。我们把被渲染的墙的矩形放在 x-y 平面，使得墙上的各个部位的 z 值都等于 0。没有凹凸映射，墙的表面法向量在墙上的任意一点将都为 (0, 0, 1)。

一、顶点程序

在示例 8-1 中的 C8E1v_bumpWall 顶点程序计算了从一个顶点到一个光源的物体空间的向量。这个程序还用传统的 modelview-projection 矩阵把顶点位置变换到剪裁空间中，而且它还会传递一个 2D 纹理坐标集来对法向量贴图纹理进行采样。

示例 8-1 C8E1v_bumpWall 顶点程序

```
void C8E1v_bumpWall(float4 position : POSITION,
                      float2 texCoord : TEXCOORD0,

                      out float4 oPosition : POSITION,
                      out float2 oTexCoord : TEXCOORD0,
                      out float3 lightDirection : TEXCOORD1,

                      uniform float3 lightPosition, // Object space
                      uniform float4x4 modelViewProj)
{
    oPosition = mul(modelViewProj, position);
    oTexCoord = texCoord;
    // Compute object-space light direction
    lightDirection = lightPosition - position.xyz;
}
```

二、片段程序

用于漫反射光照的光向量和法向量的点积需要一个单位长度的光向量。在示例 8-2 中的 C8E2f_bumpSurf 片段程序通过使用一个规范化的立方贴图 (normalization cube map) 来规范化经过插值的光的方向向量。规范化的立方贴图将在后面进行解释。现在，你可以把一个规范化的立方贴图看成是一种方法，此方法把一个被插值的、非规范化的向量当作一个纹理坐标集，并用来生成向量的一个规范化的经过范围压缩的版本。因为这个程序用一个立方贴图纹理存

取实现了规范化，这种方式的每个片段的向量规范化很快，非常适用于范围很广的多种图形处理器。

除了规范化经过插值的光向量，C8E2f_bumpSurf 程序用传统的 2D 纹理坐标集对法向量贴图进行采样。法向量贴图存取的结果是另一个范围压缩过的法向量。

其次，程序的名为 expand 的帮助函数把经过范围压缩的规范化的灯光方向和经过范围压缩的法向量转化为一个有符号的向量。然后，这个程序用一个点积模拟漫反射光照来计算最后的片段颜色。

示例 8-2 举例说明了砖墙的外表是如何随着不同的灯光位置变化的。用 C8E1v_bumpWall 和 C8E2f_bumpSurf 程序渲染的墙的表面看起来就像它真的有一个砖墙的纹理。

示例 8-2 C8E2f_bumpSurf 片段程序

```
float3 expand(float3 v)
{
    return (v - 0.5) * 2; // Expand a range-compressed vector
}

void C8E2f_bumpSurf(float2 normalMapTexCoord : TEXCOORD0,
                     float3 lightDir : TEXCOORD1,
                     out float4 color : COLOR,
                     uniform sampler2D normalMap,
                     uniform samplerCUBE normalizeCube)
{
    // Normalizes light vector with normalization cube map
    float3 lightTex = texCUBE(normalizeCube, lightDir).xyz;
    float3 light = expand(lightTex);
    // Sample and expand the normal map texture
    float3 normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
    float3 normal = expand(normalTex);
    // Compute diffuse lighting
    color = dot(normal, light);
}
```

三、构造一个规范化的立方贴图

第 7 章解释了如何使用立方贴图来对环境贴图进行编码，作为一种给物体一

个反光外表的方法。为了模拟表面反光，被用来存取环境贴图的 3D 纹理坐标向量代表了一个反射向量。但是立方贴图同样还能够对其他的方向向量的函数进行编码。向量规范化就是这样的函数之一。

Cg 标准函数库包含了一个被命名为 `normalize` 的函数来进行向量规范化。这个函数有几个重载的变体，但其中三元向量的版本是最常使用的。`normalize` 的标准实现是这样的：

```
float3 normalize (float3 v)
{
    float d = dot (v, v);           // x*x + y*y + z*z
    return d / sqrt(d);
}
```

这个 `normalize` 实现的问题是，由第二代和第三代图形处理器提供的基本的片段 profile，不能直接编译我们所展示的这个 `normalize` 函数。这是因为这些图形处理器在片段级别缺乏任意的浮点数学操作。

规范化立方贴图——一种快速规范化作为纹理坐标提供的向量的方法——在所有图形处理器上都能正常工作，无论它们是否支持高级片段的编程能力。



即使是在支持高级片段 profile 的图形处理器上，使用规范化立方贴图通常也比用数学操作实现规范化要快，因为图形处理器的设计者高度优化了纹理存取。

图 8-3 显示了一个立方贴图是如何规范化一个向量的。向量 $(3, 1.5, 0.9)$ 如图所示穿过了立方贴图的正 x 轴表面。规范化的立方贴图被构造成使得被任何给

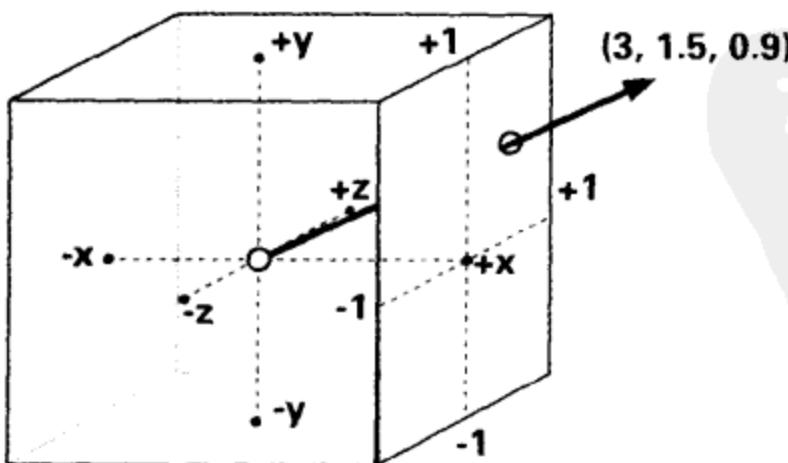


图 8-3 使用立方贴图来规范化向量

定方向向量穿过的纹理元素包含那个向量的规范化版本。当有符号的纹理分量不能被使用的时候，向量的规范化版本可以经过范围压缩再存储，然后在作为一个规范化向量使用之前要进行展开。这是在本章中的例子所假定的。因此被(3, 1.5, 0.9)穿过的范围压缩过的纹理元素大约是(0.93, 0.72, 0.63)。当这个向量被展开后，为(0.86, 0.43, 0.26)，这个向量是(3, 1.5, 0.9)近似的规范化版本。

32×32 纹理元素的分辨率对一个使用 8 位颜色分量的一个规范化立方贴图表面已经足够了。一个 16×16 或者甚至是 8×8 的分辨率也能生成可接受的结果。

8.1.4 带镜面反射的凹凸映射

你能够通过增加镜面反射和环境光照项以及增加对漫反射材质、镜面反射材质和光照颜色的控制来进一步增强前面给出的程序。下面的一对程序展示了这些。

一、顶点程序

示例 8-3 扩展了前面的 C8E1v_bumpWall 示例来计算半角向量(half-angle vector)，它被光栅器作为一个额外的纹理坐标集进行插值。C8E3v_specWall 程序通过规范化顶点的规范化的光和眼睛向量的和来计算半角向量。

示例 8-3 C8E3v_specWall 顶点程序

```
void C8E3v_specWall(float4 position : POSITION,
                      float2 texCoord : TEXCOORD0,

                      out float4 oPosition : POSITION,
                      out float2 oTexCoord : TEXCOORD0,
                      out float3 lightDirection : TEXCOORD1,
                      out float3 halfAngle : TEXCOORD2,

                      uniform float3 lightPosition, // Object space
                      uniform float3 eyePosition, // Object space
                      uniform float4x4 modelViewProj)
{
    oPosition = mul(modelViewProj, position);
    oTexCoord = texCoord;
    lightDirection = lightPosition - position.xyz;
    // Add the computation of a per-vertex half-angle vector
    float3 eyeDirection = eyePosition - position.xyz;
```

```

    halfAngle = normalize(normalize(lightDirection) +
                           normalize(eyeDirection));
}

```

二、片段程序

在示例 8-4 中所示的对应的片段程序需要更多的修改。除了像以前一样用一个规范化的立方贴图来规范化光向量，更新的片段程序还用第二个规范化立方贴图来规范化半角向量。然后，程序计算规范化的半角向量与从法向量贴图获得的被扰动过的法向量的点积。

在最初的 C8E2f_bumpSurf 程序中，程序输出漫反射点积作为最后的颜色。最后颜色的 COLOR 输出语义隐含了把点积结果的负轴映射为 0。这个映射是漫反射所需要的，因为负的光照在物理上是不可能的。C8E4f_specSurf 程序结合了漫反射和镜面反射点积，因此程序需要明确地把负值用 saturate 标准库函数映射为 0：

saturate (x)	映射一个标量或一个向量的全部分量到[0, 1]的范围。
----------------	-----------------------------

基本的片段 profile 例如 fp20 和 ps_1_3 缺乏对真正的指数的支持。为了模拟镜面反射的指数和支持很多范围的片段 profile，这个程序使用连续三次乘法来把镜面反射的点积提升到 8 次方。高级 profile 则能够使用 pow 和 lit 标准库函数。

最后，输出的颜色是通过用一个统一参数 LMd 来调整环境光照和漫反射光照项来计算的。应用程序提供的 LMd 代表了用材质的漫反射颜色预乘过的光的颜色。类似地，LMs 统一参数被用来调整镜面反射光照，它代表了材质的镜面反射颜色预乘了光的颜色。

示例 8-4 C8E4f_specSurf 片段程序

```

float3 expand(float3 v) { return (v - 0.5) * 2; }

void C8E4f_specSurf(float2 normalMapTexCoord : TEXCOORD0,
                      float3 lightDirection : TEXCOORD1,
                      float3 halfAngle : TEXCOORD2,

                      out float4 color : COLOR,

                      uniform float ambient,

```

```

uniform float4 LMd, // Light-material diffuse
uniform float4 LMs, // Light-material specular
uniform sampler2D normalMap,
uniform samplerCUBE normalizeCube,
uniform samplerCUBE normalizeCube2)

{
    // Fetch and expand range-compressed normal
    float3 normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
    float3 normal = expand(normalTex);
    // Fetch and expand normalized light vector
    float3 normLightDirTex = texCUBE(normalizeCube,
                                      lightDirection).xyz;
    float3 normLightDir = expand(normLightDirTex);
    // Fetch and expand normalized half-angle vector
    float3 normHalfAngleTex = texCUBE(normalizeCube2,
                                      halfAngle).xyz;
    float3 normHalfAngle = expand(normHalfAngleTex);

    // Compute diffuse and specular lighting dot products
    float diffuse = saturate(dot(normal, normLightDir));
    float specular = saturate(dot(normal, normHalfAngle));
    // Successive multiplies to raise specular to 8th power
    float specular2 = specular * specular;
    float specular4 = specular2 * specular2;
    float specular8 = specular4 * specular4;

    color = LMd * (ambient + diffuse) + LMs * specular8;
}

```

Advanced 进一步的改进

C8E4f_specSurf 程序能够在基本和高级片段 profile 上编译。虽然这使得凹凸映射效果能够移植到很多图形处理器上，但是如果你以高级片段为目标很多改进都是可能的。下面是一些例子。

C8E4f_specSurf 把同一个规范化立方贴图纹理绑定到两个纹理单元。正如你在第 7 章所看到的，这个重复的绑定是需要的，因为基本的片段 profile 只能对一个给定的纹理单元用这个纹理单元对应的纹理坐标集进行采样。而高级的片段 profile 则没有这种限制，因此，一个单独的 normalizeCube 立方贴图样本就能够规范化光

向量和半角向量。

C8E4f_specSurf 还通过使用连续的乘法把镜面反射点积提升到 8 次方来计算镜面反射的幂，因为基本的片段 profile 不支持任意的求幂运算。一个高级的 profile 版本能够使用下面的代码：

```
color = Kd * ( ambient + diffuse ) +
        Ks * pow ( specular, specularExponent );
```

其中 `specularExponent` 是一个统一参数，或者甚至是一个来自一个纹理的值。

C8E3v_specWall 计算了一个每个顶点的半角向量。为了完美，你应该在每个片段从光向量和观察向量计算半角向量。你能够修改顶点程序输出 `eyeDirection` 值来代替半角向量。然后，你就能够修改 C8E4f_specSurf 在每个片段计算半角向量，如下所示：

```
// 提取和展开规范化的眼向量
float3 normEyeDirTex = texCUBE ( normalizeCube, eyeDirection).xyz;
float3 normEyeDir = expand(normEyeDirTex);
// 求光向量和眼向量的和，并用规范化的立方贴图来进行规范化
float3 normHalfAngle = texCUBE ( normalizeCube,
                                    normLightDir + normEyeDir);
normHalfAngle = expand (normHalfAngle);
```

正如在第 5 章中解释的，在每个片段计算半角向量生成的反射强光要比在每个顶点计算半角向量看上去更加真实，但是花费也更加多。

8.1.5 凹凸映射其他几何图形

你已经学会了如何凹凸映射一堵砖墙，并且在图 8-2 中所显示的结果相当成功。但是，凹凸映射并不像你所认为的这些最初的例子那样简单。

在图 8-2 中渲染的墙的矩形碰巧是平坦的，并且有一个均匀的表面法向量 (0, 0, 1)。另外，通过一个均匀的线性映射赋给矩形的纹理坐标集是与顶点的位置相关的。在墙的矩形上的每个点上，`s` 纹理坐标与 `x` 位置只相差一个正的比例因子。这对 `t` 纹理坐标和 `y` 位置同样成立。

在这些具有相当多限制的环境下，你能够直接用从法向量贴图采样的法向量代替表面法向量。这正是前面的例子所实现的，而且凹凸映射的结果看上去不错。

但是，当你用 C8E1v_bumpWall 和 C8E2f_bumpSurf 程序凹凸映射到任意的几何形状上的时候，会发生什么呢？如果几何形状的表面法向量不是均匀的 (0, 0, 1)

会怎么样呢？如果被用来存取法向量贴图的 s 和 t 纹理坐标不是线性相关与几何物体的 x 和 y 物体空间位置的时候又会怎么样呢？

在这些情况下，你渲染的结果也许会很像正确的凹凸映射，但是近处观察就会发现，在场景中的光照与实际的光与眼睛的位置并不一致。产生这种结果的原因是，用来在每个片段进行凹凸映射的物体空间的光向量和半角向量不再和法向量贴图中的法向量共享一个一致的坐标系统。因此光照的结果显而易见是错的。

一、物体空间的凹凸映射

一种解决方案是确保存储在法向量贴图中的法向量被正确地定向，使得它们可以代替被渲染的几何物体在物体空间中的表面法向量。这意味着法向量贴图将有效地存储物体空间的法向量，这种方法被称为物体空间的凹凸映射 (*object-space bump mapping*)。这种方法确实有效，这就是为什么你较早的凹凸映射的墙的示例（图 8-2）是正确的原因，虽然只是针对特定的矩形墙。

不幸地是，物体空间的凹凸映射把你的法向量贴图纹理与你要进行凹凸映射的特定的几何物体捆绑在一起了。创建一个法向量贴图需要知道你将要使用法向量贴图的物体的确切的几何形状。这意味着你不能使用同一个砖模式的法向量贴图纹理来凹凸映射场景中所有不同的砖墙。相反，最后你需要为你渲染的每个不同的砖墙创建一个不同的法向量贴图。如果物体运动它的物体空间的顶点的位置，每个不同的姿态潜在地需要它自己的物体空间法向量贴图。由于这些原因，物体空间的凹凸映射是非常受限制的。

二、纹理空间的凹凸映射

正确的凹凸映射需要光的向量和半角向量与在法向量贴图中存储的法向量共享一个一致的坐标系统。你选择什么坐标系统并没有关系，只要你为在光照公式中的所有向量选择一致的坐标系统。物体空间是一个一致的坐标系统，但是它并不是惟一的选择。

你不需要使在法向量贴图中的法向量与要被凹凸映射的物体的物体空间坐标系统相配合。相反，你能够旋转物体空间的光向量和半角向量到法向量贴图的坐标系统。旋转两个方向向量到另一个可选择的坐标系统比在一个法向量贴图中调整每个法向量的工作量要少很多。被法向量贴图纹理所使用的坐标系统被称为纹理空间 (*texture space*)，因此这个方法也被称为纹理空间的凹凸映射 (*texture-space bump mapping*)，有时候也被称为切空间的凹凸映射 (*tangent-space bump mapping*)。

顶点程序对把一个向量从一个坐标系统变换到另一个系统是非常有效的。纹理空间凹凸映射所需要的向量变换类似于用 `modelview-projection` 矩阵把位置从物体空间变换到剪裁空间。

你能够对你的图形处理器进行编程，来把每个物体空间的光向量和半角向量变换到与你法向量贴图纹理相匹配的坐标系统。

但是，对一个给定的被渲染的物体 `modelview-projection` 矩阵是固定的。相反，旋转物体空间的光向量和半角向量到一个与你的法向量贴图匹配的坐标系统的变换通常在你渲染的表面上变化。你渲染的每一个顶点都需要一个不同的旋转矩阵。

虽然纹理空间的凹凸映射对每个顶点都需要一个不同的旋转矩阵，但是它有一个主要的优点。它允许你应用同一个法向量贴图纹理到多个模型——或到一个被驱动的模型——仍然能够保持凹凸映射所需要的每个片段的数学运算简单和有效，使得它能够在只支持基本片段 profile 的图形处理器上运行。

8.2 凹凸映射一个砖铺的地板

在我们考虑凹凸映射多边形网格以前，先考虑一种稍微复杂一点的情况。我们考虑渲染一个有同样砖的纹理的矩形来代替渲染凹凸映射的物体表面法向量为 $(0, 0, 1)$ 的一堵砖墙，但这个矩形被重新放置使得它变成一个砖铺的地板而不再是一堵墙。这个地板的表面法向量是 $(0, 1, 0)$ ，在 y 方向正直向上。

在这个地板例子中，应用与你在上一个例子中应用在墙上相同的砖法向量贴图到地板上。但是，“正真向上”在法向量贴图是 $(0, 0, 1)$ 向量，而“正真向上”对物体空间中的地板则是 $(0, 1, 0)$ 。这两个坐标系统是不一致的。

怎么样才能使这两个坐标系统一致呢？地板在每个点有相同的法向量，因此下面的旋转矩阵把地板的物体空间的“正真向上”的向量变换到法向量贴图的对应的“正真向上”的向量：

$$[0 \ 0 \ 1] = [0 \ 1 \ 0] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

第 8.3 和 8.4 小节将解释如何为任意的纹理映射的矩形和三角形创建这个 3×3 矩阵的细节。而目前，最重要的事情是存在这样一个矩阵，并提供了一种把向量

从物体空间变换到法向量贴图的纹理空间中的方法。

我们能够使用这个矩阵来为地板矩形旋转物体空间的光向量和半角向量，使得它们可以和法向量贴图的坐标系统相匹配。例如，如果 L 是物体空间中的光向量（被写成一个行向量），则在法向量贴图坐标系统中的 L' 能够按如下方式计算：

$$L' = \begin{bmatrix} L'_x & L'_y & L'_z \end{bmatrix} = [L_x \quad -L_z \quad L_y] = [L_x \quad L_y \quad L_z] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

要执行特定的纹理空间凹凸映射，你还必须使用同样的方法旋转在物体空间中的半角向量到纹理空间。虽然这个例子的旋转矩阵是微不足道的，但同样的原理可以应用到任意一个旋转矩阵。

关于旋转矩阵

你通常能够用一个 3×3 矩阵来表示一个 3D。一个旋转矩阵的每一行和每一列必须是一个单位长度的向量。此外，每个列向量与其他两个列向量彼此正交，这个定律对行向量也是正确的。被一个旋转矩阵变换的向量的长度在变换之后是不变的。一个 3D 旋转矩阵能够在位于两个 3D 坐标系统中的方向向量之间起到桥梁作用。

被用来把一个物体空间的方向向量变换到一个法向量贴图的纹理空间中的一个旋转矩阵的列向量被分别命名为 tangent (T)、binormal (B) 和 normal (N)。因此如公式 8-1 所示，旋转矩阵的每个元素能够被标示为：

公式 8-1 由 Tangent、Binormal 和 Normal 列向量组成的一个旋转矩阵

$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

给定一个旋转矩阵的两个列向量（或行向量），第三个列向量（或行向量）是两个已知列向量（或行向量）的外积。对于列向量，这意味着在公式 8-2 中的关系存在。

公式 8-2 在 Tangent、Binormal 和 Normal 向量之间的外积关系

$$B = N \times T$$

$$N = T \times B$$

$$T = B \times N$$

渲染一个砖铺的地板的顶点程序

你能够增强 C8E1v_bumpWall 示例使它能够使用纹理空间的凹凸映射来进行凹凸贴图。要实现这点，需要传递把物体空间向量变换到纹理空间向量所需要的旋转矩阵的 tangent 和 normal 向量。

示例 8-5 的 C8E5v_bumpAny 顶点程序与前面的 C8E2f_bumpSurf 片段程序结合在一起，能够用相同的法向量贴图纹理对砖墙和砖铺的地板进行凹凸贴图。但是要做到这点，你必须提供正确的把物体空间映射到纹理空间的旋转矩阵的 normal 向量和 tangent 向量。你必须为每个顶点指定这两个向量。程序会用外积来计算 binormal 向量。而不需要 binormal 作为另一个每个顶点的参数被传递，使用程序计算 binormal 是为了减少图形处理器为被处理的每个顶点所必须读取动态数据。动态计算 binormal 还能避免预先计算和投入内存空间来存储 binormal 向量。

纹理空间凹凸映射也被称为切空间凹凸映射，因为表面的一个切向量与表面的法向量一起就可以建立所需要的旋转矩阵。

图 8-4 比较了使用同样凹凸贴图的墙和地板的布局、同样法向量贴图纹理、同样的灯光位置和同样的 C8E2f_bumpSurf 片段程序的简单场景的两个图像。但每个图像使用了一个不同的顶点程序。在左边图像中的光照是一致和正确的，因为它使用了带有正确的纹理空间映射所需要的物体空间到纹理空间旋转的 C8E5v_bumpAny 顶点程序。但是，在右边图像中的光照是不一致的。在墙上的光照是正确的，但是在地面上的光照是错误的。不一致的光照的产生是因为，在右边的图像对墙和地板都使用了 C8E1v_bumpWall 顶点程序。

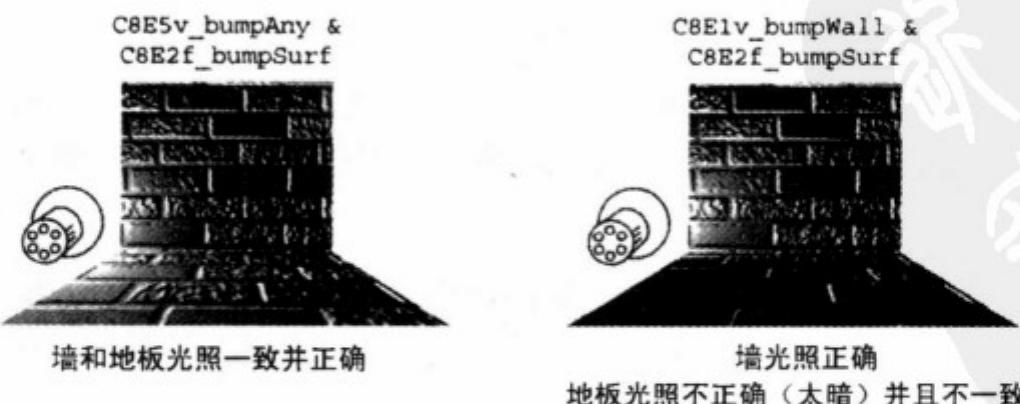


图 8-4 一致的纹理空间凹凸映射与不一致的物体空间凹凸映射的比较

示例 8-5 C8E5v_bumpAny 顶点程序

```

void C8E5v_bumpAny(float3 position : POSITION,
                    float3 normal : NORMAL,
                    float3 tangent : TEXCOORD1,
                    float2 texCoord : TEXCOORD0,

                    out float4 oPosition : POSITION,
                    out float2 normalMapCoord : TEXCOORD0,
                    out float3 lightDirection : TEXCOORD1,

                    uniform float3 lightPosition, // Object space
                    uniform float3 eyePosition, // Object space
                    uniform float4x4 modelViewProj)
{
    oPosition = mul(modelViewProj, float4(position, 1));

    // Compute the object-space light vector
    lightDirection = lightPosition - position;

    // Construct object-space-to-texture-space 3x3 matrix
    float3 binormal = cross(tangent, normal);
    float3x3 rotation = float3x3(tangent,
                                binormal,
                                normal);

    // Rotate the light vector using the matrix
    lightDirection = mul(rotation, lightDirection);

    normalMapCoord = texCoord;
}

```

按照惯例，我们把位置向量写成列向量而方向向量写成行向量。使用公式 8-2，C8E5v_bumpAny 用每个顶点的切向量和法向量的外积来计算 binromal：

```
float3 binormal = cross ( tangent, normal );
```

用于计算两个向量外积的 cross 函数是 Cg 标准函数库的一部分。
然后程序用一个 **float3x3** 矩阵构造函数来构造一个旋转矩阵：

```
float3x3 rotation = float3x3 ( tangent,
```

```
binormal,
normal);
```

构造的 rotation 矩阵的行向量是 tangent、binormal 和 normal，因此这个构造的矩阵是公式 8-1 所示的矩阵的转置。用一个矩阵乘以一个行向量与用矩阵的转置乘以一个列向量是相同的。C8E5v_bumpAny 示例的乘法是一个矩阵乘以向量的乘法，因为 rotation 矩阵是真正需要矩阵的转置，如下所示：

```
lightDirection = mul ( rotation, lightDirection );
```

用同样的方法增强 C8E3v_specWall 程序同样需要旋转半角向量：如下所示：

```
eyeDirection = mul ( rotation, eyeDirection );
halfAngle = normalize (normalize(lightDirection) +
normalize(eyeDirection) );
```

在图 8-4 中的场景只有平坦的表面。这意味着墙所需要的旋转矩阵和地板所需要的旋转矩阵在每个平坦的表面上是统一的。C8E5v_bumpAny 程序允许每个顶点有不同的 tangent 向量和 normal 向量。一个曲面或多边形的网格，需要对在每个顶点定义的从物体空间到纹理空间旋转的不同的 tangent 向量和 normal 向量的支持。图 8-5 显示了一个弯曲的三角形形状在每个顶点需要不同的 normal、tangent 和 binormal 向量。这些向量在每个顶点定义了不同的旋转矩阵来正确地旋转光向量到纹理空间。在图中，光向量是用灰色显示的。

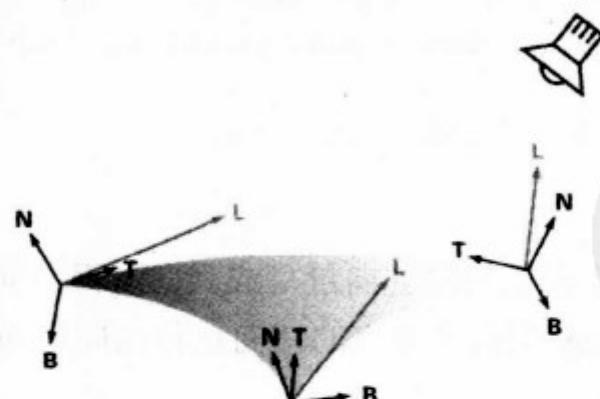


图 8-5 每个顶点的纹理空间基

下面的两个小节将解释如何把纹理空间凹凸映射推广到支持曲面和多边形网格。

8.3 凹凸映射一个圆环

Advanced

本节和下一节是给那些想更多地从数学原理上理解纹理空间的读者的。特别是，这两节解释了如何构造把物体空间向量变换到纹理空间的旋转矩阵的数学原理。如果你满足于依赖 3D 著作工具或其他的软件来为纹理空间的凹凸映射生成旋转矩阵，这些论题是不需要的。如果你对这个层次的细节不感兴趣，我们鼓励你直接跳到第 8.5 节。

在本节，我们将描述如何凹凸贴图一个如图 8-6 所示的被镶嵌的圆环。凹凸映射一个圆环比凹凸映射一堵砖墙要涉及更多的东西，因为一个圆环的表面是弯曲的。曲率意味着整个圆环的从物体空间到纹理空间的旋转不是固定的。



图 8-6 一个被镶嵌的圆环

8.3.1 圆环的数学表示

对凹凸映射，圆环提供了一个非常规则的表面，在你把这些想法应用到第 8.4 节中的更普通的任意多边形模型之前，可以用来开发你的数学直觉。

凹凸映射一个圆环比凹凸映射一个任意的多边形模型更加简单，因为在公式 8-3 中显示的一组参数化数学公式定义了一个圆环的表面。

公式 8-3 一个圆环的参数化公式

$$\begin{aligned}x &= (M + N \cos(2\pi t)) \cos(2\pi s) \\y &= (M + N \cos(2\pi t)) \sin(2\pi s)\end{aligned}$$

$$z = N \sin(2\pi t)$$

参数变量 $(s, t) \in [0, 1]$ 被映射到圆环上的 3D 位置 (x, y, z) , 其中 M 是从洞的中点到圆环管的中线的半径, 而 N 是圆环管的半径。圆环位于 $z=0$ 的平面并且它的中心被放置在原点。

通过用一组参数化公式定义一个圆环的表面, 你能够使用偏微分公式分析地决定一个圆环的准确的曲率。

在公式 8-3 中的圆环的分析定义, 使你能够根据用来定义圆环的参数变量 (s, t) 来决定一个确定方向的表面局部坐标系统, 即我们为凹凸映射寻找的纹理空间。这些参数变量还被用作纹理坐标集来为凹凸映射存取一个法向量贴图。

实际上, 这提供了一个把物体空间的光向量和物体空间中的视向量转化到一个表面局部坐标系统的方法, 这个表面局部坐标系统的定向是与存储在一个法向量贴图纹理中的法向量一致的。一旦你有了一组在这个一致坐标系统中的法向量、光向量和视向量, 在第 5 章中的光照公式就能够正常工作了。正如在第 8.1.5 小节中讨论的, 凹凸映射的技巧是找到一个一致的坐标系统并把光照所需要的所有向量正确的变换到这个空间。

如果我们假设一个表面是合理镶嵌的, 那么我们只需要在每个顶点计算被光照所需要的光向量和视向量, 然后为每个光栅化的片段值用于计算光照公式的这些向量进行插值。这个假设对一个均匀镶嵌的圆环工作得很好。

正如对平坦的砖墙所做的那样, 我们需要寻找一个用 3×3 矩阵形式表示的旋转矩阵, 使我们能够用来把物体空间的向量转换到根据圆环的 (s, t) 参数定向的一个表面局部坐标系统中。因为圆环是弯曲的, 所以圆环的每个顶点的 3×3 矩阵都不相同。

构造这个旋转矩阵需要定义圆环的参数方程的偏导数。这些偏导数公式被显示在公式 8-4 中:

公式 8-4 圆环的参数化方程的偏导数

$$\begin{aligned} \frac{\partial x}{\partial s} &= -2\pi(M + N \cos(2\pi t) \sin(2\pi s)) & \frac{\partial x}{\partial t} &= -2N\pi \sin(2\pi t) \cos(2\pi s) \\ \frac{\partial y}{\partial s} &= 2\pi(M + N \cos(2\pi t) \cos(2\pi s)) & \frac{\partial y}{\partial t} &= -2N\pi \cos(2\pi t) \sin(2\pi s) \\ \frac{\partial z}{\partial s} &= 0 & \frac{\partial z}{\partial t} &= 2N\pi \cos(2\pi t) \end{aligned}$$

我们把用 s 或 t 表示的偏导数组成的三元向量称为反向梯度(inverse gradient)，因为它很像一个传统梯度的每个分量的倒数。反向梯度用一个参数变量表明了在表面位置上的瞬时方向和量值的变化。

你能够用这些反向梯度来定义一个表面局部坐标系统。组成一个 3D 坐标系统只需要用两个正交的向量。对任何一个表面局部坐标系统都必需的一个向量是表面法向量。根据定义，表面的法向量从表面指向外部。你能够通过计算两个不重合的反向梯度的外积，为在表面上的每个点构造表面法向量。在我们圆环的例子中，表面法向量 N 由公式 8-5 给出。

公式 8-5 用表面的参数化反向梯度表示表面的法向量

$$N = \left\langle \frac{\partial x}{\partial s}, \frac{\partial y}{\partial s}, \frac{\partial z}{\partial s} \right\rangle \times \left\langle \frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, \frac{\partial z}{\partial t} \right\rangle$$

通过选择用 s 表示的反向梯度作为你的 tangent 向量，和表面法向量一起，你就能够构造一个表面局部坐标系统。

对任意一个圆环的顶点，把物体空间向量变换到纹理坐标系统所需要的旋转矩阵是

$$\begin{bmatrix} \hat{T}_x & \hat{B}_x & \hat{N}_x \\ \hat{T}_y & \hat{B}_y & \hat{N}_y \\ \hat{T}_z & \hat{B}_z & \hat{N}_z \end{bmatrix}$$

其中 \hat{V} 符号表示一个规范化的向量，而计算公式则在公式 8-6 中显示：

公式 8-6 在一个表面上的 Tangent、Binormal 和 Normal 向量

$$T = \left\langle \frac{\partial x}{\partial s}, \frac{\partial y}{\partial s}, \frac{\partial z}{\partial s} \right\rangle$$

$$B = N \times T$$

$$N = \left\langle \frac{\partial x}{\partial s}, \frac{\partial y}{\partial s}, \frac{\partial z}{\partial s} \right\rangle \times \left\langle \frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, \frac{\partial z}{\partial t} \right\rangle$$

你全部使用规范化的向量来组成这个旋转矩阵。这意味着你可以忽略在公式 8-4 中分别用 s 和 t 表示的反向梯度的固定缩放因子 2π 和 $2N\pi$ 。

然后，基于公式 8-5 的圆环的规范化的表面法向量可以简化为：

$$N = \langle \cos(s)\cos(t), \sin(s)\cos(t), \sin(t) \rangle$$

8.3.2 凹凸映射的圆环的顶点程序

示例 8-6 显示了用于渲染一个凹凸映射的圆环的顶点程序 C8E6v_torus。这个程序能够程序化地从一个在 $(s, t) \in [0, 1]$ 上指定的 2D 网格生成一个圆环，如图 8-7 所示。除了生成圆环以外，如第 8.3.1 小节所描述的这个程序还构造了正确的每个顶点的旋转矩阵。为了进行一致的纹理空间凹凸映射，这个程序还把统一的物体空间的光向量和半角向量参数旋转到纹理空间。

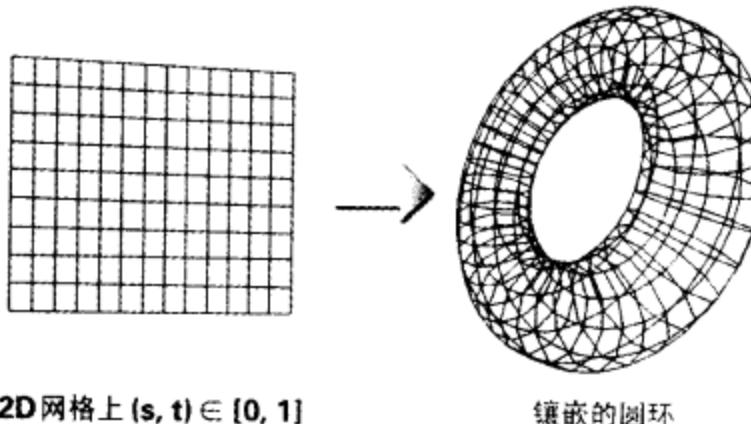


图 8-7 从一个 2D 网格程序化生成的圆环

示例 8-6 C8E6v_torus 顶点程序

```
void C8E6v_torus(float2 parametric : POSITION,
                  out float4 position      : POSITION,
                  out float2 oTexCoord     : TEXCOORD0,
                  out float3 lightDirection : TEXCOORD1,
                  out float3 halfAngle     : TEXCOORD2,
                  uniform float3 lightPosition,
                  uniform float3 eyePosition,
                  uniform float4x4 modelViewProj,
                  uniform float2 torusInfo)
{
    const float pi2 = 6.28318530; // 2 times pi
    // Stretch texture coordinates counterclockwise
    // over torus to repeat normal map in 6 by 2 pattern
    float M = torusInfo[0];
    float N = torusInfo[1];
}
```

```

oTexCoord = parametric * float2(-6, 2);
// Compute torus position from its parametric equation
float cosS, sinS;
sincos(pi2 * parametric.x, sinS, cosS);
float cost, sint;
sincos(pi2 * parametric.y, sint, cost);
float3 torusPosition = float3((M + N * cost) * cosS,
                               (M + N * cost) * sinS,
                               N * sint);
position = mul(modelViewProj, float4(torusPosition, 1));
// Compute per-vertex rotation matrix
float3 dPds = float3(-sinS * (M + N * cost), cosS *
                      (M + N * cost), 0);
float3 norm_dPds = normalize(dPds);
float3 normal = float3(cosS * cost, sinS * cost, sint);
float3 dPdt = cross(normal, norm_dPds);
float3x3 rotation = float3x3(norm_dPds,
                               dPdt,
                               normal);

// Rotate object-space vectors to texture space
float3 eyeDirection = eyePosition - torusPosition;
lightDirection = lightPosition - torusPosition;
lightDirection = mul(rotation, lightDirection);
eyeDirection = mul(rotation, eyeDirection);
halfAngle = normalize(normalize(lightDirection) +
                      normalize(eyeDirection));
}

}

```

图 8-8 显示了两个使用 C8E6v_torus 顶点程序和 C8E4f_specSurf 片段程序渲染的凹凸映射的圆环。这个例子应用了图 8-1 中的砖的法向量贴图。砖一致地向外突出，并且在每块砖上都有一个可见的反射强光。

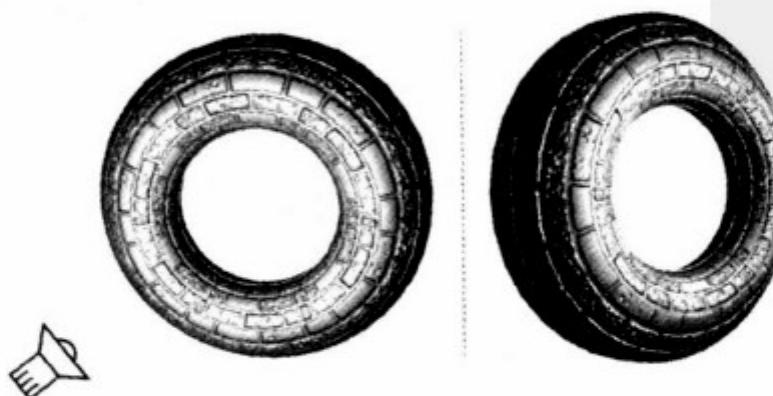


图 8-8 用 C8E6v_torus 和 C8E4f_specSurf 渲染的两个凹凸映射的圆环

8.4 凹凸映射纹理的多边形网格

现在我们考虑更普通的一个贴图的多边形模型的情况，例如在 3D 计算机游戏中为人物和物体使用的那种模型。一般而言，物体并不是平坦的像一堵墙那样的，也不是很容易地就能方便的用数学表达式描述的，就像圆环那样。相反，一个美工人员用一个贴图的三角形网格来模拟这样的物体。

我们的方法是解释如何凹凸映射贴图的多边形网格中的一个三角形，然后把这个方法推广到整个网格。

8.4.1 考察单独一个三角形

图 8-9 显示了一个异形头部的线框模型，和一个用来构造头部的法向量贴图的高度域纹理。这幅图像显示了同一个三角形 3 次。在左边的三角形版本位于 2D 的高度域纹理中。右边的三角形版本和其他组成头部的三角形被显示在 3D 的物体空间中。中间的三角形版本出现在用凹凸映射渲染的头部上。

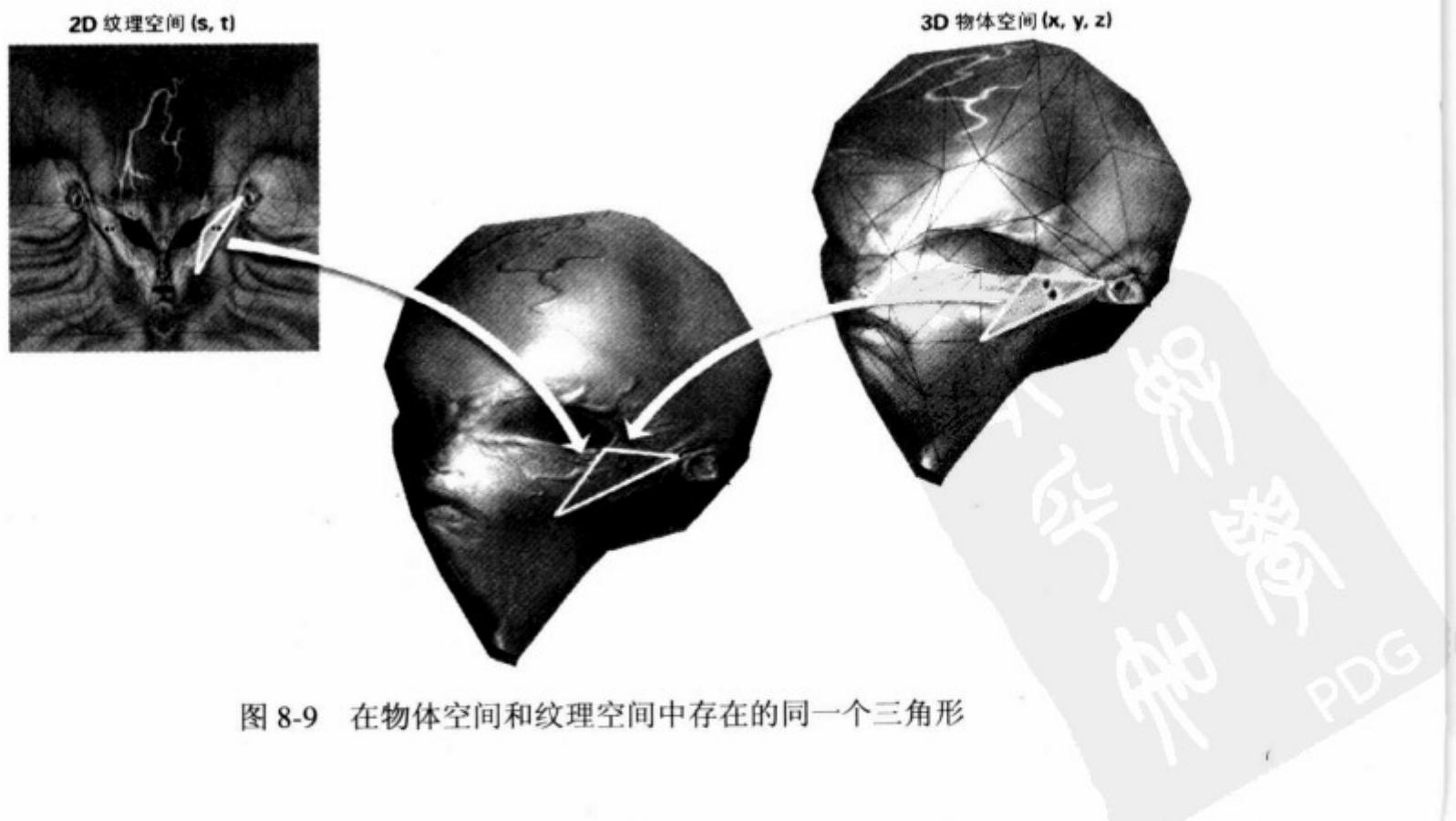


图 8-9 在物体空间和纹理空间中存在的同一个三角形

这个贴图三角形的每个顶点都有一个 3D 物体空间位置和一个 2D 纹理坐标集。你可以把这 5 个坐标的组合看成一个 5D 顶点。然后，你就可以用如下方面描述三角形的顶点 v_0 、 v_1 和 v_2 ：

$$v_0 = \langle x_0, y_0, z_0, s_0, t_0 \rangle$$

$$v_1 = \langle x_1, y_1, z_1, s_1, t_1 \rangle$$

$$v_2 = \langle x_2, y_2, z_2, s_2, t_2 \rangle$$

因为所有这些坐标位于同一个三角形的平面中，所以能够用 s 和 t 来决定 x 、 y 和 z 的平面公式：

$$A_0x + B_0s + C_0t + D_0 = 0$$

$$A_1y + B_1s + C_1t + D_1 = 0$$

$$A_2z + B_2s + C_2t + D_2 = 0$$

对这 3 个公式的每一个，你可以使用三角形的顶点坐标来计算系数 A 、 B 、 C 和 D 的值。例如， A_0 、 B_0 、 C_0 和 D_0 可以用公式 8-7 所显示的方法计算：

公式 8-7 一个纹理三角形的 (x , s , t) 平面的平面公式的系数

$$\langle A_0, B_0, C_0 \rangle = \langle \langle x_0, s_0, t_0 \rangle - \langle x_1, s_1, t_1 \rangle \rangle \times \langle \langle x_0, s_0, t_0 \rangle - \langle x_2, s_2, t_2 \rangle \rangle$$

$$D_0 = -\langle A_0, B_0, C_0 \rangle \cdot \langle x_0, s_0, t_0 \rangle$$

重写平面公式允许你用 s 和 t 来表示 x 、 y 和 z ：

$$x = \frac{-B_0s - C_0t - D_0}{A_0}$$

$$y = \frac{-B_1s - C_1t - D_1}{A_1}$$

$$z = \frac{-B_2s - C_2t - D_2}{A_2}$$

这 3 个公式提供了一种方法来把纹理空间的 2D 位置转换到三角形上在物体空间中的它们对应的 3D 位置。这些简单的 s 和 t 的线性函数提供了 x 、 y 和 z 。这些公式与公式 8-3 中圆环的公式很类似。正如在圆环中的情况那样，我们对用 s 和 t 表示的 $\langle x, y, z \rangle$ 向量的偏导数感兴趣：

公式 8-8 用纹理坐标集表示的一个纹理空间三角形的物体空间的偏导数

$$\left\langle \frac{\partial x}{\partial s}, \frac{\partial y}{\partial s}, \frac{\partial z}{\partial s} \right\rangle = \left\langle -\frac{B_0}{A_0}, -\frac{B_1}{A_1}, -\frac{B_2}{A_2} \right\rangle$$

$$\left\langle \frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, \frac{\partial z}{\partial t} \right\rangle = \left\langle -\frac{C_0}{A_0}, -\frac{C_1}{A_1}, -\frac{C_2}{A_2} \right\rangle$$

这个公式提供了类似于公式 8-4 中的反向梯度，但是这些反向梯度非常简单。实际上，每项都是一个常数。这是合理的，因为一个三角形是一致平坦的，它没有圆环表面的弯曲。

这些反向梯度的规范化版本使用与圆环一样的方法能够组成一个旋转矩阵。使用用 s 表示的规范化反向梯度作为 tangent 向量，并用 t 表示的规范化反向梯度作为 binormal 向量。你能够使用这两个反向梯度的外积作为法向量，但是如果模型为三角形的每个顶点提供了一个每个顶点的法向量，最好还是使用模型的每个顶点的法向量来代替。这是因为每个顶点的法向量确保了你的模型的凹凸贴图光照与非凹凸贴图的每个顶点的光照是一致的。

规范化两个反向梯度的外积能够给每个三角形一个统一的法向量。这将产生一个有许多小平面的外表。

8.4.2 一些告诫

一、纹理空间与物体空间的正交性

用 s 表示的反向梯度不保证与用 t 表示的反向梯度正交。虽然这正好在圆环的每个点上成立（这是在第 8.3 节中使用圆环的另一个原因），但是这并不是普遍成立的。实际上，这两个反向梯度趋于近似正交，因为，否则设计这个模型的美工人员将不得不画一个表示扭曲的相关联的纹理。美工人员自然地选择通过近似正交地反向梯度把纹理应用到模型上。

二、注意纹理空间中的零面积三角形

很可能一个三角形的两个顶点会在纹理空间中共享同一个 (s, t) 位置（或近似相同的位置），或者 3 个纹理空间的位置也许会在同一条直线上。这将会在纹理空间产生一个零面积的退化的三角形（或者面积近似为 0）。这个三角形在物体空

间也许仍然有面积；它也许只在纹理空间发生退化。美工人员应该避免创作这样的三角形，如果纹理坐标要被用于凹凸映射。

如果在纹理空间中的零面积的三角形出现在一个凹凸贴图的模型上，它们将使整个三角形只有一个单独的法向量，这将导致不正确的光照。

三、在纹理空间中的负面积三角形

当应用纹理坐标集到一个多边形模型的时候，美工人员通常镜像一个纹理的某些区域。例如，一个纹理也许只包含了一张脸的右半部分。然后，美工人员就能把脸部的纹理应用到脸的右半部分和左半部分。同样的半张脸的图像能够应用到脸的左右两边是因为脸通常是对称的。这个优化通常能够有效地使用可利用纹理的分辨率。

当应用一个贴图纹理的时候这项技术工作得非常的好，因为贴图根本不知道它们面对的是什么方向。但是，在这种方法中当多边形镜像法向量贴图的区域时，法向量贴图编码了翻转的方向向量。

这个问题可以通过下面的方法来避免。一个是当应用纹理坐标到模型的时候让美工人员放弃镜像。或者是当一个三角形在纹理空间被镜子反射的时候通过自动识别，然后再适当地翻转每个顶点的法向量方向。第二个方法更可取，因为当一个三角形在纹理空间中的面积为负的时候，你能够使用软件工具来翻转(取反)法向量，并适当地调整网格（例如，使用 NVIDIA 的 NVMeshMender 软件，这个软件可以从 NVIDIA 的开发人员网站免费获得，developer.nvidia.com）。

四、凹凸贴图纹理坐标的非均匀伸展

当给一个模型赋予纹理坐标的时候，美工人员有时候以一种非均匀的方式伸展一个纹理。此外，这对贴图通常工作得很好，但是伸展会带凹凸映射带来一些问题。非均匀地缩放纹理意味着用 s 表示的反向梯度将比用 t 表示的反向梯度长度大很多。通常，你自动地从高度域生成法向量，贴图而没有参考任何一个特定的模型。当法向量贴图应用到模型的时候，你隐含地假设任何法向量贴图的伸展是均匀的。

你能够通过要求美工人员避免非均匀伸展来避免这个问题，或者通过当转换高度域纹理到法向量贴图的时候考虑伸展来避免这个问题。

8.4.3 推广到一个多边形的网格

你能够把在前一小节描述的在多边形到多边形基础上的方法应用到你的多边形网格上的每个多边形。你为网格中的每个三角形计算 tangent、binormal 和 normal 向量。

在共享顶点的混合基

但是，在一个网格中，也许不止一个三角形会共享网格中一个给定的顶点。通常，共享一个特定顶点的每个三角形都有它自己的不同的 tangent、binormal 和 normal 向量。所以，对每个共享一个特定顶点的三角形，由 tangent、binormal 和 normal 向量组成的基同样地是不同的。

但是，如果不同三角形的这些 tangent、binormal 和 normal 在一个被共享的顶点足够类似（而且它们通常是类似的），你能够把这些向量混合在一起而不会导致明显的异常现象。可供选择的方法是为被初始顶点共享的每个三角形的每个顶点创建一份拷贝。一般地，如果向量不是太发散的话，在这样的顶点混合基是最好的办法。当一个模型没有很好地镶嵌地时候，这个方法还能帮助避免一个很多小平面的外表。

正如前面讨论的，镜像是一种向量需要分散的情况。如果发生了镜像，你需要为每个被不同映射的三角形用正确的 tangent、binormal 和 normal 向量赋给每个三角形一个不同的顶点。

8.5 把凹凸映射和其他效果结合在一起

8.5.1 印花贴图 (Decal Map)

被用于你的凹凸贴图同样的纹理坐标通常会被印花贴图纹理共享。实际上，在第 8.4 节的讨论假设了被赋予来应用一个印花纹理的纹理坐标将被用来得到用于凹凸映射的 tangent 和 binormal 向量。

通常，当一个游戏引擎不支持凹凸映射的时候，美工人员将被迫通过把光照变化绘制到纹理中来增强印花纹理。当你凹凸映射一个表面的时候，凹凸映射应该提供这些光照变化。美工人员在构造凹凸贴图和印花贴图的时，在印花贴图中必

须小心地单独地编码漫反射材质的变化而不添加光照变化。美工人员应该把几何表面的变化编码到一个高度域，从高度域能够生成一个法向量贴图。

例如，一名美工人员应该在印花贴图中把一件纯绿色的衬衫绘制成纯绿色。相反，美工人员应该在衬衫的布料上绘制折痕来表示在高度域中的光照变化，而不是在印花贴图中。如果美工人员对这些不仔细，他们将不注意地重复创建光照效果，这将使得凹凸贴图的表面太暗。

8.5.2 光泽贴图

一个物体的某些区域很有光泽（例如皮带扣和盔甲）而其他区域则比较阴暗（例如纺织物和人体的皮肤）的情况是很普通的。光泽贴图纹理是一种控制贴图，它编码了一个模型上的镜面反射强光的变化。就像印花贴图和法向量贴图那样，光泽贴图能够和其他纹理贴图共享相同的纹理坐标的参数表示。光泽贴图通常被存储在印花贴图的 alpha 分量中（或者甚至是凹凸贴图中），因为 RGBA 纹理通常与 RGB 纹理几乎一样有效。

下面的 Cg 片段代码显示了一个印花纹理是怎样同时提供印花纹理和一个光泽贴图的：

```
float4 decalTexture = tex2D ( decal, texCoord);
color = lightColor * ( decal.rgb * ( ambient + diffuse ) +
decal.a * specular);
```

8.5.3 投射自己的几何阴影 (Geometric Self-Shadowing)

投射自己的几何阴影说明了这样的一个事实，如果光源在一个表面的平面以下，这个表面不应该反光。对漫反射光照，当光向量和法向量的点积是负值的时候，这种情况将会发生。在这种情况下，点积的结果被映射到 0。你能够在 Cg 中像下面这样实现这个方法：

```
diffuse = max ( dot ( normal, light ), 0 );
```

第 5 章解释了镜面反射项当半角向量和法向量的点积是负值的时候，或者当漫反射成分被映射到 0 的时候，是如何通过把镜面反射项映射到 0 来对投射自己的几何阴影起作用的。你能够在 Cg 中像下面这样实现这个方法：

```
specular = dot ( normal, light ) > 0 ?
max ( dot ( normal, halfAngle ), 0 ) : 0 ;
```

当你进行凹凸贴图的时候，实际上会有两个表面法向量：传统插值的法向量和由法向量贴图提供的被扰动的表面法向量。一种解释这两个法向量的方法是，被插值的法向量是表面方向的大范围的近似，而被扰动的法向量则是表面方向的小范围近似。

如果任何一个法向量指向入射光方向的反方向，则这里将没有从这个光获得的光照。当你为了凹凸映射在纹理空间进行光照的时候，光的 z 向量指明了这个光是在几何的（大范围）法向量的水平线上面还是下面。如果 z 分量是负的，表面的几何方向将背向光源，因而在这个表面上将没有从这个光源获得的光照。你能够在 Cg 中像下面这样实现这个方法：

```
diffuse = light.z > 0 ? max (dot (normal, light), 0) : 0;
```

? : 测试为大范围的表面方向实施了投射自己的几何阴影。max 函数则为小范围表面方向实施了投射自己的几何阴影。你能够为镜面反射凹凸映射在 Cg 中用这种方法实现投射自己的几何阴影：

```
specular = dot (normal, light) > 0 && light.z > 0 ?
    max (dot (normal, halfAngle), 0) : 0;
```

你是否在你的凹凸映射中实现投射自己的几何阴影是个人喜好和性能平衡的问题。实现由大范围表面方向引起的投射自己的几何阴影意味着一个光源将不会照亮一些本来应该被照亮的片段。但是，如果你不实现由大范围表面方向引起的投射自己的几何阴影，则凹凸贴图表面的光源将不会为入射光的方向提供一个清楚提示。

当场景在运动的时候，由于大范围的投射自己的几何阴影，一个突然的跳跃将会引起一个凹凸贴图的表面光照不自然地闪烁。为了避免这个问题，使用一个类似 lerp 或 smoothstep 的函数来提供一个更加缓和的过渡。

8.6 练习

1. 你自己尝试一下：使用一个图像编辑程序用一个鹅卵石的图案来替换在示例 8-6 中使用的法向量贴图。提示：你能够编辑包含这个法向量贴图的文件。你不需要修改任何代码。
2. 你自己尝试一下：当你从高度域生成一个法向量贴图的时候，你能够通过一个如下所示的某个缩放因子 s 来缩放向量的差：

$$normal = \frac{\langle s(H_g - H_a), s(H_g - H_r), 1 \rangle}{\sqrt{(s(H_g - H_a))^2 + (s(H_g - H_r))^2 + 1}}$$

当你从高度域使用一个逐渐增加的 s 缩放因子重新生成一个法向量贴图，物体的外表会发生什么呢？当你减小缩放因子的时候，又会发生什么呢？如果你对 $H_g - H_a$ 使用一个缩放因子，而对 $H_g - H_r$ 使用另一个缩放因子，会发生什么呢？

3. 你自己尝试一下：通过为每个光源的影响渲染一次来实现多个光源的凹凸映射。为一个凹凸贴图的表面使用“depth equal”深度测试和加法混合来结合来自多个光源的影响。

8.7 补充阅读

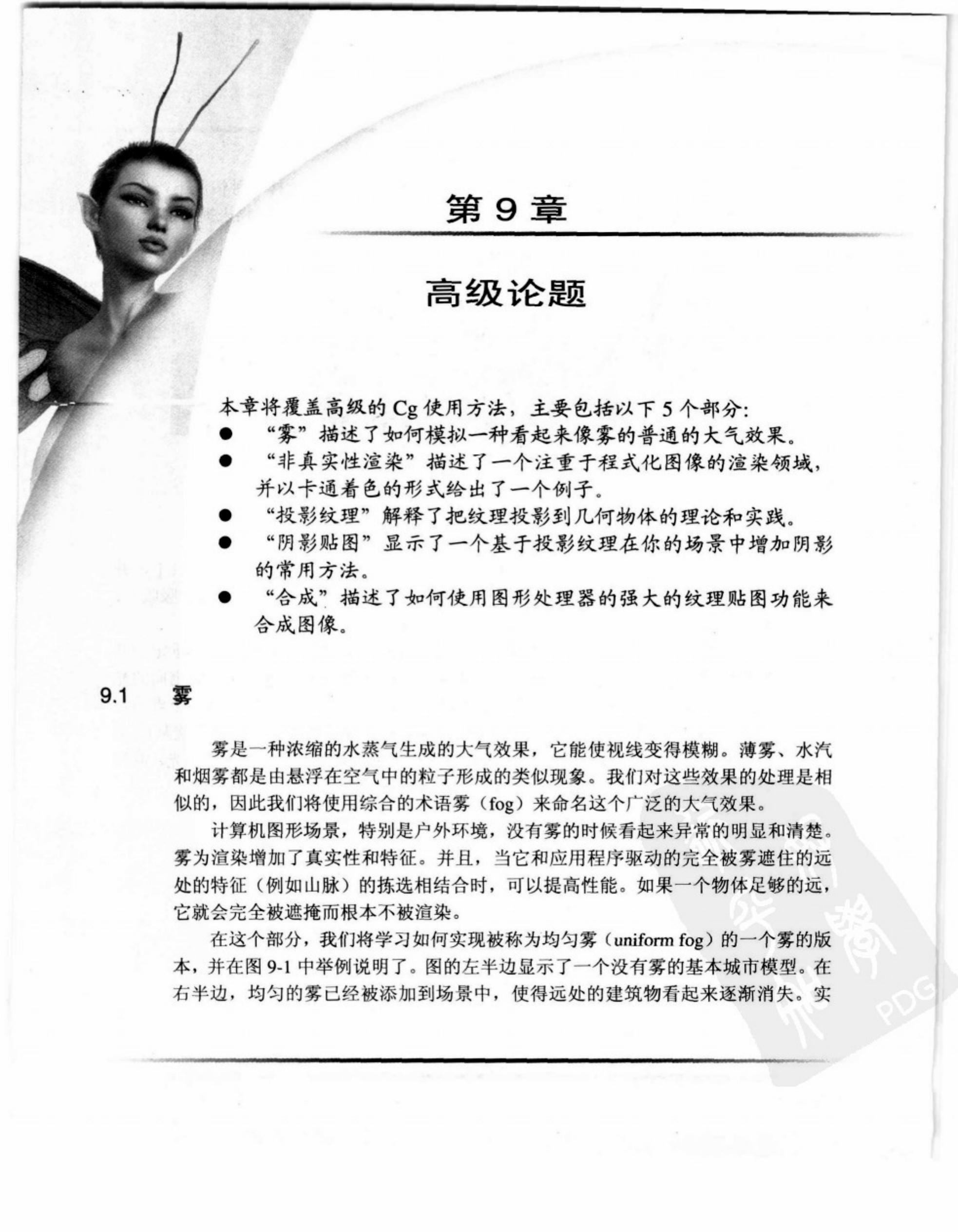
Jim Blinn 在 1978 年发明了凹凸映射。Blinn 的“Simulation of Wrinkled Surfaces”(ACM Press) 是一篇传造性的 SIGGRAPH 计算机图形学论文。

Mark Peercy、John Airey 和 Brian Cabral 在 1997 年发表了一篇题为“Efficient Bump Mapping Hardware”(ACM Press) 的 SIGGRAPH 论文，解释了切空间和它在硬件凹凸映射中的应用。

在 2000 年，Mark Kilgard 发表了“A practical and Robust Bump-Mapping Technique for Today’s GUPs”。这篇论文解释了凹凸映射的数学理论并介绍了一种适用于第三代图形处理器的技术。你能够在 NVIDIA 的开发人员的网站(developer.nvidia.com)上找到这篇论文。

Sim Dietrich 在 *Game Programming Gems* (Charles River Media, 2000 年中文版已由人民邮电出版社出版) 发表了一篇题为“Hardware Bump Mapping”的文章，介绍了为纹理空间的凹凸映射使用多边形模型的纹理坐标集。





第 9 章

高级论题

本章将覆盖高级的 Cg 使用方法，主要包括以下 5 个部分：

- “雾”描述了如何模拟一种看起来像雾的普通的大气效果。
- “非真实性渲染”描述了一个注重于程式化图像的渲染领域，并以卡通着色的形式给出了一个例子。
- “投影纹理”解释了把纹理投影到几何物体的理论和实践。
- “阴影贴图”显示了一个基于投影纹理在你的场景中增加阴影的常用方法。
- “合成”描述了如何使用图形处理器的强大的纹理贴图功能来合成图像。

9.1 雾

雾是一种浓缩的水蒸气生成的大气效果，它能使视线变得模糊。薄雾、水汽和烟雾都是由悬浮在空气中的粒子形成的类似现象。我们对这些效果的处理是相似的，因此我们将使用综合的术语雾（fog）来命名这个广泛的大气效果。

计算机图形场景，特别是户外环境，没有雾的时候看起来异常的明显和清楚。雾为渲染增加了真实性和特征。并且，当它和应用程序驱动的完全被雾遮住的远处的特征（例如山脉）的拣选相结合时，可以提高性能。如果一个物体足够的远，它就会完全被遮掩而根本不被渲染。

在这个部分，我们将学习如何实现被称为均匀雾（uniform fog）的一个雾的版本，并在图 9-1 中举例说明了。图的左半边显示了一个没有雾的基本城市模型。在右半边，均匀的雾已经被添加到场景中，使得远处的建筑物看起来逐渐消失。实

际上，如果一个建筑物足够远，雾就会把它完全遮住。在这样一种情况下，你可以对远处的建筑物使用较低级别的细节并忽略完全被遮住的建筑物来提高性能。

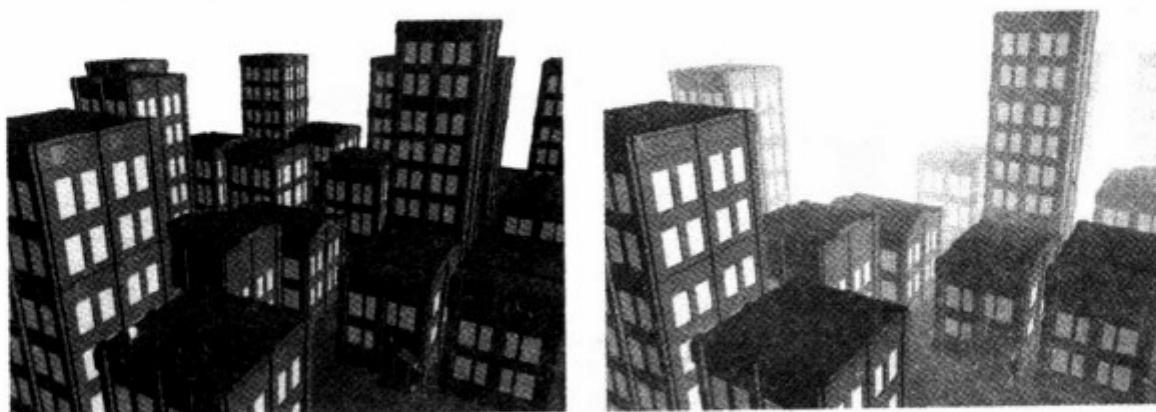


图 9-1 向一个场景添加均匀的雾

9.1.1 均匀的雾

沿着一个特别的从某个物体到你的眼睛的光线，雾的粒子悬浮在空气中，并“散射”某些原本会到达你眼睛的光。污染物、烟和空气中其他粒子还能够吸收光，而不是散射光，因此会减少到达你眼睛的光的强度。

尽管某些光在沿着光线传播的时候会被吸收或散射，雾的粒子能够重定向其他散射的光，因此额外的一些光会沿着某些光线到达你的眼睛。这些重定向的光线来自被雾的粒子散射的光的总和。图 9-2 显示了光从一个点传播到另一个点的不同的方法，这取决于雾的存在。如果没有雾的粒子介于其中，光线会直接从点 A 传播到点 B。但是，如果有雾的粒子在空气中，它们就会吸收或散射一些光，并另外从聚合的散射光随机地重定向一些光线到点 B。

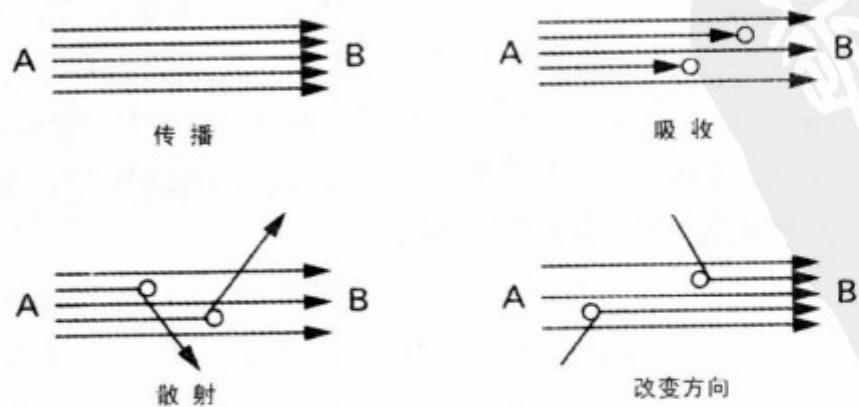


图 9-2 雾的粒子能够影响光从点 A 到点 B 的传播

当光被水蒸气散射的时候，聚合的被散射光和重定向的光的颜色趋向于灰白色。当散射是由悬浮在空气中的污染物或烟引起的时候，颜色可以从薄雾的灰色变化到浓烟的黑色。

光沿着从被观察的物体到你眼睛的光线传播得越远，雾的粒子将散射或吸收光的可能性更大。同时，有更多的沿着光线重定向额外的光到眼睛的机会。这就是为什么雾在远处更明显。

9.1.2 雾的属性

- 到达你眼睛的散射光的聚合颜色是雾的颜色 (fog color)。
- 雾的密度是，(fog density) 沿着视线在一个给定的单位距离度量多少光被雾散射或吸收。雾的密度越大，雾看上去越浓。
- 沿着从物体到眼睛的视线的距离是雾的距离 (fog distance)。雾的距离越大，一个物体看起来更模糊。

目前，假设雾的密度和雾的颜色是均匀和不变的，但是可以让从眼睛到一个可见物体的雾的距离变化。我们想要知道的和想要用 C_f 计算的是，雾如何影响在这些模糊条件下的远处物体的外表的颜色。

9.1.3 雾的数学运算

一、在一个单位距离上吸收、散射和重定向

如果光在一个有均匀的雾的空气中沿着某个光线传播一个单位距离，你可以从数学上推论出由于吸收和散射，光的强度被雾减少了一定的百分比。同时，光的重定向将会沿着光线增加光的强度。下面的公式表示了这点：

$$C_{\text{leave}} = C_{\text{enter}} - C_{\text{reduction}} + C_{\text{increase}}$$

其中：

- C_{leave} 是沿着一条光线离开一个单位线段的光的颜色亮度。
- C_{enter} 是沿着一条光线进入一个单位线段的光的颜色亮度。
- $C_{\text{reduction}}$ 是沿着单位线段所有被吸收或散射的颜色亮度。
- C_{increase} 是沿着光线方向单位线段上被雾重定向的额外的光的颜色亮度。

被吸收和散射的光的强度不能大于进入光的强度。假设雾的密度是均匀的，在入射光的颜色和被散射光的颜色之间必须有一个固定的比例。因此我们有：

$$0 \leq C_{\text{reduction}} \leq C_{\text{enter}}$$

$$C_{\text{reduction}} = h \times C_{\text{enter}}$$

$$0 \leq h \leq 1$$

其中：

- h 是一个基于雾的密度的不变的比例因子。

假设一个均匀雾的颜色，则在光线的方向上被雾散射的额外的光的颜色是：

$$C_{\text{increase}} = h \times C_{\text{fog}}$$

其中：

- C_{fog} 是不变的雾的颜色。

现在你可以根据进入光的颜色，一个恒定的光散射和消失的百分比以及一个恒定的雾的颜色来表示离开有雾的单位距离的光的颜色。关系式是：

$$C_{\text{leave}} = (1 - h) \times C_{\text{enter}} + h \times C_{\text{fog}}$$

二、用一个线性混合来表示雾

你可以一个线性混合来重新排列这个公式，其中 $g=1-h$ ，因此它可以写作：

$$C_{\text{leave}} = g \times C_{\text{enter}} + (1 - g) \times C_{\text{fog}}$$

这意味着在单位距离的雾光线段末端的光的强度等于在光线段起点的初始的光的强度，减少了初始光强度的某个 g 倍，但是增加了雾的颜色的 $(1-g)$ 倍。

三、在多个单位距离上应用雾

如果光在雾中传播了多个单位距离我们可以递归地应用这个公式，假设雾的密度和颜色在所有这些单位距离上都是均匀的。当光传播了 3 个单位距离的时候，在 3 个单位光线末端的光的强度为：

$$C_{\text{leave}} = g \times (g \times (g \times C_{\text{enter}} + (1 - g) \times C_{\text{fog}}) + (1 - g) \times C_{\text{fog}}) + (1 - g) \times C_{\text{fog}}$$

这个公式可以简化成下面的公式：

$$C_{\text{leave}} = g^3 \times C_{\text{enter}} + (1 - g^3) \times C_{\text{fog}}$$

注意我们是如何在 3 个单位的距离上应用雾公式的过程中使 g 出现了 3 次方。我们可以为任意雾距离 z 归纳出如下这个函数：

$$C_{\text{eye}} = g^z \times C_{\text{object}} + (1 - g^z) \times C_{\text{fog}}$$

我们真正感兴趣的是，一个物体被眼睛透过雾看到的颜色，而不仅仅是任意一条光线的开始和结束的两个点，因为我们把 C_{leave} 重新标为 C_{eye} ，而 C_{enter} 被重新标为 C_{object} 。

回想一下，你可以把一个指数例如 g^z 用指数和对数函数重写，因此：

$$g^z = \exp_2(\log_2(g) \times z)$$

用指数和对数函数重新表示 C_{eye} 可以获得公式 9-1。

公式 9-1 均匀的雾

$$C_{eye} = f \times C_{object} + (1 - f) \times C_{fog}$$

$$f = \exp_2(-d \times z)$$

$$d = -\log_2(g)$$

我们把 z 称为雾的距离， d 为雾的密度，而 f 为雾的因子。你为指数函数和对数函数选择的基没有什么关系，只要两个函数的基相同。

CineFX

CineFX 图形处理器计算基于 2 的指数和对数（ \exp_2 和 \log_2 ）函数非常有效。为了获得最优的性能，使用 \exp_2 和 $\log2Cg$ 标准库函数来计算这些函数。

计算雾的因子的公式中的指数是有意义的，当你考虑一下光在雾中传播的每单位距离，一定百分比的入射光强度会被散射掉。这和一个用指数衰减函数模拟的同位素放射性衰减的过程很类似。

我们假设雾的密度 d 和雾的距离 z （一个绝对距离）是非负的。在雾因子计算中对 d 的取反确保了指数的衰减，而不是增加。

9.1.4 直觉化公式

为了确保上面的这个公式与我们对雾是如何起作用的直觉相匹配，考虑一下当雾的距离为 0 或非常接近 0 的时候会发生什么。在这种情况下，物体非常接近眼睛，我们期望没有雾或很少的雾。如果 $z=0$ ，则：

$$f = \exp_2(-d \times 0) = 1$$

因此眼睛可以看到所有（100%）的物体颜色，而没有任何雾的颜色，正如期待的那样。

如果雾的距离 z 非常的大，则：

$$f = \exp_2(-d \times \infty) \approx \exp(-\infty) = 0$$

因此眼睛不会看到任何物体的颜色，而只能看见雾的颜色。这是对的，因为雾遮住了远处的物体。

雾的密度 d 的数量级越大，雾会变的越浓。同样地， d 的值越小就会减少雾。如果 d 为 0，则：

$$f = \exp_2(-0 \times z) = 1$$

这意味着看到的物体颜色是 100% 的物体的颜色和 0% 的雾的颜色。

9.1.5 用 Cg 创建均匀的雾

随后的 Cg 程序实现了我们在前面的讨论中已经解释了的雾的理论。

示例 9-1 的 C9E1f_fog 片段程序实现了在 9.1.3 节特别介绍的雾的公式。这个程序对一个贴图纹理进行采样；并用插值的颜色调制贴图的颜色；然后雾化贴图的片段颜色，假设一个被插值的雾的指数和一个均匀的雾的颜色。

示例 9-1 C9E1f_fog 片段程序

```
void C9E1f_fog(float2 texCoord : TEXCOORD0,
                float fogExponent : TEXCOORD1,
                float4 color : COLOR,
                out float4 oColor : COLOR,
                uniform sampler2D decal,
                uniform float3 fogColor)
{
    float fogFactor = exp2(-abs(fogExponent));
    float4 decalColor = tex2D(decal, texCoord);
    float4 texColor = color * decalColor;

    oColor.xyz = lerp(fogColor, texColor.xyz,
                      fogFactor);
    oColor.w = color.w;
}
```



C9E1f_fog 片段程序是一个在 CineFX 图形处理器上 fixed 数据类型能够提高性能的情况下很好的例子。数学操作例如三元和四元乘法以及 lerp 函数，当在 CineFX 图形处理器上用 fixed 量执行的时候是非常有效的。同样，C9E1f_fog 片段程序依赖于 exp2 标准库函数，它只被高级的片段 profile 支持。

使 C9E1f_fog 片段程序与示例 9-2 显示的顶点程序结合在一起。C9E2v_fog 程序从到眼睛的最短距离（用标准库函数 length 计算的）计算出了一个雾的指数和均匀的 fogDensity 参数。

其他的质量改进和性能优化是可能的。

示例 9-2 C9E2v_fog 顶点程序

```
void C9E2v_fog(float4 position      : POSITION,
                float4 color        : COLOR,
                float2 decalCoords : TEXCOORD0,

                out float4 oPosition     : POSITION,
                out float4 oColor       : COLOR,
                out float2 oDecalCoords : TEXCOORD0,
                out float fogExponent   : TEXCOORD1,

                uniform float    fogDensity,    // Based on log2
                uniform float4x4 modelViewProj,
                uniform float4x4 modelView)
{
    // Assume nonprojective modelview matrix
    float3 eyePosition = mul(modelView, position).xyz;
    float fogDistance = length(eyePosition);
    fogExponent = fogDistance * fogDensity;
    oPosition = mul(modelViewProj, position);
    oDecalCoords = decalCoords;
    oColor = color;
}
```

一、平面的雾距离

计算平面眼距离，而不是欧拉眼距离，只需要较少的顶点程序操作。可以用下面的平面雾距离代替 length 函数来计算 fogDistance，这通常是可接受的：

```
float fogDistance = eyePosition.z;
```

在非常广的视角的情况下，这种近似会导致在视野远处边界的雾的量不足，但是它通常工作得很好。

二、每个片段的欧拉雾距离

如果用顶点程序来输出光栅器线性插值的 3D 眼空间坐标，而不是插值一个欧

拉距离，均匀雾将更加精确。然后，片段程序就能够在计算雾的指数和有关雾的其他计算之前，计算在每个片段的真正的欧拉距离。虽然这个方法更加精确，但是对于合理镶嵌的场景，这个技术的代价通常超过了质量上的好处。

三、用一个纹理来编码雾的因子的转换函数

你自定义的雾的实现可以用 1D、2D 或 3D 纹理对转换雾的指数或距离到一个雾的因子的函数进行编码，而不是使用 Cg 标准库函数的 \exp_2 （或类似的）指数函数。纹理存取通常非常有效，使你能够更好地控制雾的减少。记住你将需要缩放雾的距离来匹配纹理图像的[0, 1]的范围。

9.2 非真实性渲染

许多着色器创作者尝试着不断用他们的计算机生成的图像实现对真实表现的提高。实际上，本书的大部分内容都关注于这个目标。我们从简单的概念开始，逐渐增加越来越复杂的效果——所有的目标都是教你如何生成更加真实的着色器。

但是，有时候你也许想要生成一些看上去不是那么真实的图像。在计算机辅助设计（CAD）程序中，你也许只需要用线框图来表示物体，或只用平面着色，因此它们的形状非常容易辨别。在一个基于物理的火箭模拟程序中，你也许对火箭的一个部分进行着色来表示它的温度，而不是它的物理外表。或者也许你想要创建卡通。

所有这些例子都可以归为一个被称为非真实渲染（或 NPR）的领域。在 NPR 中，主要的思想是用一种特别的方法来风格化渲染，因此它们的价值不同于来自那些使它们和在真实世界里看起来一样的表现方法。

9.2.1 卡通着色

一个完整的有关 NPR 领域的概观超出了本书的范畴，但是我们将考虑一种普通而有用的被称为卡通着色（toon shading）的 NPR 技术。卡通着色是一种渲染技术，用不变的颜色和明显的轮廓来对物体进行着色——好像它们是卡通而不是真实世界的物体。

图 9-3 给出了你可以用卡通着色获得结果的一种构思。图的左边显示了一把用普通的漫反射和镜面反射光照的光线枪；图的右边显示了一个对应的卡通着色版

本。卡通着色的光线枪表面只使用了三种色调着色：一种用于明亮的漫反射区域，另一种用于暗的漫反射区域，而第三种则用于反射强光。同样，注意卡通着色的光线枪用黑色描画了轮廓。



图 9-3 卡通着色

卡通着色的一个非常有用方面是，你不需要改变你所表示的人物和物体的方式。你仍然能用三维网格来画它们，而不用把任何东西画成二维图像（或 sprite）。其中的窍门是如何对它们进行着色。通过用一种新的着色器来替换传统光照着色器，你能够使这些渲染看起来像是卡通。

9.2.2 实现卡通着色

很多时候，光照用于给物体一个真实和着过色的外表。但是，卡通着色的目标是减少着色中的变化。

卡通着色器有 3 个主要的部件：

1. 漫反射着色只需要用两个值来表示：一个用来表示明亮的区域，而另一个用来表示暗的区域。
2. 反射强光需要鉴定并用单一颜色来表示，在那里它们的亮度足够高。
3. 物体需要被描绘轮廓来完成卡通外表。

一、漫反射着色

一个卡通着色器需要把漫反射光照从许多着色颜色转换为几种颜色。想一想

使用漫反射光照的单色例子，其中每个像素都有一个范围为从 0.0 到 1.0 的颜色。因为漫反射光照，像素的值通常逐渐地从 0.0（在没有光照的区域）到 1.0（在完全光照的区域）。

一种创建一个卡通着色器的方法是把这个连续的范围分成两个不同的范围。例如，你能够取 0.0 到 0.5 的像素值并用 0.2 替代它们，取 0.5 到 1.0 的像素值并用 1.0 表示它们。其结果将是一个两种色调的图像，并拥有一个很像卡通的外表。

从数学原理上说，我们刚刚讨论的转换规则被称为一个阶梯函数（step function）。不像其他大部分函数拥有一个连续范围的值，阶梯函数只有两个不同的值。图 9-4 举例说明了一个普通函数和一个两个值的阶梯函数。阶梯函数对卡通着色非常有用，因为它们能够告诉你如何把一个范围很广的颜色映射成只有两种颜色。你也能够使用有更多阶梯的函数，如果你喜欢的话。

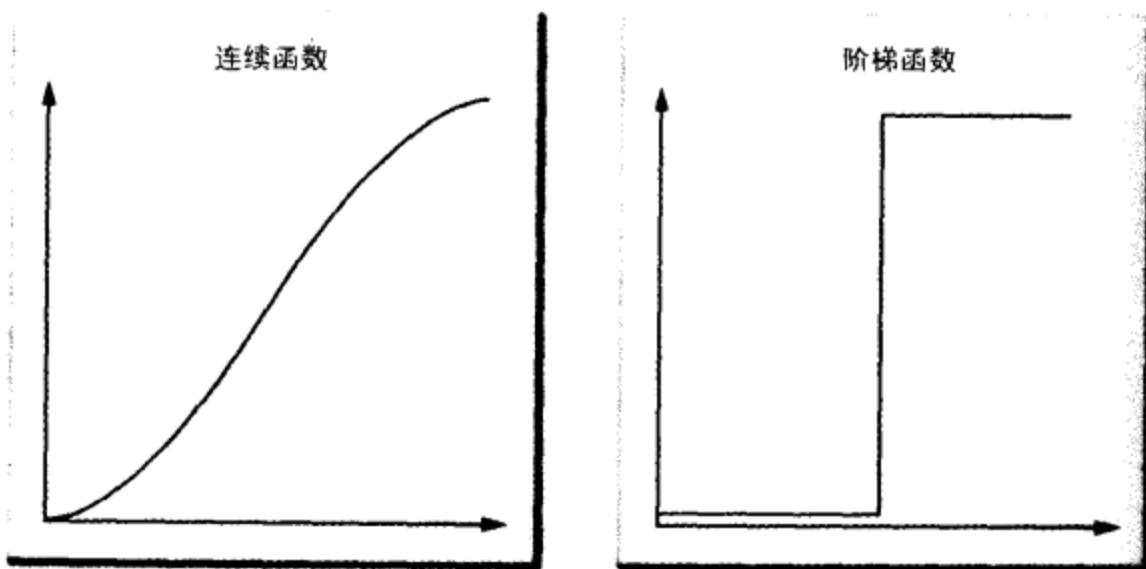


图 9-4 一个连续的函数与一个阶梯函数

基于这个原理，一个卡通着色器的漫反射部分用漫反射光照计算 $N \cdot L$ 部分并把它分类成“明亮”和“黑暗”。你能够选择你喜欢的域值，来获得一个你喜欢的“外表”。

要想把 $N \cdot L$ 的值从一个连续的范围映射到一个阶梯函数，可以使用一个纹理查找。类似你在第 8 章中使用一个规范化的立方贴图来编码一个规范化函数，你能够使用一个 1D 纹理来应用一个阶梯函数。在这种情况下，一个 1D 纹理已经足够了，因为 $N \cdot L$ 是一个标量。图 9-5 显示了一个两个纹理元素宽的 1D 纹理的例子，以及它所表示的阶梯函数。

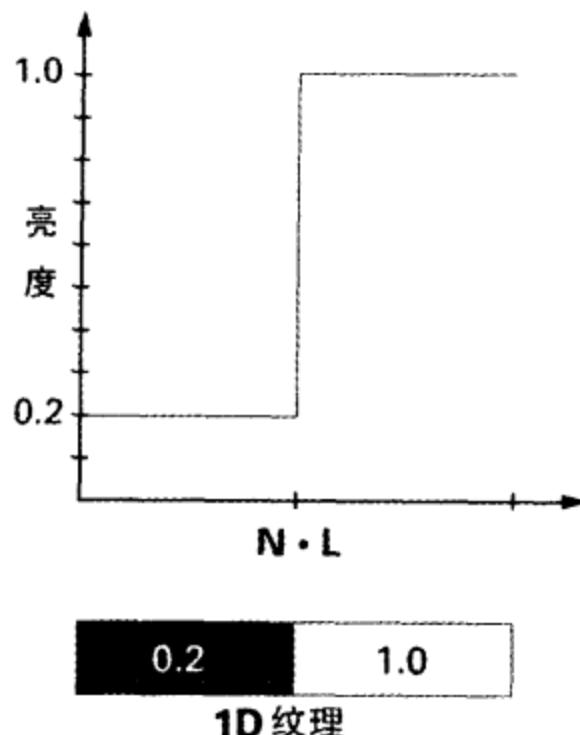


图 9-5 用一个 1D 纹理编码一个阶梯函数

假设应用程序传递了一个叫 `diffuseRamp` 的纹理到你的卡通着色器，很容易就能够应用一个阶梯函数到漫反射光照：

```
diffuseLighting = tex1D (diffuseRamp, diffuseLighting);
```

老的漫反射光照值被从阶梯函数获得的对应的值代替（假设现在没有过滤应用到 1D 纹理上。这确保了被查询的纹理只有两个不同的值，这样才能产生一个明显的阶梯函数）。

二、反射强光

处理镜面反射分量类似于处理漫反射分量。主要的想法是应用一个阶梯函数到反射强光部分，因此每个强光部分要么存在要么不存在，而不是在强度上逐渐变化。这次，一个应用程序指定的被称为 `specularRamp` 的纹理提供了阶梯函数：

```
specularLighting = tex1D ( specularRamp,
                           specularLighting );
```

三、轮廓描绘

最后，你需要描绘物体的轮廓。要实现这点，你需要找到位于物体轮廓的像素，然后用轮廓色（通常是黑色）给它们着色。

一个发现物体轮廓的容易方法是使用 $N \cdot V$ 作为一个量尺。与 $N \cdot L$ 度量了一

一个表面接收了多少光照一样， $N \cdot V$ 度量了表面从当前视点的可见程度。回想一下当你为一个点计算光照的时候，当 $N \cdot L$ 是正值时这个点被照亮，而当 $N \cdot L$ 为负值时这个点为阴影。

类似地，当 $N \cdot V$ 是正值时一个点是可见的，而当 $N \cdot V$ 是负值的时候这个点被隐藏。在 $N \cdot V$ 接近 0 的点代表了从可见到隐藏的转变——这些点就是在/或接近一个物体的轮廓上的点。再一次一个 1D 纹理应用了阶梯函数：

```
// 计算边的颜色
float edge = max ( dot ( N, V ), 0 );
edge = tex1D ( edgeRamp, edge );
```

四、解决走样问题

我们刚刚描述的方法确实创建了一种卡通着色效果，但是在动画的时候它有产生闪烁（由走样产生的）现象的倾向。这是为什么呢？因为漫反射、镜面反射和边界测试太严格了：当物体移动的时候，它们的结果趋向于相当明显的起伏。这是使用阶梯函数很自然的结果，当函数参数值变化的时候，阶梯函数的值会突然地上升。

要解决这个问题，你可以创建一个大一点的 1D 纹理（为每个测试）来编码一个平滑一点的转变。例如，你可以使用一个 16 个纹理元素宽的纹理，并用中间的 4 个纹理元素来编码函数中间的阶梯。然后你就能够打开双线性纹理过滤，它能够自动地平滑这个转变。

9.2.3 集成在一起

示例 9-3 和 9-4 显示了完整的程序并用具体的形式表现了以前的讨论。

示例 9-3 C9E3v_toonShading 顶点程序

```
void C9E3v_toonShading(float4 position : POSITION,
                        float3 normal   : NORMAL,
                        out float4 oPosition : POSITION,
                        out float diffuseLight : TEXCOORD0,
                        out float specularLight: TEXCOORD1,
                        out float edge       : TEXCOORD2,
```

```

        uniform float3 lightPosition,
        uniform float3 eyePosition,
        uniform float shininess,
        uniform float4x4 modelViewProj)
{
    oPosition = mul(modelViewProj, position);

    // Calculate diffuse lighting
    float3 N = normalize(normal);
    float3 L = normalize(lightPosition - position.xyz);
    diffuseLight = max(dot(N, L), 0);

    // Calculate specular lighting
    float3 V = normalize(eyePosition - position.xyz);
    float3 H = normalize(L + V);
    specularLight = pow(max(dot(N, H), 0), shininess);
    if (diffuseLight <= 0) specularLight = 0;

    // Perform edge detection
    edge = max(dot(N, V), 0);
}

```

示例 9-4 C9E4f_toonShading 片段程序

```

void C9E4f_toonShading(float diffuseLight : TEXCOORD0,
                        float specularLight: TEXCOORD1,
                        float edge       : TEXCOORD2,
                        out float4 color : COLOR,
                        uniform float4 Kd,
                        uniform float4 Ks,
                        uniform sampler1D diffuseRamp,
                        uniform sampler1D specularRamp,
                        uniform sampler1D edgeRamp)
{
    // Apply step functions
    diffuseLight = tex1D(diffuseRamp, diffuseLight).x;
    specularLight = tex1D(specularRamp, specularLight).x;
    edge = tex1D(edgeRamp, edge).x;
}

```

```

    // Compute the final color
    color = edge * (Kd * diffuseLight +
                    Ks * specularLight);
}

```

9.2.4 卡通着色技术存在的一些问题

我们讨论的卡通着色的方法是非常容易实现的，但是它有一个非常明显的缺点：它只对曲线物体工作得非常好。这是因为程序基于一个逐渐减少的 $N \cdot V$ 来找轮廓边界。当物体有明显的边界时， $N \cdot V$ 会在边界的地方突然改变，引起轮廓完全变成黑色。

一种普通的解决这个问题的方法是使用一个两个过程的方法。首先，沿着模型的法向量绘制一个稍微膨胀的版本。并用黑色绘制这个几何形状（或者你想用来描绘轮廓的无论什么颜色）。然后，用省略了轮廓计算的卡通着色器照常绘制剩下的几何形状。结果将是一个内部具有卡通着色外表和一个清楚地定义了每个物体轮廓外壳的渲染。不幸地是，这种方法只解决了轮廓边界，而没有解决内部边界。

另一种解决方案是在图像空间执行边界检测。这种方法可以让你发现内部边界，并确保统一的边界宽度。但是，增加这个分析将增加着色器的开销。

9.3 投影贴图

当人们谈及“贴图”的时候，他们通常讨论的是明确地赋与纹理坐标来把一个纹理应用到一个表面上。但是，这不是使用纹理的惟一方法。在这个部分，我们将学习纹理是如何被投影到物体表面的，就像是来自一个幻灯机。自然地，这种技术被称为投影贴图（projective texturing）。

图 9-6 显示了一个使用投影贴图的例子。这个场景由一个平面和漂浮在它上面的两个物体——一个立方体和一个球体组成。投影仪几乎直接在物体上面，并把一个魔鬼的图像投影到场景中。注意这里没有物体“遮住”灯光的概念。例如，魔鬼的眼睛和鼻子同时出现在球体和它下面的平面上。类似地，魔鬼的牙同时出现在立方体和平面上。在它的基本形式里，投影贴图并没有解决物体之间的阴影

关系。当你学习投影贴图是如何工作的时候，就会清楚产生这种结果的原因了。（在本章最后的练习中将会探索该问题的一种解决方案。）



图 9-6 投影贴图

9.3.1 投影纹理如何工作

那么投影贴图是如何工作的呢？一个投影纹理和普通的纹理一样——惟一的区别是纹理如何应用到几何物体上的。和标准的贴图一样，投影贴图需要在你的场景中的每个三角形都有正确的纹理坐标。如果你获得的纹理坐标正确，在每个三角形上你将以光照纹理的正确部分为结束。技巧是把正确的坐标赋给不同的多边形。

还记得在第 4 章中我们讲述了图形流水线的各种变换吗？这些概念中的大部分都可以应用到投影贴图。

我们从每个三角形的物体空间坐标开始，并使用一系列的变换来把顶点映射到一组坐标系统里。这些变换把物体空间中的顶点投影到窗口空间，因此这些三角形能够在视图中被光栅化和显示。如果你把视图看作是一个纹理，很明显你能够把每个顶点映射到纹理上的一个纹理元素。

投影贴图以相同的方式工作。通过应用一系列的变换，你可以把物体空间中的坐标映射到一个 2D 空间（一个纹理），并且你能够找出每个顶点映射到纹理的哪个部分。然后，你能够把这个位置当作纹理坐标赋给顶点，并且在渲染的时候把适当的一块纹理应用到每个三角形上。

一、比较变换

图 9-7 比较了一个传统流水线和投影贴图中的变换的并行性。

左边的是在传统流水线（一个典型的“照相机”）中的一系列的变换，而在右边的则是用于投影贴图的一组变换。注意在右半边，纹理坐标仍然是在齐次空间中。如果你想知道“齐次空间”对纹理坐标意味着什么，请不要着急。这就是你将要学习的。

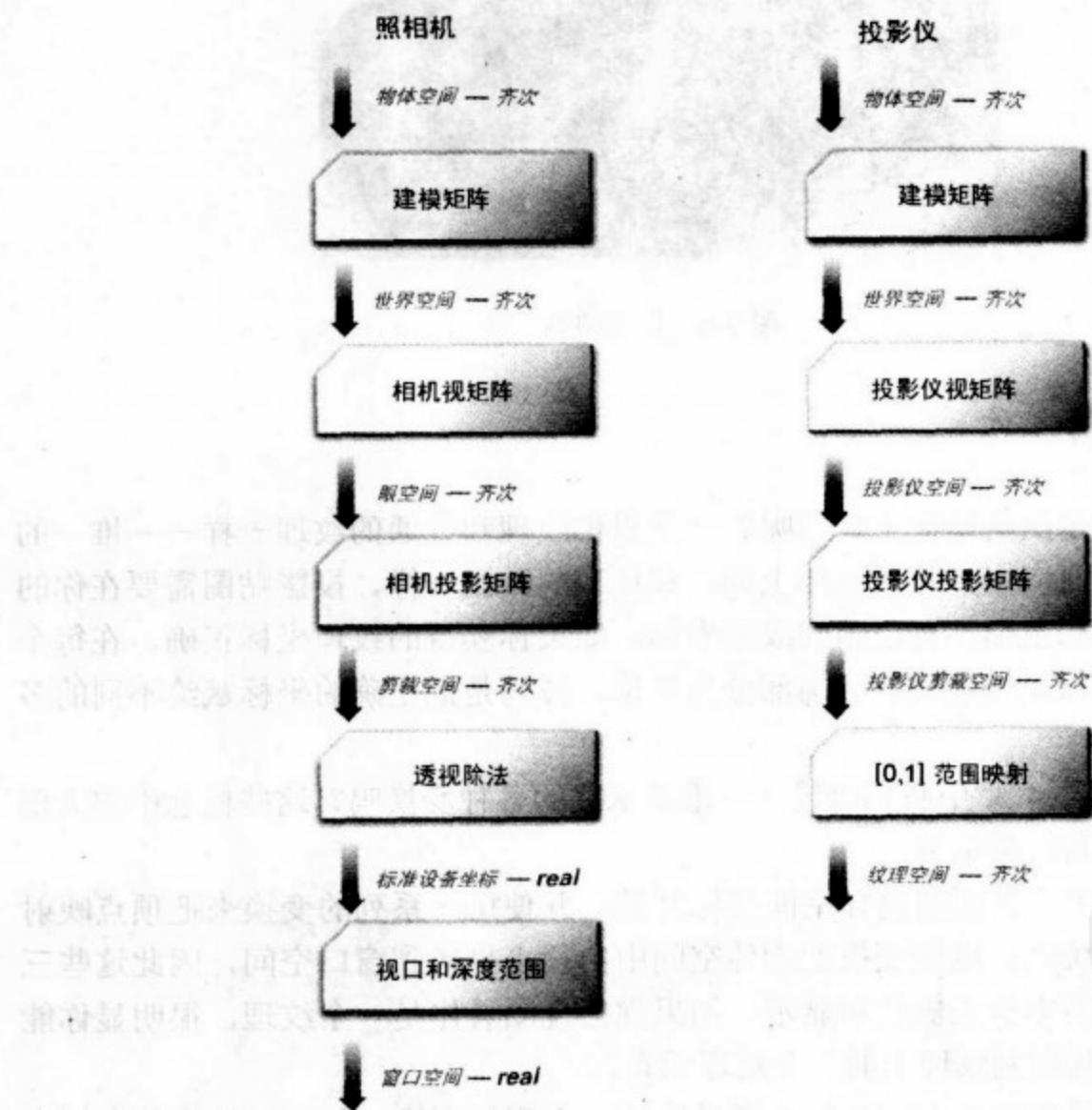


图 9-7 一个传统照相机的变换与一个投影仪的变换

二、齐次纹理坐标

当一个纹理坐标被使用的时候，它们通常是二维的并且是非投影的。传统的

2D 纹理使用一对纹理坐标来进行存取： s 和 t 纹理坐标。有时候，当一个 3D 纹理被使用的时候， r 纹理坐标被用来索引纹理的第三维。当你处理齐次坐标位置的时候，你可以把 s 、 t 和 r 纹理坐标看作 x 、 y 和 z 使用同样的方法来处理。

正如一个齐次位置有被称为 w 的第 4 个成员，齐次纹理坐标也有第 4 个分量，被称为 q 。这个分量允许你在纹理空间中表示投影、旋转、平移和缩放，所有这些都可以用方便的 4×4 矩阵乘法来表示。

当要执行一个投影纹理查找的时候，硬件获取原始的纹理坐标集 (s, t, r, q) 并用 q 除以每个分量，给出 $(s/q, t/q, r/q, 1)$ 。这和发生在顶点程序运行后面的透视除法有些类似。在这个除法之后，该坐标能够正确地索引一个 2D 或 3D 纹理。

现在应该很清楚了，用于投影贴图的理论和基本图形流水线的理论非常类似。例如，当你只指定 s 和 t 纹理坐标的时候， q 被假定为 1。这就像直接在剪裁空间中指定一个位置（使 w 等于 1），正如你在第 2 章所做的那样。惟一的区别是当处理位置的时候投影总是被使用，而当处理纹理坐标的时候投影贴图通常很谨慎地使用。

9.3.2 实现投影纹理贴图

现在是把理论用于视觉的时候了。如果你理解了这些概念，你将发现在 Cg 中根本不需要做很多工作，就可以使投影贴图正确地工作。

其中有两个操作你必须小心处理：在顶点程序中计算投影纹理坐标，和在片段程序中执行投影纹理查找。

在顶点程序中计算纹理坐标

在我们以前的一些顶点程序里，你处理了非投影纹理并且只是简单地把应用程序的纹理坐标传给片段程序。

而在投影贴图中，你不需要从应用程序明确指定每个顶点的纹理坐标。相反，你要使用一个顶点程序自动地从物体空间的每个顶点位置计算纹理坐标。在固定功能的顶点处理过程中，当允许自动纹理生成（通常被称为“texgen”）的时候这个计算将会发生。在顶点程序中实现纹理生成的最大好处是，你确切地知道发生了什么。

公式 9-2 显示了你在顶点程序中需要为投影贴图实现一系列的变换。记住，如果眼睛在光源的位置你将使用同样的顺序，但要附加一个额外的矩阵。这个矩阵

经过缩放和偏移使得结果在[0, 1]范围内，它是被用来存取纹理的。为了更加有效，我们需要把这些矩阵连接成一个单独的矩阵。我们将把这个连接后的矩阵称为纹理矩阵（texture matrix）。

公式 9-2 用于投影贴图的变换序列

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} Light \\ Frustum \\ (projection) \\ Matrix \end{bmatrix} \begin{bmatrix} Light \\ View \\ (look at) \\ Matrix \end{bmatrix} \begin{bmatrix} Modeling \\ Matrix \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ w_0 \end{bmatrix}$$

让我们仔细查看一下这些变换，解释到底发生了什么。顶点开始的时候在物体坐标系中。

1. 被建模矩阵乘。这一步把必须的建模变换应用到顶点上。无论你是否使用投影贴图，都需要应用建模矩阵。
2. 被光的观察矩阵乘。旋转和平移顶点，使它们在光的参考系里。
3. 被光的投影矩阵乘。这个矩阵定义了光的平截体，包括了它的可视域和比率。
4. 缩放和偏移结果。经过了第 1 到 3 步，被变换的顶点的值是从 -1 到 1 的。但是，纹理的索引是从 0 到 1 的，因此变换的结果必须被映射到这个范围。这可以通过先用 1/2 乘以结果的 x、y 和 z 分量，再给每个分量加上 1/2 来实现。一个简单的矩阵乘法就可以实现这个映射。

现在你需要拿出这些矩阵了，这是非常容易的。你已经有了建模矩阵，因为在第 4 章它被用来创建 modelview 矩阵。如果你从一个光源的位置渲染一个场景，光的观察和投影矩阵是你将要用到的。而缩放和偏移矩阵只不过是一些简单常量的集合。

9.3.3 投影纹理贴图的代码

一旦确信你的应用程序已经把正确的纹理矩阵传递给了顶点程序，你只需要在顶点和片段程序中增加一行代码。

在顶点程序中，你必须从物体空间位置计算投影纹理坐标：

```
float4 texCoordProj = mul ( textureMatrix, position);
```

片段程序接收到经过插值的四元纹理坐标集。在使用纹理坐标集来存储一个 2D 纹理的时候，tex2Dproj 标准库函数首先用 q 纹理坐标（第四分量）除以 s 和 t 纹理坐标（第一和第二分量）：

```
float4 textureColor = tex2Dproj (projTexture,
texCoordProj);
```

注意完整的四元纹理坐标被传给了 tex2Dproj，其中 q 分量作为第 4 个分量。

一、顶点程序

示例 9-5 显示了实现投影贴图的完整的顶点程序。

示例 9-5 C9E5v_projTex 顶点程序

```
void C9E5v_projTex(float4 position : POSITION,
                     float3 normal   : NORMAL,
                     out float4 oPosition      : POSITION,
                     out float4 texCoordProj   : TEXCOORD0,
                     out float4 diffuseLighting : COLOR,
                     uniform float Kd,
                     uniform float4x4 modelViewProj,
                     uniform float3 lightPosition,
                     uniform float4x4 textureMatrix)
{
    oPosition = mul(modelViewProj, position);

    // Compute texture coordinates for
    // querying the projective texture
    texCoordProj = mul(textureMatrix, position);

    // Compute diffuse lighting
    float3 N = normalize(normal);
    float3 L = normalize(lightPosition - position.xyz);
    diffuseLighting = Kd * max(dot(N, L), 0);
}
```

二、片段程序

示例 9-6 显示了对应的片段程序。

示例 9-6 C9E6f_projTex 片段程序

```
void C9E6f_projTex(float4 texCoordProj : TEXCOORD0,
                    float4 diffuseLighting : COLOR,
                    out float4 color : COLOR,
uniform sampler2D projectiveMap)
{
    // Fetch color from the projective texture
    float4 textureColor = tex2Dproj(projectiveMap,
                                     texCoordProj);

    color = textureColor * diffuseLighting;
}
```

三、值得注意的问题

投影贴图有两个问题你应该知道。首先，投影体图没有遮挡检查。这意味着像用幻灯机那样在三角形之间产生阴影的直觉想法是不存在的——投影纹理被应用在光的平截体内的每一个三角形。考虑到硬件只是根据给定的矩阵简单地变换顶点，并用变换后的位置来查询一个纹理，这种现象就不应该让人很惊奇了。当硬件变换每个三角形的时候，它没有这个三角形和其他三角形之间的关系的知识。

第二个问题是当 q 坐标为负数的时候，后投影这种人为现象可能会出现。后投影（Back-Projection）是指纹理被投影在光源（投影仪）后面的表面上。在许多较旧的图形处理器上，纹理插值在这种情况下将产生没有定义过的结果。

这里有几种方法可以避免当 q 是负值时的人为现象：

- 使用选择只绘制那些在灯源前面的几何体。
- 使用剪裁平面来删除在灯源后面的几何体。
- 把后投影因子放入一个 3D 衰减纹理。
- 使用一个片段程序来检查 q 是不是负值。

前两种方法实现起来很麻烦：片段程序的解决方法虽然简单有效但需要高级的片段 profile。仅仅检查 q 坐标的值。如果 q 是负的，你可以忽略投影纹理的计算

并只输出黑色。在这种情况下，你能够重写在 C9E6f_projTex 中的纹理查找：

```
float4 textureColor = texCoordProj.w < 0 ? // q 是否小于 0
    0 : // 是, 返回 0
    tex2Dproj (projTexture, texCoordProj); // 否, 对纹理进行采样
```

9.4 阴影映射

如果你曾经用过 OpenGL 或 Direct3D，你应该知道在这些 API 中光照并不自动产生阴影。有许多种方法可以用来计算阴影，但是我们只打算看一下这些流行方法中的一种被称为阴影映射（shadow mapping）的方法。图 9-8 显示了给一个简单的场景增加阴影后的效果。阴影可以帮助建立球体和立方体的相对位置关系。

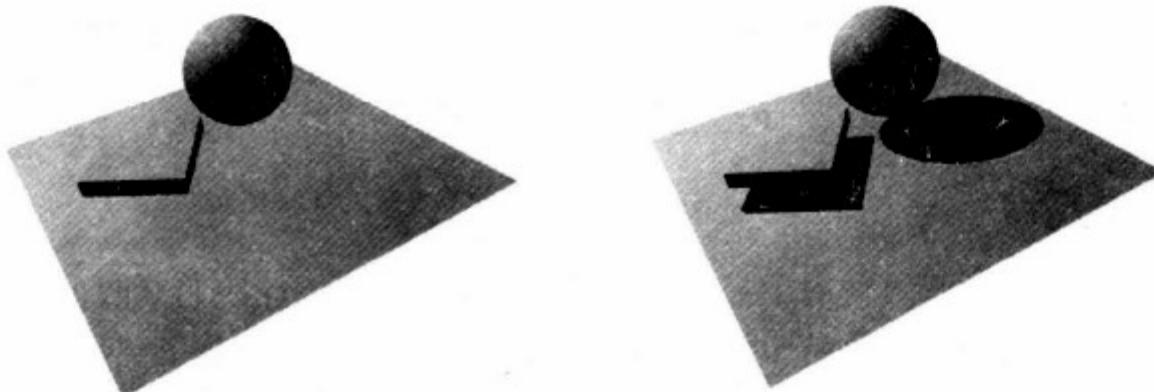


图 9-8 使用阴影映射给一个场景增加阴影

我们有两个原因需要介绍阴影映射。首先，它是基于我们刚刚介绍的投影纹理的概念。其次，最新的图形硬件的片段编程能力让你能够比以前更加精确地控制阴影映射。

阴影映射是一个双过程的技术：

1. 首先，场景以光源的位置为视点被渲染。每个渲染图像的像素的深度值被记录在一个“深度纹理”中（这个纹理通常被称为阴影贴图）。
2. 然后，场景从眼睛的位置渲染，但是用标准的投影纹理把阴影贴图从灯的位置投影到场景中。在每个像素，深度采样值（从被投影的阴影贴图纹理）与片段到灯的距离进行比较。如果后者大，该像素就不是最靠近灯源的表面。这意味着这个片段是阴影，它在着色过程中不应该接受光照。

图 9-9 举例说明了阴影映射的深度比较。在图的左边，正要被着色的点 P 在阴影中，因为这个点的深度值 (Z_B) 比记录在阴影贴图中的深度值 (Z_A) 大。相反，在图的右边显示了点 P 的深度值与在阴影贴图中记录的值相同的情况。这意味着在 P 和灯源之间没有任何物体，因此 P 不在阴影中。

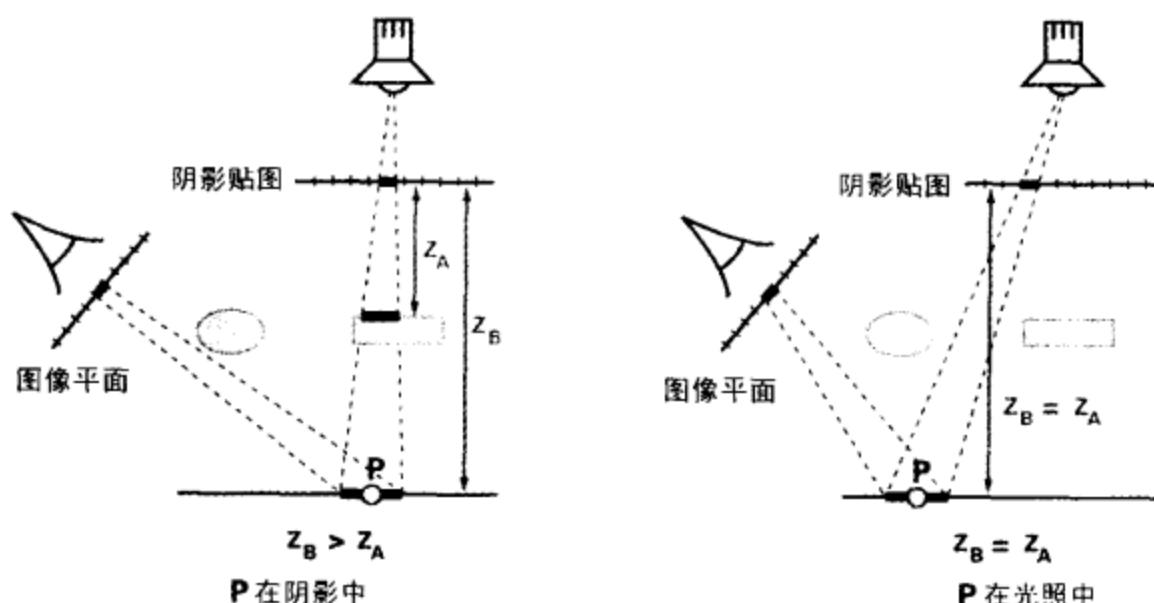


图 9-9 阴影映射的深度比较

阴影映射使用了与投影纹理相同的设置，因此阴影贴图也用 $(s/q, t/q)$ 来做索引。因为对阴影贴图来说光源是投影的中心， r/q 存储了到灯源的距离。因此，通过比较 r/q 与阴影贴图中 $(s/q, t/q)$ 位置的纹理元素的深度值，你就可以决定当前的像素是被照亮的，还是在阴影中了。

当你在一个阴影贴图上进行一个投影纹理查询，硬件自动为你执行比较：`tex2Dproj` 函数返回一个值表示当前的像素是如何被“照亮”的。更确切地说，`tex2Dproj` 函数返回一个 $(c, c, c, 1)$ 形式的四元向量，其中，如果这个像素在阴影中 c 为 0，如果这个像素被照亮 C 为 1。然后你就可以把这个向量当成一种颜色。如果双线性纹理过滤被设置了， c 将从 0 到 1 的范围取值，而不是只限制在 0 和 1 两个值。过滤在阴影边界的地方非常有用，在阴影的边界发生了从完全被遮住到完全被照亮的过渡。在这些情况下，过滤可以帮助阴影看起来更柔和，并减少边的走样（锯齿）。

下面的代码示例显示了你将如何在一个片段程序中使用阴影贴图。在这个例子中，漫反射光照使用阴影贴图查询的结果进行调整：

```

// 照常计算漫反射光照

// 从阴影贴图中取得深度值
float4 shadowCoeff; // 0 代表“在阴影中”
// 1 代表“不在阴影中”
shadowCoeff = tex2Dproj(shadowMap,
texCoordProj);

float4 shadowedColor = shadowCoeff *
diffuseLighting;

```

注意用于阴影贴图查询的代码实际上和在示例 9-6 中的投影纹理查找的代码是相同的。对一名 Cg 程序员来说，阴影贴图和其他的投影纹理很像，除了它不返回一个纹理颜色，**tex2Dproj** 返回一个值表示了阴影的覆盖之外。当一个纹理是深度纹理时，底层的硬件实现自动识别和执行阴影贴图比较来代替普通的纹理提取。

扩展

使用固定功能流水线的传统的阴影映射能够为你自动处理阴影贴图比较。但是使用片段程序，你能够完全控制这个操作并用它来创建一些新效果。

阴影的生成是一个非常难的论题，而且它已经成为了许多研究论文和科研项目主题。本章最后的“补充阅读”部分列出了一些资源，你可以用来学习更多有关阴影映射和其他产生阴影的方法。当你做实验的时候，你可能会发现集成到自己项目中的新技术能够给它们一个独特的外表。

9.5 合成

可用于高级别的三维表面着色的片段处理能力同样能够被用于 2D 图像。应用程序例如绘画程序和数字照片浏览器已经开始使用复杂的 2D 处理。并且现在大部分使用计算机图形的电影非常依赖于合成多个图像的能力，有时候会从差别非常大的源图像生成一个单一的最后图像。甚至游戏也能从用作它们原来的 3D 输出的后处理的快速和高质量的图像效果中受益。

使用 alpha 混合技术的简单合成已经出现许多年了，在 Direct3D 和 OpenGL 中都展示了这种能力。当渲染单独的 3D 多边形到一个部分填充的帧缓冲器的时候，这种技术被大量使用，但它也能够用于 2D 图像处理。但是 alpha 混合的精度

受限于定点数学，并且这个限制在合成的过程中有时候会导致严重的错误。

最近的图形处理器已经引入了浮点屏外缓冲器，允许非常复杂的合成效果，而且在本质上消除了关于精度的损失的关注。浮点缓冲器不使用硬件的 alpha 混合，而依赖于片段着色流水线来代替。浮点缓冲器被当作浮点纹理引入片段流水线。因此，你可以用片段程序操作来合成图像，并把最后的处理结果写到帧缓冲器中。片段程序不但允许我们执行所有以前在 alpha 混合中可以执行的操作，而且可以执行任何我们能够想象得到的面向片段的处理。



CineFX 构架提供了四元浮点数学操作和帧缓冲器格式。每个分量被当作一个工业标准（IEEE）的 32 位浮点值来处理和存储。

9.5.1 把输入映射到输出像素

当使用片段程序来执行 2D 图像处理的时候，我们实际上仍然在做 3D 成像。我们渲染多边形直接和屏幕对齐，然后覆盖在屏幕上。我们直接把 2D 片段程序应用到这个表面上。

最常用的方法是用一个简单的与屏幕对齐的矩形（两个三角形）覆盖屏幕。如果你定义了一个 3D 单位大小的正方形，所有的点在 x 和 y 轴范围从 -0.5 到 0.5，而 z 坐标为 0，然后示例 9-7 的 Cg 顶点程序将把它和屏幕对齐。

注意根据你所使用的图形 API，你也许不得不增加一些偏移量来正确地对齐图像。例如，在示例 9-7 中， $0.5/\text{imgWidth}$ 和 $0.5/\text{imgHeight}$ 是在 Direct3D 中对齐图像的微小的偏移量。这是必须的，因为在 Direct3D 中，纹理坐标位于每个纹理元素的中心，而不是以一个角为中心。这个微小的平移很少能够被肉眼看到，但是，如果你企图合成多个成像过程的结果，它能够产生很多问题。每个过程如果没有对准，将会偏移半个纹理元素，这本身能产生一些很酷的效果。但是如果是非故意的，这通常是令人讨厌的。

示例 9-7 C9E7v_screenAlign 顶点程序

```
void C9E7v_screenAlign(float3 position : POSITION,  
                      float4 texCoord : TEXCOORD0,  
  
                      out float4 oPosition : POSITION,  
                      out float4 oTexCoord : TEXCOORD0,
```

```

        uniform float imgWidth,
        uniform float imgHeight)
{
    float4 screen = float4(position.x, position.y,
                           2.0, 2.0);
    oPosition = screen;
    oTexCoord = float4(0.5 * (1.0 + screen.x/screen.w) +
                       (0.5/imgWidth),
                       0.5 * (1.0 + screen.y/screen.w) +
                       (0.5/imgHeight),
                       0.0,
                       1.0);
}

```

9.5.2 基本的合成操作

一旦我们已经对齐了屏幕形状的表面，并且在上面放置了纹理坐标，我们就可以合成任何我们喜欢的输入图像了。

要直接拷贝一个图像到屏幕（当你操作的时候，潜在地缩放它），你可以使用示例 9-8 所示的一个非常简单的片段程序（它包括一个染色的颜色，只是为了好玩）。

示例 9-8 C9E8f_tint 片段程序

```

void C9E8f_tint(float3 position : POSITION,
                 float2 texCoord : TEXCOORD0,
                 out float4 result : COLOR,
                 uniform sampler2D colorMap,
                 uniform float4 tintColor)
{
    result = tintColor * tex2D(colorMap, texCoord);
}

```

示例 9-9 增加了第二幅图像来打开合成的领域。

OneFX

注意示例 9.9 和随后的其他合成示例只能在高级的 profile 上工作，虽然你能够很容易地通过复制纹理坐标集为基本的 profile 重写它们。然后，你将用它自己的纹理坐标集来存取样本（即使实际上两个纹理坐标集是完全一样的），这在基本的 profile 里是允许的。

示例 9-9 C9E9f_a_over_b 片段程序

```
void C9E9f_a_over_b(float3 position : POSITION,
                     float2 texCoord : TEXCOORD0,

                     out float4 result : COLOR,

                     uniform sampler2D imageA,
                     uniform sampler2D imageB)
{
    float4 aPixel = tex2D(imageA, texCoord);
    float4 bPixel = tex2D(imageB, texCoord);
    result = (aPixel.w * aPixel) +
              ((1.0 - aPixel.w) * bPixel);
}
```

这个片段程序基于没有预乘 alpha 的输入值（有时候也叫非预乘 alpha 值），实现了一个典型的合成“over”操作符。换句话说，imageA 的像素的 RGB 部分还没有和 alpha 值结合在一起。有时候，特别是当处理经过反走样的图像的时候，alpha 和 RGB 分量已经被缩放过了，或预乘过了。如果 imageA 已经用 alpha 预乘过了，我们需要稍微地修改一下公式：

```
float4 result = aPixel + ((1.0 - aPixel.w) * bPixel);
```

注意既然在这种情况下我们已经忽略了 imageB 的 alpha 值，imageB 是否被预乘 alpha 值或后乘 alpha 值是没有关系的——除非你打算在进一步的合成过程里使用操作的结果。混合预乘和非预乘 alpha 值的图像时要小心！

只进行稍微地改动，你就可以创建其他标准的合成操作，例如下面：

- **In:** 只在 imageA 与 imageB 重叠的时候，显示 imageA，如示例 9-10 所示。
- **Out:** 只有当 imageA 与 imageB 不重叠的时候，显示 imageA，如示例 9-11 所示。
- **Dissolve:** 混合两个图像（也被称为“additive blending”或只是“mix”），如示例 9-12 所示。

把这些简单的像素到像素的操作组合起来能够创建近乎无限变化的效果。例如，当使用一个类似 dissolve 的操作 lerp 到 imageA 的 alpha 通道时，你也许想要用 imageA 覆盖 imageB。你能够用通道扭曲直接来为 alpha 创建新的“色度键（chroma-key）”值。在直接合成之前和之后，你也能够对所输入图像的颜色和外表执行有用的修改。应用在 2D 的对输入和最后的图像的颜色的修改使你能够存取更广泛的图像效果，并且不需要修改你底层的 3D 模型和渲染流水线。

示例 9-10 C9E10f_a_in_b 片段程序

```
void C9E10f_a_in_b(float3 position : POSITION,
                    float2 texCoord : TEXCOORD0,
                    out float4 result : COLOR,
                    uniform sampler2D imageA,
                    uniform sampler2D imageB)
{
    float4 aPixel = tex2D(imageA, texCoord);
    float4 bPixel = tex2D(imageB, texCoord);
    result = bPixel.w * aPixel;
}
```

示例 9-11 C9E11f_a_out_b 片段程序

```
void C9E11f_a_out_b(float3 position : POSITION,
                    float2 texCoord : TEXCOORD0,
                    out float4 result : COLOR,
                    uniform sampler2D imageA,
                    uniform sampler2D imageB)
{
    float4 aPixel = tex2D(imageA, texCoord);
    float4 bPixel = tex2D(imageB, texCoord);
    result = (1.0 - bPixel.w) * aPixel;
}
```

示例 9-12 C9E10f_a_dissolve_b 片段程序

```

void C9E12f_a_dissolve_b(float3 position : POSITION,
                        float2 texCoord : TEXCOORD0,

                        out float4 result : COLOR,

                        uniform sampler2D imageA,
                        uniform sampler2D imageB,
                        uniform float dissolve)
{
    float4 aPixel = tex2D(imageA, texCoord);
    float4 bPixel = tex2D(imageB, texCoord);
    result = lerp(aPixel, bPixel, dissolve);
}

```

9.6 练习

1. 你自己尝试一下：深度线索是一种类似于雾的非真实技术。两种技术用从观察者的距离来减弱外观的颜色。不像使用指数方程的均匀的雾，深度线索使用了一种更简单的线性衰减。深度线索能够帮助一个 CAD 设计人员观察复杂的线框模型，分辨哪条线更近哪条线更远。修改 C9E1f_fog 和 C9E2v_fog 示例来实现深度线索。提示：例如，OpenGL 通过它的 GL_LINEAR 雾模式支持深度线索。在这个模式中，公式 9-1 中的 f 按如下没有指数的方式计算：

$$f = \max(0, \min(1, (e-z)/(e-s)))$$

其中 s 是在眼空间中雾开始的地方（在 OpenGL 中的 GL_FOG_START），而 e 是雾结束的地方（在 OpenGL 中的 GL_FOG_END）。

2. 回答这个问题：描述一个均匀的雾的情况，在用欧拉距离函数 length 计算每个顶点的距离的时候会产生人为效果。提示：考虑一下当两个点离眼睛的距离相同，但这两个点相距很远的情况。

3. 你自己尝试一下：修改 C9E4f_toonShading 示例使得程序使用一个单独的 2D 纹理索引，其中漫反射分量用作 s 而镜面反射分量用作 t ，而不是使用两个 1D 纹理——一个用于漫反射，另一个用于镜面反射。

4. 你自己尝试一下：使用投影纹理来创建一个以模糊的五角星为形状的聚光

灯图案。把第 5 章的漫反射和镜面反射光照模型于来自投影纹理的聚光灯衰减项结合在一起。

5. 你自己尝试一下：尝试把练习 4 的结果和阴影映射结合在一起。
6. 你自己尝试一下：你在 9.5.1 小节看到重映射一幅图像来精确地匹配输出屏幕时，可以通过一个非常简短的顶点程序来定义 s-t 纹理坐标来实现。扩展这样一个顶点程序来包括到 2D 图像的标准 2D 仿射变换，例如缩放、平移和旋转。

9.7 补充阅读

要想学习更多有关雾的知识，请阅读由 David S. Ebert 编辑的由 Ken Musgrave、Larry Gritz 和 Steven Worley 撰写的“*Texturing and Modeling: A Procedural Approach*”第二版（Academic Press, 1998 年）的“*Atmospheric Models*”章节。有关大气效果的经典计算机图形学论著是 Victor Klassen 的“*Modeling the Effect of the Atmosphere on Light*”，发表在 1987 年的一期 *ACM Transactions on Graphics* 中。

当你放松拥有统一雾的颜色和密度的限制的时候，雾将更加有趣。Justin Legakis 在 1998 年的 SIGGRAPH 上展示了一个被称为“Fast Multi-Layer Fog”的技术梗概（ACM Press），在文中他假设雾的密度随高度变化。

Tomoyuki Nishita 和他的合作者发表了大量关于自然现象的论文，特别是大气效果，例如一束光。他们的许多工作都可以很容易地用到 Cg 程序中。他的网站是 <http://nis-lab.is.s.u-tokyo.ac.jp/~nis>。

在 2001 年，Amy 和 Bruce Gooch 撰写了“*Non-Photorealistic Rendering*”（A.K. Peters），非常详细地描述了卡通渲染，以及许多其他非真实渲染技术。Thomas Strothotte 和 Stefan Schlechtweg 撰写了“*Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation*”（Morgan Kaufmann, 2002 年）。

Mark Segal、Carl Korobkin、Rolf van Widenfelt、Jim Foran 和 Paul Haeberli 在他们的 1992 年 SIGGRAPH 论文“*Fast Shadows and Lighting Effects Using Texture Mapping*”（ACM Press）为图形硬件介绍了投影纹理。这篇论文还描述了硬件纹理映射作为投影纹理的扩展。

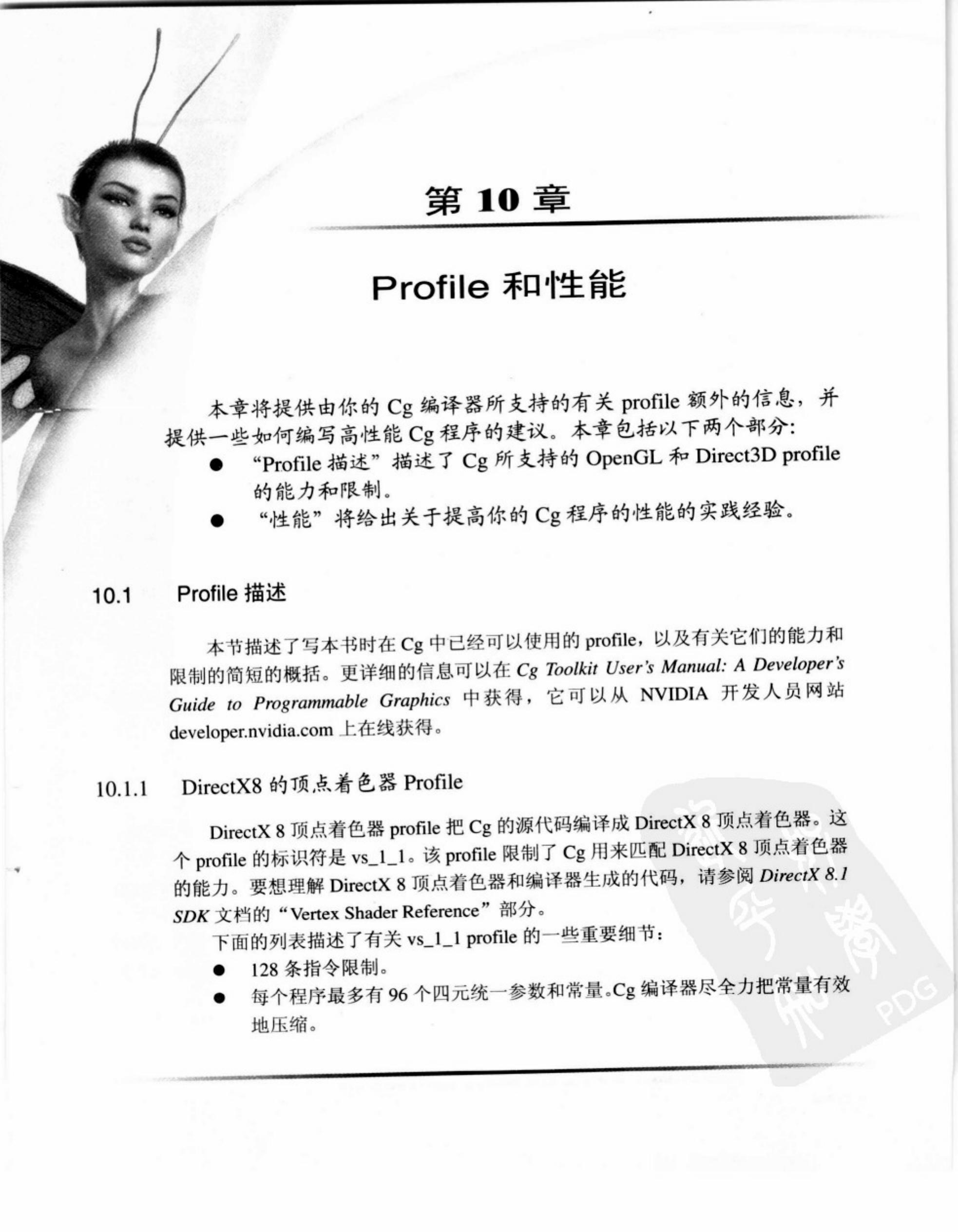
Lance Williams 1978 年最先在他的经典的 SIGGRAPH 论文“*Casting Curved Shadows on Curved Surfaces*”（ACM Press）里描述了阴影映射。William Reeves、David Salesin 和 Robert Cook 1987 年在他们的 SIGGRAPH 论文“*Rendering*

“Antialiased Shadows with Depth Maps” (ACM Press) 里描述了他们在 Pixar 关于阴影映射的工作。

OpenGL 1.4 在 2002 年为图形硬件标准化了阴影映射。Cass Everitt、Ashu Rege 和 Cem Cebenoyan 在 2002 年为 OpenGL 和 Direct3D 发表了 “Hardware Shadow Mapping”。有关的白皮书可以在 NVIDIA 的开发者网站上获得 (developer.nvidia.com)。

由 Andrew Woo、Pierre Poulin 和 Alain Fournier 撰写的 “A Survey of Shadow Algorithms” 是一篇学习更多有关阴影和阴影算法的非常优秀的论文。这篇论文发表在 1990 年 *IEEE Computer Graphics and Applications*。

Thomas Porter 和 Tom Duff 在 1984 年的 SIGGRAPH 上发表了 “Compositing Digital Images” (ACM Press)，把合成引入了图形学领域。最近，Ron Brinkmann 发表了 “The Art and Science of Digital Compositing” (Morgan Kaufmann, 1999 年)。



第 10 章

Profile 和性能

本章将提供由你的 Cg 编译器所支持的有关 profile 额外的信息，并提供一些如何编写高性能 Cg 程序的建议。本章包括以下两个部分：

- “Profile 描述” 描述了 Cg 所支持的 OpenGL 和 Direct3D profile 的能力和限制。
- “性能” 将给出关于提高你的 Cg 程序的性能的实践经验。

10.1 Profile 描述

本节描述了写本书时在 Cg 中已经可以使用的 profile，以及有关它们的能力和限制的简短的概括。更详细的信息可以在 *Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics* 中获得，它可以从 NVIDIA 开发人员网站 developer.nvidia.com 上在线获得。

10.1.1 DirectX8 的顶点着色器 Profile

DirectX 8 顶点着色器 profile 把 Cg 的源代码编译成 DirectX 8 顶点着色器。这个 profile 的标识符是 vs_1_1。该 profile 限制了 Cg 用来匹配 DirectX 8 顶点着色器的能力。要想理解 DirectX 8 顶点着色器和编译器生成的代码，请参阅 *DirectX 8.1 SDK* 文档的“Vertex Shader Reference”部分。

下面的列表描述了有关 vs_1_1 profile 的一些重要细节：

- 128 条指令限制。
- 每个程序最多有 96 个四元统一参数和常量。Cg 编译器尽全力把常量有效地压缩。

- if 语句和?: 操作符被当作条件赋值。
- while、do 和 for 语句仅当它们定义的循环可以被编译器解开的时候才被允许（这也就是说，如果编译器能够决定在循环里的重复次数）。
- 这个 profile 把所有的连续数据类型（例如 float）当作 32 位浮点值。
- 这个 profile 允许数组的变量索引，只要数组是一个统一常量。该 profile 不支持用变量索引写入数组。
- 没有纹理的采样。
- 可以任意进行混合和取反，而没有任何性能惩罚。

10.1.2 OpenGL 的基本 NVIDIA 顶点程序 Profile

NVIDIA 顶点程序 profile 编译 Cg 源代码成 NVIDIA 和 Mesa 支持的 NV_vertex_program OpenGL 扩展兼容的顶点程序。这个 profile 的标识符是 vp20。profile 限制了 Cg 用来匹配 NV_vertex_program OpenGL 扩展的能力。要理解 NVIDIA 顶点程序的能力，参考该扩展的在线说明。

该 profile 的功能与前面讨论的 vs_1_1 profile 是等同的。关于 vs_1_1 profile 的细节同样适用于 vp20 profile。

为了 vp20 程序能够正确运行，在 vp20 profile 中的常量必须被加载到正确的程序参数寄存器中。注意使用 Cg 运行库来确保正确的常量被加载。

这个 profile 不像 vs_1_1 profile，同时为朝前和朝后的主要和次要的颜色提供了输出语义，用来支持两面光照。

10.1.3 OpenGL 的 ARB 顶点程序 Profile

OpenGL 架构评审小组（Architecture Review Board, ARB）顶点程序 profile，把 Cg 源代码编译成与 ARB_vertex_program 多厂商 OpenGL 扩展兼容的顶点程序。这个 profile 的标识符是 arbvp1。profile 限制了 Cg 用来匹配 ARB_vertex_program OpenGL 扩展的能力。要了解 ARB 顶点程序的能力，请参考该扩展的在线说明。

与 vp20 profile 一样，该 profile 和前面讨论过的 vs_1_1 profile 在功能上是相同的。关于 vs_1_1 profile 的细节同样适用于 arbvp1。就像 vp20 一样，arbvp1 支持输出向前和向后的颜色。

不像 vp20 profile，arbvp1 profile 允许 Cg 程序直接引用 OpenGL 状态。但是，

如果你想编写与 vp20 和 vs_1_1 profile 兼容的顶点程序，你必须使用 Cg 运行库用所需的状态来设置统一变量的替换机制。另外，当绑定到 arbvp1 的程序的时候，直接引用 OpenGL 状态将导致轻微的驱动程序验证惩罚。

10.1.4 DirectX 9 的顶点着色器 Profile

DirectX 9 顶点着色器 profile 把 Cg 源代码编译成 DirectX 9 顶点着色器。这个 profile 的标识符是 vs_2_0 或 vs_2_x。这些 profile 限制了 Cg 用来匹配 DirectX 9 顶点着色器的能力。vs_2_x profile 是 vs_2_0 profile 的扩展版本；最主要的增强是支持了动态分支。要想了解 DirectX 9 顶点着色器的能力和由编译器生成的代码，请参考 *DirectX 9 SDK* 文档的“Vertex Shader Reference”部分。

这些 profile 比 vs_1_1 profile 支持更多的指令和临时变量。vs_2_x profile 还支持一般化的循环和条件语句。

10.1.5 OpenGL 高级 NVIDIA 顶点程序 Profile

高级的 NVIDIA 顶点程序 profile，把 Cg 源代码编译成与 NVIDIA 的 CineFX 构架所支持的 NV_vertex_program2 OpenGL 扩展兼容的顶点程序。这个 profile 的标识符是 vp30。该 profile 限制了 Cg 用来匹配 NV_vertex_program2 OpenGL 扩展的能力。

这个 profile 是 vp20 profile 的一个超集。任何用 vp20 profile 编译的程序都应该可以用 vp30 profile 编译。和 vs_2_x profile 一样，vp30 profile 比 vp20 profile 支持更多的指令和临时变量。vp30 profile 还支持一般化的循环和条件语句。

10.1.6 DirectX 8 的像素着色器 Profile

DirectX 8 像素着色器 profile 把 Cg 源代码编译成 DirectX 8.1 像素着色器。Cg 为像素着色器 1.1 版、1.2 版和 1.3 版分别提供了 profile。此 profile 的标识符是 PS-1-1、PS-1-2 和 PS-1-3。每个 profile 都限制了 Cg 用来分别匹配各个 DirectX 8 像素着色器版本的能力。要想了解 DirectX 8 像素着色器和编译器生成的代码，请参考 *DirectX 8.1 SDK* 文档的“Pixel Shader Reference”部分。

下面的列表描述了有关 ps_1_1、ps_1_2 和 ps_1_3 profile 的一些重要细节：

- 最多只能有 4 个纹理操作。
- 最多只能有 8 个数学操作。
- 纹理存取操作必须在数学操作之前。
- while、do 和 for 语句只有在它们所定义的循环可以被编译器解开的情况下才被允许。
- 这个 profile 把所有的连续数据类型(例如 float)表示成限制在范围[-1, 1]、[-2, 2]或[-8, 8]内的有符号值(取决于底层的图形处理器和 Direct3D 的能力)。
- 受限制的重组能力:
 - .x/.r、.y/.g、.z/.b、.w/.a
 - .xy/.rg、.xyz/.rgb 和.xyzw/.rgba
 - .xxx/.rrr、.yyy/.ggg、.zzz/.bbb、.www/.aaa
 - .xxxx/.rrrr、.yyyy/.gggg、.zzzz/.bbbb、.wwww/.aaaa
- 仅有有限版本的 Cg 标准库函数被支持。
- 只有一部分的数学操作被允许:
 - 修饰符(参看 DirectX 8.1 SDK 文档)。
 - 二元操作符(+、-和*)。
 - 内置函数(dot、lerp、saturate)。
- 这个 profile 不支持使用变量索引的数组。

10.1.7 用于 OpenGL 的基本 NVIDIA 片段程序 Profile

用于 OpenGL 的基本 NVIDIA 片段程序 profile, 能够把 Cg 源代码编译成 NV_register_combiner2 和 NV_texture_shader OpenGL 扩展支持的寄存器合成器和纹理着色器配置。这个 profile 的标识符是 fp20。要想了解该 profile 的能力, 请参考 NV_register_combiners、NV_register_combiners2、NV_texture_shader、NV_texture_shader2 和 NV_texture_shader3 OpenGL 扩展说明。由编译器为这个 profile 生成的代码符合 NVIDIA 的 nvparse 格式。

该 profile 的能力比 ps_1_3 profile 的能力高级一点。这个 profile 支持 samplerRECT 数据类型, 通过多厂商的 NV_texture_rectangle OpenGL 扩展可以存取非 2 次幂大小的纹理。

10.1.8 DirectX 9 像素着色器 Profile

DirectX 9 像素着色器 profile 能够把 Cg 源代码编译成 DirectX 9 像素着色器。这个 profile 的标识符是 ps_2_0 或 ps_2_x。这些 profile 限制了 Cg 用来匹配 DirectX 9 顶点着色器的能力。ps_2_x profile 是 ps_2_0 profile 的扩展版本；主要的增强是支持了更多的指令和纹理存取。要想了解 DirectX 9 顶点着色器和编译器生成的代码，请参考 *DirectX SDK* 文档的“Pixel Shader Reference”部分。

ps_2_0 或 ps_2_x profile 的关键能力和限制如下所示：

- 每个片段的浮点操作。
- 最多到 16 个纹理单元，对 ps_2_x 没有纹理取指令数目的限制。
- 对 ps_2_0，最多有 4 级的依赖纹理存取。对 ps_2_x 则没有依赖纹理存取的限制。
- 任意的重组和取反。
- 通用的比较和布尔操作。
- 对梯度指令可选择的存取。
- 数组的变量索引是不允许的（使用纹理来代替）。
- 位操作符，例如&、|和^是不支持的。

10.1.9 OpenGL 的 ARB 片段程序 Profile

ARB 片段程序 profile，能够把 Cg 源代码编译成与 ARB_fragment_program 多厂商 OpenGL 扩展兼容的片段程序。这个 profile 的标识符是 arbfp1。该 profile 限制了 Cg 用来匹配 ARB_fragment_program OpenGL 扩展的能力。要想了解 ARB 片段程序的能力，请参考该扩展的在线说明。

这个 profile 在功能上和以前讨论过的 ps_2_x profile 是一样的。关于 ps_2_x profile 的细节同样适用于 arbfp1 profile。

10.1.10 OpenGL 高级 NVIDIA 片段程序 Profile

高级的 NVIDIA 片段程序 profile，能够把 Cg 源代码编译成与 NVIDIA 的 CineFX 构架支持的 NV_fragment_program OpenGL 扩展兼容的片段程序。这个 profile 的标识符是 fp30。该 profile 限制了 Cg 用来匹配 NV_fragment_program

OpenGL 扩展的能力。

这个 profile 是写这本书的时候功能最强的片段级 profile。它是 ps_2_x profile 的超集。

除了所有在描写 ps_2_x profile 的小节里提到的功能, fp30 profile 还提供:

- 支持对非 2 次幂大小的纹理采样的 samplerRECT 数据类型。
- 支持范围是[-2, 2]的 fixed 数据类型。
- 支持半精度浮点的 half 数据类型。
- 支持 NV_fragment_program 扩展揭示的额外的指令。

10.2 性能

现代和未来的图形处理器提供的灵活性, 与 Cg 提供的易用性相结合, 使得它非常容易写出冗长的片段程序。一个实际的问题就产生了, 那就是: 你的程序变得越长, 它们执行得就越慢。在大多数情况下, Cg 编译器将帮你照料性能优化。但是它不能为你选择高级算法。因此, 该部分介绍了一些概念和技巧来帮助你从你的程序获得最优的性能。其中一些已经用某种形式或其他方法在本书的前面介绍过了, 但是为了方便参考我们把它们都放在一起并增加了额外的解释。

10.2.1 使用 Cg 标准库

当你看完这本书的示例以后, 你或许会发现 Cg 的标准库函数是非常方便的。但是标准库不仅仅是方便, 它还提供了性能。标准库函数被定义用来允许 Cg 编译器把它们转换成高度优化的汇编代码。在许多情况下, 标准库函数被编译成可以在一个图形处理器时钟周期完成的单汇编指令。这些函数的例子有 sin、cos、lit、dot 和 exp。另外, 使用标准库提供了一定量的未来证明。新版本的标准库将和未来版本的 Cg 一起得到, 这些新版本将为未来的图形处理器进行优化。

虽然有这些优点, 但是仍然会有一些你不想用标准库函数的情况。如果你需要一个函数的简单的版本, 或者是如果你能够编写一个特别快而简单的函数版本来适应你的特别需要, 你也许最好不要使用标准库。这种情况的一个例子(来自第 8 章)是, 使用标准化立方贴图而不是在片段程序中使用标准库的 normalize 函数来正规化向量。

10.2.2 充分利用统一参数

作为统一参数传递给你的顶点和片段程序时，你应该慎重。例如，如果你需要传递一个 `time` 变量给片段程序，而这个程序实际上是使用 `sin (time)`，你最好在你的应用程序里计算一次 `sin (time)`，然后把它作为统一参数传给你的片段程序。这比为每个生成的片段无谓的计算一次 `sin (time)` 要有效得多。即使你的程序同时需要 `time` 和 `sin (time)`，把这两个值作为两个统一参数传递要比只传递 `time` 然后在程序中计算 `sin (time)` 有效得多。

10.2.3 使用顶点程序与片段程序

在本书前面的内容中，你已经看到了片段程序能够比顶点程序生成更精确的图像。如果你的场景镶嵌（tessellated）的不好，顶点程序将产生许多明显地小面片的粗糙图像。但是每个片段的着色并不是免费的。一个片段程序要为每个生成的片段执行一次，因此片段程序每一帧通常要运行几百万次。另一方面，顶点程序每一帧通常只需要运行几万次（假设你的模型没有过度镶嵌）。另外，片段程序的循环和分支能力比顶点程序更受限制，虽然随着时间的推移它们的能力会得到提高。特别是，第三代和早期的图形处理器的片段程序是非常受限制的。

每次你编写一个 Cg 程序，你不得不决定如何在顶点程序和片段程序之间分配程序的工作量。一个简单的单凭经验的方法是使用顶点程序执行基本变换和纹理坐标操作，而依赖片段程序来进行剩余的计算。一般而言，这是一种好方法，因为它能够让你把最费时的数学操作移到顶点程序。通过平衡图形处理器的顶点和片段处理器之间的工作量，你可以预防它们中的一种变成一个严重的瓶颈。

在某些情况下，你也许想要为性能牺牲一些图像质量。在这种情况下，把一些计算移到顶点程序是明智的，如果这么做了就将通过减小片段程序的长度来平衡程序的工作量。图像质量的下降也许会比你预期的要小，因为插值经常工作得很好。如果你插值的量是线性变化的，当使用顶点程序代替片段程序时你将一点也看不到图像质量的下降。

另一种决定何时使用一个顶点程序或一个片段程序的方法是，考虑一下你的程序参数变化的频率。如果参数变化的非常慢（例如，在漫反射光照中），在顶点程序中进行计算或许已经足够了。但是，如果涉及到了变化非常快的参数（例如，

在镜面反射光照中), 你将需要一个片段程序来获得精确的结果。

10.2.4 数据类型和它们对性能的影响

许多 Cg 的 profile 支持各种各样的数据类型。要获得最优的性能, 你需要使用适合任何特殊计算的最小的数据类型。

`fixed` 数据类型对低精度计算非常有用, 例如颜色计算。在 `fp30` 和 `ps_2_x` 片段程序 profile 中, `fixed` 值的范围为 -2 到 +2, 并提供了 12 位固定点的精度。`fixed` 在精度上的放弃换来了性能的提升: 使用 `fixed` 数据类型的计算在速度上经常比它们使用 `half` 和 `float` 类型的对应程序在 CineFX 构架要快上几倍。

当你需要高精度和更广的范围并且你不需要可能的最高精度的时候, `half` 数据类型是非常有用的。在 `fp30` profile 中, `half` 提供了 16 位的浮点精度 (1 位的符号, 10 位的尾数和 5 位的指数), 对许多图形相关的量, 例如颜色和法向量都非常适合。特别是当你处理颜色计算的时候, `half` 类型通常是非常有效的。在 CineFX 构架中, 程序通过使用 `half` 来存储中间值来最小化它们的临时存储空间, 并且比全部使用 `float` 值运行得要快。

当你需要最高可能的范围和精度的时候, 使用 `float` 类型。例如, 在片段级, 当处理最后的纹理坐标和导数的时候, 你应该使用 `float`。

10.2.5 充分利用向量化

用向量类型来执行计算比用单个的标量要有效得多。对大部分操作, 把它们应用到一个向量和把它们应用到一个标量所花费的时间是一样的。这意味着通过谨慎地使用向量, 你可以最高 4 倍加速某些连续的指令。Cg 的重组 (swizzling) 和写屏蔽 (write-masking) 的特性使它非常容易在向量中插入标量和提取标量, 因此当可能的时候, 把你的计算尽量压缩成向量计算是非常有用的。

例如, 回忆来自第 6 章的粒子系统。在顶点程序中, 你用下面的向量化代码来计算粒子的位置:

```
float4 pFinal = pInitial +
    vInitial * t +
    0.5 * acceleration * t * t;
```

这种简洁方法能够比分别计算每个分量的位置快 4 倍:

```
float4 pFinal;
```

```

pFinal.x = pInitial.x + vInitial.x * t + 0.5 * acceleration.x * t * t;
pFinal.y = pInitial.y + vInitial.y * t + 0.5 * acceleration.y * t * t;
pFinal.z = pInitial.z + vInitial.z * t + 0.5 * acceleration.z * t * t;
pFinal.w = pInitial.w + vInitial.w * t + 0.5 * acceleration.w * t * t;

```

在像这样简单的情况下，编译器实际上能够为你对这些代码进行向量化。但是通常，用向量而不是单独分量来计算是非常自然的。充分利用 Cg 提供的方便的向量类型。除了更加有效以外，向量化的代码通常比非向量化代码更加清楚和简洁。

10.2.6 使用纹理来编码函数

一种加速你的片段程序的方法是，通过查找纹理而不是执行数学计算来估算函数。考虑一个用一个浮点值作为输入并返回一个浮点值作为结果的函数。如果输入值的范围是从 0 到 1，你可以使用输入值作为一个 1D 纹理的纹理坐标。查询的结果是一个纹理元素颜色，就是这个函数的结果。

这个技巧并不只限于 1D 纹理。你也可以使用 2D 纹理来编码一个两个变量的函数，或者你可以用一个立方贴图来查询一个 3D 函数。一个好的例子是，你在第 8 章使用的标准化的立方贴图。同样，不要忘记 RGBA 纹理允许你在 4 个分量的每个分量进行编码。请记住，通过对有用的信息（这些信息也许和颜色完全没有关系）在每个通道进行编码，你将经常发现能使你的程序更加有效的机会。

使用纹理来编码函数有许多优点。首先，不需要任何数学操作，这意味着有价值的图形处理器周期被节省了。其次，纹理查询充分利用了图形处理器特有的纹理过滤硬件，它可以自动平滑函数的输出（如果你想要的话）。也就是说，你可以用一个小的纹理来表示一个函数，并允许硬件在纹理元素之间进行插值。第三，纹理给了你编码任意函数的灵活性，因为它们不需要使用一个指定的公式或模式来创建。你甚至可以让一名美工人员描绘一个函数到纹理中，给这名美工更高级别的美感控制。

10.2.7 自由使用重组 (Swizzling) 和取反 (Negation)

在 NVIDIA 图形处理器上的顶点程序提供了变量重组和取反而没有任何性能惩罚。尽量使用这个能力。

CineFX

CineFX 图形处理器在片段级也有同样免费的取反和重组能力。

CineFX 图形处理器在片段级还能免费计算饱和 (saturation, 也就是把取值范围限制在[0, 1]内) 和绝对值。使用 `saturate` 标准库函数来进行饱和计算，使用 `abs` 标准库函数来求绝对值。

10.2.8 只对必须着色的像素进行着色

对于复杂的程序，片段程序趋向成为应用程序中的瓶颈。在这种情况下，确信在不可见的场景部分没有浪费宝贵的图形处理器周期是非常明智的。一种确保这样的好方法是，先不着色绘制一次场景，但打开深度测试。我们把这称为“先放下 z 值 (laying down z first)”。在第一次渲染过程后，帧缓存的每个像素只包含离这个像素最近的表面。在随后的渲染过程中，设置深度测试来只绘制哪些深度值和帧缓存中的深度值相匹配的片段。用这种方法，片段程序只为可见的片段运行，而不是运行于所有潜在可见的片段。

如果你熟悉 OpenGL 或 Direct3D 的话，也许想知道这种方法是如何起作用的，因为深度测试通常发生在流水线的最后一步，在片段程序已经被执行以后。幸运地是，大部分现代图形处理器拥有特别的硬件，能够在片段程序执行之前执行深度测试。这使得当片段程序不需要执行的时候，避免执行片段程序成为可能。

10.2.9 简短的汇编代码并不是更快的性能所必须的

当 Cg 编译器从你的 Cg 源代码创建汇编代码的时候，它尝试生成的代码能够在指定的目标 profile 或图形处理器上最有效地运行。现代图形处理器是非常复杂的，并且也许需要指令事先按指定的方法预定好，以获得最优的性能。在某些情况下，编译器也许会生成额外的指令，使得它输出的汇编代码比由 DirectX 9 的 `fxc` 编译器生成的汇编代码要长一些。在这种情况下，额外的汇编代码运行得更快，因为它们能够并行执行，或者因为它们没有使图形处理器着色器流水线停止的依赖性。

从这个讨论获取的教训是，不要通过编译器输出的汇编代码的长短来判断它的性能。但是，在很多时候越短的编译代码确实运行得越快。最后，最好的估算一个程序性能的方法是测试用你的程序所能得到的帧率。这将使你把所有的理论

放在一边来量化连续的实际的性能。

10.3 练习

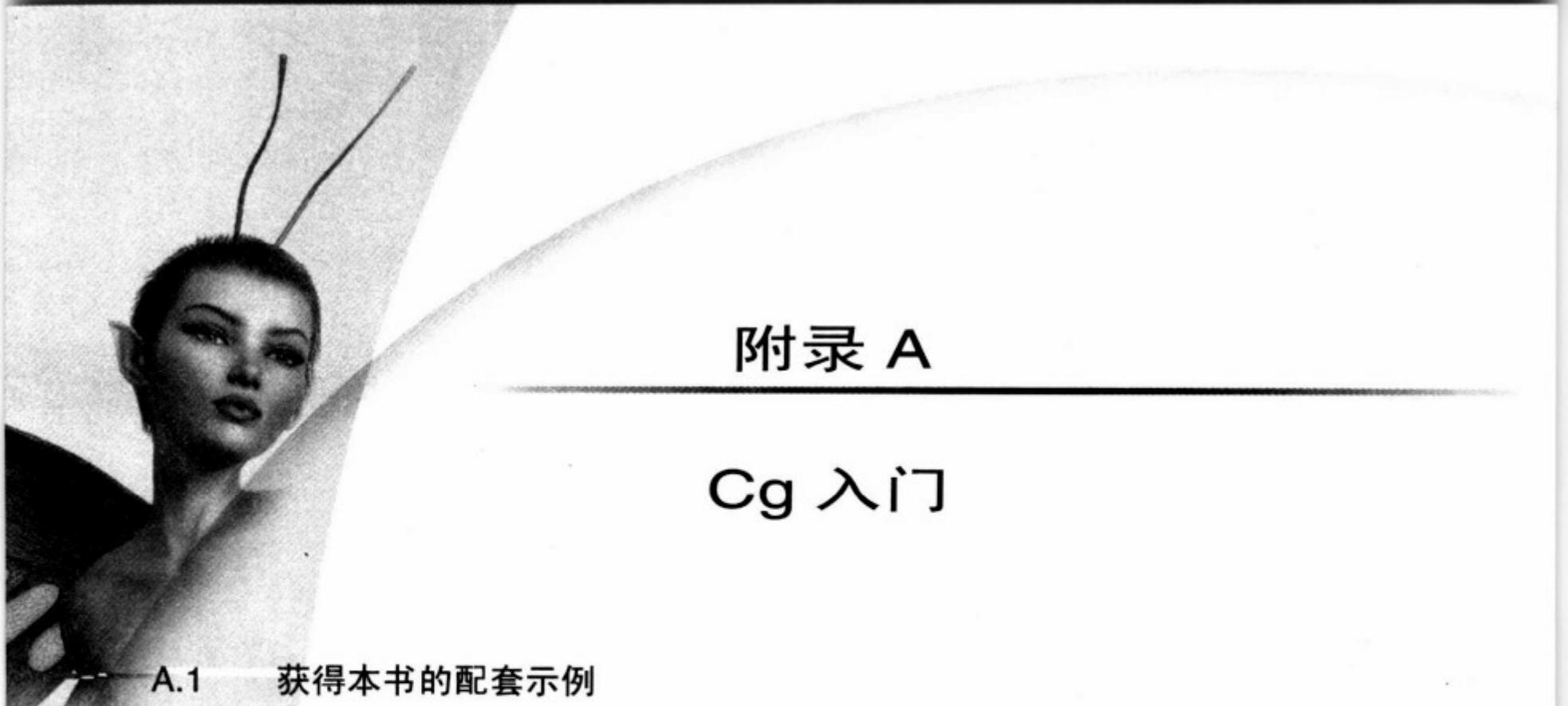
1. 你自己尝试一下：重写本书以前的一些片段示例程序，当为 fp30 和 ps_2_x profile 编译的时候，新程序要充分利用 fixed 数据类型。
2. 回答这个问题：由于图形硬件在不断向前发展，你可以预期功能更强大和通用的 profile。研究一下你的编译器所支持的能够获得的最新的 Cg profile。

10.4 补充阅读

本章所提到的 OpenGL 扩展的规范可以在网上找到，地址是 <http://oss.sgi.com/projects/ogl-sample/registry>。NVIDIA 也发表了 *NVIDIA OpenGL Extension Specifications*，它汇集了所有 NVIDIA OpenGL 驱动程序实现的 OpenGL 规范。在 NVIDIA 开发人员网站 (developer.nvidia.com) 上可以找到解释了 fp20 profile 输出的 nvparse 结构表示的文档。

Microsoft 为 DirectX 8 和 DirectX 9 的顶点和像素着色器的文档在 <http://msdn.microsoft.com/library> 上可以找到，在它的 DirectX SDK 文档中“Graphics and Multimedia”部分提供了详细的说明。





附录 A

Cg 入门

A.1 获得本书的配套示例

本书配套的示例可以从这本书的网站免费下载：

<http://developer.nvidia.com/CgTutorial>

这个网站还会列出任何对本书内容的修改、增补和修正。任何我们可以利用的补充材料都会在这个网站上发布。

A.2 获得 Cg 工具箱

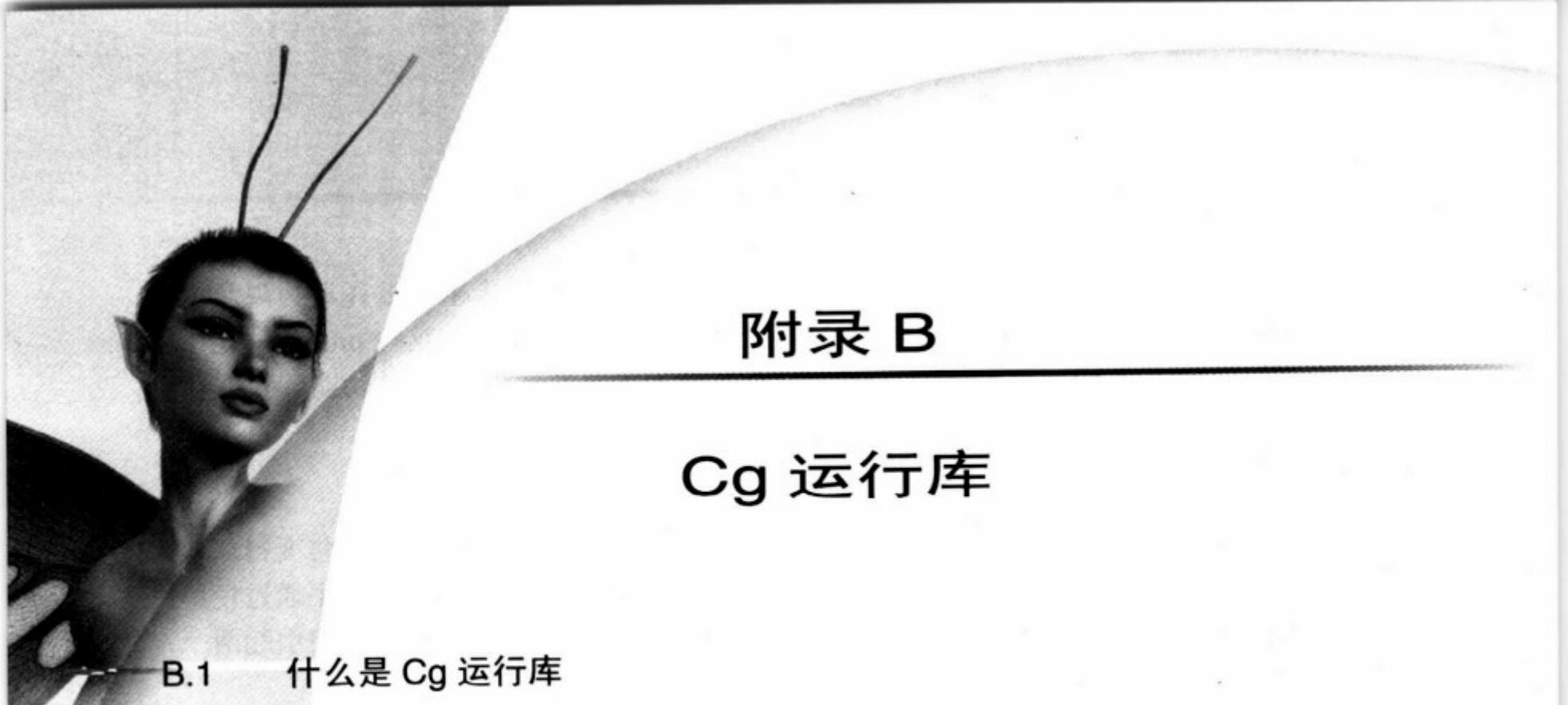
如果你准备把 Cg 加入到你自己的项目中时，你应该下载 NVIDIA 最新版本的 Cg 工具箱。你可以从 NVIDIA 的 Cg 网站得到这个工具箱：

<http://developer.nvidia.com/Cg>

经常浏览这个网站可以获得有关 Cg 语言最新的修改和增补。你也可以通过该网站报告你在发布的软件中发现的缺陷或错误的信息。







附录 B

Cg 运行库

B.1 什么是 Cg 运行库

Cg 程序提供程序给图形处理器，但是它们需要渲染图像的应用程序的支持。为了把 Cg 程序和一个应用程序连接起来，你必须做以下两件事：

1. 用适当的 profile 编译程序。这一步把你的 Cg 程序翻译成和应用程序所使用的 3D 编程接口与底层的硬件相兼容的格式。
2. 把程序连接到应用程序。在这一步允许应用程序为程序的运行进行配置，并为程序提供变化和统一的参数。

你可以选择你什么时候想要执行这些操作。你可以在编译的时候执行它们，也就是当应用程序被编译成可执行文件的时候，或者你可以在运行的时候执行它们，也就是当应用程序正在执行的时候。Cg 运行库是一组应用程序编程接口（API），它允许一个应用程序在运行的时候编译和连接 Cg 程序。

B.2 为什么使用 Cg 运行库

B.2.1 未来的证明

许多应用程序需要运行在提供了多层次功能的不同图形处理器上，因此这些应用程序需要在各种 profile 上运行。如果一个应用程序预编译了它的 Cg 程序（在编译的时候），对每个 profile，它必须为每个程序都存储一个预先编译的版本。虽然这种方法是可行的，但是预先编译的方法对一个使用了很多 Cg 程序的应用程序来说太麻烦了。更糟糕的是，Cg 程序在时间上冻结了。由于预先编译了 Cg 程序，一个应用程序牺牲了未来编译器可能提供的优化。

相反，由应用程序在运行时编译的 Cg 程序有助于未来的编译器为现存的 profile 进行优化。并且这些程序可以在未来的 profile 上运行，这些 profile 对应于在编写应用程序的 Cg 程序时并不存在的、未来的新硬件和 3D API 功能。

B.2.2 不存在依赖问题

如果你连接一个编译过的 Cg 程序到一个应用程序，这个应用程序将变得依赖于编译的结果，特别是依赖于编译器如何分配参数。应用程序将不得不通过使用 Cg 编译器输出的硬件寄存器名来引用 Cg 程序的输入参数。这种方法将引起两个严重的问题：

1. 不检查编译器的输出，寄存器名将不能很容易地匹配到在 Cg 程序中对应的有意义的名字。
2. 寄存器的分配会随着 Cg 程序、Cg 编译器或编译使用的 profile 的每一次改变而改变。这意味着你将不得不每次都更新应用程序，这将非常不方便。

相反，在运行时连接一个 Cg 程序到应用程序将解除对 Cg 编译器的依赖。使用 Cg 运行库，当你增加、删除或重命名 Cg 的输入参数时，你只需要改变应用程序的代码。

B.2.3 输入参数管理

Cg 运行库还提供了工具用来管理 Cg 程序的输入参数。特别是，它使像数组和矩阵这样的数据类型变得更容易处理。

这些附加的功能还包含了必须的 3D API 调用来将代码的长度减到最小，并帮助程序员减少错误。

B.3 Cg 运行库是如何工作的

图 B-1 显示了组成 Cg 运行库 API 的 3 个子库。

- 一组核心函数和结构封装了运行库独立于 3D API 的功能。
- 一组建立在核心函数集上的面向 OpenGL 的函数。
- 一组建立在核心函数集上的面向 Direct3D 的函数。

为了使应用程序容易编写，OpenGL 和 Direct3D 库分别采用了它们各自 API

的基本原理和数据结构。你只需要用你的应用程序所使用的 3D API 指定的 Cg 运行库进行连接。因此，大部分应用程序或者使用 OpenGL 运行库，或者使用 Direct3D 运行库。

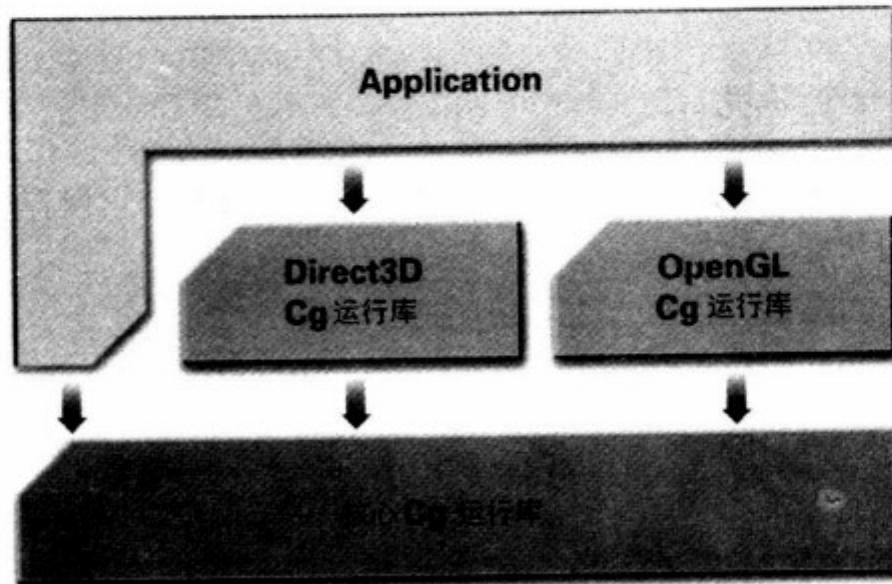


图 B-1 Cg 运行库 API 的各个部分

该附录余下的部分将提供一些用 C 编写的代码片段，教你如何在一个应用程序的框架中使用 Cg 运行库。每一个步骤都包括 OpenGL 和 Direct3D 编程的源代码。

只涉及纯粹的 Cg 资源管理的函数属于核心运行库，并有一个 cg 前缀。在这种情况下，同样的代码可以同时用于 OpenGL 和 Direct3D。

当有 OpenGL 和 Direct3D Cg 运行库的函数被使用的时候，注意 API 的名字是由函数的名字指出的。属于 OpenGL Cg 运行库的函数拥有 cgGL 前缀，而在 Direct3D Cg 运行库中的函数采用 cgD3D8 或 cgD3D9 前缀，分别对应于 DirectX8 和 DirectX9。在随后的例子中，我们只显示 DirectX 9 版本的例子。只要用“D3D8”替换“D3D9”就可以得到这些例子的 DirectX 8 的版本。注意我们列在这里的函数在 DirectX 8 和 DirectX 9 中采用了同样的参数。一般而言，并不一定总是这样。

B.3.1 头文件 (Header File)

下面是如何包括核心 Cg 运行库 API 到你的 C 和 C++ 程序的例子：

```
#include <Cg/cg.h>
```

下面是如何包括 OpenGL 指定的 Cg 运行库 API 的例子：

```
#include <Cg/cgGL.h>
```

下面是如何包括 DirectX 8 指定的 Cg 运行库 API 的例子:

```
#include <Cg/cgD3D8.h>
```

下面是如何包括 DirectX 9 指定的 Cg 运行库 API 的例子:

```
#include <Cg/cgD3D9.h>
```

B.3.2 创建一个环境

一个环境 (Context) 是一个 Cg 程序的容器。它保存了你加载的所有 Cg 程序和它们共享的数据。

下面是如何创建一个环境的例子:

```
CGcontext context = cgCreateContext();
```

B.3.3 编译一个程序

通过使用 `cgCreateProgram` 函数把一个 Cg 程序加到一个环境里，并编译这个程序:

```
CGprogram program =
cgCreateProgram(context,           // 由 cgCreateContext 得到的环境
                CG_SOURCE,        // 类型：源代码或目标代码
                programString,    // 程序文本和数据
                profile,          // profile
                "main",           // 入口函数名
                args);           // 编译参数
```

`CG_SOURCE` 参数指明了随后的字符串参数 `programString` 是包含 Cg 源代码的字节数组，而不是编译过的代码。如果你需要，Cg 运行库允许你通过使用 `CG_OBJECT` 而不是 `CG_SOURCE` 参数从编译过的代码（称为目标代码）创建一个程序。

`profile` 指定了编译程序所要使用的 `profile`——例如针对 OpenGL 应用程序的 `CG_PROFILE_ARBVP1`，或针对 Direct3D 应用程序的 `CG_PROFILE_VS_2_0`。`main` 字符串参数给出了程序中用作入口函数的函数名。最后，`args` 是一个字符串列表，提供了编译器所需要的选项。

B.3.4 载入一个程序

在你编译完一个程序以后，你需要把获得的目标代码传递给所使用的 3D API。

为了做到这点，你需要调用 Cg 运行库的 3D API 指定的函数。

在 OpenGL 里，你需要像这样加载一个程序：

```
cgGLLoadProgram (program);
```

为了进行所需要的 Direct3D 调用，Direct3D 指定的函数需要 Direct3D 设备结构。所以应用程序需要使用下面的函数调用把它传递给运行库：

```
cgD3D9SetDevice (device);
```

每一次创建一个新的 Direct3D 设备，你都必须调用这个函数，特别是在应用程序开始的时候。

你可以在 Direct3D 9 中用下面这种方法加载一个 Cg 程序：

```
cgD3D9LoadProgram (program, // CGprogram
false, // 参数阴影
0); // 汇编标记
```

或在 Direct3D 8 中用下面的方法：

```
cgD3D8LoadProgram (program, // CGprogram
false, // 参数阴影
0, // 汇编标记
0, // 顶点着色器的用法
vertexDeclaration // 顶点声明)
```

`vertexDeclaration` 是 Direct3D 顶点声明数组，描述了在顶点流中哪里可以找到需要的顶点属性。

B.3.5 修改程序的参数

运行库允许你修改程序的参数值。第一步是获得一个指向该参数的句柄：

```
CGparameter myParameter = cgGetNamedParameter ( program,
"myParameter" )
```

`myParameter` 是在程序源代码中出现的参数的名字。

第二步是设置参数的值。函数的使用依赖于参数的类型。

下面是在 OpenGL 中的一个例子：

```
cgGLSetParameter4fv (myParameter, value);
```

下面是在 Direct3D 中的同样的例子：

```
cgD3D9SetUniform (myParameter, value);
```

这些函数调用把包含在一个数组中的 4 个浮点值 `value` 赋给了参数 `myParameter`（假设它的类型为 `float4`）。

在这两种 API 中，有这样一些不同类型的函数分别用来设置矩阵、数组、纹理和纹理状态。

B.3.6 执行程序

在 OpenGL 中，在你执行一个程序之前，你必须激活它所对应的 profile。例如：

```
cgGLEnableProfile (CG_PROFILE_ARBVP1);
```

在 Direct3D 中，不需要做什么来激活一个指定的 profile。

下一步，你需要把程序绑定到当前的 3D API 状态。这意味着它在随后的渲染调用中为每个顶点（在顶点程序的情况下）或为每个片段（在片段程序的情况下）执行。

下面是如何在 OpenGL 中绑定程序的例子：

```
cgGLBindProgram (program);
```

下面是如何在 Direct3D 中绑定程序的例子：

```
cgD3D9BindProgram (program);
```

你可以一次为某个 profile 只绑定一个顶点程序和一个片段程序。因此，同样的顶点程序可以一直执行，只要没有其他的顶点程序被绑定。类似地，同样的片段程序也可以一直执行，只要没有其他的片段程序被绑定。

在 OpenGL 中，可以用下面的调用来禁止 profile：

```
cgGLDisableProfile (CG_PROFILE_ARBVP1);
```

禁止一个 profile 要基于 profile 发出一些命令，并返回 OpenGL 到它的固定功能模式。

B.3.7 释放资源

如果你的应用程序不再需要一个 Cg 程序，释放由 Cg 运行库为程序维护的资源是一个非常好的编程习惯。因为 Direct3D 运行库维护了一个到 Direct3D 设备的内部引用，当你使用完 Direct3D 运行库的时候，你必须告诉它释放这个引用。这可以通过下面的调用来实现：

```
cgD3D9SetDevice (0);
```

释放为一个单独的程序分配的资源，可以使用下面这个函数调用：

```
cgDestroyProgram (program);
```

释放为一个环境（context）分配的所有资源，可以使用下面这个函数调用：

```
cgDestroyContext (context);
```

注意结束一个环境的同时，会结束这个环境所包含的所有程序。

B.3.8 处理错误

核心 Cg 运行库通过设置一个包含错误代码的全局变量来报告错误。你可以用下面的方法来查询错误代码和它对应的错误字符串：

```
CGerror error = cgGetError ();
const char* errorString = cgGetErrorString (error);
```

每次一个错误产生，核心库还会调用由应用程序可选择提供的一个回调函数。这个回调函数将实际调用 `cgGetError`：

```
void MyErrorCallback (void)
{
    const char* errorString = cgGetErrorString (cgGetError());
    printf (logfile, "Cg error: %s", errorString);
}
```

```
cgSetErrorCallback (MyErrorCallback);
```

调用 3D API 指定的 Cg 运行库函数也会产生 API 指定的错误。对 OpenGL Cg 运行库，可以使用 `glGetError` 来检查错误。大部分的 Direct3D Cg 运行库函数都返回一个 Direct3D 错误代码 (HRESULT)。类似于 Direct3D 运行库，倘若你使用了 Direct3D Cg DLL 的调试版本，Direct3D Cg 运行库也可以在一个调试状态下运行。这个模式可以通过下面的调用激活：

```
cgD3D9EnableDebugTracing (true);
```

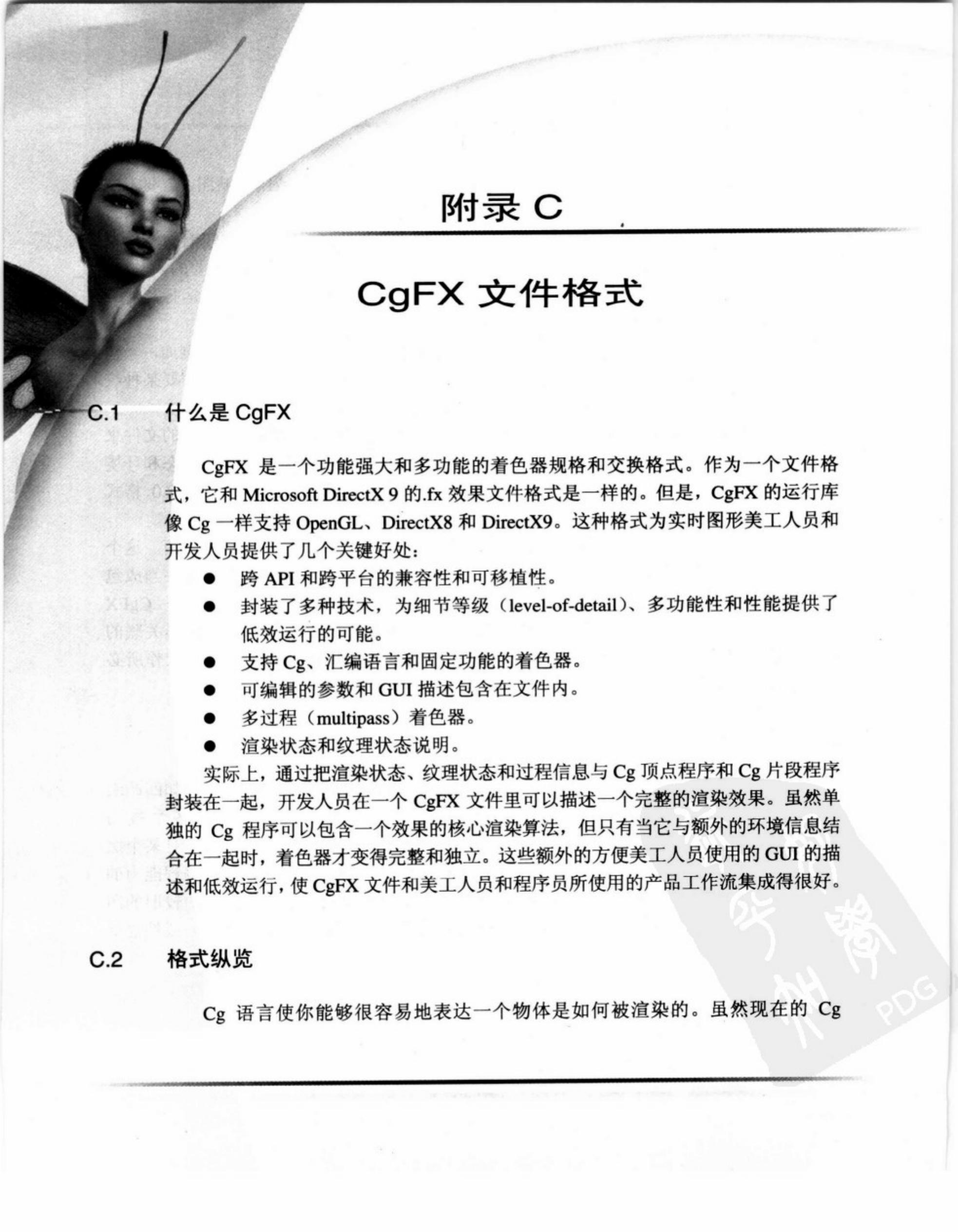
在这个模式下，许多有用的消息和跟踪结果将会被输出到调试输出控制台。

B.4 更多的细节

关于 Cg 运行库的最新信息和文档可以在 NVIDIA Cg 网站上获得：

<http://developer.nvidia.com/Cg>





附录 C

CgFX 文件格式

C.1 什么是 CgFX

CgFX 是一个功能强大和多功能的着色器规格和交换格式。作为一个文件格式，它和 Microsoft DirectX 9 的.fx 效果文件格式是一样的。但是，CgFX 的运行库像 Cg 一样支持 OpenGL、DirectX8 和 DirectX9。这种格式为实时图形美工人员和开发人员提供了几个关键好处：

- 跨 API 和跨平台的兼容性和可移植性。
- 封装了多种技术，为细节等级（level-of-detail）、多功能性和性能提供了低效运行的可能。
- 支持 Cg、汇编语言和固定功能的着色器。
- 可编辑的参数和 GUI 描述包含在文件内。
- 多过程（multipass）着色器。
- 渲染状态和纹理状态说明。

实际上，通过把渲染状态、纹理状态和过程信息与 Cg 顶点程序和 Cg 片段程序封装在一起，开发人员在一个 CgFX 文件里可以描述一个完整的渲染效果。虽然单独的 Cg 程序可以包含一个效果的核心渲染算法，但只有当它与额外的环境信息结合在一起时，着色器才变得完整和独立。这些额外的方便美工人员使用的 GUI 的描述和低效运行，使 CgFX 文件和美工人员和程序员所使用的产品工作流集成得很好。

C.2 格式纵览

Cg 语言使你能够很容易地表达一个物体是如何被渲染的。虽然现在的 Cg

profile 只描述了一个渲染过程，许多着色技术，例如阴影体或阴影贴图，需要不止一个渲染过程。

许多应用程序需要面对广泛的图形硬件功能和性能。因此，在较旧的硬件上运行的着色器的版本，和为远处的物体提供性能帮助的版本是非常重要的。

每个 Cg 程序通常只面向一个单独的 profile，并不指定如何低效地运行其他的 profile、汇编语言着色器或固定功能的顶点或片段处理。

为了使用 Cg 程序生成图像，关于它们环境的一些信息是必须的。例如，一些程序也许需要 alpha 混合来打开而需要深度写入来禁止。其他程序也许需要某种纹理格式来正常工作。这些信息在标准的 Cg 源文件中是不会出现的。

CgFX 通过一个基于文本的包含 Cg、汇编语言和固定功能的着色器的文件来处理这种类型的问题，这个文件还包含了渲染某个效果所需要的渲染状态和环境信息。正如我们以前提过的，这个文本文件的语法和 Microsoft 的.fx 2.0 格式（DirectX 9.0 效果格式）是完全匹配的。

CgFX 在一个文本文件中封装了应用一种渲染效果所需要的所有东西。这个特性使得第三方的工具或其他 3D 应用程序可以只把一个 CgFX 文本文件当成包含必须的几何信息和纹理数据而没有其他外部信息来使用。在这种情况下，CgFX 充当了一个互交换格式。CgFX 允许进行着色器交换而不带有与着色器关联的 C++ 代码，这些代码通常是使 Cg 程序在 OpenGL 和 Direct3D 下正常工作所必须的。

C.2.1 技巧 (Technique)

每个 CgFX 文件通常介绍了着色器作者尝试实现的某一种效果，例如凹凸映射、环境映射或各向异性的光照。CgFX 文件通常包含一个或多个技巧 (technique)，每个技巧描述了一种实现效果的方法。每个技巧通常针对某个级别的图形处理器的功能，因此一个 CgFX 文件可以为一个有强大片段编程能力的高级图形处理器包含一个技巧，还可以只为支持固定功能的纹理混合的较旧的图形硬件包含一个技巧。CgFX 技术还能够被多功能、细节级别 (LOD) 或性能低效使用。例如：

```
effect myEffectName
{
    technique PixelShaderVersion
```

```

{...};

technique FixedFunctionVersion
{...};

technique LowDetailVersion
{...};
}

```

C.2.2 过程 (Pass)

每个技巧可以包括一个或多个过程 (pass)。每个过程代表了在一个技巧内应用于一个单独的渲染过程的一组渲染状态和着色器。例如，过程 0 可以只设置深度值，以便过程 1 和 2 可以不需要经过多边形排序直接应用一个叠加的 alpha 混合技术。

每个过程可以包括一个顶点程序、一个片段程序，或两者都包括，并且每个过程可以使用固定功能的顶点或像素处理，或两者都使用。例如，过程 0 可以使用固定功能的像素处理来输出环境光的颜色。过程 1 可以使用 ps_1_1 片段程序，而过程 2 可以使用一个 ps_2_0 片段程序。实际上，在一个技巧内的所有过程通常都使用固定功能的处理，或都使用 Cg 或汇编程序。这种方法将预防一些图形处理器当固定功能和可编程部件用不同的方法处理同样的数据时出现的 depth-fighting 人工痕迹。

C.2.3 渲染状态

每个过程还包含渲染状态，例如 alpha 混合、深度写入和纹理过滤等模式。例如：

```

pass firstPass
{
    DepthWriteEnable = true;
    AlphaBlendingEnable = false;
    MinFilter[0] = Linear;
    MagFilter[0] = Lienar;
    MipFilter[0] = Lieanr;
}

```

```
// Pixel shader written in assembly
pixelShader = asm
{
    ps.1.1
    tex      t0;
    mov      r0, t0;
};

};

注意除了嵌入的 Cg 程序，CgFX 还允许你用 asm 关键字来编写汇编语言的顶点和片段程序。
```

C.2.4 变量和语义

最后，CgFX 文件包含了全局和每个技巧的 Cg 风格的变量。这些变量通常被当作统一参数传递给 Cg 函数，或者当作一个渲染或纹理状态设置的值。例如，一个 **bool** 变量可以被用作一个 Cg 函数的统一参数，或作为使 alpha 混合渲染状态有效或无效的值：

```
bool AlphaBlending = false;
float bumpHeight = 0.5f;
```

这些变量可以包含一个用户定义的语义，这个语义可以帮助应用程序不经过解释变量名直接提供正确的数据给着色器：

```
float4x4 myViewMatrix : ViewMatrix;
texture2D someTexture : DiffuseMap;
```

一个能够使用 CgFX 的应用程序，可以为它的变量和它们的语义查询 CgFX 文件。

C.2.5 注解（Annotation）

另外，每个变量可以有一个可选的注解。这个注解是一种变量-实例的结构，它包含了效果创作人员想要与一个支持 CgFX 的应用程序进行通信的数据，例如一名美工人员的工具。然后，应用程序能够允许基于一个适合注解类型的 GUI 元素对变量进行操作。

一个注解可以被用来描述一个用于操作统一变量的用户接口元素，或者描述了需要一个渲染过程渲染目标的类型。

```

float bumpHeight
<
    string gui = "slider";
    float uimin = 0.0f;
    float uimax = 1.0f;
    float uistep = 0.1f;
> = 0.5f;

```

注解出现在可选择语义后面，而在变量初始化前。应用程序可以查询注解，然后使用它们在支持 CgFX 的工具中展示某些参数给美工人员，例如 Discreet 的 3ds max 5 或者 Alias|Wavefront 的 Maya 4.5。

C.2.6 一个 CgFX 文件的示例

示例 C-1 显示了一个计算基本的漫反射和镜面反射光照的 CgFX 文件的例子。

例 C-1 一个示例 CgFX 文件

```

struct VS_INPUT
{
    float4 vPosition : POSITION;
    float4 vNormal : NORMAL;
    float4 vTexCoords : TEXCOORD0;
};

struct VS_OUTPUT
{
    float4 vTexCoord0 : TEXCOORD0;
    float4 vDiffuse : COLOR0;
    float4 vPosition : POSITION;
    float4 vSpecular : COLOR1;
};

VS_OUTPUT myvs(uniform float4x4 ModelViewProj,
               uniform float4x4 ModelView,
               uniform float4x4 ModelViewIT,
               uniform float4x4 ViewIT,
               uniform float4x4 View,

```

```

        const VS_INPUT vin,
        uniform float4 lightPos,
        uniform float4 diffuse,
        uniform float4 specular,
        uniform float4 ambient)

{
    VS_OUTPUT vout;
    float4 position = mul(ModelView, vin.vPosition);
    float4 normal = mul(ModelViewIT, vin.vNormal);

    float4 viewLightPos = mul(View, lightPos);
    float4 lightvec = normalize(viewLightPos - position);
    float4 eyevec = normalize(ViewIT[3]);
    float self_shadow = max(dot(normal, lightvec), 0);

    float4 halfangle = normalize(lightvec + eyevec);
    float spec_term = max(dot(normal, halfangle), 0);

    float4 diff_term = ambient + diffuse * self_shadow +
                       self_shadow * spec_term * specular;
    vout.vDiffuse = diff_term;
    vout.vPosition = mul(ModelViewProj, vin.vPosition);
    return vout;
}

float4x4 vit      : ViewIT;
float4x4 viewmat : View;
float4x4 mv       : WorldView;
float4x4 mvit     : WorldViewIT;
float4x4 mvp      : WorldViewProjection;
float4 diffuse    : DIFFUSE = { 0.1f, 0.1f, 0.5f, 1.0f };
float4 specular   : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
float4 ambient    : AMBIENT = { 0.1f, 0.1f, 0.1f, 1.0f };

float4 lightPos : Position
<
    string Object = "PointLight";
    string Space = "World";
> = { 100.0f, 100.0f, 100.0f, 0.0f };

```

```

technique t0
{
    pass p0
    {
        Zenable = true;
        ZWriteEnable = true;
        CullMode = None;
        VertexShader = compile vs_1_1 myvs( mvp, mv, mvit, vit,
                                             viewmat, lightPos,
                                             diffuse, specular,
                                             ambient);
    }
}

```

C.3 支持 CgFX 格式的 Cg 插件

在本书出版的时候，Cg 插件已经可以在主要的数字内容创作（DCC）应用程序中使用了，例如 Alias\Wavefront 的 Maya 4.5 和 Discreet 的 3ds max 5 都直接支持 CgFX 格式。

3ds max 的 Cg 插件允许一名美工人员直接在 3ds max 中观察和调整一个 CgFX 着色器的可编辑的参数。当在 DirectX 下运行 max 的时候，所有对着色器设置的修改都可以实时地显示在 3ds max 本地视图中。这为美工人员提供了更多的实时三维着色器的直接控制。

Maya 的 Cg 插件也允许一名美工人员直接在 Maya 的着色编辑窗口（例如属性编辑器和动画图表窗口）中察看和调整一个 Cg 着色器的可编辑的参数。此外，对着色器设置的修改可以实时地显示在 Maya 的 OpenGL 视图中。

C.4 学习更多有关 CgFX 的知识

与 CgFX 相关的软件可以从 NVIDIA Cg 的网站获得：

<http://developer.nvidia.com/Cg>

经常浏览这个网站可以帮助你获得最新的应用程序，插件和其他应用 CgFX 文件格式的软件。你也可以通过该网站报告你在发布的软件中找到的程序缺陷的信息。同样，有关其他规范细节和例子的信息可以查阅 DirectX9.0 效果参考文献。





附录 D

Cg 关键字

Cg 关键字列表

下面是 Cg 保留字的一个列表。带星号的字对大小写不敏感。

除了表中的这些字以外，任何以两个下划线为前缀的标识符（例如，`_newType`）都是保留字。注意矩阵和向量类型（例如 `half2x3` 或 `float4`）不在这个表中，因为它们可以被用作标识符。尽管如此，我们建议你把矩阵和向量类型也当成保留字，以避免不必要的混淆。

<code>asm*</code>	<code>explicit</code>	<code>pixelfragment*</code>	<code>template</code>
<code>asm_fragment</code>	<code>extern</code>	<code>pixelshader*</code>	<code>texture*</code>
<code>auto</code>	<code>FALSE</code>	<code>private</code>	<code>texture1D</code>
<code>bool</code>	<code>fixed</code>	<code>protected</code>	<code>texture2D</code>
<code>break</code>	<code>float*</code>	<code>public</code>	<code>texture3D</code>
<code>case</code>	<code>for</code>	<code>register</code>	<code>textureCUBE</code>
<code>catch</code>	<code>friend</code>	<code>reinterpret_cast</code>	<code>textureRECT</code>
<code>char</code>	<code>get</code>	<code>return</code>	<code>this</code>
<code>class</code>	<code>goto</code>	<code>row_major</code>	<code>throw</code>
<code>column_major</code>	<code>half</code>	<code>sampler</code>	<code>TRUE</code>
<code>compile</code>	<code>if</code>	<code>sampler_state</code>	<code>try</code>
<code>const</code>	<code>in</code>	<code>sampler1D</code>	<code>typedef</code>
<code>const_cast</code>	<code>inline</code>	<code>sampler2D</code>	<code>typeid</code>
<code>continue</code>	<code>inout</code>	<code>sampler3D</code>	<code>typename</code>
<code>decl*</code>	<code>int</code>	<code>samplerCUBE</code>	<code>uniform</code>
<code>default</code>	<code>interface</code>	<code>shared</code>	<code>union</code>
<code>delete</code>	<code>long</code>	<code>short</code>	<code>unsigned</code>

discard	matrix*	signed	using
do	mutable	sizeof	vector*
double	namespace	static	vertexfragment*
dword*	new	static_cast	vertexshader*
dynamic_cast	operator	string*	virtual
else	out	struct	void
emit	packed	switch	volatile
enum	pass*	technique*	while

附录 E

Cg 标准库函数

Cg 提供了一组内置函数和预先定义的与语义绑定的结构来简化图形处理器的编程。这些函数和 C 的标准库非常相似，提供了一组非常方便的常用函数。在很多情况下，这些函数通常映射到一条图形处理器的本地指令，因此它们执行起来非常快。对那些被映射成多条图形处理器本地指令的函数，你可以预期它们在不久的将来变得更加有效。

虽然你可以为了性能和精确度方面的原因对特定的函数编写你自己的版本，但尽可能地使用 Cg 标准库函数是一种非常明智的选择。标准库函数会继续为以后的图形处理器进行优化。现在用这些函数编写的一个程序会自动地在编译的时候为最新的架构优化。另外，标准库还为顶点和片段程序提供了一个非常方便的统一接口。

本附录描述了 Cg 标准库的内容，并分成以下 5 个部分：

- “数学函数”。
- “几何函数”。
- “纹理贴图函数”。
- “导数函数”。
- “调试函数”。

在适当的情况下，在输入和输出类型相同的时候，函数被重载用来同时支持标量和向量变量。

E.1 数学函数

表 E-1 列出了 Cg 标准函数库所提供的数学函数。这个表包括的函数对三角学、

求幂、舍入和向量及矩阵操作非常有用。所有的函数工作于大小相同的标量和向量，除非特别注明。

表 E-1

数学函数

函 数	描 述
<code>abs (x)</code>	x 的绝对值
<code>acos (x)</code>	x 的反余弦范围在 $[0, \pi]$ 内， x 在 $[-1, +1]$ 之间
<code>all (x)</code>	如果 x 的所有成员都不等于 0，则返回 <code>true</code> 否则返回 <code>false</code>
<code>any (x)</code>	如果 x 的任何一个成员不等于 0，则返回 <code>true</code> 否则返回 <code>false</code>
<code>asin (x)</code>	x 的反正弦范围在 $[-\pi/2, \pi/2]$ 内， x 的值必须在 $[-1, 1]$ 之间
<code>atan (x)</code>	x 的反正切范围在 $[-\pi/2, \pi/2]$ 内
<code>atan2 (y, x)</code>	y/x 的反正切范围在 $[-\pi, \pi]$ 内
<code>ceil (x)</code>	不小于 x 的最小整数
<code>clamp (x, a, b)</code>	按如下规则把 x 限制在范围 $[a, b]$ 内： 如果 x 小于 a ，则返回 a 如果 x 大于 b ，则返回 b 否则返回 x
<code>cos (x)</code>	x 的余弦
<code>cosh (x)</code>	x 的双曲余弦
<code>cross (A, B)</code>	向量 A 和 B 的外积； A 和 B 必须是三元向量
<code>degrees (x)</code>	弧度到角度的转换
<code>determinant (M)</code>	矩阵 M 的行列式值
<code>dot (A, B)</code>	向量 A 和 B 的点积
<code>exp (x)</code>	指数函数 e^x
<code>exp2 (x)</code>	指数函数 2^x
<code>floor (x)</code>	不大于 x 的最大整数
<code>fmod (x, y)</code>	x/y 的余数，和 x 有同样的符号。 如果 y 是 0，结果有具体的实现方法定义

续表

函 数	描 述
<code>frac (x)</code>	x 的小数部分
<code>frexp (x, out, exp)</code>	把 x 分解成一个 $[\frac{1}{2}, 1)$ 之间的标准化的分数，并把这个分数返回。并且 2^{exp} 保存在 <code>exp</code> 中 如果 x 是 0，两个部分的结果都为 0
<code>isfinite (x)</code>	如果 x 是有限的返回 <code>true</code>
<code>isinf (x)</code>	如果 x 是无限的返回 <code>true</code>
<code>isnan (x)</code>	如果 x 不是一个数字返回 <code>true</code>
<code>ldexp (x, n)</code>	$x \times 2^n$
<code>lerp (a, b, f)</code>	线性插值： $(1-f)*a + b*f$ a 和 b 是相匹配的向量或标量类型。 f 可以是一个标量或与 a 和 b 类型相同的向量
<code>lit (NdotL, NdotH, m)</code>	为环境光、漫反射光和镜面反射光成分计算光照系数。 $NdotL$ 参数被认为包含 $N \cdot L$, $NdotH$ 参数包含 $N \cdot H$ 并按如下规则返回一个四元向量： <ul style="list-style-type: none"> • 结果向量的 x 成员包含了环境光系数，这个系数通常为 1.0 • y 成员包含了漫反射系数。如果 $(N \cdot L) < 0$ 则为 0，否则为 $(N \cdot L)^m$ • z 成员包含了镜面反射系数。如果 $(N \cdot L) < 0$ 或者 $(N \cdot H) < 0$ 则为 0，否则为 $(N \cdot H)^m$ • w 成员为 1.0 这个函数没有向量的版本
<code>log (x)</code>	自然对数 $\ln(x)$; x 必须大于 0
<code>log2 (x)</code>	x 的基于 2 的对数; x 必须大于 0
<code>log10 (x)</code>	x 的基于 10 的对数; x 必须大于 0
<code>max (a, b)</code>	a 和 b 的最大值
<code>min (a, b)</code>	a 和 b 的最小值
<code>modf (x, out ip)</code>	把 x 分解成整数和分数两部分，每部分都和 x 有着相同的符号 整数部分被保存在 <code>ip</code> 中，分数部分由函数返回

续表

函 数	描 述
<code>mul (M, N)</code>	<p>矩阵 M 和矩阵 N 的积，计算方法如下所示：</p> $mul(M, N) = \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix} \begin{bmatrix} N_{11} & N_{21} & N_{31} & N_{41} \\ N_{12} & N_{22} & N_{32} & N_{42} \\ N_{13} & N_{23} & N_{33} & N_{43} \\ N_{14} & N_{24} & N_{34} & N_{44} \end{bmatrix}$ <p>如果 M 的大小是 $A \times B$，而 N 的大小是 $B \times C$，则返回的矩阵的大小是 $A \times C$</p>
<code>mul (M, v)</code>	<p>矩阵 M 和列向量 v 的积，如下所示：</p> $mul(M, v) = \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$ <p>如果 M 是一个 $A \times B$ 的矩阵而 v 是一个 $B \times 1$ 的向量，则返回一个 $A \times 1$ 的向量</p>
<code>mul (v, M)</code>	<p>行向量 v 和矩阵 M 的积，如下所示：</p> $mul(v, M) = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix} \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix}$ <p>如果 v 是一个 $1 \times A$ 的向量而 M 是一个 $A \times B$ 的矩阵而，则返回一个 $1 \times B$ 的向量</p>
<code>noise (x)</code>	根据它的参数类型，这个函数可以是一元、二元或三元噪声函数。返回的值在 0 和 1 之间，并且通常与给定的输入值一样
<code>pow (x, y)</code>	x^y
<code>radians (x)</code>	角度到弧度的转换函数
<code>round (x)</code>	离 x 最近的整数
<code>rsgqrt (x)</code>	x 的平方根的倒数； x 必须大于 0
<code>saturate (x)</code>	把 x 限制到 $[0, 1]$ 之间
<code>sign (x)</code>	如果 $x > 0$ 则返回 1；否则返回 0
<code>sin (x)</code>	x 的正弦

续表

函 数	描 述
<code>sincos (float x, out s, out c)</code>	<code>s</code> 被设为 <code>x</code> 的正弦, <code>c</code> 被设为 <code>y</code> 的余弦 如果 $\sin(x)$ 和 $\cos(x)$ 同时需要的时候, 这个函数比分别计算这两个函数有效
<code>sinh (x)</code>	<code>x</code> 的双曲正弦
<code>smoothstep (min, max, x)</code>	对在 <code>min</code> 和 <code>max</code> 之间的 <code>x</code> 的值, 返回一个从 0($x=\min$) 到 1($x=\max$) 平稳变化的值 <code>x</code> 先被限制到 $[\min, \max]$ 的范围内, 然后用如下公式进行插值: $-2*((x-\min)/(\max-\min))^3+3*((x-\min)/(\max-\min))^2$
<code>step (a, x)</code>	0 如果 $x < a$; 1 如果 $x \geq a$
<code>sqrt (x)</code>	<code>x</code> 的平方根; <code>x</code> 必须大于 0
<code>tan (x)</code>	<code>x</code> 的正切
<code>tanh (x)</code>	<code>x</code> 的双曲线切线
<code>transpose (M)</code>	矩阵 <code>M</code> 的转置矩阵 如果 <code>M</code> 是一个 $A \times B$ 矩阵, <code>M</code> 的转置是一个 $B \times A$ 矩阵, 它的第一列是 <code>M</code> 的第一行, 第二列是 <code>M</code> 的第二行, 第三列是 <code>M</code> 的第三行, 等等

E.2 几何函数

表 E-2 介绍了在 Cg 标准库中提供的几何函数。

表 E-2 几何函数

函 数	描 述
<code>distance (pt1, pt2)</code>	点 <code>pt1</code> 和 <code>pt2</code> 之间的欧氏距离
<code>faceforward (N, I, Ng)</code>	如果 $\text{dot}(N_g, I) < 0$ 为 <code>N</code> , 否则为 <code>-N</code>
<code>length (v)</code>	一个向量的欧式长度
<code>normalize (v)</code>	返回一个指向与向量 <code>v</code> 一样, 长度为 1 的向量

续表

函 数	描 述
<code>reflect (I, N)</code>	根据入射光线方向 I 和表面法向量 N 计算反射向量 仅对三元向量有效
<code>refract (I, N, eta)</code>	根据入射光线方向 I, 表面法向量 N 和折射相对系数 eta, 计算折射向量。如果对给定的 eta, I 和 N 之间的角度太大, 返回 (0, 0, 0) 只对三元向量有效

E.3 纹理贴图函数

表 E-3 介绍了在 Cg 标准库中提供的纹理贴图函数。现在，这些纹理函数完全被 ps_2_0、ps_2_x、arbfp1 和 fp30 profile（虽然只有 OpenGL 的 profile 支持 samplerRECT 函数）所支持。它们也会被所有将来的有纹理贴图能力的高级片段 profile 所支持。所有列在表 E-3 中的函数返回一个 float4 的值。

因为老硬件的受限制的像素编程能力，ps_1_1、ps_1_2、ps_1_3 和 fp20 profile 在使用纹理贴图函数上有限制。

表 E-3 纹理映射函数

函 数	描 述
<code>tex1D (sampler1D tex, float s)</code>	1D 非投影纹理查询
<code>tex1D (sampler1D tex, float s, float dsdx, float dsdy)</code>	1D 非投影使用导数的纹理查询
<code>tex1D (sampler1D tex, float2 sz)</code>	1D 非投影深度比较纹理查询
<code>tex1D (sampler1D tex, float2 sz, float dsdx, float dsdy)</code>	1D 非投影深度比较并使用导数的纹理查询
<code>tex1Dproj (sampler1D tex, float2 sq)</code>	1D 投影纹理查询
<code>tex1Dproj (sampler1D tex, float3 szq)</code>	1D 投影深度比较纹理查询
<code>tex2D (sampler2D tex, float2 s)</code>	2D 非投影纹理查询
<code>tex2D (sampler2D tex, float2 s, float2 dsdx, float2 dsdy)</code>	2D 非投影使用导数的纹理查询

续表

函 数	描 述
tex2D (sampler2D tex, float3 sz)	2D 非投影深度比较纹理查询
tex2D (sampler2D tex, float3 sz float2 dsdx, float2 dsdy)	2D 非投影深度比较并使用导数的纹理查询
tex2Dproj (sampler2D tex, float3 sq)	2D 投影纹理查询
tex2Dproj (sampler2D tex, float4 szq)	2D 投影深度比较纹理查询
texRECT (samplerRECT tex, float2 s)	2D 非投影矩形纹理查询 (OpenGL 独有)
texRECT (samplerRECT tex, float2 s, float2 dsdx, float2 dsdy)	2D 非投影使用导数的矩形纹理查询 (OpenGL 独有)
texRECT (samplerRECT tex, float3 sz)	2D 非投影深度比较矩形纹理查询 (OpenGL 独有)
texRECT (samplerRECT tex, float3 sz, float2 dsdx, float2 dsdy)	2D 非投影深度比较并使用导数的矩形纹理 查询 (OpenGL 独有)
texRECTproj (samplerRECT tex, float3 sq)	2D 投影矩形纹理查询 (OpenGL 独有)
texRECTproj (samplerRECT tex, float3 szq)	2D 投影矩形纹理深度比较查询 (OpenGL 独 有)
tex3D (sampler3D tex, float3 s)	3D 非投影纹理查询
tex3D (sampler3D tex, float3 s, float3 dsdx, float3 dsdy)	3D 非投影使用导数的纹理查询
tex3Dproj (sampler3D tex, float4 sq)	3D 投影纹理查询
texCUBE (samplerCUBE tex, float3 s)	立方贴图非投影纹理查询
texCUBE (samplerCUBE tex, float3 s, float3 dsdx, float3 dsdy)	立方贴图非投影使用导数的纹理查询
texCUBEproj (samplerCUBE tex, float4 sq)	立方贴图投影纹理查询 (忽略 q)

在这个表中，每个函数第二个参数的名字指明了在执行纹理查询的时候，它的值是如何被使用的：

- *s* 表明这是一个一元、二元或三元纹理坐标。
- *z* 表明这是一个用来进行阴影贴图查找的深度比较值。

- q 表明这是一个透视值，在进行纹理查找之前，它被用来除以纹理坐标 (s)。

当你使用的纹理函数允许你指定一个深度比较值的时候，与之相关联的纹理单元必须被设置成深度比较纹理。否则，深度比较实际上不会被执行。

E.4 导数函数

表 E-4 介绍了 Cg 标准库所支持的导数函数。顶点 profile 不需要支持这些函数。

表 E-4

导数函数

函 数	描 述
<code>ddx (a)</code>	近似 a 关于屏幕空间 x 轴的偏导数
<code>ddy (a)</code>	近似 a 关于屏幕空间 y 轴的偏导数

E.5 调试函数

表 E-5 介绍了 Cg 标准库所支持的调试函数。顶点 profile 不需要支持这个函数。

`debug` 函数的目的是允许一个程序被编译两次——一次使用 DEBUG 选项，另一次则不选择。通过执行这两个程序，有可能获得一个包含程序的最后输出的帧缓冲器和另一个包含一个为了调试目的需要检查的中间值的帧缓冲器。

表 E-5

调试函数

函 数	描 述
<code>void debug (float4 x)</code>	如果一个编译器的 DEBUG 选项被选择了，调用这个函数将使得 x 的值被拷贝到程序的 COLOR 输出，并且正在执行的程序会被终止 如果编译器的 DEBUG 选项没被选中，这个函数什么也不做

[General Information]

书名 : Cg 教程 可编程实时图形权威指南

作者 : Randima Fernando, Mark J. Kilgard 著 ; 洪伟等译

页数 : 280

出版社 : 人民邮电出版社

出版日期 : 2004.09

简介 : 本书解释了如何在当今可编程 GPU 构架上实现基本和高级的技术, 主要涉及的方面包括 : 3D 变换、每个顶点和每个像素的光照、顶点混合与关键帧插值、环境贴图、凹凸贴图、雾、性能优化、投影纹理、卡通着色、合成等。

主题词 : 图象处理 (学科 : 应用软件) 图象处理 应用软件

SS号 : 11275772

DX号 : 000005109067

http://book2.duxiu.com/bookDetail.jsp?dxNum
ber=000005109067&d=363BE174FF97087CBDA5AE60
DF2CCD42&fenlei=18170411010401&sw=Cg%BD%CC%
B3%CC+%BF%C9%B1%E0%B3%CC%CA%B5%CA%B1%CD%BC%
D0%CE%C8%A8%CD%FE%D6%B8%C4%CF

封面

书名

版权

前言

目录

第1章	简介
第2章	最简单的程序
第3章	参数、纹理和表达式
第4章	变换
第5章	光照
第6章	动画
第7章	环境映射技术
第8章	凹凸映射
第9章	高级论题
第10章	Profile 和性能
附录A	Cg入门
附录B	Cg运行库
附录C	Cg FX文件格式
附录D	Cg关键字
附录E	Cg标准库函数