Cg Tutorial

目录

第 1 章：简介

（Chapter 1. Introduction）

本章由以下四个部分组成：

1）　“**什么是 Cg**”　将介绍 Cg　编程语言

2）　“**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。

3）　“**Cg　的开发历史**”　提供了 Cg 开发的一些背景知识。

4）　“**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 3D 应用程序编程接口（APIs）。

This chapter has the following four sections:

- **"What Is Cg?"** introduces the Cg programming language.
- **"Vertices, Fragments, and the Graphics Pipeline"** describes the data flow of modern graphics hardware and explains how Cg fits into this data flow.
- **"Cg's Historical Development"** provides some background on how Cg was developed.
- **"The Cg Environment"** explains how applications go about using C g programs through the Cg runtime and existing 3D application programming interfaces (APIs).

1.1　什么是 Cg

（1.1 What Is Cg?）

本书将教你如何使用一种叫 Cg　的编程语言。Cg 语言使得通过控制可编程图形硬件绘制形状，外形和动作成为了可能。它把将这些可编程控制的属性和当今高速度、高性能的 GPU 很好的结合在一起。在此之前，从来没有计算机图形从业者（无论是设计师还是工程师）在他们生成自己的实时图像的时候有如此多的控制能力。

This book teaches you how to use a programming language called C g. The C g language makes it possible for you to control the shape, appearance, and motion of objects drawn using programmable graphics hardware. It marries programmatic control of these attributes with the incredible speed and capabilities of today's graphics processors. Never before have computer graphics practitioners, whether artists or programmers, had so much control over the real-time images they generate.

Cg　为开发人员提供了一整套使用便捷的编程平台，使得在多种平台上迅速生成特殊效果、制作实时的、如同电影级别的体验成为可能。通过一个新的抽象层，Cg　语言使开发者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　和像　Xbox　这样的游戏平台。Cg　是在和微软公司密切合作的基础上开发的，因此　Cg 与 OpenGL API　和　Microsoft 的 HLSL 完全兼容。

Cg provides developers with a complete programming platform that is easy to use and enables the fast creation of special effects and real-time cinematic-quality experiences on multiple platforms. By providing a new level of abstraction, C g removes the need for developers to program directly to the graphics hardware assembly language, and thereby more easily target OpenGL, DirectX, Windows, Linux, Macintosh OS X, and console platforms such as the Xbox. C g was developed in close collaboration with Microsoft Corporation and is compatible with both the OpenGL API and Microsoft's High-Level Shading Language (HLSL) for DirectX 9.0.

Cg 代表“C for Graphics”。C 语言是一种开发与 20 世纪 70 年代、非常流行的通用编程语言。由于它的流行简洁的设计，C 语言为之后的多种编程语言提供了基础。例如，C++和 Java 语法结构都是基于 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。

Cg stands for "C for graphics." The C programming language is a popular, general-purpose language invented in the 1970s. Because of its popularity and clean design, C provided the basis for several subsequent programming languages. For example, C ++ and Java base their syntax and structure largely on C. The C g language bases itself on C as well. If you are familiar with C or one of the many languages derived from C, then C g will be easy to learn.

另一方面，如果你不熟悉 C 语言，或者没接触过一般的编程语言，但是你非常喜欢计算机图形学并想要学习一些新东西。那么无论如何都要阅读本书。Cg 程序都十分简短且容易理解。

On the other hand, if you are not familiar with C or even programming languages in general but you enjoy computer graphics and want to learn something new, read on anyway. C g programs tend to be short and understandable.

本章的大部分所讲述背景知识对于你理解和有效的使用 Cg 语言将非常有价值。另一方面，你将发现通过实践学习 Cg 是十分容易的。如果你想马上进入 Cg 教程，那么可以随时跳到第 2 章节开始学习。

Much of this chapter is background that provides valuable context for understanding C g and using it effectively. On the other hand, you may find C g is easier to learn by doing. Feel free to skip to Chapter 2 at any time if you feel more comfortable just diving into the tutorial.

1.1.1 为可编程图形硬件设计的语言

（**1.1.1 A Language for Programming Graphics Hardware**）

Cg 不同与 C、C++和 Java，因为他非常特别。没有人会用 Cg 写电子制表软件或者文字处理程序。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渲染语言。但是 Cg 可以做光照意外的很多事情。例如，Cg 程序可以实现物理模拟、混合和其他非光照任务。

Cg is different from C , C ++, and Java because it is very specialized. No one will ever write a spreadsheet or word processor in C g. Instead, C g targets the ability to programmatically control the shape, appearance, and motion of objects rendered using graphics hardware. Broadly, this type of language is called a *shading language*. However, C g can do more than just shading. For example, C g programs can perform physical simulation, compositing, and other nonshading tasks.

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来渲染物体的解决方案。例如，你可以编写一个 Cg 程序来使得物体的外表看起来凸凹不平，或者让一个虚拟人物活动起来。在之后的 1.3 节中，你将学习许多光照渲染语言的历史和 Cg 是如何融入的该历史中

的。

Think of a C g program as a detailed recipe for how to render an object by using programmable graphics hardware. For example, you can write a C g program to make a surface appear bumpy or to animate a virtual character. Later, in Section 1.3, you will learn more about the history of shading languages and where C g fits into this history.

1.1.2 Cg 语言的数据流模型

（**1.1.2 Cg's Data-Flow Model**）

除了专门为图形设计以外，Cg 与其他光照语言是和传统编程语言是截然不同的，，因为这些语言是基于数据流模型的。在这样一个模型里，计算的是为了响应一系列，流经处理序列的数据而发生的。

In addition to being specialized for graphics, C g and other shading languages are different from conventional programming languages because they are based on a data-flow computational model. In such a model, computation occurs in response to data that flows through a sequence of processing steps.

在渲染一副图片时，Cg 程序运行在处理顶点（Vertices）和片段（Fragments）上（如果你还不理解片段是什么，那么现在暂时把它当作像素）。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。

Cg programs operate on vertices and fragments (think "pixels" for now if you do not know what a fragment is) that are processed when rendering an image. Think of a C g program as a black box into which vertices or fragments flow on one side, are somehow transformed, and then flow out on the other side. However, the box is not really a black box because you get to determine, by means of the C g programs you write, exactly what happens inside.

在渲染三维场景时，每当处理一个顶点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。

Every time a vertex is processed or the rasterizer generates a fragment while rendering a 3D scene, your corresponding vertex or fragment C g program executes. Section 1.3 explains C g's data-flow model further.

大部分最新的 PC 和所有最近的游戏平台都包括一个 GPU，专门用来处理图形任务，例如变换和光栅化三维模型。你的 Cg 程序实际上是在你的计算机中的 GPU 上执行的。

Most recent personal computers—and all recent game consoles—contain a graphics processing unit (GPU) that is dedicated to graphics tasks such as transforming and rasterizing 3D models. Your C g programs actually execute within the GPU of your computer.

1.1.3 GPU 的特殊性和 CPU 的通用性

（**1.1.3 GPU Specialization and CPU Generalization**）

无论是 PC 还是或者游戏平台是否拥有一个 GPU， 它都必须有一个 CPU 来运行操作系统和应用程序。CPU 是以多用途为目的设计的，其所执行的程序（例如，文字处理器和统计软件包）是用多用途语言编写的，例如 C++或者 Java。

Whether or not a personal computer or game console has a GPU, there must be a C PU that runs the operating system and application programs. CPUs are, by design, general purpose. CPUs execute applications (for example, word processors and accounting packages) written in general-purpose languages, such as C++ or Java.

因为 GPU 是为专门目的设计，因此它的图形处理任务的速度要比通用的 CPU 快得多，例如渲染 3D 场景。新的 GPU 可以在 1s 内处理好几千万的顶点和光栅化几亿甚至几十亿片段。而且未来的 GPU 会更快。这绝对比 CPU 处理类似数量的顶点和片段的速度快许多。但是 GPU 不能像 CPU 那样运行任意的、多种多样的程序。

Because of the GPU's specialized design, it is much faster at graphics tasks, such as rendering 3D scenes, than a general-purpose CPU would be. New GPUs process tens of millions of vertices per second and rasterizer hundreds of millions or even billions of fragments per second. Future GPUs will be even speedier. This is overwhelmingly faster than the rate at which a C PU could process a similar number of vertices and fragments. However, the GPU cannot execute the same arbitrary, general-purpose programs that a CPU can.

GPU 专用性和高性能的特质是 Cg 为什么能存在的原因。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 也同样提供了一个和 GPU 的执行模型相符合的抽象执行模型。你将在 1.2 小节学习 GPU 独特的执行模型。

The specialized, high-performance nature of the GPU is why Cg exists. General-purpose programming languages are too open-ended for the specialized task of processing vertices and fragments. In contrast, the Cg language is fully dedicated to this task. C g also provides an abstract execution model that matches the GPU's execution model. You will learn about the unique execution model of GPUs in Section 1.2.

1.1.4 Cg 性能的基本原理

（1.1.4 The Performance Rationale for Cg）

为了保持一个看起来连续的交互，3D应用程序需要维持每秒超过15帧的图像刷新率。通常，我们认为不低于 60 帧/s 的刷新率为实时的（real-time），在这样的帧率下与应用程序的交互看起来就像是即时发生的。计算机的显示器也许有 100 万后者更多的像素需要重绘。对于一个 3D 场景而言， GPU 对显示器上的每个像素会处理每个像素好几次。原因可能是场景中的物体一般是相互重叠的，或者要提高每个像素的表现。这意味着，实时 3D 应用程序每秒需要更新上亿个像素。伴随这些需要处理的像素的是，由顶点汇成的三维模型，这些顶点只有在被正确的变换后，才能被组成多边形、线段和点，然后被光栅化成像素。这就需要每秒变换上千万的顶点。

To sustain the illusion of interactivity, a 3D application needs to maintain an animation rate of 15 or more images per second. Generally, we consider 60 or more frames per second to be "real time," the rate at which interaction with applications appears to occur instantaneously. The computer's display may have a million or more pixels that require redrawing. For 3D scenes, the GPU typically processes every pixel on the screen many times to account for how objects occlude each other, or to improve the appearance of each pixel. This means that real-time 3D applications can require hundreds of millions of pixel updates per second. Along with the required pixel processing, 3D models are composed of vertices that must be transformed properly before they are assembled into polygons, lines, and points that will be rasterized into pixels. This can require transforming tens of millions of vertices per second.

而且，在这种图形处理之外还需要 CPU 做相当大的工作来为每个新的图像更新显示。事实上，我们同时需要 CPU 和 GPU 专门面向图形的功能。以一个可交互的刷新率和 3D 应用程序和游戏所要求的质量标准来渲染场景同事需要这两种处理器。这意味着一个开发人员可以用 C++写一个 3D 应用或游戏，然后用 Cg 来充分利用 GPU 的额外的图形处理能力。

Moreover, this graphical processing happens in addition to the considerable amount of effort required of the CPU to update the animation for each new image. The reality is that we need both the CPU and the GPU's specialized graphics-oriented capabilities. Both are required to render scenes at the interactive rates and quality standards those users of 3D applications and games demand. This means a developer can write a 3D application or game in C++ and then use Cg to make the most of the GPU's additional graphics horsepower.

1.1.5 与传统编程语言共存

（**1.1.5 Coexistence with Conventional Languages**）

Cg 绝不是用来代替显存的多用途编程语言。Cg 是一种辅助语言，是专门为 GPU 设计的。使用 C 或者 C++可以使用 Cg 的运行程序（将会在 1.4.2 小节进行解释）来加载 GPU 上执行的 Cg 程序。Cg 运行库（Cg runtime）是一个用来加载、编译、操作和设置 Cg 程序在 GPU 上执行的标准子程序集合。应用程序使用 Cg 程序指示 GPU 如何完成这些可编程的渲染效果，这些效果在 CPU 上不可能达到图形处理器所能获得的渲染速度。

In no way does Cg replace any existing general-purpose languages. C g is an auxiliary language, designed specifically for GPUs. Programs written for the C PU in conventional languages such as C or C++ can use the Cg runtime (described in Section 1.4.2) to load C g programs for GPUs to execute. The C g runtime is a standard set of subroutines used to load, compile, manipulate, and configure C g programs for execution by the GPU. Applications supply C g programs to instruct GPUs on how to accomplish the programmable rendering effects that would not otherwise be possible on a C PU at the rendering rates a GPU is capable of achieving.

Cg 使得专门类型的并行处理成为可能。当你的 CPU 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 GPU 对 Vertices 和 fragments 进行并行处理。

Cg enables a specialized style of parallel processing. While your C PU executes a conventional application, that application also orchestrates the parallel processing of vertices and fragments on the GPU, by programs written in C g.

如果一个实时光照语言是这样好的一个主意的话，为什么没有人早一点发明 Cg 呢？这个问题的答案食欲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，采用硬件连接的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。

If a real-time shading language is such a good idea, why didn't someone invent Cg sooner? The answer has to do with the evolution of computer graphics hardware. Prior to 2001, most computer graphics hardware—certainly the kind of inexpensive graphics hardware in PC s and game consoles—was hard-wired to the specific tasks of vertex and fragment processing. By "hard-wired," we mean that the algorithms were fixed within the hardware, as opposed to being programmable in a way that is accessible to graphics applications. Even though these hard-wired graphics algorithms could be configured by graphics applications in a variety of ways, the applications could not reprogram the hardware to do tasks unanticipated by the designers of the hardware. Fortunately, this situation has changed.

图形硬件的设计已经发展了，而且近来的 GPU 中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，没有理由为他提供一种编程语言。现在，既然这种硬件能够被使用了，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 GPU 程序就像使用 C 编写 CPU 程序那样容易。

Graphics hardware design has advanced, and vertex and fragment processing units in recent GPUs are truly programmable. Before the advent of programmable graphics hardware, there was no point in providing a programming language for it. Now that such hardware is available, there is a clear need to make it easier to program this hardware. C g makes it much easier to program GPUs in the same manner that C made it much easier to program CPUs.

在 Cg 存在之前，战士 GPU 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 GPU 技术的发展，使得可能需要更冗长的汇编语言，因此对更高级语言的需求就更加迫切。现在那些为了优化性能而存在的大量低级编程，被委托给为优化代码输出、处理乏味调度指令的编译器。图 1.1 是一个用来表现皮肤的复杂汇编语言片段的一小部分。很显然，这个程序难以理解，特别是引用了大量特殊硬件寄存器的地方。

Before Cg existed, addressing the programmable capabilities of the GPU was possible only through low-level assembly language. The cryptic instruction syntax and manual hardware register manipulation required by assembly languages—such as DirectX 8 vertex and pixel shaders and some OpenGL extensions—made it a painful task for most developers. As GPU technology made longer and more complex assembly language programs possible, the need for a high-level language became clear. The extensive low-level programming that had been required to achieve optimal performance could now be delegated to a compiler, which optimizes the code output and handles tedious instruction scheduling. Figure 1-1 is a small portion of a complex assembly language fragment program used to represent skin. Clearly, it is hard to comprehend, particularly with the specific references to hardware registers.

Example 1-1. A Snippet of Assembly Language Code

```
. . .
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;
. . .
```

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

In contrast, well-commented C g code is more portable, more legible, easier to debug, and easier to reuse. Cg gives you the advantages of a high-level language such as C while delivering the performance of low-level assembly code.

1.1.6 Cg 的其他方面

（**1.1.6 Other Aspects of Cg**）

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，他现在还没有包含太多软件工程任务所需要的复杂功能。与 C++和 Java 不同，Cg 不需要支持类和其他在面向对象编程中大量出现的特征。现在的 Cg 没有实现指针和内存分配（虽然已经保留的关键字，可能在将来将会实）。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的永久显示，也不是当今高性能 GPU 的标志性功能。由于技术发展到允许 GPU 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 是基于 C 的，未来对 Cg 的更新可能也会采用 C 和 C++的语言特性。

Cg is a language for programming "in the small." That makes it much simpler than a modern general-purpose language such as C++. Because Cg specializes in transforming vertices and fragments, it does not currently include many of the complex features required for massive software engineering tasks. Unlike C++ and Java, Cg does not support classes and other features used in object-oriented programming. Current Cg implementations do not provide pointers or even memory allocation (though future implementations may, and keywords are appropriately reserved). C g has absolutely no support for file input/output operations. By and large, these restrictions are not permanent limitations in the language, but rather are indicative of the capabilities of today's highest performance GPUs. As technology advances to permit more general programmability on the GPU, you can expect C g to grow appropriately. Because C g is closely based on C , future updates to C g are likely to adopt language features from C and C++.

Cg 提供了数组和数据结构。它拥有所有现代语言的流控制特性：循环，条件跳转和函数调用。

Cg provides arrays and structures. It has all the flow-control constructs of a modern language: loops, conditionals, and function calls.

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 有一个函数库叫标准函数库（Standard Library），这个函数库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）的函数用来计算反射向量。

Cg natively supports vectors and matrices because these data types and related math operations are fundamental to graphics and most graphics hardware directly supports vector data types. C g has a library of functions, called the Standard Library, that is well suited for the kind of operations required for graphics. For example, the C g Standard Library includes a reflect function for computing reflection vectors.

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。

Cg programs execute in relative isolation. This means that the processing of a particular vertex or fragment has no effect on other vertices or fragments processed at the same time. There are no side effects to the execution of a C g program. This lack of interdependency among vertices and fragments makes Cg programs extremely well suited for hardware execution by highly pipelined and parallel hardware.

1.1.7 Cg 的有限执行环境

（1.1.7 The Limited Execution Environment of Cg Programs）

当你使用一种语言在现代操作系统上为先进的 CPU 编写一个程序的时候，你可以预期到大部分程序只要程序本身正确，就可以正常的编译和执行。这是因为 CPU 是设计用来执行多用途程序的，而这样的程序综合系统拥有充足的资源。

When you write a program in a language designed for modern C PUs using a modern operating system, you expect that a more-or-less arbitrary program, as long as it is correct, will compile and execute properly. This is because C PUs, by design, execute general-purpose programs for which the overall system has more than sufficient resources.

但是,GPU 是专用的，而且 GPU 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 GPU 上执行。Cg 包含了一个硬件的概念："配置"，当你在编译一个 Cg 程序的时候，你必须指定一个特定的配置。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要求你编写的每个片段不能使用超过四个纹理。

However, GPUs are specialized rather than general-purpose, and the feature set of GPUs is still evolving. Not everything you can write in C g can be compiled to execute on a given GPU. C g includes the concept of hardware "profiles," one of which you specify when you compile a C g program. Each profile corresponds to a particular combination of GPU architecture and graphics API. Your program not only must be correct, but it also must limit itself to the restrictions imposed by the particular profile used to compile your C g program. For example, a given fragment profile may limit you to no more than four texture accesses per fragment.

随着 GPU 的发展，Cg 将会支持更多，与性能能更强的 GPU 结构相对应的 profile。在不久的将来，当 GPU 变得功能越来越全面的时候，profile 将会变得不重要。但是，现在的 Cg 设计者将需要限制程序来确保他们可以再现有的 GPU 上编译和执行。一般情况下，未来的 profile 将会是目前 profile 的超集，因此所有为现在 profile 编写的程序可以在未来的 profile 上不经或修改直接编译。

As GPUs evolve, additional profiles will be supported by C g that correspond to more capable GPU architectures. In the future, profiles will be less important as GPUs become more full-featured. But for now Cg programmers will need to limit programs to ensure that they can compile and execute on existing GPUs. In general, future profiles will be supersets of current profiles, so that programs written for today's profiles will compile without change using future profiles.

这种情况听起来像是一种限制。但是实际上本书所展示的 Cg 程序可以在上千万的 GPU 上工作并生成引人注目的渲染效果。其他限制程序大小和范围的原因是：你的 Cg 程序越小越有效，它们运行的也会越快。实时图形学常常需要在增长的环境复杂度、动画速度和光照改善中取得平衡。因此通过明智的 Cg 编程最大化渲染效果始终是一件好事。

This situation may sound limiting, but in practice the C g programs shown in this book work on tens of millions of GPUs and produce compelling rendering effects. Another reason for limiting program size and scope is that the smaller and more efficient your C g programs are, the faster they will run. Real-time graphics is often about balancing increased scene complexity, animation rates, and improved shading. So it's always good to maximize rendering efficiency through judicious Cg programming.

记住 profile 所加强的限制是当前 GPU 的限制所造成的，而不是 Cg 本身。Cg 语言本身功能是非常强大的，足够表达目前所有 GPU 还没能支持的光照技术。随着时间的过去，GPU 处理器的功能将会发展到 Cg profile 能够运行的令人惊讶的复杂 Cg 程序。Cg 是一种适于现在和未来 GPU 的语言。

Keep in mind that the restrictions imposed by profiles are really limitations of current GPUs, not C g. The Cg language is powerful enough to express shading techniques that are not yet possible with all GPUs. With time, GPU functionality will evolve far enough that C g profiles will be able to run amazingly complex Cg programs. Cg is a language for both current and future GPUs.

1.2 顶点、片段和图形流水线

（1.2 Vertices, Fragments, and the Graphics Pipeline）

为了把 Cg 放在一个合适的位置，你需要理解 GPU 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硬件处理器的渲染流水线。

To put C g into its proper context, you need to understand how GPUs render images. This section explains how graphics hardware is evolving and then explores the modern graphics hardware-rendering pipeline.

1.2.1 计算机图形硬件的发展历史

（1.2.1 The Evolution of Computer Graphics Hardware）

计算机图形硬件正在以不可思议的速度前进着。如图 1-2 所示，一共有三个力量驱使这种创新的步伐。首先，半导体工业自身每 18 个月就使一个微芯片上得晶体管数目增加一倍。计算机能力则以这个在历史上被称为摩尔定律的固定加倍速度下发展着。这意味着更便宜和更快的计算机硬件，会成为我们这个时代的普通标准。

Computer graphics hardware is advancing at incredible rates. Three forces are driving this rapid pace of innovation, as shown in Figure 1-2. First, the semiconductor industry has committed itself to doubling the number of transistors (the basic unit of computer hardware) that fit on a microchip every 18 months. This constant redoubling of computer power, historically known as Moore's Law, means cheaper and faster computer hardware, and is the norm for our age.

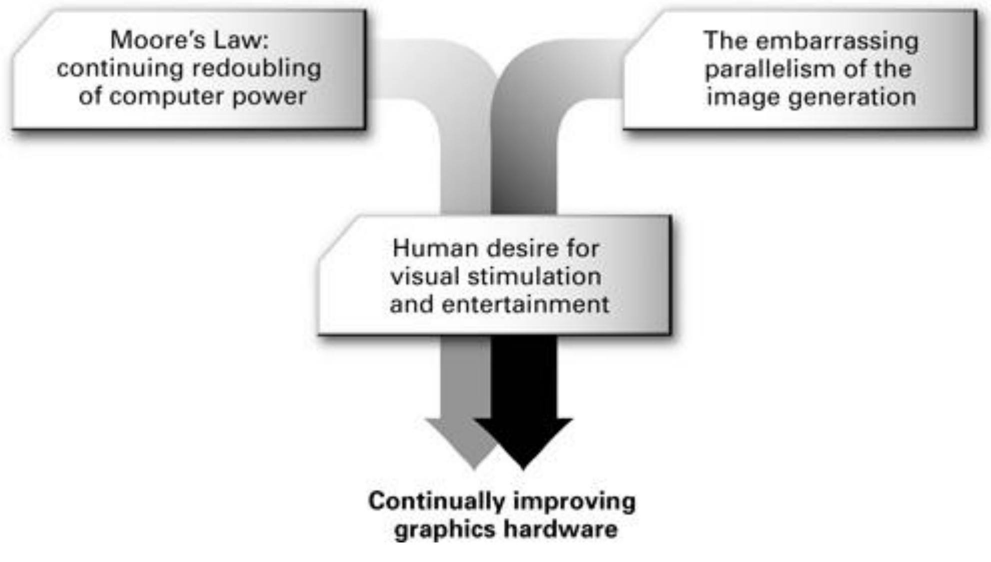

Moore's Law: continuing redoubling of computer power
The embarrassing parallelism of the image generation
Human desire for visual stimulation and entertainment
Continually improving graphics hardware

Figure 1-2 Forces Driving Graphics Hardware Innovation

第二种推动计算机图形硬件的原因是：模拟我们周围的世界需要大量的计算。人的眼睛和大脑以一种令人惊讶的速度敏锐地接触、观察和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。现实世界是在是太真实了。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“使人为难的并行问题”是指，图形硬件的设计者们可以把创造真实世界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。

The second force is the vast amount of computation required to simulate the world around us. Our eyes consume and our brains comprehend images of our 3D world at an astounding rate and with startling acuity. We are unlikely ever to reach a point where computer graphics becomes a substitute for reality. Reality is just too real. Undaunted, computer graphics practitioners continue to rise to the challenge. Fortunately, generating images is an embarrassingly parallel problem. What we mean by "embarrassingly parallel" is that graphics hardware designers can repeatedly split up the problem of creating realistic images into more chunks of work that are smaller and easier to tackle. Then hardware engineers can arrange, in parallel, the ever-greater number of transistors available to execute all these various chunks of work.

我们所说的第三种力量是，我们期望能够体验到更加真实的模拟和娱乐感受。这种力量把持续增长的计算机硬件资源和模拟更加真实的虚拟现实任务“连接起来”。

Our third force is the sustained desire we all have to be stimulated and entertained visually. This is the force that "connects" the source of our continued redoubling of computer hardware resources to the task of approximating visual reality ever more realistically than before.

正如图 1-2 所显示的那样，这些敏锐地观察使我们有信心预言计算机图形硬件的处理速度将会变得更为迅速。这些创新使我们拥有更好的交互性与令人炫目的二维体验拥有更强烈的欲望。而满足这些需求是激发我们开发 Cg 预言的动力。

As Figure 1-2 illustrates, these insights let us confidently predict that computer graphics hardware is going to get much faster. These innovations whet our collective appetite for more interactive and compelling 3D experiences. Satisfying this demand is what motivated the development of the C g language.

1.2.3 图形硬件流水线

（1.2.3 The Graphics Hardware Pipeline）

*流水线*是一系列可以并行和按照固定顺序操作的阶段。每个阶段都从它的前一阶段接受输入，然后把输出发送给随后的阶段。就像同时制造不同阶段不同汽车的汽车装配线一样。传统的图形硬件流水线以流水的方式处理大量的顶点、片段和几何图形。

A *pipeline* is a sequence of stages operating in parallel and in a fixed order. Each stage receives its input from the prior stage and sends its output to the subsequent stage. Like an assembly line where dozens of automobiles are manufactured at the same time, with each automobile at a different stage of the line, a conventional graphics hardware pipeline processes a multitude of vertices, geometric primitives, and fragments in a pipelined fashion.

图 1-3 显示了当今 GPU 所使用的图形硬件流水线。3D 应用程序传给 GPU 一系列由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 所示，有许多方法来制定几何图元。

Figure 1-3 shows the graphics hardware pipeline used by today's GPUs. The 3D application sends the GPU a sequence of vertices batched into geometric primitives: typically polygons, lines, and points. As shown in Figure 1-4, there are many ways to specify geometric primitives.

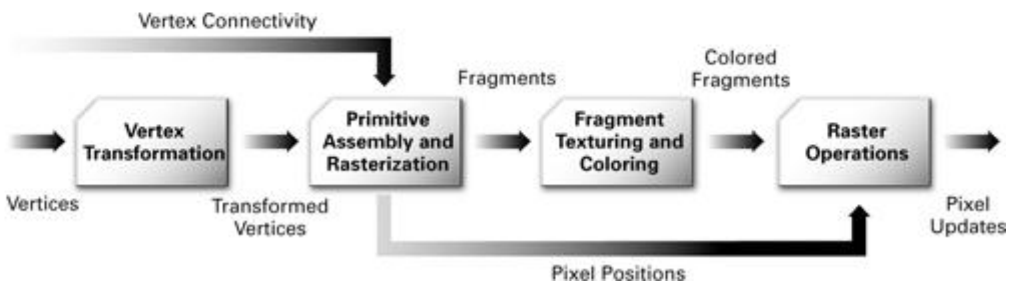


Figure 1-3 The Graphics Hardware Pipeline

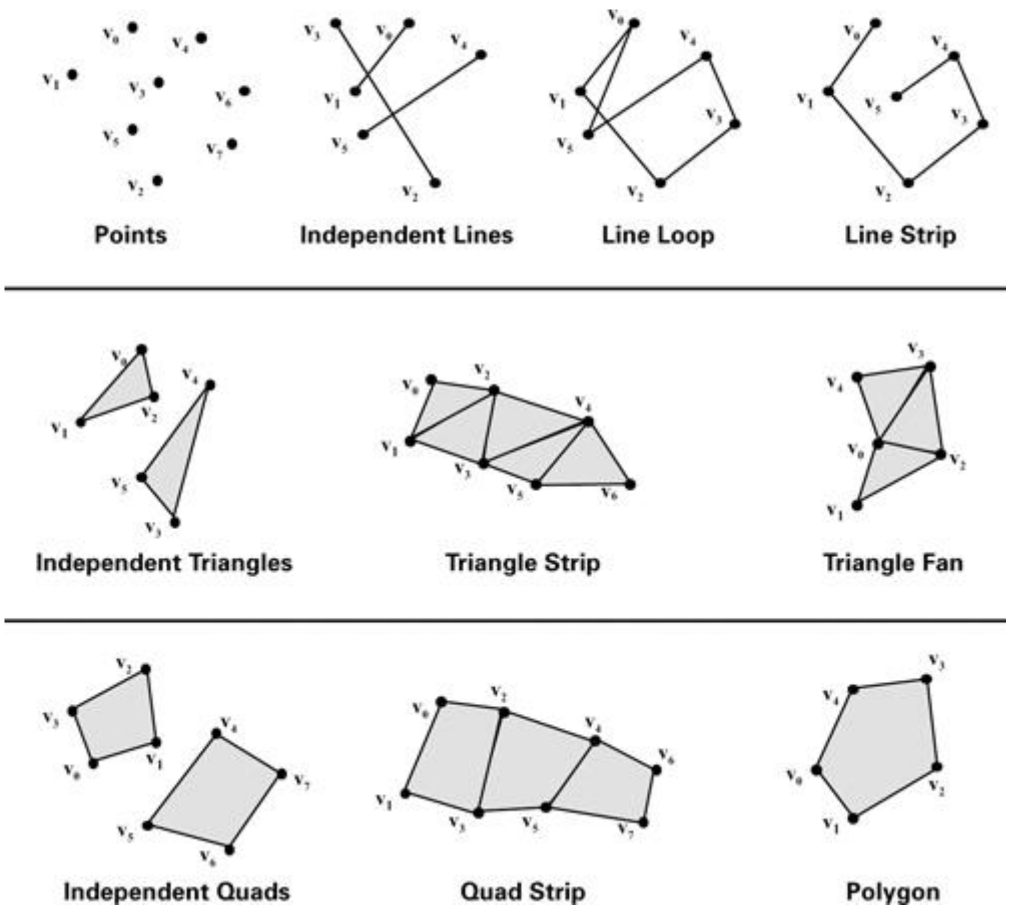


Figure 1-4 Types of Geometric Primitives

每个顶点除了有位置信息外，还有一些其他属性，例如颜色、间色（反射），一个或多个纹理坐标集合一个法向量（normal vector）。法向量指示了物体表面在改顶点的方向，他是专门用来计算光照的。

Every vertex has a position but also usually has several other attributes such as a color, a secondary (or *specular*) color, one or multiple texture coordinate sets, and a normal vector. The normal vector indicates what direction the surface faces at the vertex, and is typically used in lighting calculations.

1.2.3.1 顶点变换

（**Vertex Transformation**）

顶点变换（Vertex transformation）是图形硬件流水线中第一个处理阶段。顶点变换在每个顶点上执行一系列数学操作。这些操作包括把顶点位置变换到屏幕位置以供光栅器使用、为贴图产生纹理坐标、以及决定顶点颜色的光照运算。我们将在后续的几章解释这些任务中的大部分内容。

Vertex transformation is the first processing stage in the graphics hardware pipeline. Vertex transformation performs a sequence of math operations on each vertex. These operations include transforming the vertex position into a screen position for use by the rasterizer, generating texture coordinates for texturing, and lighting the vertex to determine its color. We will explain many of these tasks in subsequent chapters.

1.2.3.2 图元装配和光栅化

（**Primitive Assembly and Rasterization**）

经过变换的顶点数据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转配成几何图元。这将产生一系列的三角形，线段和点。这些图元需要经过剪裁到可视平截体（view frustum）（一个三维空间的可见矩形区域）和任何应用程序指定的有效剪裁平面。光栅器还可以根据空间多边形的朝向丢弃一些多边形。这个过程被称为挑选（culling）。

The transformed vertices flow in sequence to the next stage, called *primitive assembly and rasterization*. First, the primitive assembly step assembles vertices into geometric primitives based on the geometric primitive batching information that accompanies the sequence of vertices. This results in a sequence of triangles, lines, or points. These primitives may require clipping to the *view frustum* (the view's visible region of 3D space), as well as any enabled application-specified clip planes. The rasterizer may also discard polygons based on whether they face forward or backward. This process is known as *culling.*

经过剪裁和挑选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，如果一个由一个顶点组成的三角形可以占据整个屏幕，则需要生成上百万的片段！

Polygons that survive these clipping and culling steps must be rasterized. Rasterization is the process of determining the set of pixels covered by a geometric primitive. Polygons, lines, and points are each rasterized according to the rules specified for each type of primitive. The results of rasterization are a set of pixel locations as well as a set of fragments. There is no relationship between the number of vertices a primitive has and the number of fragments that are generated when it is rasterized. For example, a triangle made up of just three vertices could take up the entire screen, and therefore generate millions of fragments!

我们在前面已经告诉过你可以把片段看成一个像素。但是，现在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的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。一个片段则是更新一个特定像素所需的一个潜在必要的状态。

Earlier, we told you to think of a fragment as a pixel if you did not know precisely what a fragment was.

At this point, however, the distinction between a fragment and a pixel becomes important. The term *pixel* is short for "picture element." A pixel represents the contents of the frame buffer at a specific location, such as the color, depth, and any other values associated with that location. A *fragment* is the state required potentially to update a particular pixel.

所以，术语“片段”是用来描述因光栅化而把每个几何图元（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关联的像素位置、深度值和经过插值的参数，例如颜色、间色（反射）和一个或多个纹理坐标集。这些不同的插值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，如果一个片段通过光栅化测试（光栅化测试将在光栅操作阶段被简单介绍），这个片段将被用于更新帧缓存中得像素。

and one or more texture coordinate sets. These various interpolated parameters are derived from the transformed vertices that make up the particular geometric primitive used to generate the fragments. You can think of a fragment as a "potential pixel." If a fragment passes the various rasterization tests (in the raster operations stage, which is described shortly), the fragment updates a pixel in the frame buffer.

1.2.3.3 插值、贴图和着色

（**Interpolation, Texturing, and Coloring**）
当一个图元被光栅化为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刻插值，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。这个阶段还会确定一个新的深度，或者为了避免更新帧缓存的对应像素而丢弃这个片段。这个阶段允许丢弃片段信息，因此这个阶段可能不会对片段着色。

Once a primitive is rasterized into a collection of zero or more fragments, the *interpolation, texturing, and coloring* stage interpolates the fragment parameters as necessary, performs a sequence of texturing and math operations, and determines a final color for each fragment. In addition to determining the fragment's final color, this stage may also determine a new depth or may even discard the fragment to avoid updating the frame buffer's corresponding pixel. Allowing for the possibility that the stage may discard a fragment, this stage emits one or zero colored fragments for every input fragment it receives.

1.2.3.4 光栅操作

（**Raster Operations**）
光栅操作阶段是在帧缓存更新前，执行一系列针对每个片段的操作的最后一个阶段。这些操作时 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被隐藏。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。

The *raster operations* stage performs a final sequence of per-fragment operations immediately before updating the frame buffer. These operations are a standard part of OpenGL and Direct3D. During this stage, hidden surfaces are eliminated through a process known as *depth testing*. Other effects, such as blending and stencil-based shadowing, also occur during this stage.

光栅操作阶段将根据许多测试结果来检查每个片段，这些测试包括剪切、alpha、模板和深度等测试。这些测试设计了片段最后的颜色和深度，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（虽然一个模板写入操作依旧发生）。通过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。

The raster operations stage checks each fragment based on a number of tests, including the scissor, alpha, stencil, and depth tests. These tests involve the fragment's final color or depth, the pixel location, and per-pixel values such as the depth value and stencil value of the pixel. If any test fails, this stage discards the fragment without updating the pixel's color value (though a stencil write operation may occur). Passing the depth test may replace the pixel's depth value with the fragment's depth. After the tests, a blending operation combines the final color of the fragment with the corresponding pixel's color value. Finally, a frame buffer write operation replaces the pixel's color with the blended color. Figure 1-5 shows this sequence of operations.

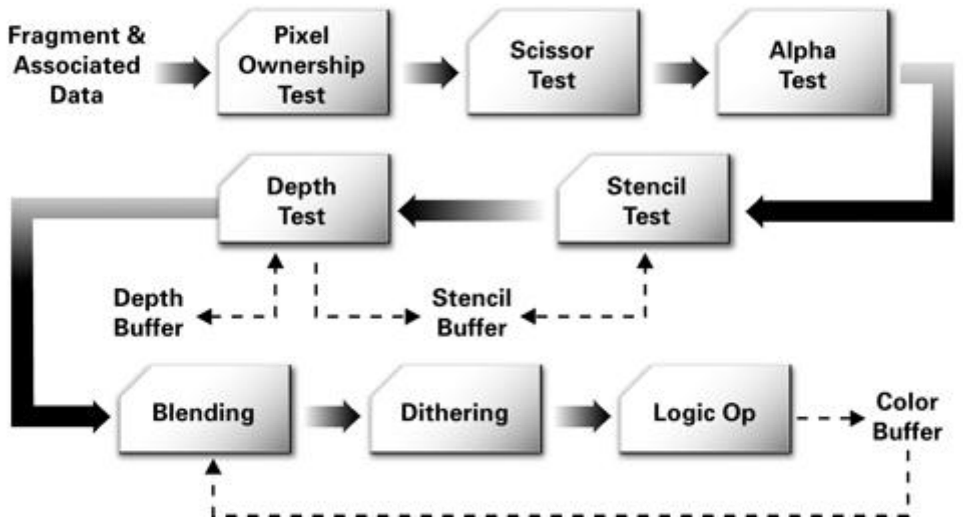


Figure 1-5 Standard OpenGL and Direct3D Raster Operations

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阶段。

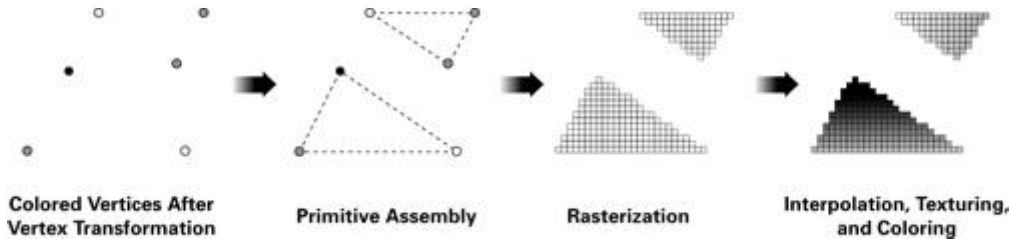
Figure 1-5 shows that the raster operations stage is actually itself a series of pipeline stages. In fact, all of the previously described stages can be broken down into substages as well.

1.2.3.5 形象化图形流水线

（**Visualizing the Graphics Pipeline**）

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。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多个片段。

Figure 1-6 depicts the stages of the graphics pipeline. In the figure, two triangles are rasterized. The process starts with the transformation and coloring of vertices. Next, the primitive assembly step creates triangles from the vertices, as the dotted lines indicate. After this, the rasterizer "fills in" the triangles with fragments. Finally, the register values from the vertices are interpolated and used for texturing and coloring. Notice that many fragments are generated from just a few vertices.



Colored Vertices After Vertex Transformation
Primitive Assembly
Rasterization
Interpolation, Texturing, and Coloring

Figure 1-6 Visualizing the Graphics Pipeline

1.2.4 可编程图形流水线

（**1.2.4 The Programmable Graphics Pipeline**）

当今图形硬件设计上最明显的趋势是在 GPU 内提供更多的可编程管线。图 1-7 显示了一个可编程 GPU 中的顶点处理和片段处理阶段。

The dominant trend in graphics hardware design today is the effort to expose more programmability within the GPU. Figure 1-7 shows the vertex processing and fragment processing stages in the pipeline of a programmable GPU.

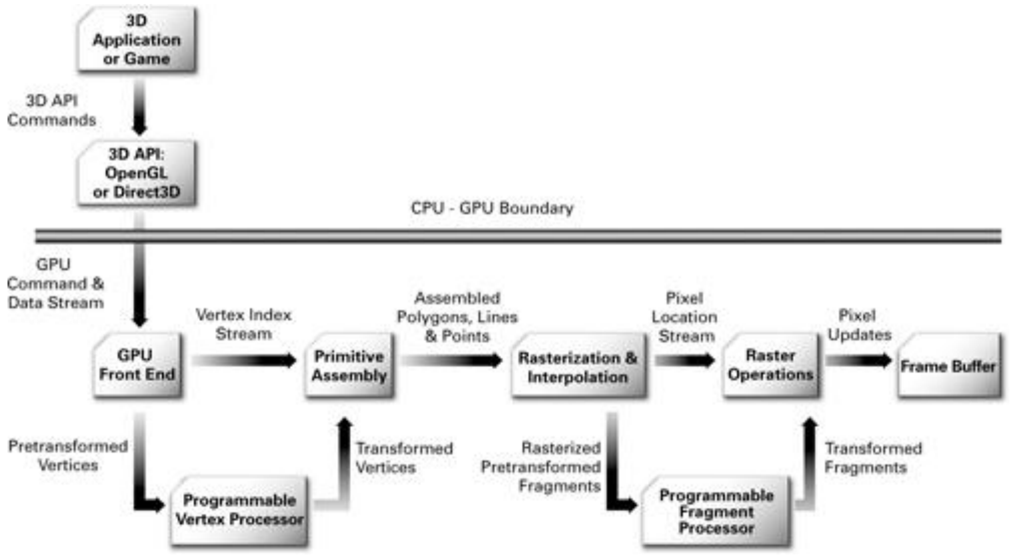


Figure 1-7 The Programmable Graphics Pipeline

图 1-7 比图 1-3 展示了更多的细节，更重要的是他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，可运行你的 Cg 顶点程序，而可编程片段处理器则是一个可运行你的 Cg 片段程序的单元。

Figure 1-7 shows more detail than Figure 1-3, but more important, it shows the vertex and fragment processing broken out into programmable units. The *programmable vertex processor* is the hardware unit that runs your C g vertex programs, whereas the *programmable fragment processor* is the unit that runs your Cg fragment programs.

正如第 1.2.2 小节中解释的，GPU 的设计水平已经发展为使得 GPU 中顶点和片段处理器已经从可配置转变为可编程了。随后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。

As explained in Section 1.2.2, GPU designs have evolved, and the vertex and fragment processors within the GPU have transitioned from being configurable to being programmable. The descriptions in the next two sections present the critical functional features of programmable vertex and fragment processors.

1.2.4.1 可编程顶点处理器

（**The Programmable Vertex Processor**）

图 1-8 显示了一个典型的可编程顶点处理器的流程图。顶点处理器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读取下一个指令并执行它，知道顶点程序结束。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，，包含了应用程序制定的顶点属性集。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寄存器。

Figure 1-8 shows a flow chart for a typical programmable vertex processor. The data-flow model for vertex processing begins by loading each vertex's attributes (such as position, color, texture coordinates, and so on) into the vertex processor. The vertex processor then repeatedly fetches the next instruction and executes it until the vertex program terminates. Instructions access several distinct sets of registers banks that contain vector values, such as position, normal, or color. The vertex attribute registers are read-only and contain the application-specified set of attributes for the vertex. The temporary registers can be read and written and are used for computing intermediate results. The output result registers are write-only. The program is responsible for writing its results to these registers. When the vertex program terminates, the output result registers contain the newly transformed vertex. After triangle setup and rasterization, the interpolated values for each register are passed to the fragment processor.

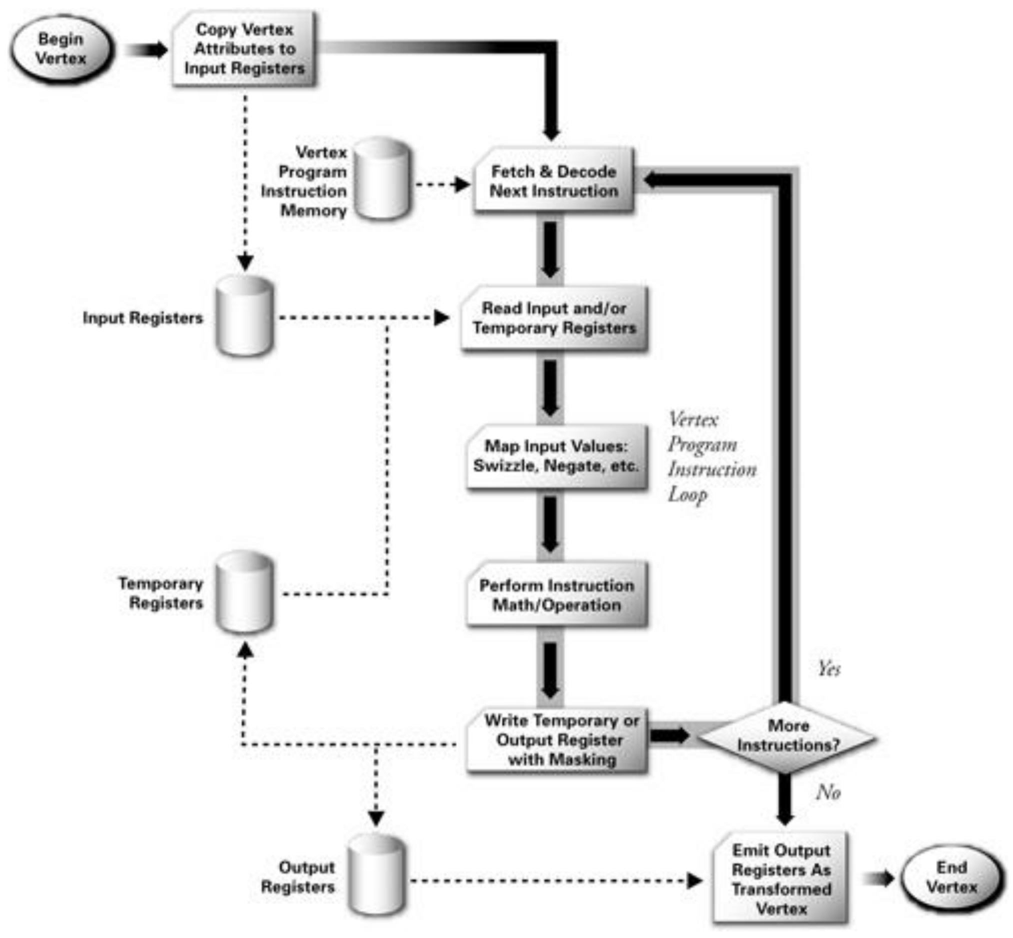

Begin Vertex
Copy Vertex Attributes to Input Registers
Vertex Program Instruction Memory
Fetch & Decode Next Instruction
Input Registers
Read Input and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Vertex Program Instruction Loop
Temporary Registers
Perform Instruction Math/Operation
Yes
Write Temporary or Output Register with Masking
More Instructions?
No
Output Registers
Emit Output Registers As Transformed Vertex
End Vertex

Figure 1-8 Programmable Vertex Processor Flow C hart

大部分的顶点处理仅适用了有限的操作，作用于二维、三维和四维浮点向量的向量数学操作时十分必要的。这些操作包括加、减、乘加、内积、最小值和最大值等。对向量取反和位移（任意重新安排向量分量位置的能力）的硬件支持能够使向量数学指令支持取反、减法和外积操作。分量掩码写入控制了所有的指令输出，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标准化。指数、对数和近似三角函数使得光照、雾化和几何计算变得更加便捷。专门的指令能够使得光照和淡化效果更加容易计算。

Most vertex processing uses a limited palette of operations. Vector math operations on floating-point vectors of two, three, or four components are necessary. These operations include add, multiply, multiply-add, dot product, minimum, and maximum. Hardware support for vector negation and component-wise swizzling (the ability to reorder vector components arbitrarily) generalizes these vector math instructions to provide negation, subtraction, and cross products. Component-wise write masking controls the output of all instructions. Combining reciprocal and reciprocal square root operations with vector multiplication and dot products, respectively, enables vector-by-scalar division and vector normalization. Exponential, logarithmic, and trigonometric approximations facilitate lighting, fog, and geometric computations. Specialized instructions
can make lighting and attenuation functions easier to compute.

其他更进一步的功能，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获得。

Further functionality, such as relative addressing of constants and flow-control support for branching and looping, is also available in more recent programmable vertex processors.

1.2.4.2 可编程片段处理器

（**The Programmable Fragment Processor**）
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，但是他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结果。

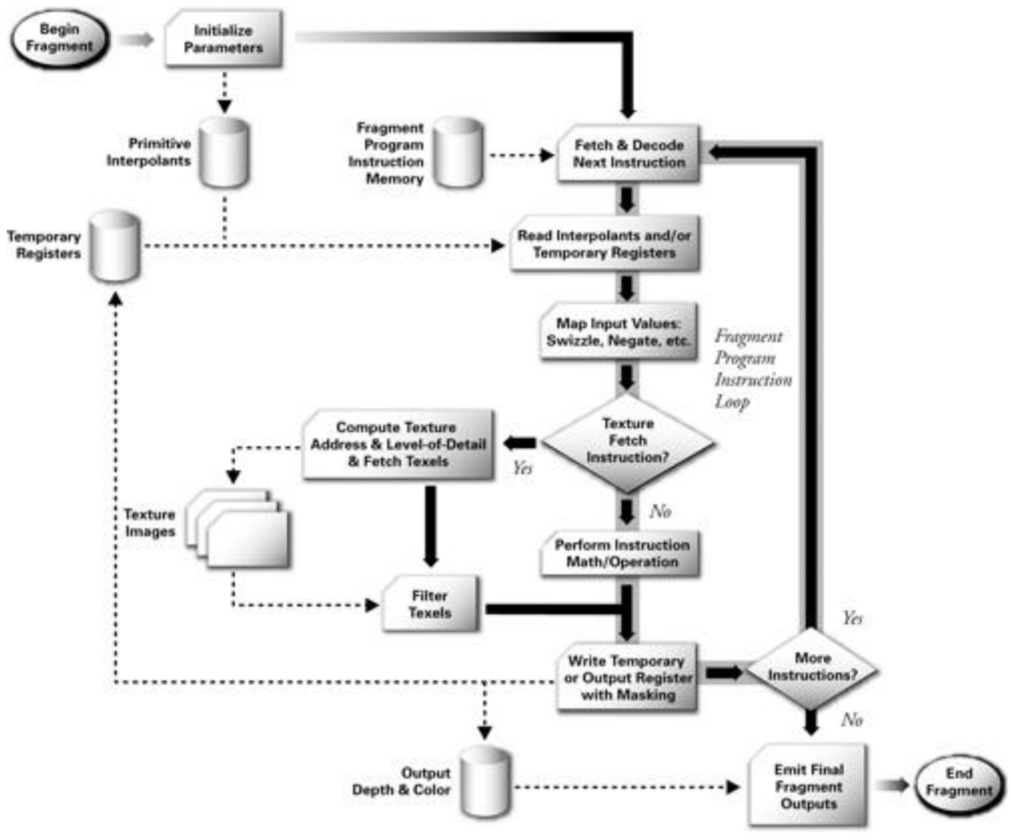
Programmable fragment processors require many of the same math operations as programmable vertex processors do, but they also support texturing operations. Texturing operations enable the processor to access a texture image using a set of texture coordinates and then to return a filtered sample of the texture image.

最新的 GPU 提供了浮点数支持：旧一些的 GPU 只有有限的定点数据类型。相比使用在处理低精度数据类型，片段操作使用浮点运算变得更加有效。GPU 必须同时处理相当多的片段。当代 GPU 还没有任意分支功能，但是随着硬件的发展，这一点很可能会改变。Cg 允许你通过使用条件赋值和循环操作模拟分支和迭代。

Newer GPUs offer full support for floating-point values; older GPUs have more limited fixed-point data types. Even when floating-point operations are available, fragment operations are often more efficient when using lower-precision data types. GPUs must process so many fragments at once that arbitrary branching is not available in current GPU generations, but this is likely to change over time as hardware evolves. C g still allows you to write fragment programs that branch and iterate by simulating such constructs with conditional assignment operations or loop unrolling.

图 1-9 显示了当代可编程片段处理器的流程图。就像顶点处理器那样，数据流设计执行一系列指令直到程序终止。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。

Figure 1-9 shows the flow chart for a current programmable fragment processor. As with a programmable vertex processor, the data flow involves executing a sequence of instructions until the program terminates. Again, there is a set of input registers. However, rather than vertex attributes, the fragment processor's read-only input registers contain interpolated per-fragment parameters derived from the per-vertex parameters of the fragment's primitive. Read/write temporary registers store intermediate values. Write operations to write-only output registers become the color and optionally the new depth of the fragment. Fragment program instructions include texture fetches.



Begin Fragment
Initialize Parameters
Primitive Interpolants
Fragment Program Instruction Memory
Fetch & Decode Next Instruction
Temporary Registers
Read Interpolants and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Fragment Program Instruction Loop
Compute Texture Address & Level-of-Detail & Fetch Texels
Texture Fetch Instruction?
Yes
No
Texture Images
Perform Instruction Math/Operation
Filter Texels
Yes
Write Temporary or Output Register with Masking
More Instructions?
No
Output Depth & Color
Emit Final Fragment Outputs
End Fragment

Figure 1-9 Programmable Fragment Processor Flow Chart

1.2.5 Cg 提供了顶点和片段的可编程能力

（1.2.5 Cg Provides Vertex and Fragment Programmability）

你的 GPU 中得这两种可编程处理器需要你——一个应用程序员，为每个处理器提供一个程序来执行：Cg 提供的事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。

These two programmable processors in your GPU require you, the application programmer, to supply a program for each processor to execute. What C g provides is a language and a compiler that can translate your shading algorithm into a form that your GPU's hardware can execute. With C g, rather than program at the level shown in Figures 1-8 and 1-9, you can program in a high-level language very similar to C .

第 2 章 最简单的程序

（**Chapter 2. The Simplest Programs**）

本章介绍了通过 Cg 进行一系列简单的顶点和片段程序进行编程。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得顶点程序，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 GPU 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，并介绍了片段程序的 profile。
“用顶点和片段的示例程序进行渲染”教你如何使用 OpenGL 和 Direct3D 来渲染简单的集合图形，这节还会涉及到剪裁的概念。

This chapter introduces C g programming through a series of simple vertex and fragment programs. The chapter has the following four sections:

- **"A Simple Vertex Program"** presents a straightforward vertex program and explains the basic elements and syntax of the C g language.
- **"Compiling Your Example"** explains how to compile programs for different GPUs, using the concept of profiles.
- **"A Simple Fragment Program"** defines a basic fragment program and introduces fragment profiles.
- **"Rendering with Your Vertex and Fragment Program Examples"** shows how to render simple geometry with OpenGL or Direct3D. This section also mentions the concept of clipping.

2.1 一个简单的顶点程序

（**2.1 A Simple Vertex Program**）

绿色是代表没有经验和成长的颜色，所以这个 Cg 程序将通过渲染一个绿色的三角形来开始 Cg 语言的学习。

Green is the color associated with inexperience and growth, so a C g program for rendering a green 2D triangle is a fitting way to start learning C g.

示例图 2-1 介绍了你用 Cg 编写的第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，内置函数和内置数据结构。我们这样标记将有助于你辨认程序中对 Cg 编译器有特殊意义的单词。在 Cg 中注释的用法和 C++一样：你可以使用 /* 和 */ 作为分隔符，或者你可以在注释前使用//符号。

Example 2-1 shows the complete source code for your first vertex program in C g. Source code examples in this book use boldface to indicate C g keywords, built-in functions, and built-in data types. This notation will help you identify words in programs that have special meaning to the C g compiler. In addition, comments in code samples are set in gray type, to distinguish them from the rest of the code. Comments in Cg work just as in C ++: you can use the /* and */ delimiters, or you can precede comments with the // characters.

例子中的命名规范

（**The Naming Convention for Examples**）

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 第 1 例 中的顶点程序 ”。我们使用这种标志方法是为了更容易地在书中各章和配套的软件中找到所需内容。这种命名规则使读者更容易找到这本书中不同的例子，但 Cg 语言本身并没有这种命名规范，而且在你自己的程序中也可以不使用这种命名规规则。

The vertex program in Example 2-1 is quite simple. The " C2E1v " prefix used in various parts of the program stands for "Chapter 2, Example 1 vertex program." We use this notation to make it easier to find examples across chapters and in the accompanying software framework. This convention makes it easier to keep track of the various examples in this book, but it is not a requirement of the C g language itself and indeed is not a convention intended for your own programs.

Example 2-1. The C2E1v_green **Vertex Program**

```
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;
}
```

如果你熟悉 C 或者 C++，你可能可以推断出这个程序所做的事情。该程序是将输入的二维顶点坐标的位置赋值到输出的三维顶点的坐标位置，并且将代表绿色的 RGBA 常量付给了顶点的输出颜色。

2.1.1 输出结构

（**2.1.1 Output Structures**）

C2E1v_green 程序是从以下这个声明开始的：

The C2E1v_green program begins with this declaration:

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

这里声明了一个专用结构。这个结构被称为输出结构。这个结构包含了一组给定 Cg 程序的返回值。

This declaration is for a special structure known as an *output structure*. This structure contains the bundle of values that represent the output (or result) of a given Cg program.

一个用多用途 CPU 语言（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，一直相反，Cg 被限制只能输出一组值。对一个给定的 Cg 程序，其输出结构封装了输出值的潜在范围。

A program written in a general-purpose CPU language such as C can perform a wide variety of tasks, such as reading and writing files, soliciting input from users, printing text, displaying graphics, and communicating over a network. C g programs, on the other hand, are limited to outputting a bundle of values. A Cg program's output structure encapsulates the potential range of output values for a given Cg program.

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。紧随其后的事结构的名字。大括号里是结构的定义，它包含了一系列结构成员，每个成员都有一个名字和类型。

Cg declares structures with the same syntax used in C and C ++. A structure declaration begins with the struct keyword, followed by the name of the structure. Enclosed in curly brackets, a structure definition contains a list of structure members, each with a name and a type.

输出结构域传统的 C 或者 C++结构不同，因为他的每个成员还包括了一个语义项。我们很快会子在 2.1.6 小节介绍语义的概念。

An output structure differs from a conventional C or C ++ structure because it includes *semantics* for each member. We will return to the concept of semantics soon in Section 2.1.6.

2.1.2 标识符

（**2.1.2 Identifiers**）

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。一个标识符不能是以数字开头，也不能是一个关键字。

When you declare a structure, you provide an *identifier,* or name, after the struct keyword; you do this for each structure you declare. An identifier in C g has the same form as in C and C ++. Identifiers consist of a sequence of one or more uppercase or lowercase alphabetic characters, the digits 0 to 9, and the underscore character (_). For example, Matrix_B and pi2 are valid identifiers. An identifier cannot begin with a digit and cannot be a keyword.

标示符不仅可以命名结构，也可以命名类型声明、结构成员、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

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

Identifiers not only name structures, but also name type declarations, members of structures, variables, functions, and semantics (you will learn more about each of these shortly). Other identifiers in Example 2-1 are these:

- C2E1v_green —the entry function name
- position —a function parameter
- OUT —a local variable
- float4 —a vector data type that is part of C g's Standard Library
- color and position —structure members
- POSITION and COLOR —semantics

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持了不同的命名空间。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 maintains different namespaces, based on the context of an identifier, in the same manner as C and C++. For example, the identifier position identifies a function parameter and a member of the C2E1v_Output structure.

Cg 中的关键字

许多 Cg 的关键字和 C 与 C++中的关键字相同，但是 Cg 使用了部分 C 与 C++中没有出现的关键字。随着阅读的进展，我们将解释这些关键字中的大部分内容。附录 D 是包含了全部 Cg 关键的一个列表。与 C 和 C++一样，Cg 的关键字也不能作为标识符使用。

Many Cg keywords are also C and C++ keywords, but Cg uses additional keywords not found in C or C++. Over the course of the book, we will explain most of these keywords. Appendix D contains the complete list of Cg keywords. As in C and C++, Cg's keywords are not available for use as identifiers.

2.1.3 结构成员

（**2.1.3 Structure Members**）

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称。

Within the curly brackets of a structure declaration, you will find one or more structure members. Each member is a data type with an associated member name.

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，如他们的类型 float4 所指示的。

In the C2E1v_Output structure, there are two members: position and color . Both members are four-component floating-point vectors, as indicated by their type, float4 .

2.1.4 向量

（**2.1.4 Vectors**）
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++中没有原生的向量类型，因此向量只不过是一组标量而已。因为向量对顶点和片段的处理是不可缺少的，并且 GPU 内置了对向量数据类型的支持，CG 具有向量数据类型。

The fundamental data types in C or C ++ are scalar quantities, such as int or float . In C or C++, there is no native "vector" type, so vectors are typically just arrays of scalar values. Because vectors are essential to vertex and fragment processing and because GPUs have built-in support for vector data types, C g has vector data types.

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，他是定义在 Cg 标准库中的标准类型。和 C 和 C++ 不同，包含 Cg 标准库的声明不需要指定一些预处理器指令（例如#include），Cg 自动的包含了大部分 Cg 程序所需要的声明。

The members (position and color) are declared using the float4 data type. This name is not a reserved word in Cg; it is a standard type definition in the C g Standard Library. Unlike in C and C ++, there is no need to specify a preprocessor statement (such as #include) to include declarations for C g's Standard Library. Instead, C g automatically includes the declarations needed by most Cg programs.

你需要依赖于由 Cg 标准库所提供的预先定义的向量数据类型。例如 float2、float3、float4 和其他一些类似的数据类型。以确保你的程序可以最有效的使用你的 GPU 最有效的使用你的 GPU 的向量处理能力。

You should rely on the predefined vector data types provided by the C g Standard Library, such as float2 , float3 , float4 , and other such types to ensure that your programs make the most efficient use of the vector processing capabilities of your programmable GPU.

Note
Cg 中的向量类型，如 float3、float4 等，并不是和一组 float 类型完全相等。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）。这些通常被称为向量的压缩数组，告诉编译器分配压缩数组类型的元素，使得这些变量的向量操作效率更高。如果两个向量以压缩的方式存储，GPU 通常能够在一个单独的指令里执行三元或四元的操作，例如乘法、加法和内积。

Vector types in C g, such as float3 and float4 , are not 100 percent equivalent to arrays of however many float s. For example, float x[4] is not the same declaration as float4 x . These vector types are, in fact, *packed arrays*. Packed arrays, often just called vectors, tell the compiler to allocate the elements of packed arrays so that vector operations on these variables are most efficient. If two input vectors are stored in packed form, programmable graphics hardware typically performs three-component or four-component math operations—such as multiplications, additions, and dot products—in a single instruction.

在传统的编程语言像 C 和 C++中压缩数组并不存在。最近的 CPU 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多数多用途编程语言本身所支持。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 GPU 提供的快速向量操作。

Packed arrays are not available in conventional programming languages like C and C++. Recent CPU instruction sets—such as Intel's SSE2, SSE, and MMX; AMD's 3DNow!; and Motorola's AltiVec—have additional vector instructions, but packed arrays are not natively supported by most general-purpose programming languages. C g, however, provides specific support for packed arrays because vector quantities are integral to vertex and fragment processing. Packed arrays help the C g compiler take advantage of the fast vector operations provided by programmable GPUs.

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

As with many aspects of C g, how you use packed arrays depends on the C g profile you select. For example, packed arrays are usually limited to four or fewer components. They are often extremely efficient for vector operations such as assignment, negation, absolute value, multiplication, addition, linear interpolation, maximum, and minimum. Dot-product and cross-product operations with packed operands are also very efficient.

但是，使用变量的数组索引随机访问压缩数组将使效率变低，或者 profile 根本不支持该类操作。例如：

On the other hand, accessing packed arrays with a non-constant array index is either inefficient or unsupported, depending on the profile. For example:

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

最好的规则就是通过 Cg 内置的向量类型把二元、三元和四元的向量（例如颜色、位置、纹理坐标和方向）声明为压缩数组类型。

The rule of thumb is to declare all vectors that consist of two, three, or four components (such as colors, positions, texture coordinate sets, and directions) as packed arrays by using C g's built-in vector types.

2.1.5 矩阵

（**2.1.5 Matrices**）

除了向量类型以外，Cg 本身还支持矩阵类型。这里有一些在 Cg 中声明的矩阵例子：

In addition to vector types, C g natively supports matrix types. Here are some examples of matrix declarations in C g:

```
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 C 和 C++一样的初始化数组的方法来声明初始化矩阵。

You can declare and initialize a matrix with the same notation that would be used to initialize an array in C or C++:

```
float2x3 matrix4 = { 1.0, 2.0,
                     3.0, 4.0,
                     5.0, 6.0 };
```

与向量类似的，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 GPU 上可以非常有效的执行。

Like vectors, matrices are packed data types in C g, so operations using standard matrix types execute very efficiently on the GPU.

2.1.6 语义

（**2.1.6 Semantics**）

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 成员之后。在某种意义上，意义是一种混合剂，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 程序返回他的输出结果时，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。

A colon and a special word, known as a *semantic*, follow the position and color members of the C2E1v_Output structure. Semantics are, in a sense, the glue that binds a C g program to the rest of the graphics pipeline. The semantics POSITION and COLOR indicate the hardware resource that the respective member feeds when the Cg program returns its output structure. They indicate how the variables preceding them connect to the rest of the graphics pipeline.

POSITION 语义（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给空间的位置。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变换后的剪裁空间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到，正式的解释将放在第 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窗口中的位置。

The POSITION semantic (in this case, in an output structure used by a C g vertex program) is the clip-space position for the transformed vertex. Later graphics pipeline stages will use the output vector associated with this semantic as the post-transform, clip-space position of the vertex for primitive assembly, clipping, and rasterization. You will be introduced to clip space later in this chapter, and more formally in Chapter 4. For now, you can think of a 2D vertex's clip-space position simply as its position within a window.

在本文中的 COLOR 语义是 Direct3D 中的“漫反射顶点颜色”和 OpenGL 中的“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。

The COLOR semantic in this context is what Direct3D calls the "diffuse vertex color" and OpenGL calls the "primary vertex color." Color interpolation for a triangle or other geometric primitive during rasterization depends on the primitive's per-vertex colors.

Note
不要把一个成员名字和他的语义搞混。在示例 2-1 中，position 成员与 POSITION 语义相关联。但是，是因为成员名称后的 POSIITON 语义（而不是 position 成员本身），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 选择得并不是很恰当。尽管这些变量名有误导性，但 Cg 仍然严格按照语义的内容执行。

Do not confuse a member name with its semantic. In Example 2-1, the position member is associated with the POSITION semantic. However, it is the use of the POSITION semantic after the member name— not the name of the member itself—that makes the rasterizer treat the position member as a position. In the following output structure, the member names density and position are poorly chosen, but Cg abides by the specified semantics despite the misleading names:

```
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。并不是所有语义在所有的 profile 里都存在，但是在我们的示例中，我们将使用被现有的 profile 广泛支持的语义。

Subsequent examples will introduce other output semantic names. Not all semantics are available in all profiles, but in our examples, we will use the semantics that are broadly supported by existing profiles.

你还可以创建自己的语义名，但是在本书中，我们把示例限制在只是用标准的语义集。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*.

You can also create your own semantic names, but in this book, we limit our examples to the standard set of semantics. For more information about using your own semantic names, see the *Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics*.

2.1.7 函数

（**2.1.7 Functions**）
在Cg 中声明函数与在C和C++中采用的方式一样。你需要为函数制定一个返回类型（如果不返回任何东西，则使用void）、他的名字和一个放在括号里用逗号隔开的参数列表。在函数声明之后，函数体描述了该函数所要执行的计算内容。

Declaring functions in C g works in much the same way as it does in C and C ++. You specify a return type for the function (or void if nothing is returned), its name, and a comma-separated parameter list in parentheses. After the declaration, the body of the function describes the computation performed by the function.

函数既可以是入口函数，也可以是内部函数。

Functions can be either entry functions or internal functions.

2.1.7.1 入口函数

（**Entry Functions**）

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，被命名为C2E1v_green 的入口函数是按如下方式定义的：

An *entry function* defines a vertex program or fragment program; it is analogous to the main function in C or C ++. A program's execution starts in its entry function. In Example 2-1, the entry function called C2E1v_green is defined as follows:

```
C2E1v_Output C2E1v_green(float2 position : POSITION)
```

这个函数返回的输出结构C2E1v_Output在前面已描述过。这意味着该函数同时输出一个位置和一个颜色。这些输出拥有结构所定义的语义。

This function returns the output structure C2E1v_Output described earlier. This means that the function outputs both a position and a color. These outputs have semantics defined by the structure.

这个函数还接受命名为position的输入参数。该参数是float2类型的，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之后的时候，就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，就告诉顶点处理器采用每个由应用程序指定的函数处理的顶点来初始化参数。

The function also accepts an input parameter named position . This parameter has the type float2 ,so it is a floating-point vector with two components. When a colon and semantic name follow an input parameter name, this indicates the semantic associated with that input parameter. When POSITION is used as an input semantic, this tells the vertex processor to initialize this parameter with the application-specified position of every vertex processed by the function.

2.1.7.2 内部函数

（**Internal Functions**）

内部函数是辅助函数，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吗，也可以使用自己定义的内部函数。

Internal functions are helper functions called by entry functions or other internal functions. You can use the internal functions provided by the C g Standard Library, and you can define your own internal functions as well.

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

Internal functions ignore any semantics applied to their input or output parameters or return values; only entry functions use semantics.

2.1.8 输入和输出语义是不同的

（**2.1.8 Input and Output Semantics Are Different**）

图2-1显示了C2E1v_green顶点程序的输入和输出语义流程。

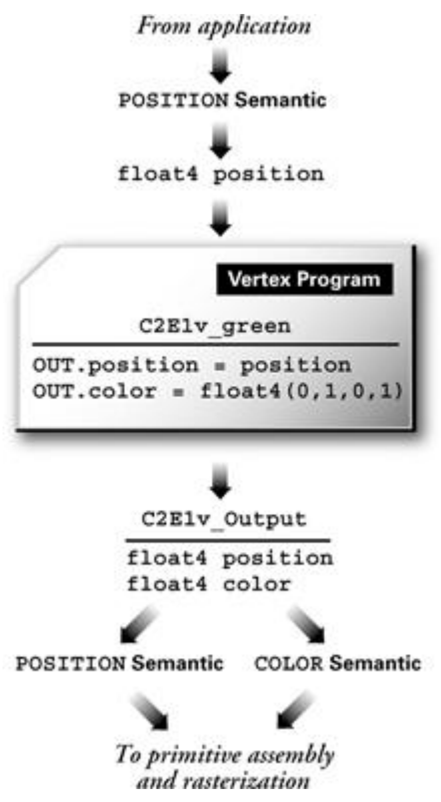
Figure 2-1 shows the flow of input and output semantics for the C2E1v_green vertex program.

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：当一个顶点传递给GPU处理的时候，由程序指定的位置。然而，一个输出结构中使用POSITION语义的成员就表示要输入给硬件光栅器的剪裁空间位置。

Input and output semantics are not the same, even though some have the same names. For example, for an input parameter to a vertex program, POSITION refers to the application-specified position assigned by the application when it sends a vertex to the GPU. However, an output structure element with the POSITION semantic represents the clip-space position that is fed to the hardware rasterizer.

两个语义都命名为POSITION，而且实际上两者确实都在同一个位置。但是每个POSITION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转变为合适图元装配、剪裁和光栅化的位置信息。在C2E1v_green中的转换是微不足道的（顶点位置在传递前没有丝毫改变），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变换顶点的方法。

Both semantics are named POSITION and, indeed, each is a position. However, each position semantic refers to a position at different points along the graphics pipeline. Your C g vertex program transforms the application-supplied vertex position into a vertex suited for primitive assembly, clipping, and rasterization. In the C2E1v_green program, this transformation is trivial (the vertex position is passed along unchanged), but later in this book, particularly in C hapters 4 and 5, you will learn more useful and interesting ways to transform vertices.



2.1.9 函数体

（**2.1.9 The Function Body**）

C2E1v_green函数的主要内容包含在它的函数体内。

The substance of the C2E1v_green function is contained in its body:

```
{
C2E1v_Output OUT;
OUT.position = float4(position, 0, 1);
OUT.color = float4(0, 1, 0, 1); // RGBA green
return OUT;
}
```

由于函数的返回类型是C2E1v_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成为输出结构。函数体为这个结构变量的每个成员赋值并返回这个结构（请注意虽然我们在所有的示例中选择使用与入口函数有相同的前缀的返回类型，但这并非必须的）。

Because the function's return type is C2E1v_Output , you must declare a variable of this type to hold the value that the function returns. We often call a structure returned by an entry function an *output structure*. The function body sets both elements of this structure variable and returns the structure. (Note that the entry function's return type is not required to have the same prefix as the entry function, although we've chosen to make them the same in our examples.)

OUT 和position以及OUT和 color之间的句点是成员操作符，他是你能够存取结构中的一个成员。这与C和C++存取结构成采用的方法是一样的。你可以把结构想象成一个容纳多个值的容器。成员操作符使你能够获得这个结构所包含的值：

The dot between OUT and position , as well as OUT and color , is the *member operator* and gives access to a member within a structure. This is the same way that C and C++ access structure members. Think of a structure as a container for multiple values. The member operator lets you retrieve the values contained in a structure:

```
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赋值给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）分别通过吧第三和第四成员赋值为0和1，把一个二元位置向量转变化为一个四元向量。

First, the program assigns the position input parameter to OUT.position . However, the output structure member OUT.position is of type float4 (Chapter 4 explains why). The expression float4 (position, 0, 1) converts a two-component position vector to a four-component vector by setting the third and fourth components to 0 and 1, respectively.

其次，程序要把代表绿色的RGBA颜色赋值给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，因为他是一个RGBA颜色。RGBA中的“A”代表“alpha”，它通常表示一个颜色值的透明通道属性。Alpha值为1时表示该颜色完全不透明。

Second, the program assigns the RGBA color value for green to OUT.color . To provide the numeric value for green, construct the appropriate four-component color vector. The type of the color member is float4 because the color is an RGBA color. The "A" in RGBA stands for "alpha," which normally encodes a measure of how opaque or transparent a color value is. The value 1 for alpha means the color is fully opaque.

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。构造函数将使用列在括号内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，构造是为向量和矩阵准备的。

When you use the float4 or similar vector type name like a function (for example, float4(0, 1, 0, 1)), this is a called a *constructor*. This constructor creates a value of the type specified out of the values listed in the parentheses. C ++ has the concept of constructors, but C does not. Constructors in C g are provided for vectors and matrices.

语法float4（0，　1，　0，　1）创建一个向量<0, 1, 0, 1>，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顺序的颜色成员里只有绿色和alpha是有效的，而红色和蓝色没有任何贡献。

The syntax float4(0, 1, 0, 1) creates a vector <0, 1, 0, 1> that is assigned to the color member of type float4 in OUT . The vector <0, 1, 0, 1> is green because the color components are in the red, green, blue, alpha (RGBA) order with a green (and alpha) contribution specified, but no red or blue contribution.

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。

Finally, the return statement returns the output structure you initialized. The collection of values within OUT is passed along to the next stage of the graphics pipeline, as indicated by the semantics assigned to each member.

2.2 编译你的例子

（**2.2 Compiling Your Example**）

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：

1） 将要编译的入口函数名
2） 编译入口函数所需要的profile名

示例2-1的入口函数名是C2e1v_green。

You use the C g runtime to load and compile C g programs. When you compile a program, you must specify two things in addition to the text of the program:

- The name of the entry function to compile
- The profile name for the entry function to compile

The name of the entry function in Example 2-1 is 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GPU硬件处理能力来选择合适的profile。

C2E1v_green is a vertex program, so you need to compile for a vertex profile. The vertex profile you choose depends on the programming interface your application uses for 3D rendering (OpenGL or Direct3D), as well as the hardware capabilities of your GPU.

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GPU一定会支持功能更强的profile。

Profile Name	**Programming Interface**	**Description**
arbvp1	OpenGL	Basic multivendor vertex programmability (corresponding to ARB_vertex_program functionality)
vs_1_1	DirectX 8	Basic multivendor vertex programmability
vp20	OpenGL	Basic NVIDIA vertex programmability (corresponding to NV_vertex_program functionality)

vs_2_0 vs_2_x	DirectX 9	Advanced multivendor vertex programmability
vp30	OpenGL	Advanced NVIDIA vertex programmability (corresponding to NV_vertex_program2 functionality)

Table 2-1. Cg Vertex Profiles

There are several appropriate vertex profiles for compiling our example, as listed in Table 2-1. Future GPUs will no doubt support profiles that are more capable.

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，你将遇到需要高级的顶点或片段profile编译的的复杂Cg 程序。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。

Your first example is very simple, so there is no problem compiling it with any of the profiles in Table 2-1, or any future vertex profile for that matter. As you progress through the book, you'll encounter some complex Cg programs that will require advanced vertex or fragment profiles. When an advanced profile is required, we are careful to point that out. Most examples in this book are written to compile for a broad range of Cg profiles.

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OpenGL最好的profile。Vs_1_1是DirectX8最好的profile。是否有选择其他profile的理由呢？当然，如果你想要使用高级的顶点可编程功能，比如在基本profile中不存在的复杂的流控制或快速的硬件指令的时候， 你就需要另外的profile。比如，如果你选择vp30 profile，你可以编写一个可以执行任意次数循环的Cg顶点程序。

Because you'll want your C2E1v_green example to compile on the broadest range of GPUs, the best profiles for your example are arbvp1 for OpenGL and vs_1_1 for DirectX 8. Is there any reason to choose another profile? Yes, if you want to use advanced vertex programmability functionality, such as complex flow control or fast hardware instructions, which are not available in the basic profiles. For example, if you choose the vp30 profile, you can write a C g vertex program that loops a varying (non-constant) number of times.

Note
如果有足够编译你所有Cg程序的一个基本的profile，使用它可以获得更广泛的硬件支持。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规范来编写程序。

If there is a basic profile that is sufficient for compiling your C g example, use it to gain the broadest hardware support. However, if you choose a more advanced profile, your C g program may execute more efficiently and you can program using more general programming practices.

那么，什么是更加高级的profile的缺点呢？他将会限制你的程序只能在比较新的GPU上运行，为了两方面都取得较好的结果——广泛的硬件支持和对最新硬件的支持——你需要为基本的profile提供一个可靠的Cg程序，而为高级的profile提供一个高级的Cg程序。CgFX格式通过提供一个统一的方法，在一个单独的文件中封装针对某一渲染效果的多个Cg实现来简化这种方法。附录C解释了更多关于CgFX的知识。

So what is the disadvantage of a more advanced profile? It may limit your program to newer GPUs. To get the best of both worlds—broad hardware support as well as the latest available hardware—you might provide both a fallback C g program for basic profiles and a more advanced C g program for more advanced profiles. The CgFX format simplifies this approach by providing a unified way to encapsulate multiple Cg implementations of a given rendering effect in a single source file. Appendix C explains more about CgFX.

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，你的程序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有选择的二维编程接口（OpenGL或Direct），完成你程序的渲染。

Refer to Appendix B to learn how an application can use the C g runtime library to load and compile Cg programs. In general, you call a set of C g runtime routines that require your program text, your entry function name, and your chosen profile. If there are errors in your program, the compilation will fail. You can request a list of compile errors to assist you in correcting your code. Once your program compiles successfully, other C g runtime routines assist you in configuring your 3D programming interface of choice (OpenGL or Direct3D) to render with your program.

2.2.2 Cg 编译错误类型

（2.2.2 Classes of Cg Compilation Errors）

Cg程序有两个类型的编译错误：普通编译错误和基于profile配置导致的错误。

There are two classes of compilation errors for C g programs: conventional and profile-dependent.

普通编译错误是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C与C++程序员处理的编译错误基本没有区别。

Conventional errors are caused either by incorrect syntax, usually due to typos, or by incorrect semantics, such as calling a function with the wrong number of parameters. These types of errors are not fundamentally different from the everyday compile errors that C and C++ programmers deal with.

基于profile配置导致的错误是由于在使用Cg的时候虽然语法和语义是正确的、但却不被你所指定的profile支持造成的。你也许写了有效的Cg程序，但你所指定的profile却不能编译。多用途语言程序没有这种类型的错误。

Profile-dependent errors result from using C g in a way that is syntactically and semantically correct but not supported by your specified profile. You may have written valid C g code, but it might not compile because of the profile you specified. General-purpose programming languages do not have this type of error.

2.2.3 基于 profile 的错误

（**2.2.3 Profile-Dependent Errors**）

基于profile的错误通常是由三维编程接口和你试图用来编译程序的下层GPU硬件限制所造成的。Cg 中有三类基于profile的错误：功能（capability）、上下文环境（context）和容量（capacity）

Profile-dependent errors are usually caused by limitations of the 3D programming interface and the underlying GPU hardware for which you are attempting to compile your program. There are three categories of profile-dependent errors: capability, context, and capacity.

2.2.3.1 功能

（**Capability**）

所有当前片段程序的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。之所以这样的原因是非常简单的：目前大部分GPU中的可编程顶点处理器不支持纹理存取。未来的顶点profile可能会允许纹理存取操作。

All current profiles for fragment programs permit texture accesses, but no current vertex profiles do. The reason for this is simple. The programmable vertex processors in most current GPUs do not support texture accesses. Future vertex profiles are likely to permit texture accesses.

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（而你却在程序中尝试存取纹理），一个基于profile的功能错误将会产生。硬件或三维编程接口缺乏对应的功能来实现你试图通过Cg表达的任务。

Cg does not allow you to compile a program that is impossible to execute, given the specified profile for compilation. If a vertex profile does not support texture accesses, a profile-dependent error of capability occurs. The hardware, or the 3D programming interface, lacks the ability to do what C g allows you to express.

2.2.3.2 上下文错误

（**Context**）

一个基于profile的上下文错误是非常基本的，虽然非常罕见。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会产生位置信息。

An error of profile-dependent context is more fundamental, though rare. For example, it is an error to write a vertex program that does not return exactly one parameter that is bound to the POSITION semantic. This is because the remainder of the graphics pipeline assumes that all vertices have a position.

同样， 一个片段profile不可以像顶点profile必须返回一个POSITION那样返回一个POSIOIN语义。这样的错误是由于使用了与图形流水线不一致的数据流规则造成的。

Likewise, a fragment profile cannot return a POSITION the way a vertex profile must. Such errors are caused by using C g in a manner inconsistent with the data flow of the graphics pipeline.

2.2.3.3 容量错误

（**Capacity**）

容量错误是由于GPU处理能力的限制而产生的。有些GPU只能在一个渲染过程中执行4个纹理存取。而其他的GPU片段程序指令数目允许的情况下，在一个渲染过程中可以执行任意多的纹理存取。

Capacity errors stem from a limit on a GPU's capability. Some GPUs can perform only four texture accesses in a single rendering pass. Other GPUs can perform any number of texture accesses in a single rendering pass, restricted only by the number of fragment program instructions the hardware supports. If you access more than four textures in a profile that does not permit access to more than four textures, you receive a capacity error.

容量错误可能是最让人沮丧的，因为从程序的角度来看究竟是哪部分容量超出了限制是非常不明显的。例如，你也许超过了GPU所允许的一个顶点程序的最大指令数目，但是这个现象并不是那么明显。

Capacity errors are probably the most frustrating, because it may not be apparent from looking at your program what the exceeded capacity is. For example, you may have exceeded the maximum number of vertex program instructions allowed by the GPU in a single program, but that fact may not be obvious.

2.2.3.4 预防错误

（**Preventing Errors**）

有两种方法可以避免令人灰心的错误发生，一种是用跟高级的profile。一个profile越高级，你碰到的profile能力和容量限制的可能就越小。随着可编程GPU的功能提高，你将不用为功能和容量的限制而担忧。

There are two ways to avoid these frustrating errors. One is to use a more advanced profile. The more advanced a profile is, the less chance you have of bumping into the capability and capacity limits of the profile. As the functionality of programmable graphics hardware improves, you will worry less and less out capability and capacity limits.

另一种解决办法是扩展你自己在三维应用程序内是用profile的功能、上下文和容量限制方面的知识。参加Cg Toolkit的配套文档可以学习到有关这些限制的知识。

Another solution is to educate yourself about the limitations of capability, context, and capacity for the profiles you use in your 3D application. Consult the documentation that accompanies the C g Toolkit to learn about these limits.

通过了解你所使用的最基本的三维编程接口的限制，可以增强你对由于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可能受到基于profile限制的构造。

You can often get a good sense of the profile-dependent restrictions by knowing the limitations of the raw 3D programming interface you are using. Consult your OpenGL and Direct3D documentation; it will help you identify C g constructs that might be subject to profile-dependent limitations.

2.2.4 标准：多入口函数

（**2.2.4 The Norm: Multiple Entry Functions**）

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是一个完整的程序，但它没有一个被命名为main的过程。为什么呢？因为我们命名了C2E1v_green为入口函数。在这本书中，我们保持我们的命名规则来区分我们的示例程序与其他程序，并且是他们更容易被找到。然而在你自己的3D应用程序里，你可以用任何你选择的名称来命名入口函数。只要这个函数是一个有效的标识符就可以了。

C and C ++ programs begin executing when the operating system invokes a program instance and calls the program's main routine (or WinMain routine for Windows programs). Example 2-1 is complete, but it has no routine named main . Why? Because, instead, we named the entry function C2E1v_green . In this book, we adhere to our naming convention to distinguish our examples from each other and to make them easy to locate. In your own 3D application, however, you can name the entry function whatever you choose, as long as the name is a valid identifier.

你的3D应用程序通常会使用一系列的Cg程序。最少的情况下，你也会使用一个顶点程序和一个片段程序，虽然你也可以使用固定管线来处理顶点、片段或者两者同时处理。一个复杂的应用程序也许会有上百个Cg程序。因为Cg程序可以在运行时便已，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来Cg程序本身的文本来生成新的Cg程序。

Your 3D application will typically use a collection of C g programs, not just one. At a minimum, you will probably have one vertex program and one fragment program, though you can use the fixed-function pipeline for vertex processing, fragment processing, or even both if you wish. A complex application may have hundreds of Cg programs. Because Cg programs can be compiled at runtime, you can even generate new Cg programs while your application is running, by formatting C g program text procedurally.

当然，你也可以使用main这个名称。当你编译Cg源码的时候如果没有明确地指出入口函数名，那么main将被作为默认入口函数使用。

Of course, you can still use the name main , which is the default entry function name for C g programs if no explicit entry function name is specified when you compile C g source code.

Note
请为你的入口函数找到一个描述性的名称以避免混淆。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，即使他是被运行库假设为缺省的入口函数名。

To avoid confusion, find descriptive names for your entry functions. If all your entry functions are named main , it will be difficult to have several entry functions in a single C g source file—even if that is the default entry function name assumed by the runtime.

2.2.5 加载和配置顶点和片段程序

（**2.2.5 Downloading and Configuring Vertex and Fragment Programs**）

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例程里面的代码。如果main例程结束返回，程序就会终止。

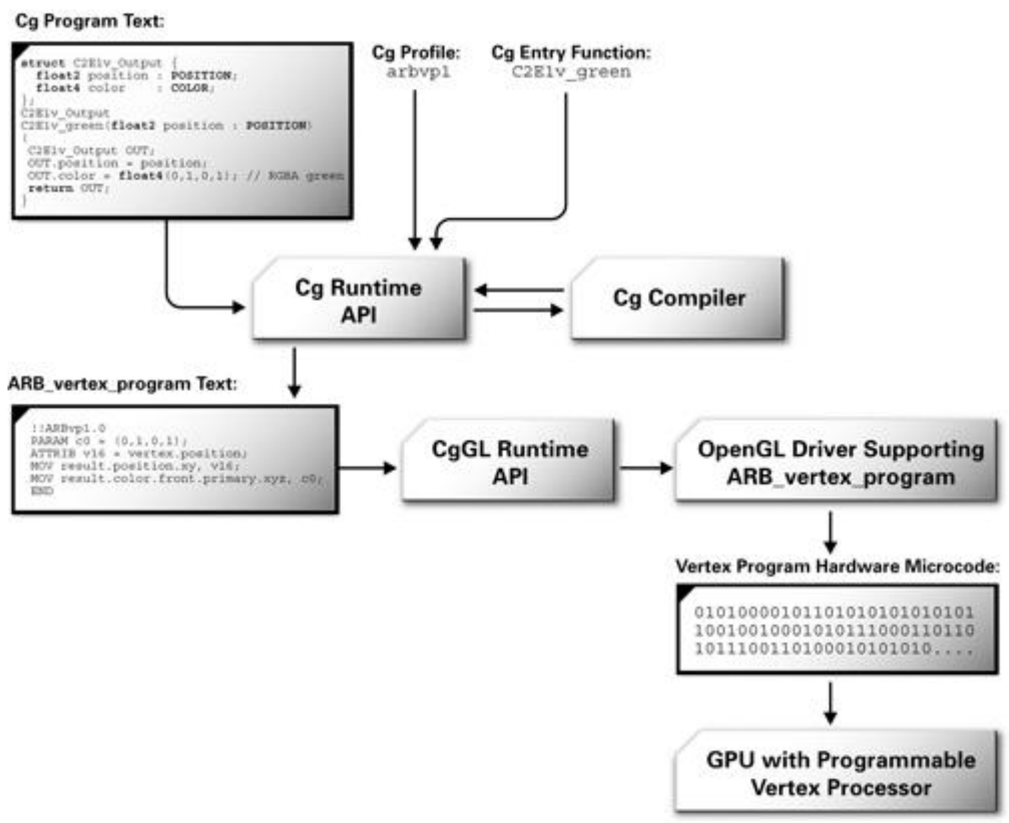
In a general-purpose language, the operating system invokes the main (or WinMain) routine and the program executes the code contained in that main routine. If the main routine returns, the program terminates.

但是，在Cg里你不需要像现在C和C++中那样，调用一个程序运行直到他终止。相反，Cg编译器把你的程序翻译成可以加载到硬件的3D编程接口的格式。这就是说，将由你的应用程序来调用必要的Cg运行库和3D编程接口例程来加载和配置你的程序，使他们可以被GPU使用。

However, in C g, you do not invoke a program that runs until it terminates, as you would in C or C++. Instead, the C g compiler translates your program into a form that your 3D programming interface can download to hardware. It is up to your application to call the necessary Cg runtime and 3D programming interface routines to download and configure your program for use by the GPU.

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代码，使得GPU的顶点处理器在变换顶点的时候可以直接执行。

Figure 2-2 shows how an application compiles a C g program and converts it into a binary microcode that the GPU's vertex processor directly executes when transforming vertices.



```
struct C2E1v_Output {
  float2 position : POSITION;
  float4 color    : COLOR;
};
C2E1v_Output
C2E1v_green(float2 position : POSITION)
{
 C2E1v_Output OUT;
 OUT.position = position;
 OUT.color = float4(0,1,0,1); // RGBA green
 return OUT;
}
```

```
!!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
```

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

Figure 2-2 Compiling and Loading a C g Program into the GPU

一旦加载了顶点程序，在你GPU中的顶点处理器每当应用程序提供一个顶点给GPU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。

Once loaded with a vertex program, the programmable vertex processor in your GPU runs that program every time the application feeds a vertex to the GPU. When it renders a complex model with thousands of vertices, your current vertex program processes every vertex in the model. The vertex program runs once for each vertex.

当一个片段程序应用到你的GPU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3D编程接口来加载你的程序并把它绑定为当前片段程序，来处理有光栅器产生的片段。当你的片段程序绑定后，3D图元被光栅化为片段，然后当前的片段程序将处理每一个生成的片段。片段程序为每个片段运行一次。

This same concept of a single current program applies to the programmable fragment processor in your GPU too. You compile your Cg fragment program and use the C g runtime, along with your 3D programming interface, to download your program and bind it as the current fragment program for processing fragments generated by the rasterizer. After your fragment program is bound, 3D primitives are rasterized into fragments and your current fragment program processes each generated fragment. The fragment program runs once for each fragment.

通常，3D应用程序同时对GPU中的可编程顶点和片段处理器进行编程，已达到某种特殊的渲染效果。因为这个方法对GPU中的可编程顶点和片段处理器的并行性和高度流水线化是非常

有效的。

Typically, 3D applications co-program the programmable vertex and fragment processors in the GPU to achieve a particular rendering effect. This approach is very efficient because of the parallel and highly pipelined nature of the programmable vertex and fragment processors in GPUs.

2.3 一个简单的片段程序

（**2.3 A Simple Fragment Program**）

到目前为止，我们的示例仅涉及了一个顶点程序，C2E1v_green，现在将介绍一个简单的片段程序，你可以使用它来里顶点程序的输出。

So far, our example involves only a vertex program, C2E1v_green . This section presents a simple fragment program that you can use with our vertex program.

示例2-2显示了我们第一个片段程序的全部Cg源码。

Example 2-2 shows the complete Cg source code for our first fragment program.

Example 2-2. The C2E2f_passthrough **Fragment Program**

```
struct C2E2f_Output {
float4 color : COLOR;
};
C2E2f_Output C2E2f_passthrough(float4 color : COLOR)
{
C2E2f_Output OUT;
OUT.color = color;
return OUT;
}
```

这个程序甚至比顶点示例程序C2E1v_green更加简单。实际上，这个程序什么也没有做。这个程序知识把光栅器赋给了每个片段的插值颜色不经改变就直接输出。如果片段通过了各种光栅操作测试，如剪切和深度测试，GPU的光栅操作硬件就使用这个颜色来更新帧缓存。

This program is even simpler than the example for the C2E1v_green vertex program. In fact, it does almost nothing. The program outputs the unchanged interpolated color assigned for every fragment generated by the rasterizer. The GPU's raster operation hardware uses this color to update the frame buffer if the fragment survives the various raster operations, such as scissoring and depth testing.

这是一个片段程序C2E2f_passthrough返回的的输出结构：

Here is the output structure returned by C2E2f_passthrough :

```
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。而一个片段程序将这些输出精简到一个单独的颜色结构中以用来更新帧缓存（在高级的profile中，片段程序也可以输出其他数据，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。

Fragment programs have a simpler output structure than vertex programs. A vertex program must output a position and may return one or more colors, texture coordinate sets, and other per-vertex outputs. A fragment program, however, must reduce everything to a single color that will update the frame buffer. (In some advanced profiles, fragment programs can write additional data such as a depth value as well.) The COLOR semantic assigned to the color member in a fragment program indicates that the member is the color to be used to update the frame buffer.

C2E2f_passthrough声明的入口函数是

The entry function declaration for C2E2f_passthrough is this:

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

这个函数的返回只含有一个颜色的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输入语义是片段使用光栅器基于图元的顶点颜色插值而产生的颜色。

The function returns the C2E2f_Output output structure with one color. The function receives a single four-component vector, named "color," bound to the COLOR input semantic. The COLOR input semantic for a fragment program is the color of the fragment interpolated by the rasterizer, based on the primitive's assigned vertex colors.

C2E2f_passthrough的函数体定义如下：

The body of C2E2f_passthrough is this:

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

在用C2E2f_Output输出结构类型定义了一个OUT变量之后，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。

After declaring an OUT variable with the C2E2f_Output output structure type, the program assigns the fragment's interpolated color (the single input parameter) to the final fragment color in the output structure. Finally, the program returns the OUT structure.

2.3.1 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是给顶点程序使用的。想

要编译C2E2f_passthrough，你必须选择一个适合的片段profile。

Just as you needed a profile to compile the C2E1v_green example, you also need a profile to compile the C2E2f_passthrough example. However, the profiles for compiling the C2E1v_green example were for vertex programs. To compile C2E2f_passthrough , you must choose an appropriate fragment profile.

表2-2列出了各种用于编译片段程序的profile。

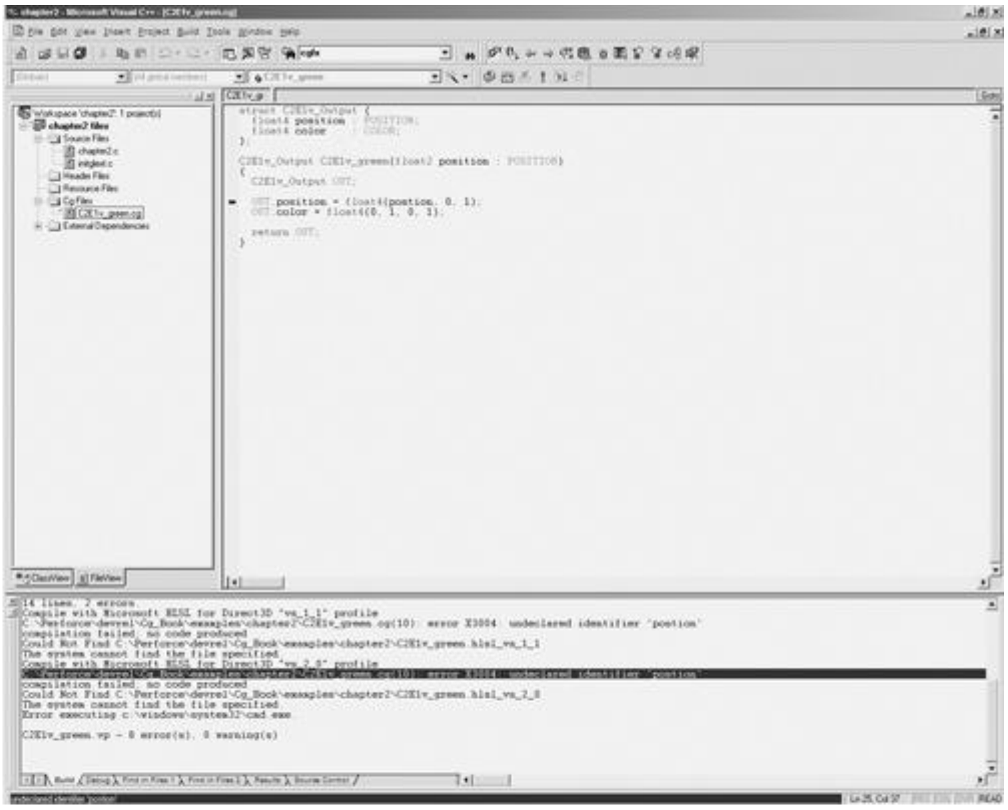
Table 2-2 lists various profiles for compiling fragment programs.

Table 2-2. Cg Fragment Profiles

Profile Name	**Programming Interface**	**Description**
ps_1_1 ps_1_2 ps_1_3	DirectX 8	Basic multivendor fragment programmability
fp20	OpenGL0	Basic NVIDIA fragment programmability (corresponding to NV_texture_shader and NV_register_combiners functionality)
arbfp1	OpenGL	Advanced multivendor fragment programmability (corresponding to ARB_fragment_program functionality)
ps_2_0 ps_2_x	DirectX 9	Advanced multivendor fragment programmability
fp30	OpenGL	Advanced NVIDIA fragment programmability (corresponding to NV_fragment_program functionality)

和早些的顶点示例一样，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中的任何一种profile来编译C2E2f_passthrough示例。Cg有一个被简称为cgc的命令行编译器，在运行是时的动态编译功能及其强大，因此我们极力推荐。但是，当你在编写Cg程序的时候，你通常想要不运行你的3D应用程序，只要检验编译结果是否正确。如果想要在编写程序的时候查找Cg程序的编译错误，试着运行下cgc。他将尝试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IDE中使用cgc将使得发现编译错误更便捷。一个好的IDE甚至可以给予错误信息中的行号，帮助你迅速而正确地定位错误代码所在行，就上你编译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。

Like the earlier vertex program example, this first fragment program example is so simple that you can compile the C2E2f_passthrough example with any of the profiles in Table 2-2. Cg has a command-line compiler known as cgc , which is short for "C g compiler." Dynamic compilation at runtime can be very powerful and is highly recommended. However, when you are writing Cg programs, you often want to verify that they compile correctly without having to run your 3D application. To detect Cg compiler errors while writing programs, try running cgc . It will "test compile" the C g program files used by your application as part of your regular application build process. Using cgc in an integrated development environment (IDE) such as Microsoft's Visual C ++ can make it quick and easy to find compilation errors. A good IDE will even help you quickly locate the appropriate line of code, based on line numbers in the error message, just as you would with C or C ++ programs. Figure 2-3 shows an example of debugging compiler errors in Microsoft Visual Studio.



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

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

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

  return OUT;
}
```

```
14 lines, 2 errors
Compile with Microsoft HLSL for Direct3D "vs_1_1" profile
C:\Perforce\devrel\Cg_Book\examples\chapter2\C2E1v_green.cg(10): error X3004: undeclared identifier 'postion'
compilation failed, no code produced
Could Not Find C:\Perforce\devrel\Cg_Book\examples\chapter2\C2E1v_green.hlsl_vs_1_1
The system cannot find the file specified.
Compile with Microsoft HLSL for Direct3D "vs_2_0" profile
C:\Perforce\devrel\Cg_Book\examples\chapter2\C2E1v_green.cg(10): error X3004: undeclared identifier 'postion'
compilation failed, no code produced
Could Not Find C:\Perforce\devrel\Cg_Book\examples\chapter2\C2E1v_green.hlsl_vs_2_0
The system cannot find the file specified.
Error executing c:\windows\system32\cmd.exe

C2E1v_green.vp - 0 error(s), 0 warning(s)
```

Figure 2-3 Locating Error Lines in an Integrated Development Environment

Note

Cg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。及时他们主要使用其中一种编程接口。但是对这些3D编程接口而言，基于profile的区别可以再对应的profile中有效的地方存在。因此，你需要使用从刚才编译你的Cg程序两次。一次用适当的OpenGL的profile，一次用适当的Direct3D的profile。

Cg developers often write a single C g program that works for both OpenGL and Direct3D, even if they predominantly use one programming interface or the other. However, profile-dependent differences can exist between what is valid in the corresponding profiles for these 3D programming interfaces. So make it your practice to compile your C g programs twice with cgc：once for the appropriate OpenGL profile, and again for the appropriate Direct3D profile.

2.4 用顶点和片段示例程序渲染

（**2.4 Rendering with Your Vertex and Fragment Program Examples**）

现在是看你的两个简单Cg程序运行结果的时候了。不要期望太多，因为他们都是非常简单的成程序。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化一个绿色的三角

形，你仍然可以学到许多知识。

Now it's time to see your two simple C g programs in action. Don't expect too much, because these are both very simple programs. However, you can still learn a lot by examining how the programs work together—and with the rest of the graphics pipeline—to draw a green triangle.

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。

Look at the 2D triangle in Figure 2-4. This is the geometry that your vertex and fragment program will operate on in this example.

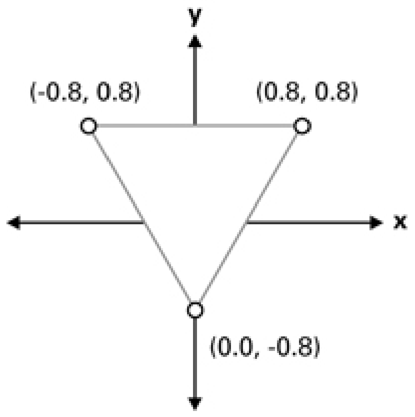


Figure 2-4 A 2D Triangle for Rendering

2.4.1 用 OpenGL 渲染一个三角形

（2.4.1 Rendering a Triangle with OpenGL）

在OpenGL中，你可以用下面的命令渲染一个二维三角形：

In OpenGL, you can render this 2D triangle with the following commands:

```
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

（2.4.2 Rendering a Triangle with Direct3D）

在Direct3D中，你可以使用如下代码渲染同样的三角形：

In Direct3D, you can render the same triangle with the following code:

```
D3DXVECTOR4 vertices[3] =
{
D3DXVECTOR4(-0.8f, 0.8f, 0.f, 1.f),
D3DXVECTOR4( 0.8f, 0.8f, 0.f, 1.f),
D3DXVECTOR4( 0.0f, -0.8f, 0.f, 1.f),
};
m_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST, 1,
vertices, sizeof(D3DXVECTOR4));
```

在OpenGL和Direct3D中，在许多有效的方法来把顶点传递给GPU。当你使用Cg程序来处理顶点时，应用程序如何处理传递并不是十分重要。

There are other, more efficient ways to transfer vertices to the GPU in OpenGL or Direct3D. When you use Cg programs to process vertices, it doesn't matter how the application sends the vertices to the GPU.

2.4.3 获得同样的结果

（**2.4.3 Getting the Same Results**）

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，这个结果都是一样的。我们并不否则，这个不透明的绿色三角形不是很令人激动。

Figure 2-5 shows the result of rendering this triangle with the C2E1v_green vertex program and C2E2f_passthrough fragment program configured. The result is the same, whether rendered with OpenGL or Direct3D. Admittedly, it's not very exciting, but the triangle is solid green.

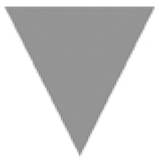


Figure 2-5 Rendering a Triangle

顶点程序把每个指定的二维位置传递给光栅器，光栅器认为这些指定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。如果顶点程序提供二维坐标的话。如图2-5所示的情况，被光栅化的部分是图元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。

The vertex program passes the specified 2D position of each vertex to the rasterizer. The rasterizer expects positions to be specified as coordinates in clip space. C lip space defines what is visible from the current viewpoint. If the vertex program supplies 2D coordinates, as is the case in Figure 2-5, the portion of the primitive that is rasterized is the portion of the primitive where *x* and *y* are between -1 and +1. The entire triangle is within the clipping region, so the complete triangle is rasterized.

图2-6显示了二维剪裁空间光栅化的区域。

Figure 2-6 shows the region of rasterization for 2D clip space.

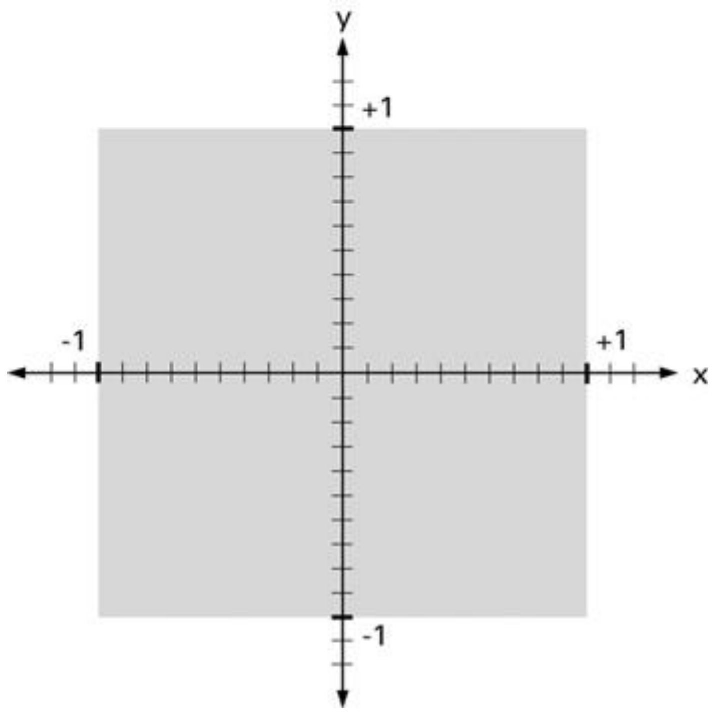


Figure 2-6 A 2D View of C lip Space

如果一个图元全部落在灰色的区域里（整个区域是剪裁空间的x和y坐标介于-1和1之间的区域）。整个图元将被渲染到帧缓存中。你将在第4章学习到更多有关剪裁空间的知识。

Primitives are rendered into the frame buffer if they fall within the gray region (the region of clip space where *x* and *y* are between -1 and +1). You'll learn more about clip space in Chapter 4.

图2-7显示了当前使用C2E1v_green顶点程序和C2E2f_passthrough片段程序渲染不是完全在二维裁剪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星形。如果一个图元没有一个部分是剪裁空间的可见区域里的，光栅器将丢弃这个图元。

Figure 2-7 shows what happens when you use the C2E1v_green vertex program and C2E2f_passthrough fragment program with 2D geometry that is not entirely within the visible region of 2D clip space. When the stars include vertices that have *x* or *y* coordinates outside the -1 to +1 region, the rasterizer clips some of the green stars by the edge of the visible region. If no part of a primitive is within the viewable region of clip space, the rasterizer eliminates the primitive.

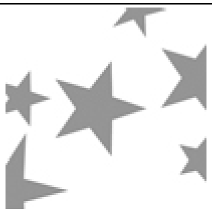


Figure 2-7 Primitives Rendered with and Require 2D Clipping

GPU自动对剪裁空间进行变换（通过一个简单的缩放和偏移）到光栅器使用的窗口坐标。GPU在光栅化之前根据需要伸缩剪裁空间的坐标，以有效的渲染视图和窗口位置。

The GPU then transforms clip space automatically (by a simple scale and bias) into window space coordinates for the rasterizer. Effectively, the GPU stretches clip-space coordinates as necessary, prior to rasterization, to fit the rendering viewport and window position.

第4章将把二维剪裁空间的概念推广到三维。并将介绍透视图。但是在本章和第三章，所有的示例程序都将是用二维渲染，以使情况变得简单一点。

Chapter 4 generalizes this 2D notion of clip space to 3D, and includes perspective views. But in this chapter and Chapter 3, the examples use 2D rendering to keep things simple.

第 3 章 参数、纹理和表达式

（Chapter 3. Parameters, Textures, and Expressions）

本章将通过一系列简单的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：
1） “**参数**”解释了Cg程序是如何处理参数的。
2） “**纹理样本**”解释了片段程序是如何存取纹理的。
3） “**数学表达式**”展示了数字表达式是如何计算新的顶点和片段值的。

This chapter continues to present C g concepts through a series of simple vertex and fragment programs.
The chapter has the following three sections:

- **"Parameters"** explains how C g programs handle parameters.
- **"Texture Samplers"** explains how fragment programs access textures.
- **"Math Expressions"** shows how math expressions compute new vertex and fragment values.

3.1 参数

（3.1 Parameters）

第2章C2E1v_green和C2E2f_passthrough示例程序是非常基本的，我们将通过扩展这些示例的复杂性来介绍附加参数。

The C2E1v_green and C2E2f_passthrough examples from Chapter 2 are very basic. We will now broaden these examples to introduce additional parameters.

3.1.1 Uniform 参数

（3.1.1 Uniform Parameters）

C2E1v_green顶点程序总是把绿色赋给顶点颜色。如果你给C2E1v_green程序重命名，并且更改OUT.color的赋值语句，你可以使用你喜欢的任何颜色产生一个不同的顶点程序。

C2E1v_green (see page 38 in C hapter 2) always assigns green for the vertex color. If you rename the C2E1v_green program and change the line that assigns the value of OUT.color , you can potentially make a different vertex program for any color you like.

比如，改变适当的代码将其更改为一个生成热情的粉红色的着色器：

For example, changing the appropriate line results in a hot pink shader:

```
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编写一个不同的Cg程序。你可以通过传递一个只是当前被请求颜色的参数来声明这个程序。

The world is a colorful place, so you wouldn't want to have to write a different C g program for every color under the sun. Instead, you can generalize the program by passing it a parameter that indicates the currently requested color.

示例3-1的顶点程序提供了一个名为constantColor的参数，是你的应用程序可以指派任何颜色，而不是只用某种固定的颜色。

The C3E1v_anyColor vertex program in Example 3-1 provides a constantColor parameter that your application can assign to any color, rather than just a particular constant color

Example 3-1. The C3E1v_anyColor **Vertex Program**

```
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的区别是函数的接口定义和每个程序付给OUT.color的值。

The difference between C3E1v_anyColor and C2E1v_green is the function interface definition and what each program assigns to OUT.color .

更新过的函数定义是这样的：

The updated function definition is this:

```
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过的，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。

In addition to the position parameter, the new function definition has a parameter named constantColor that the program defines as type uniform float4 . As we discussed earlier, the float4 type is a vector of four floating-point values—in this case, assumed to be an RGBA color. What we have not discussed is the uniform type qualifier.

3.1.1.1 uniform 类型限制符

（**The** uniform **Type Qualifier**）

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，他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3D编程接口状态和其他通过Cg运行库建立起来的 name-value 对。

The uniform qualifier indicates the source of a variable's initial value. When a C g program declares a variable as uniform , it conveys that the variable's initial value comes from an environment that is external to the specified C g program. This external environment contains your 3D programming interface state and other name/value pairs established through the C g runtime.

C3E1v_anyColor示例中的constantColor变量的会让Cg编译器生成的顶点程序从GPU顶点处理器常量寄存器取回变量的初始值。

In the case of the constantColor variable in the C3E1v_anyColor example, the C g compiler generates a vertex program that retrieves the variable's initial value from a vertex processor constant register within the GPU.

使用Cg运行库，你的3D应用程序可以在一个Cg程序里为一个uniform参数名设置一个参数句柄——在现在的情况下是constantColor——并使用该句柄把uniform变量的正确值载入到GPU中。Uniform参数的值是如何指定和加载的细节随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。

Using the C g runtime, your 3D application can query a parameter handle for a uniform parameter name within a Cg program—in this case, constantColor —and use the handle to load the proper value for the particular uniform variable into the GPU. The details of how uniform parameter values are specified and loaded vary by profile, but the C g runtime makes this process easy. Appendix B explains how to do this.

我们的C3E1v_anycolor顶点程序把他的uniform变量constantColor的值付给顶点输出颜色，如下所示：

Our C3E1v_anyColor vertex program assigns the vertex output color to the value of its constantColor uniform variable, as shown:

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

无论应用程序为uniform的变量constantColor指定了什么颜色，当C3E1v_anycolor变换了一个顶点的时候，这种颜色就是Cg程序赋给输出顶点的颜色。

Whatever color the application specifies for the constantColor uniform variable is the color that the Cg program assigns to the output vertex color when C3E1v_anyColor transforms a vertex.

增加了一个uniform参数，使我们能够将最初渲染绿色颜色的示例程序扩展为可渲染任意自定颜色。

The addition of a uniform parameter lets us generalize our initial example to render any color, when originally it could render only green.

3.1.1.2 当没有 uniform 限制符的时候

（**When There Is No** uniform **Qualifier**）

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

1） 使用明确的初始值：

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

2） 使用一个语义:

```
float4 position : POSITION;
```

3） 由profile决定不定义或初始化为0：

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

When a Cg program does *not* include the uniform qualifier to specify a variable, you can assign the initial value for the variable in one of the following ways:

- Using an explicit initial assignment:
- float4 green = float4 (0, 1, 0, 1);
- Using a semantic:
- float4 position : POSITION;
- Leaving it undefined or equal to zero, depending on the profile:
- float whatever; // May be initially undefined or zero

3.1.1.3 uniform 在 RenderMan 和 Cg 中分别意味着什么

（**What** uniform **Means in RenderMan vs. Cg**）

Note

在RenderMan里编写过着色器的程序员是很熟悉uniform关键字的。但是，在Cg中uniform的意思与RenderMan中并不一样。

The uniform reserved word will be familiar to programmers who have written shaders in RenderMan. However, the meaning of uniform in C g is different from its meaning in RenderMan.

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

In RenderMan, the uniform storage modifier indicates variables whose values are constant over a shaded surface, whereas varying variables are those whose values can vary over the surface.

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他的初始值。除了初始化不同以外，uniform变量和其他变量是完全一样的。除非变量被指定为const类型，Cg允许所有的变量被改变。不想RenderMan，Cg没有varying保留关键字。

Cg does not have this same distinction. In C g, a uniform -qualified variable obtains its initial value from an external environment and, except for this initialization difference, is the same as any other variable. Cg permits all variables to vary, unless the variable has the const type qualifier specified. Unlike RenderMan, Cg has no varying reserved word.

尽管RenderMan的uniform概念和Cg的概念有语义上的区别，但是在RenderMan中声明的uniform变量和Cg中定义的uniform与Cg中的是类似的，反之亦然。

Despite the semantic difference between RenderMan's concept of uniform and Cg's concept of it, variables declared uniform in RenderMan correspond to variables declared uniform in C g, and vice versa.

3.1.2 const 类型限制符

（**3.1.2 The** const **Type Qualifier**）

Cg也提供了const限制符。Const限制符与C和C++中的限制符影响变量的方式是一样的。他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const常量的变量赋值或改变他。使用const限制符指明了某个值永远不能被改变。如果Cg编译器发现了一个声明为const的变量将被修改，那么会他会产生一个编译错误。

Cg also provides the const qualifier. The const qualifier affects variables the same way that the const qualifier does in C and C ++: it restricts how a variable in your program may be used. You cannot assign a value to, or otherwise change, a variable that is specified as constant. Use the const qualifier to indicate that a certain value should never change. The C g compiler will generate an error if it detects usage that would modify a variable declared as const .

当一个程序用const限制了一个变量的时候，这里有一些禁止的使用方式：

Here are some examples of usage *not* allowed when a program qualifies a variable with const :

```
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类型限制符是相互独立的，因此一个变量可以使用const或uniform来限定，或同时使用const或uniform来限定。

The const and uniform type qualifiers are independent, so a variable can be specified using const or uniform , both const and uniform , or neither.

3.1.3 Varying 参数

（**3.1.3 Varying Parameters**）

你已经在C2E1v_green和C3E1v_anycolor中看到的每个顶点都改变的参数的例子。跟刚在

C2E1v_green和C3E1v_anycolor的position参数后面的POSITION输入语义，指明GPU要用各个程序处理的每个顶点输入位置来初始化每个position参数。

You have already seen examples of a per-vertex varying parameter in both C2E1v_green and C3E1v_anyColor . The POSITION input semantic that follows the position parameter in C2E1v_green and C3E1v_anyColor indicates that the GPU is to initialize each respective position parameter with the input position of each vertex processed by each respective program.

语义提供了一种在顶点程序中使用随顶点（或在片段程序中随片段）变化而变化的值来初始化Cg程序的方法。

Semantics provide a way to initialize C g program parameters with values that vary either from vertex to vertex (in vertex programs) or fragment to fragment (in fragment programs).

在示例3-2中，我们队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，而是可以随顶点而改变的颜色或纹理坐标集（用来存取纹理）。

A slight modification to C3E1v_anyColor , called C3E2v_varying , in Example 3-2, lets the program output not merely a single constant color, but rather a color and texture coordinate set (used for accessing textures) that can vary per vertex.

Example 3-2. The C3E2v_varying **Vertex Program**

```
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示例用两个未标记的参数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。

The C3E2v_varying example replaces the constantColor parameter declared as a uniform parameter in the C3E1v_anyColor example with two new non-uniform parameters, color and texCoord . The program assigns the COLOR and TEXCOORD0 semantics, respectively, to the two parameters. These two semantics correspond to the application-specified vertex color and texture coordinate set zero, respectively.

这个新程序用如下代码通过输出每个顶点的位置、颜色和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单一颜色。

```
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片段渲染原来的三角形的结果。在这里，我们假设你使用了OpenGL或Direct3D来给每个顶点赋颜色：上面的两个顶点用亮蓝色而下面的顶点则用浅蓝色。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，但之后的C2E2f_passthrough片段程序忽略了这些纹理坐标。

Figure 3-1 shows the result of rendering our original triangle using the C3E2v_varying vertex program and the C2E2f_passthrough fragment program. Here, we assume that you have used OpenGL or Direct3D to assign the vertices of the triangle the per-vertex colors bright blue for the top two vertices and off-blue for the bottom vertex. Color interpolation performed by the rasterization hardware smoothly shades the interior fragments of the triangle. Although per-vertex texture coordinates are input and output by the C3E2v_varying vertex program, the subsequent C2E2f_passthrough fragment program ignores the texture coordinates.

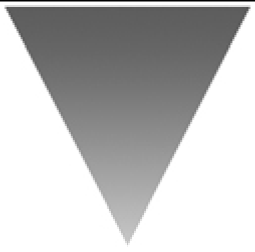


Figure 3-1 Rendering a Gradiated 2D Triangle

3.2 纹理样本

（**3.2 Texture Samplers**）

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的片段程序使用了纹理坐标对图像进行采样。

The C3E2v_varying example passed per-vertex texture coordinates through the vertex program. Although the C2E2f_passthrough fragment program ignores texture coordinates, this next fragment program, called C3E3f_texture and shown in Example 3-3, uses the texture coordinates to sample a texture image.

Example 3-3. The C3E3f_texture Fragment Program

```
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中的新概念在他的声明中被显示出来：

The C3E3f_Output structure is essentially the same as the C2E2f_Output structure used by C2E2f_passthrough, our prior fragment program example. What is new about the C3E3f_texture example is in its declaration:

C3E3f_texture片段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。

The C3E3f_texture fragment program receives an interpolated texture coordinate set but ignores the interpolated color. The program also receives a uniform parameter called decal of type sampler2D.

3.2.1 样本对象

（**3.2.1 Sampler Objects**）

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后缀指明了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。在以后的章节中你会遇到一些这样的样本对象。

A *sampler* in C g refers to an external object that C g can sample, such as a texture. The 2D suffix for the sampler2D type indicates that the texture is a conventional two-dimensional texture. Table 3-1 lists other sampler types supported by C g that correspond to different kinds of textures. You will encounter some of these in later chapters.

Table 3-1. Cg Sampler Types

Sampler Type	**Texture Type**	**Applications**
sampler1D	One-dimensional texture	1D functions
sampler2D	Two-dimensional texture	Decals, normal maps, gloss maps, shadow maps, and others
sampler3D	Three-dimensional texture	Volumetric data, 3D attenuation functions
samplerCUBE	Cube map texture	Environment maps, normalization cube maps
samplerRECT	Non-power-of-two,	Video images, photographs,

	non-mipmapped 2D texture	temporary buffers

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[0, 1]，但是你也查找超出这个范围的值。在这里我们不对这个问题进行详细的解释，因为结果是取决于在OpenGL和Direct3D中如何设置你的纹理。

Texture coordinates specify where to look when accessing a texture. Figure 3-2 shows a 2D texture, along with a query based on the texture coordinates (0.6, 0.4). Typically, texture coordinates range from 0 to 1, but you can also use values outside the range. We will not go into detail about this here, because the resulting behavior depends on how you set up your texture in OpenGL or Direct3D.

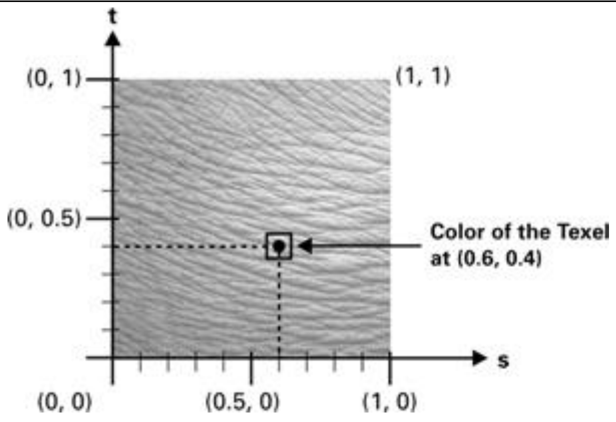


Figure 3-2 Querying a Texture

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

The semantic for the texture coordinate set named texCoord in Example 3-3 is TEXCOORD0 , corresponding to the texture coordinate set for texture unit 0. As the name of the sampler parameter decal implies, the intent of this fragment program is to use the fragment's interpolated texture coordinate set to access a texture.

3.2.2 纹理采样

（**3.2.2 Sampling Textures**）

C3E3f_texture的下一个有趣的代码行使用经过插值的纹理坐标存取decal纹理：

The next interesting line of C3E3f_texture accesses the decal texture with the interpolated texture coordinates:

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

tex2D例程属于Cg标准库。他是一系列可以使用指定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位置采样的数据。

The routine tex2D belongs to the C g Standard Library. It is a member of a family of routines that access different types of samplers with a specified texture coordinate set and then return a vector result. The result is the sampled data at the location indicated by the texture coordinate set in the sampler object.

实际上，这等同于纹理查找。那么一个纹理如何采样和过滤的：这是与纹理类型以及与Cg样本变量相关联的纹理对象所包含的纹理参数所决定的。你可以使用OpenGL和Direct3D的纹理指令来决定一个给定纹理的纹理属性，这取决于你选择哪个3D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。

In practice, this amounts to a texture lookup. How the texture is sampled and filtered depends on the texture type and texture parameters of the texture object associated with the C g sampler variable. You can determine the texture properties for a given texture by using OpenGL or Direct3D texture specification commands, depending on your choice of 3D programming interface. Your application is likely to establish this association by using the C g runtime.

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，并需要一个一个三元纹理坐标集作为它第二个参数。

The 2D suffix indicates that tex2D must sample a sampler object of type sampler2D . Likewise, the texCUBE routine returns a vector, accepts a sampler of type samplerCUBE for its first argument, and requires a three-component texture coordinate set for its second argument.

基本片段profile（例如ps_1_1和fp20）把纹理采样例程，例如tex2D和texCUBE，限制到对应与样本纹理单元的纹理坐标集。为了尽量简单并且支持所有的片段profile，C3E3f_texture示例遵守了这个限制（请看2.3.1小节对profile的简单介绍）。

Basic fragment profiles (such as ps_1_1 and fp20) limit texture-sampling routines, such as tex2D and texCUBE , to the texture coordinate set that corresponds to the sampler's texture unit. To be as simple as possible and support all fragment profiles, the C3E3f_texture example follows this restriction. (See Section 2.3.1 for a brief introduction to profiles.)

高级的片段profile（例如ps_2_x、arbfp和fp30）允许使用从其他纹理单元取得的纹理坐标集，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。

Advanced fragment profiles (such as ps_2_x, arbfp1 , and fp30) allow a sampler to be sampled using texture coordinate sets from other texture units, or even texture coordinates computed in your Cg program.

3.2.3 在对一个纹理采样的时候，发送纹理坐标

（3.2.3 Sending Texture Coordinates While Sampling a Texture）

C3E2v_varying顶点程序把每个顶点的位置、颜色和纹理坐标集传递给光栅器。C3E3f_texture片段程序忽略经过插值的颜色，但使用经过插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

个顶点纹理坐标来渲染我们的简单三角形的结果。

The C3E2v_varying vertex program passes a per-vertex position, color, and texture coordinate set to the rasterizer. The C3E3f_texture fragment program ignores the interpolated color, but samples a texture image with the interpolated texture coordinate set. Figure 3-3 shows what happens when you first bind both C g programs with a texture that contains the image of a gruesome face, and then render our simple triangle with additional per-vertex texture coordinates assigned.

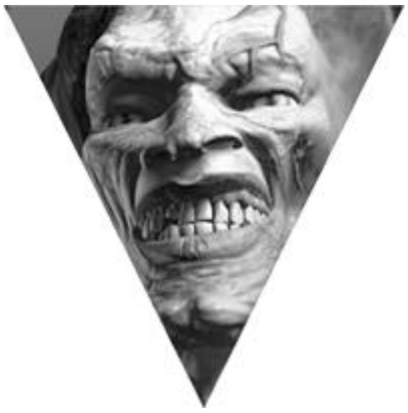


Figure 3-3 Rendering a Textured 2D Triangle

3.3 数学表达式

（**3.3 Math Expressions**）

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所能实现的仅此而已。这些示例的意图是把Cg介绍给你，并显示简单的Cg程序的结构。

So far, all the Cg examples we've presented have done little more than pass along parameters, or use a parameter to sample a texture. Conventional nonprogrammable 3D programming interfaces can accomplish just as much. The point of these examples was to introduce you to C g and show the structure of simple Cg programs.

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

More interesting C g programs perform computations on input parameters by using operators and built-in functions provided by the C g Standard Library.

3.3.1 操作符

（**3.3.1 Operators**）

Cg支持的数学、关系和其他操作符与C和C++提供的一样。这意味着加法可以用一个“+”、乘法用“*”，大于等于用“>=”。你已经在以前的示例中看到了用“=”完成的赋值。

So far, all the C g examples we've presented have done little more than pass along parameters, or use a parameter to sample a texture. Conventional nonprogrammable 3D programming interfaces can accomplish just as much. The point of these examples was to introduce you to C g and show the structure of simple Cg programs.

这里有一些Cg表达式的例子：

Here are some examples of Cg expressions:

```
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++，因为他对向量的数学操作提供了内置的支持。你可以在C++中通过重载某些类的操作符来实现这点，但向量数学操作在Cg语言中是一个标准内容。

Cg is different from C and C ++ because it provides built-in support for arithmetic operations on vector quantities. You can accomplish this in C ++ by writing your own classes that use operator overloading, but vector math operations are a standard part of the language in C g.

下面这些操作符以点积的方式产生结果

Operator	**Name**
*	Multiplication
/	Division
-	Negation
+	Addition
-	Subtraction

当一个标量和一个向量被用作上述某个操作符的操作数的时候，标量被复制到一个大小匹配的向量中。

When a scalar and a vector are used as operands of one of these component-wise operators, the scalar value is replicated (sometimes called "smeared") into a vector of the matching size.

这里有一些向量Cg表达式的例子：

Here are some examples of vector Cg expressions:

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

表3-2呈现了全部操作符的列表，并伴随了他们的优先级、结合律和使用方法。被标记的操作符是被保留的。但是，没有Cg profile支持这些保留操作符，因为当前的GPU不支持位元的整数操作。

Table 3-2 presents the complete list of operators, along with their precedence, associativity, and usage. Operators marked with a reverse highlight are currently reserved. However, no existing C g profiles support these reserved operators because current graphics hardware does not support bitwise integer operations.

Table 3-2. Precedence, Associativity, and Usage of Operators

Operators	**Associativity**	**Usage**
( ) [ ] -> .	Left to right	Function call, array reference, structure reference, component selection
! ~ ++ - + - * & (type) ***sizeof***	Right to left	Unary operators: negation, increment, decrement, positive, negative, indirection, address, cast
*** / %**	Left to right	Multiplication, division, remainder
+ -	Left to right	Addition, subtraction
<< >>	Left to right	Shift operators
< <= > >=	Left to right	Relational operators
== !=	Left to right	Equality, inequality
&	Left to right	Bitwise AND
^	Left to right	Bitwise exclusive OR
|	Left to right	Bitwise OR
&&	Left to right	Logical AND
||	Left to right	Logical OR
? :	Right to left	Conditional expression
= += -= *= -= &= ^= |= <<= >>=	Right to left	Assignment, assignment expressions
,	Left to right	Comma operator

Notes

1） 操作时的优先级从上到下为从高级到低级。
2） 在同一行的操作符有相同的优先级。
3） 被反向加亮的操作符是为以后使用保留的

- Operators are listed top to bottom, from highest to lowest precedence.
- Operators in the same row have the same precedence.
- Operators marked with a reverse highlight are currently reserved for future use.

3.3.2 依赖于 profile 的数值数据类型

（**3.3.2 Profile-Dependent Numeric Data Types**）

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

When you program in C or C ++ and declare variables, you pick from a few different-sized integer data types (int , long , short , char) and a couple of different-sized floating-point data types (float , double).

你的CPU对所有这些基本的数据类型提供了硬件支持。虽然随着GPU的发展，在未来会提供更多的数据类型，但是GPU通常不支持这么多的数据类型。例如，现存的GPU的顶点和片段处理器中不支持指针类型。

Your CPU provides the hardware support for all these basic data types. However, GPUs do not generally support so many data types—though, as GPUs evolve, they promise to provide more data types. For example, existing GPUs do not support pointer types in vertex or fragment programs.

3.3.2.1 连续数据类型

（**Representing Continuous Data Types**）

Cg 提供了float、half和double浮点类型。Cg定义这些类型的方法与C语言相似——语法不要求十分精确。可以理解为：half的范围和精度不高于float的范围和精度，而float的范围和精度不高于double的范围和精度。

Cg provides the float , half , and double floating-point types. C g's approach to defining these types is similar to C 's—the language does not mandate particular precisions. It is understood that half has a range and precision less than or equal to the range and precision of float , and float has a range and precision less than or equal to the range and precision of double .

half数据类型在C和C++中并不存在。有Cg介绍的这个新的数据类型可以保留float精度的一般（通常为16位），他在存储和性能方面比标准的float类型（通常是32位）更有效。

The half data type does not exist in C or C ++. This new data type introduced by C g holds a half-precision floating-point value (typically 16-bit) that is more efficient in storage and performance than standard-precision floating-point (typically 32-bit) types.

Note
NVIDIA CineFX GPU架构在片段程序中支持half精度的值。half数据类型在片段程序中适合被用作中间值，例如颜色和单位化的向量。通过在需要的时候使用half而不是float，你可以加速你的片段处理程序的性能。

The NVIDIA CineFX GPU architecture supports half-precision values for fragment programs. The half data type is often appropriate for intermediate values in fragment programs, such as colors and normalized vectors. By using half values when possible rather than float , you speed up the performance of your fragment programs.

GPU被设计用来提供可表示的连续数据类型，例如颜色和变量。GPU现在还不支持原生的离散数据类型（例如字母和位掩码），因为GPU通常不对这类数据设计特殊操作。

GPUs, by design, provide data types that represent continuous quantities, such as colors and vectors.

GPUs do not (currently) support data types that represent inherently discrete quantities, such as alphanumeric characters and bit masks, because GPUs do not typically operate on this kind of data.

连续数据类型并不限制于整形，当用CPU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。GPU处理的连续值，特别是在片段程序上，已经被限制在一个很小的范围内，例如[0, 1]和 [-1, 1]，而不支持浮点提供的广泛的范围。例如，颜色值通常被限制在[0, 1]的范围内，而单位化的向量其定义被限制在[-1, 1]的范围内。这些范围受限制的数据类型被认为是“定点数”，而不是“浮点数”。

Continuous quantities are not limited to integer values. When programming a C PU, programmers typically use floating-point data types to represent continuous values because floating-point types can represent fractional values. Continuous values processed by GPUs, particularly at the fragment level, have been limited to narrow ranges such as [0, 1] or [-1, +1], rather than supporting the expansive range provided by floating-point. For example, colors are often limited to the [0, 1] range, and normalized vectors are, by definition, confined to the [-1, +1] range. These range-limited data types are known as "fixed-point," rather than floating-point.

虽然定点数据类型使用了有限的精度，但是他们可以表示连续量。然而，他们缺乏浮点数据类型的范围，他的编码方式与他的科学符号相似。一个浮点值除了位数以外还编码了一个可变指数（这与科学计数法写出来的方式一样，例如2.00 * 10^8），而一个顶点值假定了一个固定的指数。例如，一个没有单位化的向量或者足够大的纹理坐标也许需要浮点类型来避免数据溢出。

Although fixed-point data types use limited precision, they can represent continuous quantities. However, they lack the range of floating-point data types, whose encoding is similar to scientific notation. A floating-point value encodes a variable exponent in addition to a mantissa (similar to how numbers are written in scientific notation, such as 2.99 x 10_8), whereas a fixed-point value assumes a fixed exponent. For example, an unnormalized vector or a sufficiently large texture coordinate may require floating-point for the value to avoid overflowing a given fixed-point range.

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GPU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并描述了他们是如何表示不同的数据类型的。这暗示了Cg程序员float在各种profile环境中也须并非真正地意味着浮点数。

Cg must be able to manipulate fixed-point data types to support programmability for GPUs that lack floating-point fragment programmability. This means that certain fragment profiles use fixed-point values. Table 3-3 lists various C g profiles and describes how they represent various data types. The implication for Cg programmers is that float may not actually mean floating-point in all profiles in all contexts.

Table 3-3. Data Types for Various Profiles

Profile Names	Types	Numerics
arbfp1 arbvp1 vs_1_1 vs_2_0 vp20 vp30	float double half fixed	Floating-point
	int	Floating-point clamped to integers

fp20	float double half int fixed	Floating-point for texture mapping; fixed point with [-1, +1] range for fragment coloring
ps_1_1 ps_1_2 ps_1_3	float double half int fixed	Floating-point for texture mapping; fixed-point with GPU-dependent range for fragment coloring; range depends on underlying Direct3D capability
ps_2_0 ps_2_x	float double	24-bit floating-point (minimum)
	int	Floating-point clamped to integers
	half	16-bit floating-point (minimum)
	fixed	Depends on compiler settings
fp30	float double	Floating-point
	int	Floating-point clamped to integers
	half	16-bit floating-point
	fixed	Fixed-point with [-2, 2) range

Note

fp20和ps_1_1 profile把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，我们想要数学操在纹理贴图之后执行。如果你想要用真正的浮点数类型，需要使用arbftp1,fp30或vp_2_0 profile，但请注意这些高级的profile不被较旧的GPU支持。

The fp20 and ps_1_1 profiles treat variables in fragment coloring as fixed-point values in the range [-1, +1]. By *fragment coloring,* we mean math operations performed after the texture mapping results. If you want true floating-point data types, use the arbfp1 , fp30 , or vp_2_0 profiles, but be aware these are advanced profiles not supported by older GPUs.

Note

CineFX 构架在片段程序中还支持了一种被称为fixed的特殊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) .在其他的profile中，fixed数据类型和可用的最小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（和向量类型比如fixed3和fixed4），但是Microsoft的HLSL编译器（fxc）并不支持。

The CineFX architecture also supports a special high-performance continuous data type called fixed for fragment programs. The fixed data type has a [-2, +2) range (meaning, ranging from negative 2 to notquite positive 2) for the fp30 profile. In other profiles, the fixed data type is synonymous with the smallest continuous data type available. Although the C g compiler (cgc) and runtime support the fixed data type (and vector versions such as fixed3 and fixed4), Microsoft's HLSL compiler (fxc) does not.

3.3.3 标准库内置的函数

（**3.3.3 Standard Library Built-In Functions**）

Cg标准库包含许多内置的函数以用来简化GPU编程。在许多情况下，这些函数被映射为一个单独的本地GPU指令，因此他们非常有效。

The Cg Standard Library contains many built-in functions that simplify GPU programming. In many cases, the functions map to a single native GPU instruction, so they can be very efficient.

这些内置的函数和C的标准库函数很相似。Cg标准库提供了一套使用的三角函数、指数、向量、矩阵和纹理函数。但是Cg标注库函数没有提供I/O、字符串处理和内存分配，因为Cg不支持这些操作（虽然你的C和C++应用程序肯定能够使用）。

These built-in functions are similar to C 's Standard Library functions. The C g Standard Library provides a practical set of trigonometric, exponential, vector, matrix, and texture functions. But there are no Cg Standard Library routines for input/output, string manipulation, or memory allocation, because C g does not support these operations (though your C or C ++ application certainly could).

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。

We already used one Cg Standard Library function, tex2D , in Example 3-3. Refer to Table 3-4 for a select list of other functions that the C g Standard Library provides. You can find a complete list of Cg Standard Library functions in Appendix E.

Table 3-4. Selected Cg Standard Library Functions

Function Prototype	Profile Usage	Description
abs(x)	All	Absolute value
cos(x)	Vertex, advanced fragment	Cosine of angle in radians
cross(v1, v2)	Vertex, advanced fragment	Cross product of two vectors

ddx(a)
ddy(a)
	Advanced fragment	Approximate partial derivatives of a with respect to window-space x or y coordinate, respectively

determinant(M)	Vertex, advanced fragment	Determinant of a matrix
dot(a, b)	All, but restricted basic fragment	Dot product of two vectors
floor(x)	Vertex, advanced fragment	Vertex, advanced fragment Largest integer not greater than x
isnan(x)	Advanced vertex and fragment	True if x is not a number (NaN)
lerp(a, b, f)	All	Linear interpolation between a and

		b based on f
log2(x)	Vertex, advanced fragment	Base 2 logarithm of x
max(a, b)	All	Maximum of a and b

mul(M, N)
mul(M, v)
mul(v, M)
	Vertex, advanced fragment	Matrix-by-matrix multiplication
		Matrix-by-vector multiplication
		Vector-by-matrix multiplication

pow(x, y)	Vertex, advanced fragment	Raise x to the power y
radians(x)	Vertex, advanced fragment	Degrees-to-radians conversion
reflect(v, n)	Vertex, advanced fragment	Reflection vector of entering ray v and normal vector n
round(x)	Vertex, advanced fragment	Round x to nearest integer
rsqrt(x)	Vertex, advanced fragment	Reciprocal square root
tex2D(sampler, x)	Fragment, restricted for basic	2D texture lookup
tex3Dproj(sampler, x)	Fragment, restricted for basic	Projective 3D texture lookup
texCUBE(sampler, x)	Fragment, restricted for basic	Cube-map texture lookup

3.3.3.1 函数重载

（**Function Overloading**）

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。

The Cg Standard Library "overloads" most of its routines so that the same routine works for multiple data types. As in C ++, function overloading provides multiple implementations for a routine by using a single name and differently typed parameters.

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：

Overloading is very convenient. It means you can use a function, for example abs , with a scalar parameter, a two-component parameter, a three-component parameter, or a four-component parameter. In each case, Cg "calls" the appropriate version of the absolute value function:

```
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，在第一个示例中，abs接受了一个四元向量。在第二个示例中，abs接受了一个二元向量。根据传递给函数的参数类型的不同，编译器能够自动调用正确版本的abs函数。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能够自动选择你指定函数名的正确实现。

The code fragment calls the abs routine twice. In the first instance, abs accepts a four-component vector. In the second instance, abs accepts a two-component vector. The compiler automatically calls the appropriate version of abs , based on the parameters passed to the routine. The extensive use of function overloading in the C g Standard Library means you do not need to think about what routine to call for a given-size vector or other parameter. C g automatically picks the appropriate implementation of the routine you name.

函数重载并不只限于Cg标准库。另外，你也可以使用重载编写你的内部函数。

Function overloading is not limited to the C g Standard Library. Additionally, you can write your own internal functions with function overloading.

函数重载在Cg中甚至可以应用于不同profile的某一函数的不同实现。例如，一个新的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顶点profile则不会有这样专门的指令。虽然这样会损失一些精度， 你仍然可以用一系列被支持的顶点指令来近似正弦和余弦函数。你可以编写两个函数，并为每个都指定一个专门的profile。

Function overloading in C g can even apply to different implementations of the same routine name for different profiles. For example, an advanced vertex profile for a new GPU may have special instructions to compute the trigonometric sine and cosine functions. A basic vertex profile for older GPUs may lack that special instruction. However, you may be able to approximate sine or cosine with a sequence of supported vertex instructions, although with less accuracy. You could write two functions and specify that each require a particular profile.

Cg对依赖于profile重载的支持能够帮助你把Cg程序中依赖于profile的限制分离到辅助函数中去。The *Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics* 中有更多与profile有关的信息。

Cg's support for profile-dependent overloading helps you isolate profile-dependent limitations in your Cg programs to helper functions. The *Cg Toolkit User's Manual: A Developer's Guide to Programmable Graphics* has more information about profile-dependent overloading.

3.3.3.2 Cg 标准函数库的有效性和精确性

（**The Cg Standard Library's Efficiency and Precision**）

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标准函数和你编写类似函数一样甚至比你的函数更加的有效和精确。

Whenever possible, use the Cg Standard Library to do math or other operations it supports. The Cg Standard Library functions are as efficient and precise as—or more efficient and precise than—similar functions you might write yourself.

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：

For example, the dot function computes the dot product of two vectors. You might write a dot product function yourself, such as this one:

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

这与dot函数所实现的数学原理是一样的。但是，dot函数会映射到一个专门的GPU指令，因此Cg标准库所提供的点积运算可能会比myDot函数更快更精确。

This is the same math that the dot function implements. However, the dot function maps to a special GPU instruction, so the dot product provided by the C g Standard Library is very likely to be faster and more accurate than the myDot routine.

Note
尽可能使用Cg标准库，这样你能够让Cg编译器为你的GPU生成最有效最精确的程序。

By using C g Standard Library functions wherever possible, you guide the C g compiler to generate the most efficient and precise program for your particular GPU.

3.3.4 二维扭曲

（**3.3.4 2D Twisting**）

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。顶点程序将根据顶点离窗口的远近决定顶点围绕窗口中心旋转的程度。

In the next example you will put expressions, operators, and the C g Standard Library to work. This example demonstrates how to twist 2D geometry. The farther a vertex is from the center of the window, the more the vertex program rotates the vertex around the center of the window.

在示例3-4中的C3E4v_twwist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。

The C3E4v_twist program shown in Example 3-4 demonstrates scalar-by-vector multiplication, scalar addition and multiplication, scalar negation, the length Standard Library routine, and the sincos Standard Library routine.

Example 3-4. The C3E4v_twist **Vertex Program**

```
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。

The C3E4v_twist program inputs the vertex position and color as varying parameters and a uniform scalar twisting scale factor. Figure 3-4 shows the example with various amounts of twisting.

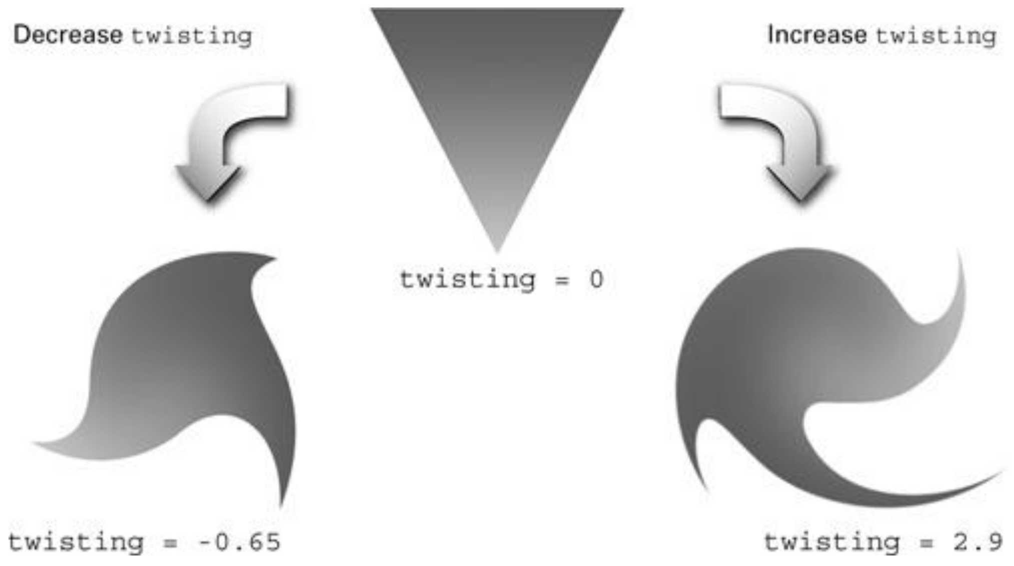


Figure 3-4 Results with Different Parameter Settings

3.3.4.1 length 和 sincos 标准库函数

（**The** length **and** sincos **Standard Library Routines**） length函数有一个重载的原型，其中SCALAR可以是任意的标量数据类型，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：

The length routine has an overloaded prototype, where SCALAR is any scalar data type and VECTOR is a vector of the same scalar data type as SCALAR with one, two, three, or four components:

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

Cg标准库函数length返回他唯一参数的标量长度：

The Cg Standard Library routine length returns the scalar length of its single input parameter:

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

这个程序计算了弧度，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。

The program computes an angle in radians that is the twisting parameter times the length of the input position. Then the sincos Standard Library routine computes the sine and cosine of this angle.

sincos函数有如下的重载类型，其中SCALAR是任何标量类型：

The sincos routine has the following overloaded prototype, where SCALAR is any scalar data type:

```
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）的正弦和余弦更新参数s和c。

3.3.4.2 返回类型参数的传递

（**Call-by-Result Parameter Passing**）

限定词out指定了当函数返回的时候，Cg必须把由out限定的形式参数最终值传递给他的调用参数。初始的时候，out参数的值是没有定义的。这个规定被称为返回类型参数的传递。

An out qualifier indicates that when the routine returns, C g must assign the final value of a formal parameter qualified by out to its corresponding caller parameter. Initially, the value of an out parameter is undefined. This convention is known as *call-by-result* (or *copy-out*) parameter passing.

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&标注）。但是这被称为引用数值传递，而不是Cg的返回类型参数的传递。

C has no similar parameter-passing convention. C ++ allows a reference parameter to function (indicated by & prefixed to formal parameters), but this is a *call-by-reference* parameter-passing convention, not Cg's call-by-result convention.

Cg还提供了in和inout限定词。in类型限定符指明了Cg通过值来传递参数。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限定词返回的时候，Cg会丢弃谢谢参数的值，除非这些参数也被out限定了。

Cg also provides the in and inout keywords. The in type qualifier indicates that C g passes the parameter by value, effectively *call-by-value*. The calling routine's parameter value initializes the corresponding formal parameter of the routine called. When a routine with in -qualified parameters returns, Cg discards the values of these parameters unless the parameter is also out -qualified.

C对所有的参数使用了值传递的参数传递方式。C++对传递引用以外的所有参数使用值传递的参数传递方式。

C uses the copy-by-value parameter-passing convention for all parameters. C ++ uses copy-by-value for all parameters, except those passed by reference.

inout限定词（或者in 和 out类型限定词被同时使用在同一个参数上）结合了值传递和返回类型（也被称为call-by-value-result 或者 copy-in-copy-out）。

The inout type qualifier (or the in and out type qualifiers that are specified for a single parameter) combine call-by-value with call-by-result (otherwise known as *call-by-value-result* or *copy-in-copy-out*).

in限定词是可以选择得，因为如果你不指定一个in、out或inout限定符，参数将被默认指定为in限定符。

The in qualifier is optional because if you do not specify an in , out , or inout qualifier, the in qualifier is assumed.

我们可以在使用out和inout参数的同时，仍然返回一个常规的返回值。

You can use out and inout parameters and still return a conventional return value.

3.3.4.3 旋转顶点

（**Rotating Vertices**）

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。

Once the program has computed the sine and cosine of the angle of rotation for the vertex, it applies a rotation transformation. Equation 3-1 expresses 2D rotation.

Equation 3-1 2D Rotation

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

下面的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运算，但是现在我们将用简单易懂的方法来实现这个公式：

The following code fragment implements this equation. In Chapter 4, you will learn how to express this type of matrix math more succinctly and efficiently, but for now we'll implement the math the straightforward way:

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

3.3.4.4 镶嵌对顶点程序的重要性

（**The Importance of Tessellation for Vertex Programs**）

C3E4v_twist程序通过围绕图像中心旋转来实现效果。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操作。

The C3E4v_twist program works by rotating vertices around the center of the image. As the magnitude of the twist rotation increases, an object may require more vertices—thus higher tessellation—to reproduce the twisting effect reasonably.

通常的情况下，当一个顶点程序设计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多的镶嵌。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。如图3-5显示了如何增加镶嵌程度来提高C3E4v_twist示例的扭曲效果。

Generally, when a vertex program involves nonlinear computations, such as the trigonometric functions in this example, sufficient tessellation is required for acceptable results. This is because the values of the vertices are interpolated linearly by the rasterizer as it creates fragments. If there is insufficient tessellation, the vertex program may reveal the tessellated nature of the underlying geometry. Figure 3-5 shows how increasing the amount of tessellation improves the twisted appearance of the C3E4v_twist example.

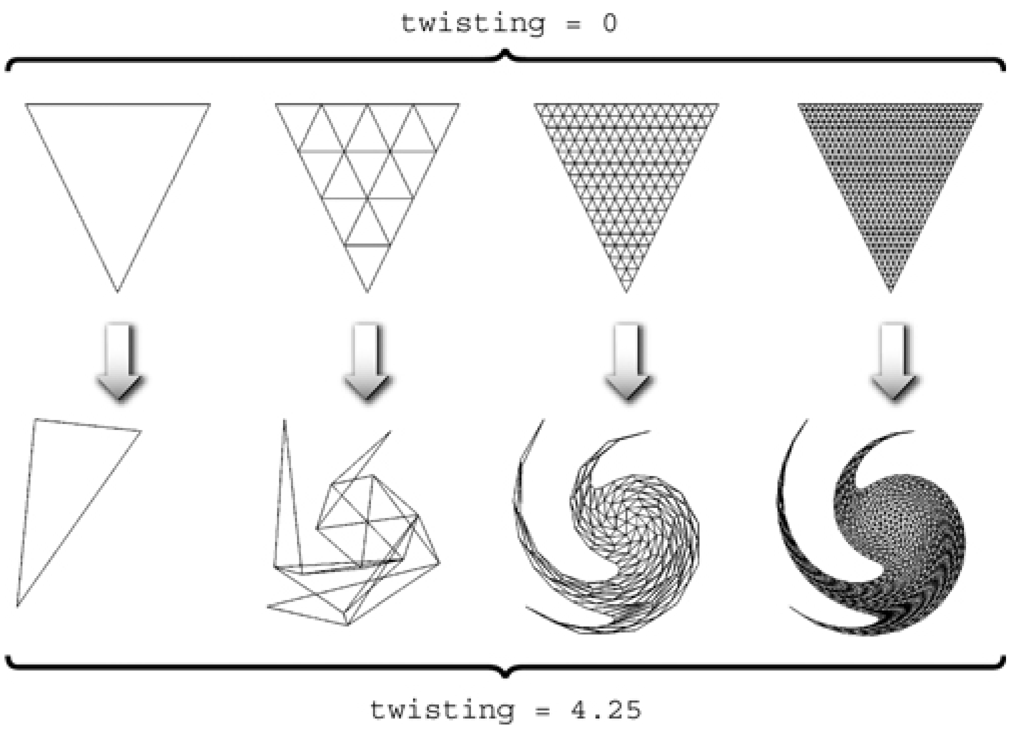


Figure 3-5 Improving the Fidelity of by Increasing Tessellation

3.3.5 叠加效果

（**3.3.5 Double Vision**）

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叠加效果。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。然后再把采样值平等地混合。

Now we demonstrate how to combine a vertex program and a fragment program to achieve a textured "double vision" effect. The idea is to sample the same texture twice, based on slightly shifted texture coordinates, and then blend the samples equally.

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。

The C3E5v_twoTextures vertex program shown in Example 3-5 shifts a single texture coordinate position twice, using two distinct offsets to generate two slightly separated texture coordinate sets. The fragment program then accesses a texture image at the two offset locations and equally blends the two texture results. Figure 3-6 shows the rendering results and the required inputs.

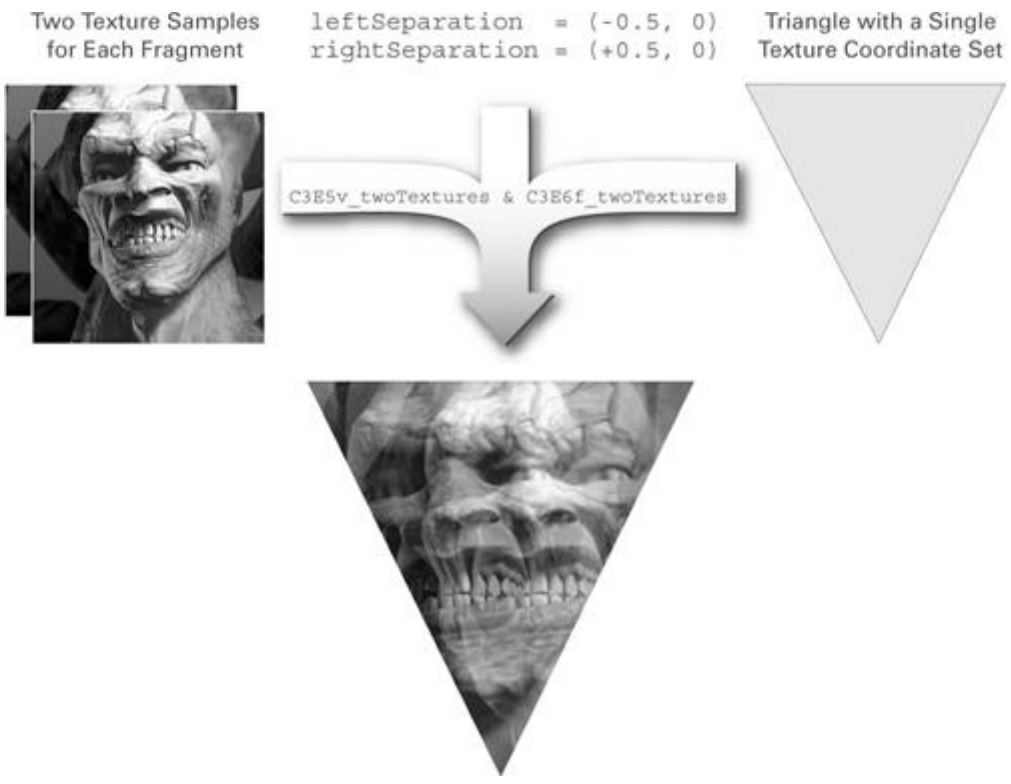

Two Texture Samples for Each Fragment
leftSeparation = (-0.5, 0)
rightSeparation = (+0.5, 0)
Triangle with a Single Texture Coordinate Set
C3E5v_twoTextures & C3E6f_twoTextures

Figure 3-6 Creating a Double Vision Effect with and

Example 3-5. The C3E5v_twoTextures **Vertex Program**

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

3.3.5.1 重影顶点程序

（**The Double Vision Vertex Program**）

在示例3-5中显示的C3E5v_twoTextures程序只是简单的传递了顶点位置。另外这个程序输出了统一输入纹理坐标两次，一次用统一参数leftSeparation平移，另一次用rightSeparation平移。

The C3E5v_twoTextures program in Example 3-5 passes through the vertex position. The program outputs the single input texture coordinate twice, once shifted by the leftSeparation uniform parameter and then shifted by the rightSeparation uniform parameter.

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

3.3.5.2 out参数与Output结构对比

（**Out Parameters vs. Output Structures**）

C3E5v_twoTextures示例还显示了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词（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。你甚至能够混合地使用它们。

The C3E5v_twoTextures example also shows a different approach to outputting parameters. Rather than return an output structure, as all our previous examples have done, the C3E5v_twoTextures example returns nothing; the function's return type is void . Instead, out parameters with associated semantics, which are part of the entry function's prototype, indicate which parameters are output parameters. The choice of using out parameters or an output return structure to output parameters from an entry function is up to you. There is no functional difference between the two approaches. You can even mix them.

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，例如position和oPosition参数，否则它们将有相同的名字。

The remainder of this book uses the out parameter approach, because it avoids having to specify output structures. We add an " o " prefix for out parameters to distinguish input and output parameters that would otherwise have the same name—for example, the position and oPosition parameters.

Example 3-6. The C3E6f_twoTextures **Fragment Program**

```
void C3E6f_twoTextures(float2 leftTexCoord : TEXCOORD0,
float2 rightTexCoord : TEXCOORD1,
out float4 color : COLOR,
uniform sampler2D decal)00
{
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。这种凡事需要额外的工作量来格式化代码，但是我们在这本书中的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。

In Example 3-5 and subsequent examples, we also line up and group the parameters to the entry function as input, output, and uniform parameters. This style takes extra work to format code, but we use it in this book to make the examples easier to read, particularly when the examples have many parameters.

3.3.5.3 用于公安机片段 profile 的重影片段程序

（**The Double Vision Fragment Program for Advanced Fragment Profiles**）

在示例3-6中的C3E6f_twoTextures片段程序用C3E5f_twoTextures程序计算的两个平移和插值的纹理坐标集来对统一纹理采样两次，如图3-6所示。

The C3E6f_twoTextures fragment program in Example 3-6 takes the two shifted and interpolated texture coordinate sets computed by C3E5v_twoTextures and use them to sample the same texture image twice, as shown in Figure 3-6.

```
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。

Then the program computes the average of the two color samples:

```
color = lerp(leftColor, rightColor, 0.5);
```

lerp函数计算两个类型相同的向量的加权线性插值。为了帮助记忆，其实lerp表示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是一个标量或者与VECTOR有相同维数和元素类型的向量。

The lerp routine computes a weighted linear interpolation of two same-sized vectors. The mnemonic *lerp* stands for "linear interpolation." The routine has an overloaded prototype in which VECTOR is a vector with one, two, three, or four components and TYPE is a scalar or vector with the same number of components and element types as VECTOR：

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

lerp函数按照如下方式计算：（The lerp routine computes:）

*result =(1-weight) * xa + weight * xb*

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

不幸的是，C3E6f_twoTextures片段程序将无法使用基本的片段profile编译，例如fp20和ps_1_1（你将马上学习到为什么不行），这个程序用高级片段profile，例如fp30和ps_2_2，编译得很好。

Unfortunately, the C3E6f_twoTextures fragment program will not compile with basic fragment profiles such as fp20 and ps_1_1 (you will learn why shortly). It compiles fine, however, with advanced fragment profiles, such as fp30 and ps_2_0.

3.3.5.4 用于基本片段 profile 的重影片段程序

（**The Double Vision Fragment Program for Basic Fragment Profiles**）

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，这段程序无法使用基本的片段程序profile来编译。由于第三代和早期的GPU限制，这样的profile只能用于一个给定的纹理坐标集存取它对应的纹理单元。

The C3E6f_twoTextures example uses two texture coordinate sets, 0 and 1, to access texture unit 0. Because of this, the program does not compile with basic fragment program profiles. Such profiles can use only a given texture coordinate set with the set's corresponding texture unit due to limitations in third-generation and earlier GPUs.

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的片段profile上。在示例3-7中的C3E7f_twoTextures版本包含了必要的修改。

You can alter the C3E6f_twoTextures program slightly so that it works with basic and advanced fragment profiles. The C3E7f_twoTextures version in Example 3-7 contains the necessary alterations.

Example 3-7. The C3E7f_twoTextures **Fragment Program**

```
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
```

这两种方法的性能是相同的。这个示例证明了简单的Cg程序（而不是非常非常复杂的程序）通常能个通过简单的改写，就能够在只支持较早期的profile的GPU上运行的和最新GPU一样。

The performance of these two approaches is comparable. This example demonstrates that *simpler* Cg programs—those that are not too complicated—can often be written with a little extra care to run on older GPUs, which support basic vertex and fragment profiles, as well as on recent GPUs, which support advanced profiles.

第 4 章 变换

（**Transformations**）

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，但不逐一解释相关的数学原理。本章包括以下两个部分：

1） **“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。

2） **“理论应用”**描述如何在Cg 中应用矩阵和坐标系统理论。

When you write vertex or fragment programs, it is important to understand the coordinate systems that you are working with. This chapter explains the transformations that take place in the graphics pipeline, without going into detail about the underlying mathematics. The chapter has the following two sections:

• **"Coordinate Systems"** explains the various coordinate systems used to represent vertex positions as they are transformed prior to rasterization.

• **"Applying the Theory"** describes how to apply the theory of coordinate systems and matrices in Cg.

4.1 坐标系统

（**4.1 Coordinate Systems**）

图形流水线的目的是生成图像并把他们显式在屏幕上。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。生成的图形通常表示为一个观察点或摄像机从一个比较有利的特殊位置看到的图像。

The purpose of the graphics pipeline is to create images and display them on your screen. The graphics pipeline takes geometric data representing an object or scene (typically in three dimensions) and creates a two-dimensional image from it. Your application supplies the geometric data as a collection of vertices that form polygons, lines, and points. The resulting image typically represents what an observer or camera would see from a particular vantage point.

当几何数据流经流水线的时候，GPU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。每个坐标系统都有其特殊的目的。Cg顶点程序为你自己编程控制这些变换提供了一种方法。

As the geometric data flows through the pipeline, the GPU's vertex processor transforms the constituent vertices into one or more different coordinate systems, each of which serves a particular purpose. C g vertex programs provide a way for you to program these transformations yourself.

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完成的任务。你不能在顶点程序中不输出经过变换的位置信息，因为光栅器需要经过变换的位置信息来装配图元和产生片段。

Vertex programs may perform other tasks, such as lighting (discussed in Chapter 5) and animation (discussed in Chapter 6), but transforming vertex positions is a task *required* by all vertex programs. You cannot write a vertex program that does not output a transformed position, because the rasterizer needs transformed positions in order to assemble primitives and generate fragments.

到目前位置，你所遇到的顶点程序都把他们的位置处理限制为简单的二维变换，本章将结石如何实现传统的三维变换来渲染三维物体。

So far, the vertex program examples you've encountered limited their position processing to simple 2D transformations. This chapter explains how to implement conventional 3D transformations to render 3D objects.

图4-1距离说明了用来处理顶点位置的传统变换流程。当位置从一个变换进入到下一个变换的时候，这个图表用被顶点位置所使用的坐标空间来注解变换的转变过程。

Figure 4-1 illustrates the conventional arrangement of transforms used to process vertex positions. The diagram annotates the transitions between each transform with the coordinate space used for vertex positions as the positions pass from one transform to the next.

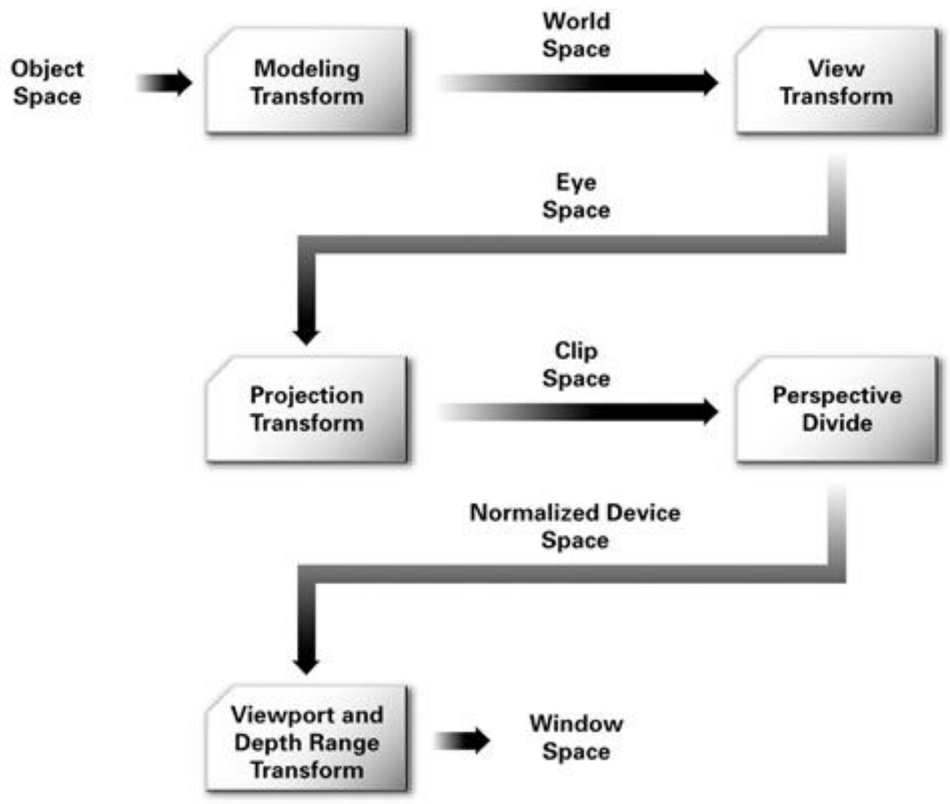


Figure 4-1 Coordinate Systems and Transforms for Vertex Processing

之后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，因此我们只概括地解释流水线的每一个阶段。

The following sections describe each coordinate system and transform in this sequence. We assume that you have some basic knowledge of matrices and transformations, and so we explain each stage of the pipeline with a high-level overview.

4.1.1 物体空间

（**4.1.1 Object Space**）

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、比例和位置来放置组成模型的顶点。一个物体的物体空间可以与其他物体的物理空间没有任何关系。例如，一个圆柱体可以创建一个以底面圆心为原点、对称轴方向沿z正方向的空间坐标系统。

Applications specify vertex positions in a coordinate system known as *object space* (also called *model space*). When an artist creates a 3D model of an object, the artist selects a convenient orientation, scale, and position with which to place the model's constituent vertices. The object space for one object may have no relationship to the object space of another object. For example, a cylinder may have an object-space coordinate system in which the origin lies at the center of the base and the *z* direction points along the axis of symmetry.

无论在物体空间还是在之后的某个空间，你可以把顶点位置表示成向量。一般情况下，你的应用程序为每个物体保持一个形同<x, y, z>的向量为的空间三维顶点位置。每个顶点也许还有一个随附在物体空间表面的法向量，也被存储为一个<x, y, z>向量。

You represent each vertex position, whether in object space or in one of the subsequent spaces, as a vector. Typically, your application maintains each object-space 3D vertex position as an <*x*, *y*, *z*> vector. Each vertex may also have an accompanying object-space surface normal, also stored as an <*x*, *y*, *z*> vector.

4.1.2 齐次坐标

（**4.1.2 Homogeneous Coordinates**）

通常我们认为<x, y, z>位置向量只不过是四元组<x, y, z, w>的一种特殊情况。这种四元组被称为一个齐次位置。当我们表达一个向量位置为<x, y, z>的时候，我们假设有一个隐含的1作为它的w成员。

More generally, we consider the <*x*, *y*, *z*> position vector to be merely a special case of the four-component <*x*, *y*, *z*, *w*> form. This type of four-component position vector is called a *homogeneous position*. When we express a vector position as an <*x*, *y*, *z*> quantity, we assume that there is an implicit 1 for its *w* component.

在数学上，w值是你用来作为x、y和z成员的除数的，以获得的一个传统的三维（非齐次）位置，如公式4-1所示。

Mathematically, the *w* value is the value by which you would divide the *x*, *y*, and *z* components to obtain the conventional 3D (non-homogeneous) position, as shown in Equation 4-1.

Equation 4-1 Converting Between Non-homogeneous and Homogeneous Positions

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

用这种齐次形式表示位置有许多优点。其中一个优点是多重组合变换，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4.2节解释。同样，使用齐次位置不必执行耗时的除法和创建设计投影透视等特殊情况的中间运算。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。

Expressing positions in this homogeneous form has many advantages. For one, multiple transformations, including projective transformations required for perspective 3D views, can be combined efficiently into a single 4x4 matrix. This technique is explained in Section 4.2. Also, using homogeneous positions makes it unnecessary to perform expensive intermediate divisions and to create special cases involving perspective views. Homogeneous positions are also handy for representing directions and curved surfaces described by rational polynomials.

当讨论投影变换的时候我们将继续讨论w成员。

We will return to the *w* component when discussing the projection transform.

4.1.3 世界空间

（4.1.3 World Space）

一个物体的物体孔家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。然后房间里的所有物体就可以相对房间的中心和某个比例（一个单位距离是一尺还是一米）和某个方向（y轴的正方向是向上的么？北是在x轴的正方向上么?）来放置了。

Object space for a particular object gives it no spatial relationship with respect to other objects. The purpose of *world space* is to provide some absolute reference for all the objects in your scene. How a world-space coordinate system is established is arbitrary. For example, you may decide that the origin of world space is the center of your room. Objects in the room are then positioned relative to the center of the room and some notion of scale (Is a unit of distance a foot or a meter?) and some notion of orientation (Does the positive *y* -axis point "up"? Is north in the direction of the positive *x*-axis?).

4.1.4 建模变换

（4.1.4 The Modeling Transform）

在物体空间中指定的物体被放置到世界空间是通过建模变换完成的。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房间的世界坐标系统中。在同一个房间中的两把椅子可以使用同样的三维模型、但使用不同的建模变换使椅子放在房间

中的不同位置。

The way an object, specified in object space, is positioned within world space is by means of a modeling transform. For example, you may need to rotate, translate, and scale the 3D model of a chair so that the chair is placed properly within your room's world-space coordinate system. Two chairs in the same room may use the same 3D chair model but have different modeling transforms, so that each chair exists at a distinct location in the room.

你能用一个数学上的4*4的矩阵表示本章中的所有变换。随用矩阵的性质，通过把几个矩阵相乘，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矩阵中。当你用这种方法连接矩阵是，矩阵的组合也代表了各种变换的组合。正如你将要看到的，这个结果会非常有用。

You can mathematically represent all the transforms in this chapter as a 4x4 matrix. Using the properties of matrices, you can combine several translations, rotations, scales, and projections into a single 4x4 matrix by multiplying them together. When you concatenate matrices in this way, the combined matrix also represents the combination of the respective transforms. This turns out to be very powerful, as you will see.

如果你用4*4的矩阵乘以齐次形式的物体空间位置来表示建模位置（如果没有明确w成员，则假设w成员是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本章之后的所有变换讨论中。

If you multiply the 4x4 matrix representing the modeling transform by the object-space position in homogeneous form (assuming a 1 for the *w* component if there is no explicit *w* component), the result is the same position transformed into world space. This same matrix math principle applies to all subsequent transforms discussed in this chapter.

图4-2说明了几个不同的建模变换效果。左边的图显示了一个机器人以一个基本的姿态放置、没有应用任何建模变换的效果。当你在机器人的各个身体部件使用一系列建模变换后，会呈现出右图的结果。例如，你必须要平移和旋转右臂到如图所示的位置。要在世界空间中把机器人放置在核实的方向和位置需要进一步平移和旋转变换。

Figure 4-2 illustrates the effect of several different modeling transformations. The left side of the figure shows a robot modeled in a basic pose with no modeling transformations applied. The right side shows what happens to the robot after you apply a series of modeling transformations to its various body parts. For example, you must rotate and translate the right arm to position it as shown. Further transformations may be required to translate and rotate the newly posed robot into the proper position and orientation in world space.

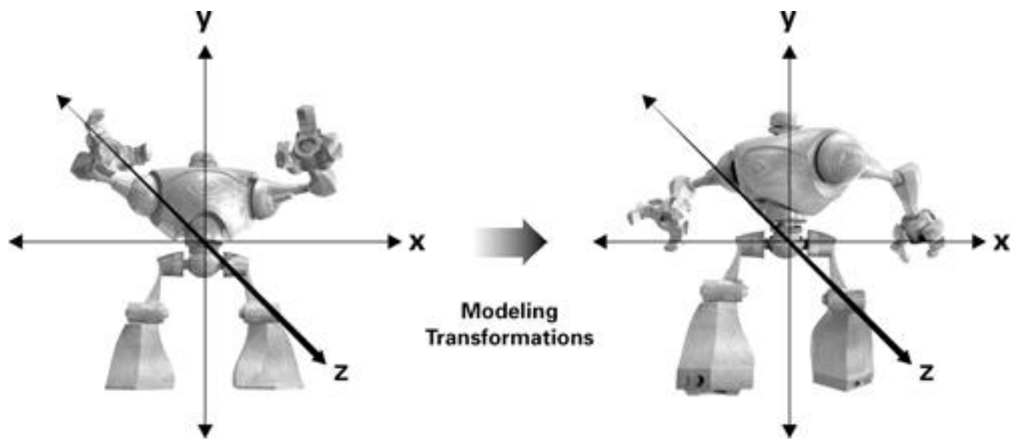


Figure 4-2 The Effect of Modeling Transformations

4.1.5 眼空间

（**4.1.5 Eye Space**）

最后，你要从一个特殊的视点（“眼睛”）观看你的场景。在称为眼空间（或视觉空间）的坐标系统里，眼睛位于坐标系统的原点。朝“上”的方向通常遵循惯例是y轴正方向。你可以设定场景的方向是使眼睛向z轴的负方向望去。

Ultimately, you want to look at your scene from a particular viewpoint (the "eye"). In the coordinate system known as *eye space* (or *view space*), the eye is located at the origin of the coordinate system. Following the standard convention, you orient the scene so the eye is looking down one direction of the *z*-axis. The "up" direction is typically the positive *y* direction.

眼空间对光栅特别有用，这将会在第5章讨论到。

Eye space, which is particularly useful for lighting, will be discussed in Chapter 5.

4.1.6 视变换

（**4.1.6 The View Transform**）

从世界空间位置到眼空间位置的变换是视变换。你可以再次使用4*4的矩阵表示视变换。典型的视变换结合了一个使眼睛在世界空间的位置移动到眼空间原点的平移。然后适当地旋转眼睛。视变换通过这些定义了视点的位置和方向。

The transform that converts world-space positions to eye-space positions is the *view transform*. Once again, you express the view transform with a 4x4 matrix. The typical view transform combines a translation that moves the eye position in world space to the origin of eye space and then rotates the eye appropriately. By doing this, the view transform defines the position and orientation of the viewpoint.

图4-3举例说明了视变换。左图显示了图4-2中的机器人和被放置在世界坐标(0, 0 ,5)位置的眼

睛。右图则在眼空间中显示了他们。你会观察到眼空间的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他移动到眼空间里的正确位置。经过平移之后，机器人站在了眼空间的(0, 0,- 5)位置吗，而眼睛则处在原点。在这个例子里，眼空间和世界空间都把y轴正方向当成“朝上”的方向，并且平移完全在z轴上进行，否则，还会需要一个和平移类似的旋转。

Figure 4-3 illustrates the view transform. The left side of the figure shows the robot from Figure 4-2 along with the eye, which is positioned at <0, 0, 5> in the world-space coordinate system. The right side shows them in eye space. Observe that eye space positions the origin at the eye. In this example, the view transform translates the robot in order to move it to the correct position in eye space. After the translation, the robot ends up at <0, 0, -5> in eye space, while the eye is at the origin. In this example, eye space and world space share the positive *y*-axis as their "up" direction and the translation is purely in the *z* direction. Otherwise, a rotation might be required as well as a translation.

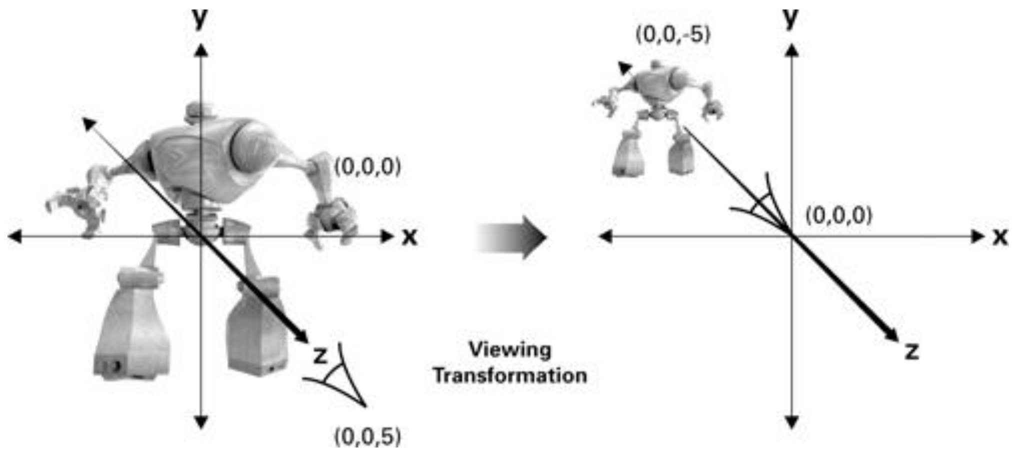


Figure 4-3 The Effect of the Viewing Transformation

4.1.6.1 modelview 矩阵

（**The modelview Matrix**）

大部分光照和其他着色计算都会涉及到位置和表面法向量。通常情况下，他们在眼空间和物体空间中变换过程中被计算效率会更高。你的应用程序中，世界空间为场景里的物体建立整体的空间关系时非常有效，但是它对光照和其他着色计算并非如此。

Most lighting and other shading computations involve quantities such as positions and surface normals. In general, these computations tend to be more efficient when performed in either eye space or object space. World space is useful in your application for establishing the overall spatial relationships between objects in a scene, but it is not particularly efficient for lighting and other shading computations.

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起来，组成一个被称为modelview的单独的矩阵。你可以通过简单地使用建模矩阵乘以视矩阵把他们结合起来。

For this reason, we typically combine the two matrices that represent the modeling and view transforms into a single matrix known as the *modelview matrix*. You can combine the two matrices by simply multiplying the view matrix by the modeling matrix.

4.1.7 剪裁空间

（**4.1.7 Clip Space**）

当位置处在眼空间之后，你需要决定什么样的位置是你在最终渲染的图像中是可见的。在眼空间转变之后的坐标系统被称为剪裁空间，这个空间中的坐标系统被称为剪裁坐标。

Once positions are in eye space, the next step is to determine what positions are actually viewable in the image you eventually intend to render. The coordinate system subsequent to eye space is known as *clip space*, and coordinates in this space are called *clip coordinates*.

一个Cg顶点程序输出的顶点的位置是在剪裁空间中的。每个顶点程序可以有选择的输出某些参数，例如纹理坐标和颜色。但是任意一个顶点程序必须输出一个剪裁空间的位置。你在之前的例子中看到的，POSITION语义正是被用来特别指明一个顶点程序输出的是剪裁空间的位置的。

The vertex position that a C g vertex program outputs is in clip space. Every vertex program optionally outputs parameters such as texture coordinates and colors, but a vertex program *always* outputs a clip-space position. As you have seen in earlier examples, the POSITION semantic is used to indicate that a particular vertex program output is the clip-space position.

4.1.8 投影变换

（**4.1.8 The Projection Transform**）

对应于投影变换的这个4*4矩阵被称为投影矩阵。

The transform that converts eye-space coordinates into clip-space coordinates is known as the *projection transform*.

投影变换定义了一个可视平截体（view frustum），代表了眼空间中物体的可见区域。多边形、线段和点只有在可视平截体中时，才有可能在被光栅化到一幅图像中时被看到。OpenGL和Direct3D对剪裁空间有些微不同的规则。在OpenGL中，任何可见的物体必须在一个轴对称的立方体中，使得他的剪裁空间位置的x、y和z成员不大于有它对应的w成员。这意味着-w <= x、y、z <= w。Direct3D对X和y也有相同的剪裁要求，而z必须是0 <= z <= w。因为这些都依赖于w，因此剪裁规则假设空间位置是齐次的。

The projection transform defines a *view frustum* that represents the region of eye space where objects are viewable. Only polygons, lines, and points that are within the view frustum are potentially viewable when rasterized into an image. OpenGL and Direct3D have slightly different rules for clip space. In OpenGL, everything that is viewable must be within an axis-aligned cube such that the *x*, *y*, and *z* components of its clip-space position are less than or equal to its corresponding *w* component. This implies that $-w \le x \le w$, $-w \le y \le w$, and $-w \le z \le w$. Direct3D has the same clipping requirement for *x* and *y*, but the *z* requirement is 0 <= *z* <= *w.* These clipping rules assume that the clip-space position is in homogeneous form, because they rely on *w*.

投影变换提供了从眼空间的可视区域（也就是可视平截体）到包含剪裁空间的可视区域周堆成立方体的映射。你可以把这个映射表示成一个4*4的矩阵。

The projection transform provides the mapping to this clip-space axis-aligned cube containing the viewable region of clip space from the viewable region of eye space—otherwise known as the view frustum. You can express this mapping as a 4x4 matrix.

4.1.8.1 投影矩阵

（**The Projection Matrix**）

对用于投影变换的这个4*4矩阵被称为投影矩阵。

The 4x4 matrix that corresponds to the projection transform is known as the *projection matrix*.

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

Figure 4-4 illustrates how the projection matrix transforms the robot in eye space from Figure 4-3 into clip space. The entire robot fits into clip space, so the resulting image should picture the robot without any portion of the robot being clipped.

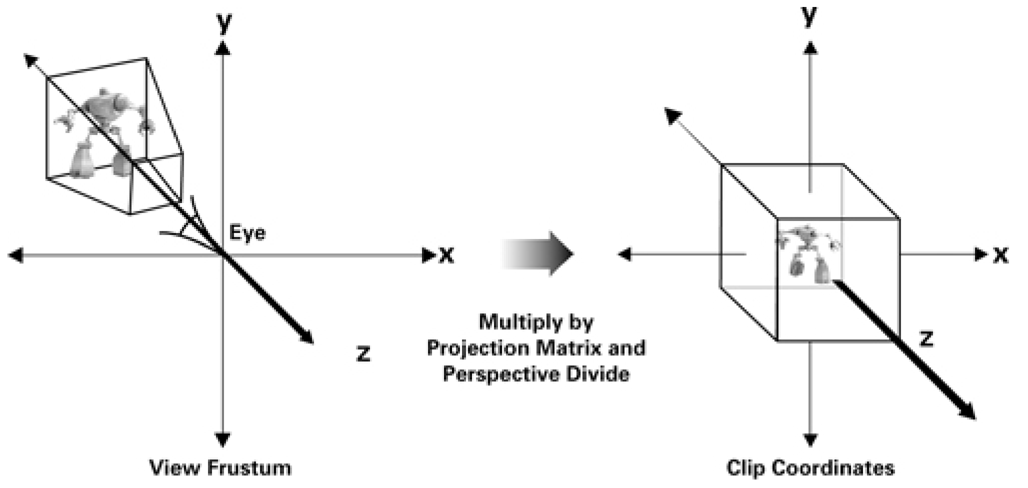


Figure 4-4 The Effect of the Projection Matrix

OpenGL和Direct3D的剪裁空间规则是不同的，并分别被设置在各自的投影矩阵中。所以，如果Cg程序员使用了恰当的投影矩阵，那么这两种剪裁空间的区别就显得不难么明显了。通常由应用程序负责提供合适的投影矩阵给Cg程序。

The clip-space rules are different for OpenGL and Direct3D and are built into the projection matrix for each respective API. As a result, if C g programmers rely on the appropriate projection matrix for their choice of 3D programming interface, the distinction between the two clip-space definitions is not apparent. Typically, the application is responsible for providing the appropriate projection matrix to Cg programs.

4.1.9 规格化设备坐标

（**4.1.9 Normalized Device Coordinates**）

剪裁坐标是齐次形式的，但是我们仍然需要计算一个表示深度值的二维位置（深度值是为了表示深度缓冲（depth buffering），一种渲染可见表面的硬件加速方法）。

Clip coordinates are in the homogenous form of <*x, y, z, w*>, but we need to compute a 2D position (an *x* and *y* pair) along with a depth value. (The depth value is for *depth buffering*, a hardware-accelerated way to render visible surfaces.)

4.1.9.1 透视除法

（**Perspective Division**）

用w除x、y和z能完成这项任务。生成的结果坐标被称为规格化设备坐标（normalized device coordinates）。现在所有的可见几何数据在OpenGL都位于坐标(-1, -1, -1)到(1, 1, 1)的正方体中，在Direct3D中的坐标范围则为(-1, -1, 0)到(1, 1, 1,)。

Dividing *x, y,* and *z* by *w* accomplishes this. The resulting coordinates are called *normalized device coordinates*. Now all the visible geometric data lies in a cube with positions between <-1, -1, -1> and <1, 1, 1> in OpenGL, and between <-1, -1, 0> and <1, 1, 1> in Direct3D.

在第2章和第3种中的二维顶点程序输出的就是你现在知道的规格化设备坐标。在这些例子中输出的二维位置假设隐含的z值为0并且w值为1。

The 2D vertex programs in Chapters 2 and 3 output what you now know as normalized device coordinates. The 2D output position in these examples assumed an implicit *z* value of 0 and a *w* value of 1.

4.1.10 窗口坐标（屏幕坐标）

（**4.1.10 Window Coordinates**）

最后一步是取每个顶点的规格化设备坐标，然后把他们转换成为使用像素度量的二维坐标系统。这一步骤被命名为视图变换（viewport transform），他为GPU的光栅器提供数据。然后，光栅器从顶点组成的店、线段和多边形中决定最后图像的片段。另一个被称为深度范围变换的变换过程则将z值缩放到深度缓冲可以使用的范围内。

The final step is to take each vertex's normalized device coordinates and convert them into a final coordinate system that is measured in pixels for *x* and *y*. This step, called the *viewport transform,* feeds the GPU's rasterizer. The rasterizer then forms points, lines, or polygons from the vertices, and generates fragments that determine the final image. Another transform, called the *depth range transform,* scales the *z* value of the vertices into the range of the depth buffer for use in depth buffering.

4.2 理论应用
（**4.2 Applying the Theory**）

尽管所有的讨论都是关于坐标空间的，但你所需要的能正确进行顶点变换的代买都是非常繁琐的。通常顶点程序接受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，这样就只需要进行一次（而不是两次）乘法。图4-5通过展示从物体坐标到剪裁坐标的两个方法，来举例说明这个原理。

Despite all the discussion about coordinate spaces, the C g code you need for transforming vertices correctly is quite trivial. Normally, the vertex program receives vertex positions in object space. The program then multiplies each vertex by the modelview and projection matrices to get that vertex into clip space. In practice, you would concatenate these two matrices so that just one multiplication is needed instead of two. Figure 4-5 illustrates this principle by showing two ways to get from object coordinates to clip coordinates.

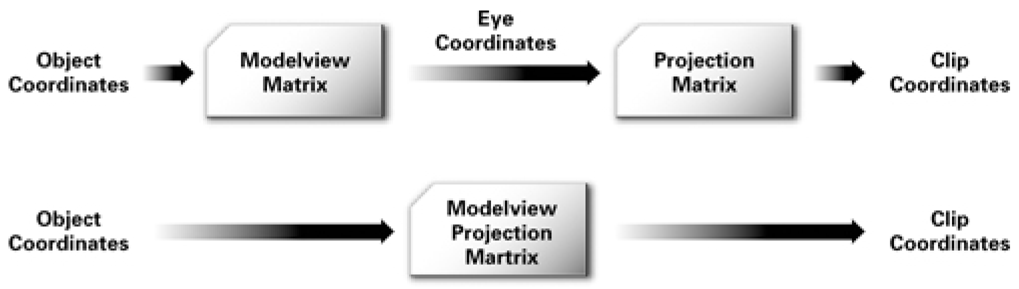


Figure 4-5 Optimizations for Transforming to C lip Space

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。

Example 4-1 shows how a typical C g program would efficiently handle 3D vertex transformations from object space directly to clip space.

Example 4-1. The C4E1v_transform **Vertex Program**

```
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与投影矩阵作为输入参数。你的OpenGL或Direct3D应用程序将负责提供这些数据。有几个Cg运行里程能够帮助你基于当前的OpenGL或Direct3D的变换状态来输入正确的矩阵。然后，使用一个矩阵乘法对position参

数进行了变换，并且把结果写到oPosition中：

The program takes the object-space position (position) and concatenated modelview and projection matrices (modelViewProj) as input parameters. Your OpenGL or Direct3D application would be responsible for providing this data. There are C g runtime routines that help you load the appropriate matrix based on the current OpenGL or Direct3D transformation state. The position parameter is then transformed with a matrix multiplication, and the result is written out to 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”前缀来区别有相同名字的输入和输出参数。

In this book, we explicitly assign all output parameters, even if they are simply being passed through. We use the " o " prefix to differentiate input and output parameters that have the same names.

第 5 章 光照

（**Chapter 5. Lighting**）

本章将教会你如何在场景中使用光源来照亮物体。我们首先从构造一个已简化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。本找包括以下 5 个部分：

1） “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。

2） “实现基本的逐顶点光照模型”主要介绍一个在 OpenGL 和 Direct3D 中使用的光照模型的简化版本。这部分还会逐步的解释如何在一个顶点程序中实现该光照模型。

3）“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。

4） “创建一个光照函数”解释如何创建自己的函数。

5） “扩扎基本模型”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 的关键感念，例如创建函数、数组和结构。

This chapter shows you how to illuminate the objects in your scenes with light sources. We start by building a simplified version of a commonly used lighting model. Then, we gradually add functionality to the basic model to make it more useful in common situations. This chapter has the following five sections:

• **"Lighting and Lighting Models"** explains the importance of lighting and introduces the concept of a lighting model.

• **"Implementing the Basic Per-Vertex Lighting Model"** presents a simplified version of the lighting model used in OpenGL and Direct3D. This section also goes through a step-by-step implementation of this lighting model in a vertex program.

• **"Per-Fragment Lighting"** describes the differences between per-vertex and per-fragment lighting and shows you how to implement per-fragment lighting.

• **"Creating a Lighting Function"** explains how to create your own functions.

• **"Extending the Basic Model"** describes several improvements to the basic lighting model, including texturing, attenuation, and spotlight effects. While explaining these improvements, we introduce several key C g concepts, such as creating functions, arrays, and structures.

5.1 光照和光照模型

（**5.1 Lighting and Lighting Models**）

到目前位置，我们所有的例子都是简单明了的，击中在一些编写 Cg 程序所必须的基本概念。下面几章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。

So far, all our examples have been straightforward, focusing on the fundamental concepts that you need to start writing programs. The next few chapters will show you how to add some more interesting effects. This chapter explains lighting.

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的原因。因为他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（不幸的是，在计算机图形学中，当你在一个场景中增加光源的时候，阴影并不能够狠轻易的生成）。第 9 章将讨论有关生成阴影的主题。

Adding a light to a scene causes many variations in shading and creates more interesting images. This is why movie directors pay such close attention to lighting: it plays a big part in telling a compelling story. Dark areas of a scene can evoke a sense of mystery and heightened tension. (Unfortunately, in computer graphics, shadows do not come "for free" when you add lights to a scene. C hapter 9 visits the separate topic of shadow generation.)

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。在过去的几十年里，许多光照模型被开发和使用，从简单的近似一直到极其精确的模拟。

Together, lighting and an object's material properties determine its appearance. A lighting model describes the way light interacts with an object, based on the light's characteristics and the object material's characteristics. Over the years, numerous lighting models have been developed and used, ranging from simple approximations to extremely accurate simulations.

图 5-1 显示了一组使用不同的光照所渲染的物体。注意不同的公式能够模拟多种多样的真实世界中的材质。

Figure 5-1 shows a set of objects that were rendered using various lighting models. Notice how the different formulations resemble an assortment of real-world materials.

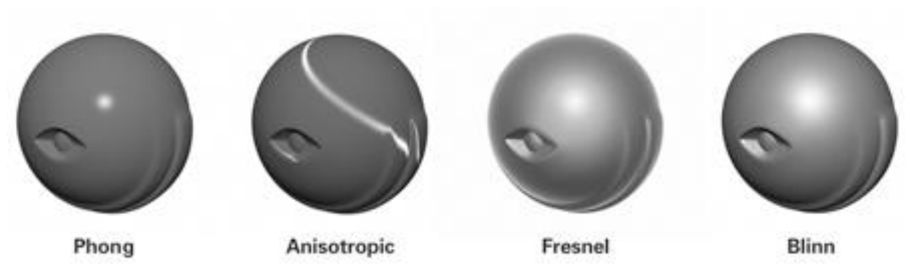


Figure 5-1 Different Lighting Models

在过去，固定功能的图形流水线被显示只能使用一个光照模型，这个模型被我们称为固定功能光照模型。这个功能固定的模型是基于众所周知的 Phong 光照模型设计的，但是经过了一些调整和增强。这个模型有几个优点：他看上去已经足够了，计算量很小，而且他有许多只管的参数可以用来调整和控制物体的外观。但是，他的问题是只能针对非常有限的一些材质，才能够工作得很好。一个塑料或橡胶的外表是使用固定光照模型最普遍的例子。 这解释了为什么许多计算机渲染的图像看起来并不真实。

In the past, fixed-function graphics pipelines were limited to one lighting model, which we call the *fixed-function lighting model*. The fixed-function model is based on what is known as the Phong lighting model, but with some tweaks and additions. The fixed-function lighting model has several advantages: it looks adequate, it's cheap to compute, and it has a number of intuitive parameters that can be tweaked to control appearance. The problem with it, however, is that it works well for only a limited set of materials. A plastic or rubbery appearance is the most common symptom of using the fixed-function lighting model, and this explains why many computer graphics images do not look realistic.

为了解除固定功能光照模型的限制，图形程序员发现了许多创新的方法来使用渲染流水线的一些其他特征。例如，聪明的程序员使用基于纹理的方法能个偶模拟一组范围很广的材质表面特征。

To get around the limitations of the fixed-function lighting model, graphics programmers found innovative ways to use other features of the pipeline. For example, clever programs used texture-based methods to mimic the surface characteristics of a wider range of materials.

随着 Cg 和可编程图形硬件的到来，你现在能够使用一种高级语言简洁地表达复杂的光照模型。你不再需要设置优先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，你不需要限定你的光照模型来适应固定功能流水线的能力。相反，你能够将你自定义的光照模型描述为一个在 GPU 中执行的 Cg 程序。

With the advent of C g and programmable hardware, you can now express complicated lighting models concisely using a high-level language. You no longer have to configure a limited set of graphics pipeline states or program tedious assembly language routines. And, you don't have to limit your lighting model to fit the fixed-function pipeline's capabilities. Instead, you can express your own custom lighting model as a Cg program that executes within your programmable GPU.

5.2 基本的逐顶点光照模型的实现

（5.2 Implementing the Basic Per-Vertex Lighting Model）

本节将解释如何用一个顶点程序实现一个固定功能光照模型的简化版本。大家对这个光照模型的而熟悉程度和这个模型的简单新使它成为一个很好的起点。首先，我们将介绍固定功能光照模型的背景知识。如果你已经对光照模型很熟悉了，你可以随意地跳过下一节直接到有关实现细节的第 5.2.2 节。

This section explains how to implement a simplified version of the fixed-function lighting model using a vertex program. The familiarity and simplicity of this lighting model make it an excellent starting point. First we give some background about the fixed-function lighting model. If you are already familiar with this lighting model, feel free to skip ahead to the implementation in Section 5.2.2.

5.2.1 基本的光照模型

（5.2.1 The Basic Lighting Model）

OpenGL 和 Direct3D 提供了几乎完全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，是对经典的放射（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、绿和蓝三原色的 float3 向量来表示每种光照作用。

OpenGL and Direct3D provide almost identical fixed-function lighting models. In our example, we will use a simplified version that we will refer to as the "Basic" model. The Basic model, like the OpenGL and Direct3D models, modifies and extends the classic Phong model. In the Basic model, an object's surface color is the sum of emissive, ambient, diffuse, and specular lighting contributions. Each contribution depends on the combination of the surface's material properties (such as shininess and material color) and the light source's properties (such as light color and position). We represent each contribution as a float3 vector that contains the red, green, and blue color components.

基本模型的高级公式的数学描述如下：

This high-level equation describes the Basic model mathematically:

surfaceColor = emissive + ambient + diffuse + specular

5.2.1.1 放射项

（**The Emissive Term**）

放射项表示了由表面所发出的光。这种关照作用是独立于所有光源的。放射项是一个 RGB 值，指明了表边所发光的颜色。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 从概念上距离说明了放射项，图 5-3 显示了一个放射性物体的渲染。这个渲染比较单调沉闷，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，一个物体放射性发光在场景中并不照亮其他物体。一个放射性物体本身并不是一个光源——他不照亮其他物体或投下阴影。另一个解释放射项的方法是：他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放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

The emissive term represents light emitted or given off by a surface. This contribution is independent of all light sources. The emissive term is an RGB value that indicates the color of the emitted light. If you were to view an emissive material in a completely dark room, it would appear to be this color. The emissive term can simulate glowing. Figure 5-2 illustrates the emissive term conceptually, and Figure 5-3 shows a rendering of a purely emissive object. The rendering is understandably boring, because the emissive color is the same all over the object. Unlike in the real world, an object's emissive glow does not actually illuminate other nearby objects in the scene. An emissive object is not itself a light source—it does not illuminate other objects or cast shadows. Another way to think of the emissive term is that it is a color added after computing all the other lighting terms. More advanced global illumination models would simulate how the emitted light affects the rest of the scene, but these models are beyond the scope of this book.

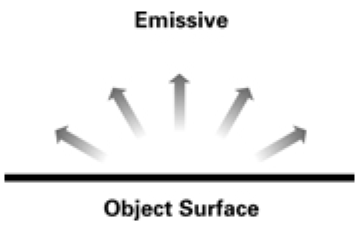


Figure 5-2 The Emissive Term

Figure 5-3 Rendering the Emissive Term

下面使我们用于放射项的数学公式：

emissive = K_e

其中：

K_e 代表材质的反射光颜色

Here is the mathematical formulation we use for the emissive term:

emissive = K_e

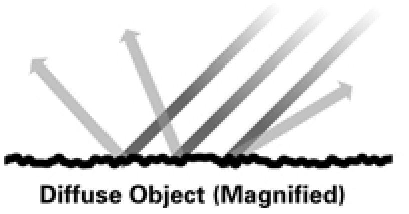
where:

- K_e is the material's emissive color.

5.2.1.2 环境反射项

（**The Ambient Term**）

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像来自各个方向一样，环境反射光看起来并不是来自某个方向的。想法它看起案例像是来自所有方向。因为这个原因，环境反射光照项并不依赖与光源的位置。图 5-4 显示了一个只收到环境反射光照的物体的渲染。环境反射向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。但是和放射颜色不同，环境反射项收全局光照的影响。



The ambient term accounts for light that has bounced arou nd so much in the scene that it seems to come from everywhere. Ambient light does not appear to come from any particular direction; rather, it appears to come from all directions. Because of this, the ambient lighting term does not depend on the light source position. Figure 5-4 illustrates this concept, and Figure 5-5 shows a rendering of an object that receives only ambient light. The ambient term depends on a material's ambient reflectance, as well as the color of the ambient light that is incident on the material. Like the emissive term, the ambient term on its own is just a constant color. Unlike the emissive color, however, the ambient term is affected by the global ambient lighting.

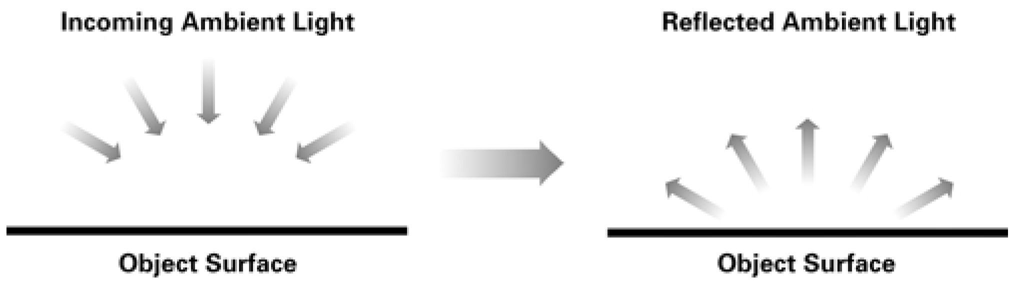


Figure 5-4 The Ambient Term

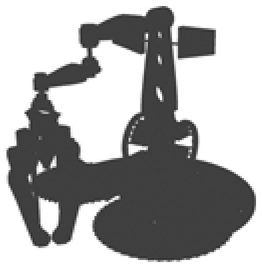


Figure 5-5 Rendering the Ambient Term

下面是用于环境反射项的数学公式

$ambient = K_a \times globalAmbient$

其中：

K_a是材质的环境反射系数

globalAmbient 是入射环境光的颜色

Here is the mathematical formulation we use for the ambient term:

$ambient = K_a \times globalAmbient$

where:

- K_a is the material's ambient reflectance and
- *globalAmbient* is the color of the incoming ambient light.

5.2.1.3 漫反射项

（**The Diffuse Term**）

漫反射项代表了从一个表面向所有方向反射出去的相同的方向光。通常，漫反射表面在微观尺度上是非常粗糙的，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。

The diffuse term accounts for directed light reflected off a surface equally in all directions. In general, diffuse surfaces are rough on a microscopic scale, with small nooks and crannies that reflect light in many directions. When incoming rays of light hit these nooks and crannies, the light bounces off in all directions, as shown in Figure 5-6.

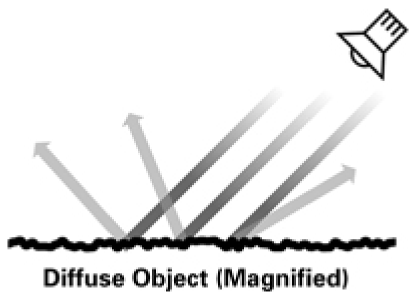


Figure 5-6 Diffuse Light Scattering

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，表面上任何一点的漫反射影响都是一样的。图 5-7 距离说民挂了漫反射项，而图 5-8 则显示了一个漫反射物体的渲染。

The amount of light reflected is proportional to the angle of incidence of the light striking the surface. Surfaces with a dull finish, such as a dusty chalkboard, are said to be diffuse. The diffuse contribution at any particular point on a surface is the same, regardless of where the viewpoint is. Figure 5-7 illustrates the diffuse term, and Figure 5-8 shows a rendering of a diffuse object.

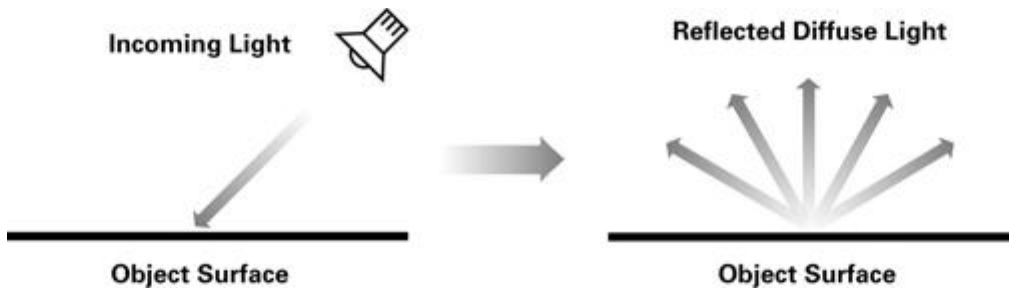


Figure 5-7 The Diffuse Term

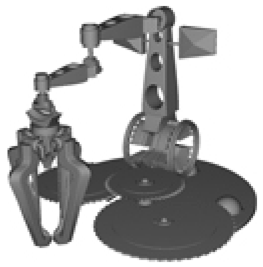


Figure 5-8 Rendering the Diffuse Term

下面是计算漫反射项的数学公式（在图 5-9 中显示）

$diffuse = K_d \times lightColor \times \max(N*L, 0)$

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*是规格化表面法向量

*L*是规格化指向光源法向量

P 是着色点

Here is the mathematical formulation we use for the diffuse term (illustrated in Figure 5-9):

diffuse = K_d x *lightColor* x max(*N***L*, 0)

where:

- K_d is the material's diffuse color,
- *lightColor* is the color of the incoming diffuse light,
- *N* is the normalized surface normal,
- *L* is the normalized vector toward the light source, and
- *P* is the point being shaded.

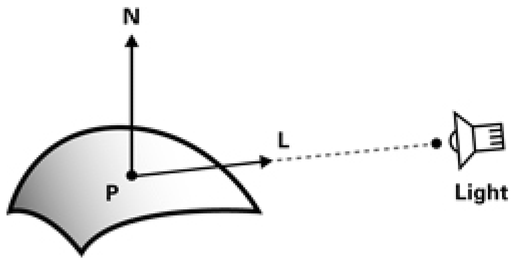


Figure 5-9 Calculating Diffuse Lighting

规格化的向量 N 和 L 的点积（或内积）是两个向量之间夹角的一个度量。两个向量之间夹角越小，点积值越大，而表面会接收到更多的入射光照。背向光源的表面将产生负的点积值，因此在公式中 max(N*L, 0)确保了这样的表面不会显示漫反射的光照。

The vector dot product (or inner product) of the normalized vectors *N* and *L* is a measure of the angle between the two vectors. The smaller the angle between the vectors, the greater the dot-product value will be, and the more incident light the surface will receive. Surfaces that face away from the light will produce negative dot-product values, so the *max*(*N* ?*L*, 0) in the equation ensures that these surfaces show no diffuse lighting.

5.2.1.4 镜面反射项

（**The Specular Term**）

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明显的，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。

The specular term represents light scattered from a surface predominantly around the mirror direction. The specular term is most prominent on very smooth and shiny surfaces, such as polished metals. Figure

5-10 illustrates the concept of specular reflection, and Figure 5-11 shows a rendering of a completely specular object.

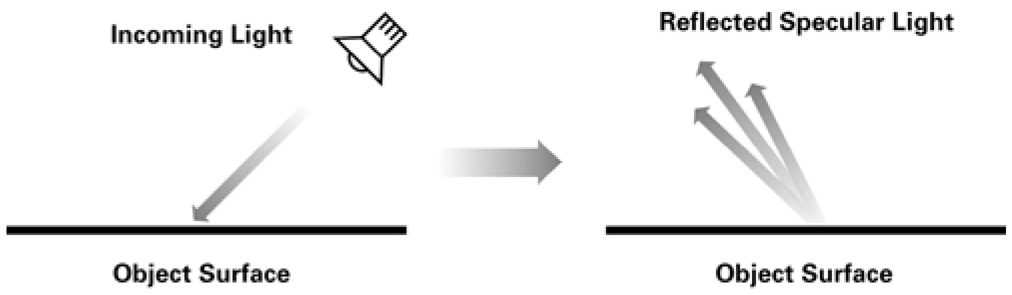


Figure 5-10 The Specular Term

Figure 5-11 Rendering the Specular Term

与其他光照项不同，镜面反射的作用更依赖与观察者的位置。如果观察者不在一个能够接受反射光线的位置，观察者将不可能在表面上看到一个镜面反射的高光效果。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。而较少光泽度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。

Unlike the emissive, ambient, and diffuse lighting terms, the specular contribution depends on the location of the viewer. If the viewer is not at a location that receives the reflected rays, the viewer will not see a specular highlight on the surface. The specular term is affected not only by the specular color properties of the light source and material, but also by how shiny the surface is. Shinier materials have smaller, tighter highlights, whereas less shiny materials have highlights that are more spread out. Figure 5-12 shows some examples of shininess, with the shininess exponent increasing from left to right.

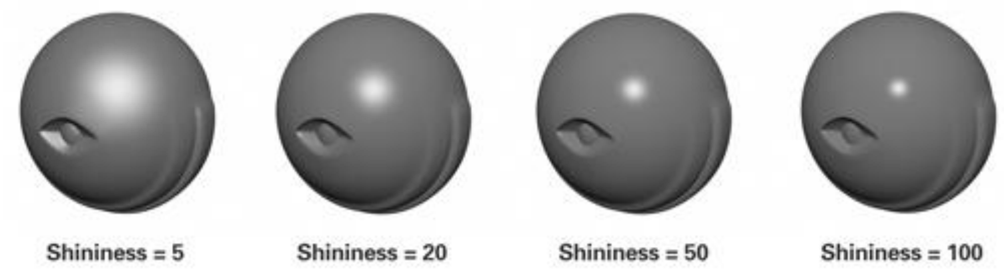


Figure 5-12 Examples of Different Shininess Exponents

下面是计算镜面反射的数学公式（如图 5-13 所示）：

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

其中：

K_s是材质的镜面反射颜色

*lightColor*是入射镜面的反射光的颜色

N是规格化的表面法向量

V是指向视点的规格化向量

*L*是指向光源的规格化向量

H是V和L中间向量的规格化向量

P 是着色点

*facing 如果N*L 大于0 则为1，否则为0*

Here is the mathematical formulation we use for the specular term (illustrated in Figure 5-13):

specular = K_s x *lightColor* x *facing* x (max(*N* * *H*, 0)) ^{shininess}

where:

- K_s is the material's specular color,
- *lightColor* is the color of the incoming specular light,
- *N* is the normalized surface normal,
- *V* is the normalized vector toward the viewpoint,
- *L* is the normalized vector toward the light source,
- *H* is the normalized vector that is halfway between *V* and *L*,
- *P* is the point being shaded, and
- *facing* is 1 if *N* * *L* is greater than 0, and 0 otherwise.

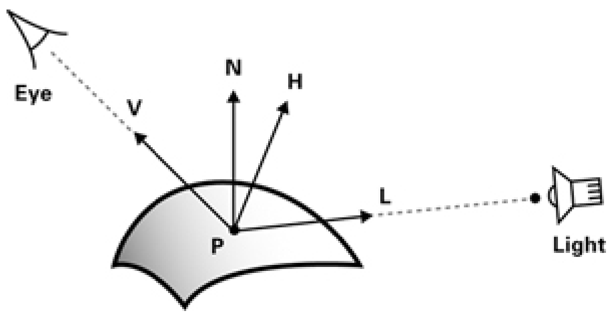


Figure 5-13 Calculating the Specular Term

当视向量 V 和半角向量 H 之间的夹角很小的时候，材质的镜面反射将表现得很明显。N 和 H 的点积的幂确保了当 H 和 V 分开的时候,镜面反射能够迅速减弱

When the angle between the view vector *V* and the half-angle vector *H* is small, the specular appearance of the material becomes apparent. The exponentiation of the dot product of *N* and *H* ensures that the specular appearance falls off quickly as *H* and *V* move farther apart.

另外，如果因为 N*L（来自漫反射光照）是复值，漫反射项为 0 的话，镜面反射项将被强制设置为 0，这能确保镜面反射高光不出现在背向光源的集合表面上。

Additionally, the specular term is forced to zero if the diffuse term is zero because *N* ?*L* (from diffuse lighting) is negative. This ensures that specular highlights do not appear on geometry that faces away from the light.

5.2.1.5 把所有的项加在一起

（**Adding the Terms Together**）

把环境反射、漫反射和镜面反射结合在一起将给出最后的光照。如图 5-14 所示。在这幅图中，我们故意地把散射项排除在外，因为他通常被用来实现特殊的效果而不是用来照亮物体。

Combining the ambient, diffuse, and specular terms gives the final lighting, as shown in Figure 5-14. In the figure, we deliberately exclude the emissive term, because it is normally used to achieve special effects rather than for lighting ordinary objects.

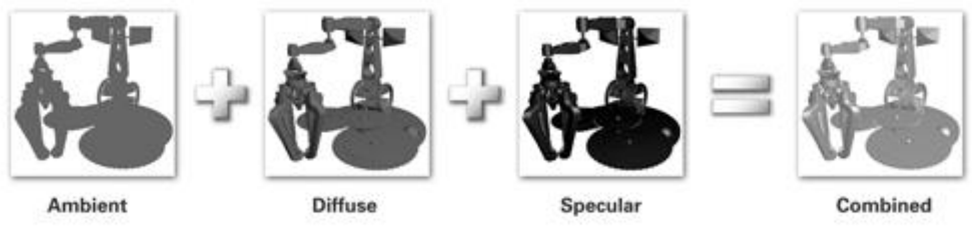


Figure 5-14 Putting the Terms Together

5.2.1.6 简化
（**Simplifications**）

如果你使用 OpenGL 或 Direct3D 的经验，你也许已经注意到了对基本光照模型有很多简化方法。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。我们还为光的漫反射和镜面反射使用了同样的颜色，儿不允许他们每个取不同的值。另外，我们没有引入衰减和聚光灯的效果。

If you are experienced with OpenGL or Direct3D, you might have noticed a number of simplifications for the Basic lighting model. We are using a global ambient color instead of a per-light ambient color. We are also using the same value for the light's diffuse and specular colors instead of allowing different values for each of these. In addition, we do not account for attenuation or spotlight effects.

5.2.2 一个基本的逐顶点光照的顶点程序

（**5.2.2 A Vertex Program for Basic Per-Vertex Lighting**）

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

This section explains a C g vertex program that implements the Basic lighting model described in Section 5.2.1

在示例 5-1 中的 C5E1v_basicLight 顶点程序作了以下工作：

1） 把顶点的位置从物体弓箭变换到剪裁空间。

2） 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。

The C5E1v_basicLight vertex program in Example 5-1 does the following:

- Transforms the position from object space to clip space.
- Computes the illuminated color of the vertex, including the emissive, ambient, diffuse, and specular lighting contributions from a single light source.

在这个例子中，我们在物体空间执行光照计算。加入你把所有需要的变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 在眼空间而不是在物体空间执行他们的光照计算。当场景中有多个光源的时候，眼空间要比物体空间更有效，但物体空间更容易实现。

In this example, we perform the lighting calculations in object space. You could also use other spaces, provided that you transform all necessary vectors into the appropriate coordinate system. For example, OpenGL and Direct3D perform their lighting computations in eye space rather than object space. Eye space is more efficient than object space when there are multiple lights, but object space is easier to implement.

Example 5-1. The C5E1v_basicLight **Vertex Program**

```
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;
}
```

在本章结尾的练习部分讨论了眼空间和物体空间光照性能的问题。

The exercises at the end of this chapter explore the trade-offs between eye-space and object-space lighting.

5.2.2.1 由应用程序制定的数据

（**Application-Specified Data**）

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换（varying）（如果这项随着每个顶点而改变），或者统一（uniform）（如果这项几乎不变）（例如在每个物体的基础上）。

Table 5-1 lists the various pieces of data that the application needs to send to the graphics pipeline. We classify each item as *varying* if the item changes with every vertex, or *uniform* if the item changes less frequently (such as on a per-object basis).

Table 5-1. Application-Specified Data for the Graphics Pipeline

Parameter	**Variable Name**	**Type**	**Category**
GEOMETRIC PARAMETERS			
Object-space vertex position	position	float4	Varying
Object-space vertex normal	normal	float3	Varying
Concatenated modelview and projection matrices	modelViewProj	float4x4	Uniform
Object-space light position	lightPosition	float3	Uniform
Object-space eye position	eyePosition	float3	Uniform
LIGHT PARAMETERS			
Light color	lightColor	float3	Uniform
Global ambient color	globalAmbient	float3	Uniform
MATERIAL PARAMETERS			
Emissive reflectance	Ke	float3	Uniform
Ambient reflectance	Ka	float3	Uniform
Diffuse reflectance	Kd	float3	Uniform
Specular reflectance	Ks	float3	Uniform
Shininess	shininess	float	Uniform

5.2.2.2 一个调试技巧

（**A Debugging Tip**）

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非常简单的时候，慢慢地建立这个程序是一个好主意。每当你增加一些代码，你最好运行以下程序并查看一下结果，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，然后期望它输出正确结果要明智得多。如果你犯了一个微笑的错误，如果你知道也许是最近的修改引起的，那么跟踪一个问题将变得容易得多。

As you can see, the code for lighting is significantly more complex than any C g code you have seen so far. When you are working on a program that is not trivial, it is a good idea to build it up slowly, piece by piece. Run the program as you make each incremental addition, and check to make sure that the results are what you expect. This is much smarter than writing all the code for the program and then hoping that it generates the right result. If you make a minor error, tracking down a problem will be a lot easier if you know the recent changes that probably caused it.

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作用的小模块（放射、环境反射、漫反射和镜面反射）。因此，有一个好的方法是先计算 emissive，然后只把 color 设为 emissive, 随后计算 ambient，并把 color 设置成 emissive+ambient。通过这种方法逐渐创建你的 Cg 程序，你将能够少受很多折磨。

This tip applies particularly to lighting code, because lighting can be broken down into various contributions (emissive, ambient, diffuse, and specular). Therefore, a good approach is to calculate emissive , and then set color to just emissive . After that, calculate ambient , and set color to emissive plus ambient . By building up your C g programs gradually in this manner, you will save yourself a lot of frustration.

5.2.2.3 顶点程序体

（**The Vertex Program Body**）

5.2.2.3.1 计算剪裁空间

（**Calculating the Clip-Space Position**）

我们从计算通常给光栅器的剪裁空间位置开始，如在第 4 章所解释的那样：

We start by computing the usual clip-space position calculation for the rasterizer, as explained in Chapter 4:

```
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 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、光源位置和眼睛位置）也都是 float3 的。

Next, we instance a variable to store the object-space vertex position, because we will need this information later on. We are going to use a float3 temporary variable because the other vectors used in lighting (such as the surface normal, light position, and eye position) are also float3 types.

```
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这个被称为重组（swizzling）的一个功能。

Here we see an interesting new piece of syntax: position.xyz . This is our first look at a feature of Cg called *swizzling*.

我们忽略了物体空间的 w 分量，事实上我们假设 w 值为 1。

By ignoring the object-space *w* component, we are effectively assuming that *w* is 1.

5.2.2.3.2 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式。重组由与结构成员连接符相同的“点”（“.”），加上你希望重组的尾缀组成。这个后缀是字母 x、y、z 和 w 的一些组合。用于 RGBA 的颜色字母 r、g、b 和 a 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 对应向量的第一个分量,y 和 g 对应第二个分量，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，并使用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。

Swizzling allows you to rearrange the components of a vector to create a new vector—in any way that you choose. Swizzling uses the same period operator that is used to access structure members, plus a suffix that indicates how you would like to rearrange the components of a vector that you're working on. The suffix is some combination of the letters x , y , z , and w . The letters r , g , b , and a —appropriate for RGBA colors—can also be used. However, the two sets of suffix letters cannot be mixed. These letters indicate which components of the original vector to use when constructing the new one. The letters x and r correspond to the first component of a vector, y and g to the second component, and so on. In the previous example, position is a float4 variable. The .xyz suffix extracts the *x, y,* and *z* components of position and creates a new three-component vector from these three values. This new vector is then assigned to the float3 variable called 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类型的支持。但是，重组在 Cg 中对有效的地操作向量是非常有用的。

Neither C nor C ++ supports swizzling because neither language has built-in support for vector data types. However, swizzling is quite useful in C g to manipulate vectors efficiently.

下面是一些重组的例子：

Here are some more examples of swizzling:

```
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)
```

仔细看以下这四行代码。第一行声明了一个名为 vec1 的 float4 类型的变量。第二行提取了 vec1 的 y 和 x 分量，并从他们创建了一个新的交换了 x 和 y 的 float2 类型向量，并把这个向量赋值给 vec2。第三行，vec1 的 w 分量被付给了单独的 float，名为 scalar。在最后一行，一个 float3 类型的向量是通过 scalar 的三次复制创建的。这被称为拖影（smearing），它证明了 Cg 将标量视为只就像只有一个分量的向量（也就是 x 后缀被用来存取标量的值）。

Take a closer look at these four lines of code. The first line declares a float4 called vec1 . The second line takes the y and x components of vec1 and creates a swapped float2 out of them. This vector is then assigned to vec2 . In the third line, the w component of vec1 is assigned to a single float , called scalar . Finally, in the last line, a float3 vector is created by replicating scalar three times. This is known as *smearing*, and it illustrates that C g treats scalar values just like one component vector (meaning that the .x suffix is used to access the scalar's value).

你还能够重组矩阵，所以基于一系列的矩阵元素来创建向量。你可以使用.m<行><列>的符号实现这点。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向量。例如：

You can also swizzle matrices to create vectors based on a sequence of matrix elements. To do this, use the ._m *<row><col>* notation. You can chain together a series of matrix swizzles, and the result will be an appropriately sized vector. For example:

```
float4x4 myMatrix;
float myFloatScalar;
float4 myFloatVec4;'
// Set myFloatScalar to myMatrix[3][2]
myFloatScalar = myMatrix._m32;
// Assign the main diagonal of myMatrix to myFloatVec4
myFloatVec4 = myMatrix._m00_m11_m22_m33;
```

另外，你能够使用[]数组操作符来存取矩阵单独的一行。例如使用在前面代码示例中的声明变量：

In addition, you can access an individual row of a matrix using the [] array operator. Using the variables declared in the preceding code sample:

```
// Set myFloatVector to the first row of myMatrix
myFloatVec4 = myMatrix[0];
```

5.2.2.3.3 掩码写入

（**Write Masking**）

Cg 支持的另一个与重组相关的操作就是写入掩码（write masking），他只允许指定的向量通过一个赋值语句被更新。例如，你只能通过使用一个 float2 类型的向量写入一个 float4 类型向量的 x 和 y 分量：

Cg supports another operation, related to swizzling, called *write masking,* that allows only specified components of a vector to be updated by an assignment. For example, you could write to just the *x* and *w* components of a float4 vector by using a float2 vector:

```
// Assume that initially vec1 = (4.0, -2.0, 5.0, 3.0)
// and vec2 = (-2.0, 4.0);
vec1.xw = vec2; // Now vec1 = (-2.0, -2.0, 5.0, 4.0)
```

写入掩码后缀可以用任何顺序勒出 x、y、z 和 w(或 r、g、b 和 a)分量。每个字母在给定的写入掩码后缀中最多只能够出现一次，而且你不能再一个写入掩码后缀中回合 xyzw 和 rgba 字母。

Note
在最先进的 GPU 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和有效性。

On most modern GPUs, swizzling and write masking are operations that have no performance penalty. So use both features whenever they help improve the clarity or efficiency of your code.

5.2.2.3.4 放射光照作用

（**The Emissive Light Contribution**）

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

There is nothing much to do for the emissive term. For the sake of coding clarity, we instance a variable, named emissive , for the emissive light contribution:

```
// Compute emissive term
float3 emissive = Ke;
```

Note
当 Cg 编译器把你的程序翻译成可执行代码的时候，他还对翻译的代码进行优化。因此上面的片段代码中创建中间变量（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是你的代码可读性更强，所以应该鼓励为中间结果使用实例命名的方法。

When the Cg compiler translates your program to executable code, it also optimizes the translated code so that there is no performance penalty for creating intermediate variables such as the emissive variable in the above code fragment. Because instancing such variables makes your code more readable, you are encouraged to instance named variables for intermediate results to improve the clarity of your code.

5.2.2.3.5 环境反射光照作用

（**The Ambient Light Contribution**）

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他。这是一个对每个分量的乘法，这意味着我们必须提供 Ka 的每个分量，并用全局环境反射光的对应颜色乘以他。下面的代码同时使用重组和掩码写入来返程这项工作。

For the ambient term, recall that we have to take the material's ambient color, Ka , and multiply it by the global ambient light color. This is a per-component multiplication, meaning that we want to take each color component of Ka and multiply it with the corresponding color component of the global ambient light color. The following code, which uses both swizzling and write masking, would get the job done:

```
// An inefficient way to compute the ambient term
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，但是他看起来确实很费事，而且形式上也不够优雅。因为 Cg 对向量有原生的支持，它允许你更简洁地表达这种类型的操作。下面是一个更简洁的使用向量缩放另一个向量的方法：

This code works, but it certainly looks like a lot of effort, and it isn't very elegant. Because C g has native support for vectors, it allows you to express this type of operation concisely. Here is a much more compact way to scale a vector by another vector:

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

非常简单，不是么？在一个对向量和矩阵及对他们执行的普通操作原生的支持的语言下工作是十分方便的。

Pretty simple, isn't it? It is convenient to work in a language that has built-in support for vectors and matrices, as well as the common operations that you perform on them.

5.2.2.3.6 漫反射光照作用

（**The Diffuse Light Contribution**）

现在，我们进入光照模型更加有趣的部分。为了漫反射的计算，我们需要从顶点到光照的向量。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：

Now we get to the more interesting parts of the lighting model. For the diffuse calculation, we need the vector from the vertex to the light source. To define a vector, you take the end point and subtract the starting point. In this case, the vector ends at lightPosition and starts at P：

```
// Compute the light vector
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单位化。幸运的是，在 Cg 标准库中声明了一个 normalize 函数，能够返回一个单位化的向量。如果一个向量没有正确的单位化，光照将会太暗或太亮。

We are interested in direction only, not magnitude, so we need to normalize the vector. Fortunately, there is a normalize function, which is declared in the C g Standard Library, that returns the normalized version of a vector. If the vector is not normalized correctly, the lighting will be either too bright or too dark.

normalize(v) ： Returns a normalized version of vector v

下一步，我们需要进行实际的光照计算。这是一个稍微复杂一些的表达式，因此，我们一段段地分析它。首先，公式里有一个内积。回忆一下，内积是一个返回单独值的基本数学函数，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的内积。

Next, we need to do the actual lighting computation. This is a slightly complicated expression, so look at it in pieces. First, there is the dot product. Recall that the dot product is a basic math function that computes a single value, which represents the cosine of the angle between two unit-length vectors. In C g, you can use the dot function to calculate the dot product between two vectors:

dot(a, b) ： Returns the dot product of vectors a and b

因此，计算 N 和 L 间的点积代码片段是：

Therefore, the code fragment that finds the dot product between N and L is this:

```
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，而且在光照公式中和其他项加载一起的时候会引起错误。为了处理这个问题，你就必须把这样的结果隐射为 0。这意味着如果点积的值小于 0，则将它设为 0。这个隐射操作使用它 Cg 的 max 函数非常容易实现：

max(a, b) ： Returns the maximum of a and b

把隐射增加到前面的表达式中得到：

Adding clamping to the previous expression gives:

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

因此，最后的代码片段看起来是这样的：

So the final piece of code looks like this:

```
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 的值是一个标量。记住，在 Cg 中你能够用一个标量乘以一个向量，这么做会用这个标量缩放向量中的每一个分量。因此，你能够用两个乘法把这些颜色很容易地结合起来：

Finally, you have to factor in the diffuse material color (Kd) and the light color (lightColor). The diffuseLight value that you just calculated is a scalar quantity. Remember that in C g, you can multiply a vector by a scalar; doing so will scale each component of the vector by the scalar. So you can combine all the colors easily with two multiplications:

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

5.2.2.3.7 镜面反射光照作用

（**The Specular Light Contribution**）

镜面反射的计算需要更多小的工作。返回去看以下图 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。

The specular calculation requires a little more work. Take a look back at Figure 5-13, which shows the various vectors that you'll need. You already have the L vector from the diffuse lighting calculation, but the V and H vectors still need to be calculated. This is not too hard, given that you already have the eye position (eyePosition) and vertex position (P).

我们从计算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，我们应该规格化这个向量。下面的代码是所需的结果:

Start by finding the vector from the vertex to the eye. This is typically called the *view vector,* or simply V in the example code. Because we are trying to define a direction, we should normalize the vector. The following code is the result:

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

下一步，你需要，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 需要被规格化，因为它代表了一个方向。要计算 H，你可以使用以下表达式：

Next, you need H , the vector that is halfway between the light vector L and the view vector V . For this reason, H is known as the *half-angle vector*. Like V , H needs to be normalized, because it represents a direction. To find H , you could use the following expression:

```
// An inefficient way to calculate H
float3 H = normalize(0.5 * L + 0.5 * V);
```

但是，因为你正在做一个规格化操作，用 0.5 来缩放 L 和 V 没有任何作用，因为在规格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：

However, since you are doing a normalize operation, scaling L and V by 0.5 has no effect, because the scaling factor cancels out during the normalization process. So the actual code looks like this:

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

现在，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：

At this point you're ready to calculate the specular term. As with the diffuse term, build up the expression for the specular term piece by piece. Start with the dot product of H and N :

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

和漫反射光照一样，结果也需要被限制为大于 0 的数：

This needs to be clamped to zero, just like the diffuse lighting:

```
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，这将缩小镜面反射高光的效果。要求一个值的幂可以使用 Cg 的 pow 函数：

The result has to be raised to the power indicated by shininess . This has the effect of narrowing the specular highlight as the shininess value increases. To raise a quantity to a power, use C g's pow function:

pow(x, y) ： Returns x^y

增加 pow 函数到镜面反射光照表达式得到：

Adding the pow function to the specular lighting expression gives:

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

把所有结果综合在一起，你将得到镜面反射光照：

Putting it all together, you've got the specular lighting:

```
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），镜面反射的高光不会出现。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 条件表达式功能的机会。

Finally, you must ensure that specular highlights do not show up when the diffuse lighting is zero (because the surface faces away from the light source). In other words, if the diffuse lighting is zero, then set the specular lighting to zero. Otherwise, use the calculated specular lighting value. This is a good opportunity to use C g's conditional expression functionality.

5.2.2.3.8 条件表达式

（**Conditional Expressions**）

和在 C 中一样，Cg 允许你使用关键字 if 和 else 来表示条件表达式，比如：

As in C , C g allows you to use the keywords if and else to evaluate conditional expressions. For example:

```
if (value == 1) {
color = float4(1.0, 0.0, 0.0, 1.0); // Color is red
} else {
color = float4(0.0, 1.0, 0.0, 1.0); // Color is green
}
```

还是和 C 中一样，你可以使用“？ ：” 符号来非常简洁地实现条件表达式。“？ ：”符号的使用方法如下：

Also like C , you can use the ?: notation to implement conditional expressions very concisely. The ?: notation works as follows:

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

因此上面的示例可以表达成：

The previous example can therefore be expressed as:

```
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，下面是包括了条件判断的镜面反射光照代码：

Getting back to the example, here is the specular lighting code, with the conditional test included:

```
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）。起先，是用两个单独的颜色来控制镜面反射高光看起来有些奇怪。但这是非常有用的，以为内有些材质（如金属）有与材质颜色相似的镜面反射高光，但是其他材质（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调制。调整 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。

As with the diffuse lighting calculation, you have to factor in the material's specular color (Ks) as well as the light's color (lightColor). At first, it might seem odd to have two separate colors that control the specular highlight. However, this is useful because some materials (such as metals) have specular highlights that are similar to the material color, whereas other materials (such as plastics) have specular highlights that are white. Both kinds of highlights are then modulated by the light's color. The Ks and lightColor variables are convenient ways to tweak the lighting model to achieve a particular appearance.

镜面反射分量可以按如下的方式计算得到：

The specular component is calculated as follows:

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

5.2.2.3.9 集成

（**Putting It All Together**）

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付给名为 color 的输出参数：

The final step is to combine the emissive, ambient, diffuse, and specular contributions to get the final vertex color. You also need to assign this color to the output parameter called color .

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

5.2.3 逐顶点的光照片段程序。

（**5.2.3 The Fragment Program for Per-Vertex Lighting**）

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，我们完成了所有的任务。

Because the vertex program has already performed the lighting calculations, your fragment program needs only to take the interpolated color and pass it to the frame buffer. We reuse the C2E2f_passthrough fragment program for this task, and we're done.

5.2.4 逐顶点光照渲染效果

（**5.2.4 Per-Vertex Lighting Results**）

图 5-15 显示了逐顶点光照程序的一个渲染示例。

Figure 5-15 shows a sample rendering from the per-vertex lighting program.

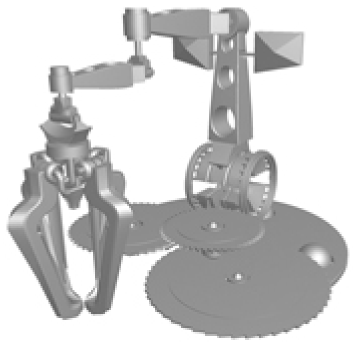


Figure 5-15 Per-Vertex Lighting Results

5.3 逐片段光照

（**5.3 Per-Fragment Lighting**）

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。这意味着如果模型足够简单，你可以分辨出底层的网络结构。如果模型具有很少的顶点，每个模型的光照将不够充分。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。这副图显示了 3 个不同镶嵌等级的圆柱体。在每个被光照渲染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。镶嵌的数量从左向右依次增加，可以看到光照效果的改善十分明显。

You may have noticed that the per-vertex lighting results look somewhat coarse. The shading tends to be a bit "triangular" looking, meaning that you can make out the underlying mesh structure, if the model is simple enough. If the model has very few vertices, per-vertex lighting will often be inadequate. However, as your models acquire more and more vertices, you will find that the results start to improve considerably, as in Figure 5-16. The figure shows three tessellated cylinders with different levels of tessellation. Below each lit cylinder is the wireframe version of the model showing each cylinder's tessellation. As the amount of tessellation increases from left to right, notice that the lighting improves significantly.

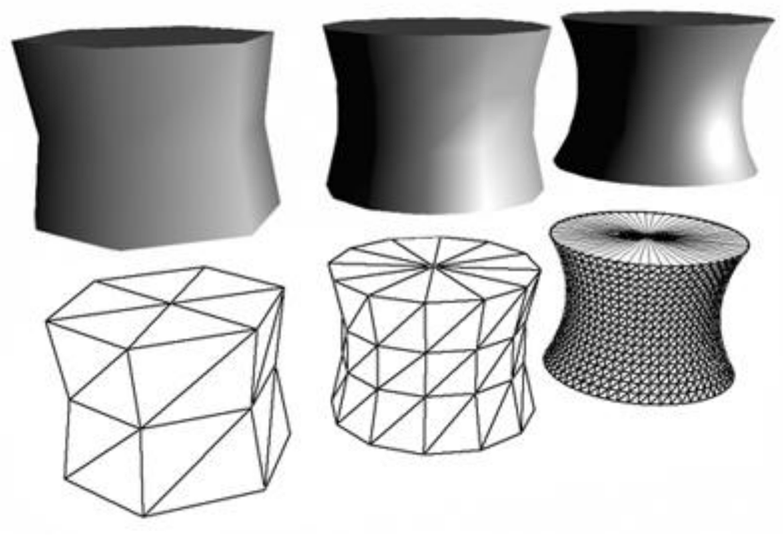


Figure 5-16 The Effects of Tessellation on Lighting

由于数据被插值和使用的方式，镶嵌少的模型使用逐顶点的光照看起来效果很差。使用逐顶点光照，光照只在每个三角形的顶点计算。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。因为光照公式并不是真正得为每个片段估算的，所以这种着色会丢失一些细节。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，即使他本来应该显示。

Less tessellated models look bad with per-vertex lighting because of the way data is interpolated and used. With per-vertex lighting, lighting is calculated only at each vertex of each triangle. The lighting is then interpolated for each fragment that is generated for the triangle. This approach, called smooth color interpolation or *Gouraud shading*, can miss details because the lighting equation is not actually evaluated for each fragment. For example, a specular highlight that is not captured at any of the triangle's vertices will not show up on the triangle, even if it should appear inside the triangle.

使用其那面的逐顶点光照示例所产生的情况就是这样的。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。

This is exactly what could happen with the preceding per-vertex lighting example: your vertex program calculated the lighting, and then the rasterizer interpolated the colors for each fragment.

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，而不是仅仅为每个顶点计算。因此，将被插值是表面法向量，而不是插值最后的光照颜色。然后，片段程序使用插值过的表面法向量在每个像素点计算光照。这种技术被称为 Phong 着色（不要与 Phong 光照模型混淆，Phong 光照模型指的是基本光照模型中使用的镜面反射近似模型）或更普通一点逐像素光照（per-pixel lighting）或逐片段光照（per-fragment lighting）。正如你所预期的，逐片段光照给出了更好的结果，以为整个光照公式是为了每个三角形的片段计算的。如图 5-17 所示，图的左边显示了一个用逐顶点光照渲染的一个被镶嵌的圆柱体，右边显示了一个用逐片段光照渲染的同样的圆柱体。每个圆柱体有这同样粗糙程度的镶嵌。注意一下，高光在左边的圆柱体上比较粗糙，而右边的圆柱体则比较明显。

To get a more accurate result, you need to evaluate the whole lighting model for each fragment, instead of just for each vertex. So instead of interpolating the final lit color, the surface normals are interpolated. Then, the fragment program uses the interpolated surface normals to calculate the lighting at each pixel. This technique is called *Phong shading* (not to be confused with the Phong lighting model, which refers to the specular approximation used in the Basic model) or, more commonly, *per-pixel lighting* or *per-fragment lighting*. As you might expect, per-fragment lighting gives much better results because the whole lighting equation is evaluated for each fragment of each triangle, as shown in Figure 5-17. The left side of the figure shows a tessellated cylinder rendered with per-vertex lighting, and the right side shows the same cylinder rendered with per-fragment lighting. Each cylinder has the same coarse tessellation. Notice that the highlights are coarse on the left cylinder and much sharper on the right cylinder.

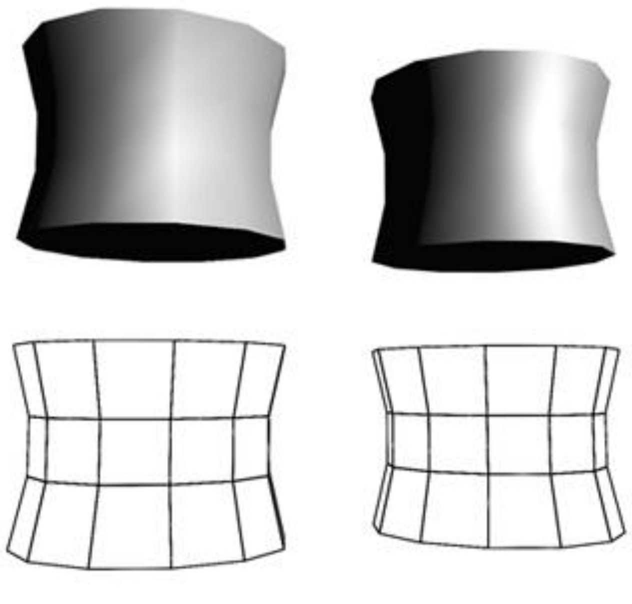


Figure 5-17 C omparing Per-Vertex and Per-Fragment Lighting

5.3.1 实现逐片段的光照

（**5.3.1 Implementing Per-Fragment Lighting**）

Note
在这个例子中的片段程序需要第四代的 GPU，例如 NVidia 的 GeForceFX 或者 ATL 的 Radeon 9700。

The fragment program in this example requires a fourth-generation GPU, such as NVIDIA's GeForce FX or ATI's 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比较有趣的工作，而顶点程序将只是做基础的辅助工作，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运行。这并不令人吃惊，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精细的控制。使用一个顶点或片段程序还有其他一些限制，例如性能。在第 10 章将会进一步讨论这个主题。

In this example, the computational burden on vertex and fragment programs will be swapped; this time, the fragment program will do the interesting work. The vertex program will only help set the stage by passing some parameters over to the fragment program. Many advanced techniques follow this pattern as well, which is probably not very surprising, because fragment programs give you more detailed control over your final image than vertex programs do. Using a vertex or fragment program also has other implications, such as performance. Chapter 10 discusses this topic further.

你将会发现用逐片段光照的片段程序与用逐顶点光照的顶点程序看上去非常相似。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。这种能力证明对逐片段的光照非常有用，因为你对所需要的代码已经很熟悉了。和逐顶点的光照一样，逐片段的光照也是在物体空间中实现的。

You will find that the fragment program for per-fragment lighting looks remarkably similar to the vertex program for per-vertex lighting. Again, that is because C g allows you to use a common language for both vertex and fragment programming. This capability turns out to be very useful for per-fragment lighting, because the required code will already be familiar to you. Like the per-vertex lighting, the per-fragment lighting is done in object space, for ease of implementation.

5.3.2 用于逐片段光照的顶点程序

（**5.3.2 The Vertex Program for Per-Fragment Lighting**）

这个示例的顶点程序知识一个传输管道：他执行最小限度的计算和传输必要的数据给流水线，因此被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，他们被输出到纹理坐标集 0 和 1 中。

The vertex program for this example is just a conduit: it performs minimal computations and essentially forwards data down the pipeline, so that the fragment program can do the interesting work. After writing out the homogeneous position, the vertex program also passes along the object-space position and object-space normal, outputting them as texture coordinate sets 0 and 1.

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，因此扎住这个机会以确保你已经完全理解了程序中的每行代码。

Example 5-2 shows the complete source code for the vertex program. There are no new concepts here, so take this opportunity to make sure that you completely understand each line in the program.

Example 5-2. The C5E2v_fragmentLighting **Vertex Program**

```
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 用于逐片段光照的片段程序

（**5.3.3 The Fragment Program for Per-Fragment Lighting**）

C3E3f_basicLight 程序和用于逐顶点光照的顶点程序几乎完全一样，因此我们不会对他进行详细地讨论。示例 5-3 显示了用于逐片段光照程序的源代码。

The C5E3f_basicLight program is almost identical to the vertex program for per-vertex lighting, and so we will not go through it in detail. Example 5-3 shows the source code for the per-fragment lighting program.

假设物体空间的每个顶点法向量已经正常地被规格化。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逐片段法向量重新进行规格化：

It is common to assume that the object-space per-vertex normal is already normalized. In such a case, one operation that C5E3f_basicLight performs that the corresponding vertex program would not require is renormalizing the interpolated per-fragment normal:

```
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在规格化。

This normalize is necessary because linear interpolation of a texture coordinate set can cause the per-fragment normal vector to become de-normalized.

C5E3f_basicLight 片段程序显示了 Cg 真的语序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 GPU 足够强大——编写这个程序的时候需要第四代或更好的 GPU）。但是逐片段的计算并不是无开销的。在大多数情况，在一帧中的片段数目将比顶点数目多很多，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的造成更严重的影响。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。

The C5E3f_basicLight fragment program shows that Cg really does allow you to express your ideas the same way in both vertex and fragment programs (as long as your GPU is powerful enough to keep up— this program, as written, requires a fourth-generation GPU or better). But per-fragment calculations aren't free. In most cases, there will be more fragments in the frame than there are vertices, which means that the fragment program needs to run many more times than the vertex program. Therefore, longer fragment programs tend to have a more significant impact on performance than longer vertex programs. Chapter 10 discusses in more detail the trade-offs between using vertex and fragment programs.

Example 5-3. The C5E3f_basicLight **Fragment Program**

```
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;
}
```

在本章的其他部分和本书的其余部分中，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在更广泛的 GPU 中。但是，如果你希望的话，你一般能够把一个逐顶点的计算移植到片段程序中。

In the rest of this chapter, and the remainder of this book, we avoid complex fragment programs where possible, to make the examples accessible to a broad range of GPU generations. But you can usually move a per-vertex computation to the fragment program if you want to.

5.4 创建一个光照函数

（5.4 Creating a Lighting Function）

在前面的小节中，我们只是简单得从逐顶点示例中拷贝了大部分代码在逐片段的实例中，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中。

In the preceding section, we simply copied most of the code from the per-vertex example to the per-fragment example, but there's a better solution: encapsulating the key aspects of lighting in a function.

在一个复杂的 Cg 程序中，光照也学知识程序多执行的几个计算中的一个。在逐顶点的光照例子中，你看到了计算光照所需的各个步骤。但是，无论何时你计算光照的时候，你都不会想要重写这些所有的代码。幸运的事，你能够编写一个内部函数来封装光照功能，然后可以在同的入口函数中重用相同的光照函数。

In a complex Cg program, lighting might be only one of several computations that the program performs. In the per-vertex lighting example, you saw the various steps that were necessary to calculate lighting. But you would not want to rewrite all this code whenever you wanted to compute lighting. Fortunately, you don't have to. As we mentioned in C hapter 2, you can write an internal function that encapsulates the lighting task and reuse the same lighting function in different entry functions.

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关联函数的调用开销。因此，当有可能的时候你应该使用函数，因为他们能够提高可读性、简化调试、鼓励重用，并且能够使得未来的优化更加容易。

Unlike functions in C or C ++, functions in C g are typically inlined (though this may depend on the profile— advanced profiles such as vp30 can support function calls in addition to inlining). Inlining functions means that they have no associated function-call overhead. Therefore, you should use functions whenever possible, because they improve readability, simplify debugging, encourage reuse, and make future optimization easier.

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。

Cg, like C , requires that you declare functions before using them.

5.4.1 声明一个函数

（**5.4.1 Declaring a Function**）

在 Cg 中，函数就像 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：

In C g, functions are declared just as in C . You can optionally specify parameters to pass to the function, as well as values that will be returned by the function. Here is a simple function declaration:

```
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：

This function takes a three-component vector v as a parameter and returns a float that is the *x* component of v . The return keyword is used to return the function's result. You call the getX function just as you would call any other C g function:

```
// Declare a scratch vector
float3 myVector = float3(0.5, 1.0, -1.0);
// Get the x component of myVector
float x = getX(myVector);
// Now x = 0.5
```

有些时候，你想要一个函数返回不仅仅是一个结果。在这种情况下，你需要使用 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，然后返回他的 x、y 和 z 分量:

Sometimes, you want a function to return several results instead of just one. In these situations, you can use the out modifier (as explained in Section 3.3.4) to specify that a particular parameter to a program be for output only. Here is an example that takes a vector and returns its *x, y,* and *z* components:

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

注意这些函数被声明为 void 理性，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 是如何被调用的：

Note that this function is declared void , because it returns all its values through parameters. Here is a code sample that shows how getComponents would be used:

```
// Declare a scratch vector
float3 myVector = float3(0.5, 1.0, -1.0);
// Declare scratch variables
float x, y, z;
// Get the x, y, and z components of myVector
getComponents(myVector, x, y, z);
// Now x = 0.5, y = 1.0, z = -1.0
```

5.4.2 一个光照函数

（**5.4.2 A Lighting Function**）

因为光照是一个复杂的过程，你能够编写许多不同类型的函数，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，并未他创建一个函数。下面是这个函数最基本的样子。

Because lighting is a complex process, you can write many different types of lighting functions, each of which can take different parameters. For now, take the Basic model that you implemented and create a function for it. Here is a first shot at this function:

```
float3 lighting(float3 Ke,
float3 Ka,
float3 Kd,
float3 Ks,
float shininess,
float3 lightPosition,
float3 lightColor,
float3 globalAmbient,
float3 P,
float3 N,
float3 eyePosition)
{
// Calculate lighting here
}
```

这个方法的一个主要问题是这个函数需要很多参数。把这些参数组成的“材质参数”和“光照参数”，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。

One major problem with this approach is that the function requires so many parameters. It would be far neater to group the parameters into "material parameters" and "light parameters," and then to pass each set as an individual variable. Fortunately, C g supports structures, which provide exactly this functionality.

5.4.3 结构

（**5.4.3 Structures**）

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，他封装了基于基本光照模型的某个材质的所有性质：

As we mentioned in Chapter 2, C g structures are declared the same way they are in C or C++. The struct keyword is used, followed by the list of structure members. Here is an example of a structure that encapsulates all the properties of a particular material based on the Basic lighting model:

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

结构的成员可以通过“,”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。

Structure members are accessed using the period operator. The following code snippet shows how to declare and access a structure:

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

你还能够创建第二个结构来保存光照性质：

You could create a second structure to hold light properties:

```
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：

Now, you can redefine the lighting function using structures as parameters:

```
float3 lighting(Material material,
Light light,
float3 globalAmbient,
float3 P,
float3 N,
float3 eyePosition)
{
// Calculate lighting here
}
```

使用这个方法能够在以后使你的光线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。

With this approach, you could later make your light or material model more complex, without having to add more parameters to the lighting function itself. Another advantage is that you can calculate the effects of multiple lights by using an array of Light structures instead of just one.

5.4.4 数组

（**5.4.4 Arrays**）

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：

Cg supports arrays much as C does. Because C g currently does not support pointers, you must always use array syntax rather than pointer syntax when dealing with arrays. Here is an example of an array declaration and access in C g:

```
// Declare a four-element array
float3 myArray[4];
int index = 2;
// Assign a vector to element 2 of the array
myArray[index] = float3(0.1, 0.2, 0.3);
```

Note

一个与 C 的重要区别是数组在 Cg 中是第一类数据类型（first-class type）。这意味着数组赋值实际上是拷贝了整个数组，而且作为参数传递的数组是通过值传递过来的（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的。

An important difference from C is that arrays are first-class types in C g. This means that array assignments actually copy the entire array, and arrays that are passed as parameters are passed by value (the entire array is copied before making any changes), rather than by reference.

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

You can also pass arrays as parameters to functions. We're going to use this feature to create a function that computes the lighting from two distinct light sources, as shown in Example 5-4.

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 循环来从这个两个光源累积满发射和镜面反射作用。这些作用通过使用 C5E5_computeLighting 辅助函数来计算，我们满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 程序流程的一些概念。

As you look over the code for C5E4v_twoLights , you will notice that it starts by calculating the emissive and ambient terms, which are independent of the light sources. The function then loops over the two light sources using a for loop, accumulating the diffuse and specular contributions from the lights. These contributions are calculated using the C5E5_computeLighting helper function, which we will define shortly. First, let's learn a little more about for loops and other constructs that control the flow of Cg programs.

5.4.5 流控制

（**5.4.5 Flow Control**）

Cg 提供了 C 的流程控制能力的一个子集。特别需要指出的事，Cg 支持：

1） 函数和 return 指令

2） if-else

3） for

4） while 和 do-while

Cg offers a subset of C 's flow-control capabilities. In particular, C g supports:

- functions and the return statement
- if - else
- for
- while and do - while

除了有一些 profile 指定的限制存在外，这些指令与他们在 C 中的对应指令完全相同。例如，一些 profile 只有在循环迭代次数能够实现被 Cg 编译器确定的情况下才允许 for 或者 while 循环。

These are identical to their C counterparts, except for profile-specific restrictions that may exist. For example, some profiles allow for or while loops only if the number of loop iterations can be determined ahead of time by the C g compiler.

Cg 保留了用于其他 C 的流程控制的更关键字，例如 goto 和 switch。但是，这些关键字到现在位置还未被支持。

Cg reserves keywords for other C flow-control constructs, such as goto and switch . However, these constructs are not currently supported.

Example 5-4. The C5E4v_twoLights **Vertex Program**

```
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++) {
C5E5_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;
}
```

5.4.6 计算漫反射和镜面反射光照

（**5.4.6 Computing the Diffuse and Specular Lighting**）

这个难题的最后一部分是 C5E5_computeLighting 函数，他负责为某个特定的光源计算漫反射和镜面反射效果。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。

The final piece of the puzzle is the C5E5_computeLighting function, which is responsible for calculating the diffuse and specular contributions for a particular light source. Example 5-5 re-implements the diffuse and specular lighting code that we wrote earlier.

Example 5-5. The C5E5_computeLighting **Internal Function**

```
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

（**5.5 Extending the Basic Model**）

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他变得更加有用。下面几个小节将介绍 3 中放大器：距离衰减、聚光灯效果和方向光。每种放大器都将同时在顶点阶段和片段阶段工作。

Now that you have implemented a Basic lighting model, let's look at how we can make it a little more useful. The following sections present three enhancements: distance attenuation, spotlight effects, and directional lights. Each enhancement will work at both the vertex level and the fragment level.

光照是一种非常复杂的问题，其中有许多技术需要探索。而我们的目标是使你入门。

Lighting is a very complicated topic, and there are a huge number of techniques to explore. Our goal is just to get you started.

5.5.1 距离衰减

（**5.5.1 Distance Attenuation**）

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前度总是一样的。虽然对某这些光源这是一个合理的近似（例如，当太阳光照射地球上物体的时候），我们通常想要创建一个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：

The Basic lighting model assumes that the intensity of the light source is always the same no matter how far the light is from the surface being shaded. Although that is a reasonable approximation for some light sources (such as when the Sun lights objects on Earth), we usually want to create lights for which intensity diminishes with distance. This property is called *distance attenuation*. In OpenGL or Direct3D, the attenuation at any particular point is modeled using the following formula:

$$attenuationFactor = \frac{1}{k_C + k_L d + k_Q d^2}$$

其中：

1） d 是到达光源的距离

2） K_C、K_L 和 K_Q 是控制衰减量的常量

where:

• *d* is the distance from the light source and

• k_C , k_L , and k_Q are constants that control the amount of attenuation.

在这个衰减公式中，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 t/d^2 衰减，但是有时候这可能不会是你获得想要的效果。使用这 3 个衰减参数的想法是你对场景光照的表现能够有更多的控制。

In this formulation of attenuation, k_C , k_L , and k_Q , respectively, are the constant, linear, and quadratic coefficients of attenuation. In the real world, the intensity of illumination from a point light source decays as $1/d_2$, but sometimes that might not give the effect you want. The idea behind having three attenuation parameters is to give you more control over the appearance of your scene's lighting.

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，整个公式变成了：

The attenuation factor modulates the diffuse and specular terms of the lighting equation. So the equation becomes:

lighting = *emissive* + *ambient* + *attenuationFactor* x (*diffuse* + *specular*)

示例 5-6 描述了在给定表面位置和 Light 结构（我们 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。

Example 5-6 describes a C g function that calculates attenuation, given the surface position and Light structure (to which we have now added kC , kL , and kQ).

Example 5-6. The C5E6_attenuation **Internal Function**

```
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，这个函数是另一个标准库函数之一。下面是 distance 的正式定义：

We take advantage of the distance function that is yet another one of C g's Standard Library functions. Here is the formal definition of distance :

distance(pt1, pt2) Euclidean distance between points pt1 and pt2

衰减计算需要加入到 C5E5_computeLighting 函数中，因为他同时影响了来自光源的漫反射和镜面反射作用。你应该首先计算衰减，这样你就能够用他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。

The attenuation calculation needs to be added to the C5E5_computeLighting function, because it affects both the diffuse contribution and the specular contribution from the light source. You should calculate the attenuation first, so that you can use it to modulate the diffuse and specular contributions. The C5E7_attenuateLighting internal function in Example 5-7 shows the necessary modifications.

Example 5-7. The C5E7_attenuateLighting **Internal Function**

```
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

（**5.5.2 Adding a Spotlight Effect**）

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。只有聚光灯圆锥体内的物体才能受到光照。

Another commonly used extension for the Basic lighting model is making the light a spotlight instead of an unidirectional light. A spotlight cut-off angle controls the spread of the spotlight cone, as shown in Figure 5-18. Only objects within the spotlight cone receive light.

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），如图 5-19 所示的那样。

To create the spotlight cone, you need to know the spotlight position, spotlight direction, and position of the point that you are trying to shade. With this information, you can compute the vectors *V* (the vector from the spotlight to the vertex) and *D* (the direction of the spotlight), as shown in Figure 5-19.

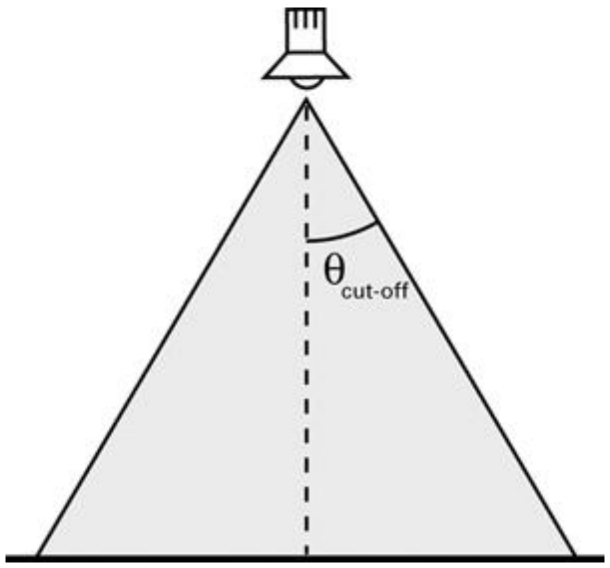


Figure 5-18 Specifying a Spotlight Cut-Off Angle

通过计算这两个规格化向量的点积，你能够得到他们之间的夹角的余弦。然后用他来找出 P 是否在聚光灯圆锥体内。P 只有当 dot(V, D)大于聚光灯取舍角的余弦值的时才受到聚光灯的影响。

By taking the dot product of the two normalized vectors, you can find the cosine of the angle between them, and use that to find out if *P* lies within the spotlight cone. *P* is affected by the spotlight only if dot(V, D) is greater than the cosine of the spotlight's cut-off angle.

基于这个数学原理，我们能够为聚光计算创建一个函数。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——假设已经被规格化了）和 cosLightAngle

（聚光灯取舍角的余弦值）。

Based on this math, we can create a function for the spotlight calculation, as shown in Example 5-8. The function C5E8_spotlight returns 1 if *P* is within the spotlight cone, and 0 otherwise. Note that we have added direction (the spotlight direction—assumed to be normalized already) and cosLightAngle (the cosine of the spotlight's cut-off angle) to the Light structure from Example 5-6.

Example 5-8. The C5E8_spotlight **Internal Function**

```
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;
}
```

5.5.2.1 强度变化

（**Intensity Variation**）

直到现在为止，我们假设由聚光灯发出的光的强度在聚光灯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个部分：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的逐渐减少，如图 5-20 所示，这个通用的方法近似的创造出各家复杂的方法。在图 5-21 的右图演示了这种效果。

So far, we have assumed that the intensity of light given off by the spotlight is uniform within the spotlight's cone. Rarely, if ever, is a real spotlight so uniformly focused. To make things more interesting, we will divide the cone into two parts: an inner cone and an outer cone. The inner cone (or "hotspot") emits a constant intensity, and this intensity drops off smoothly outside the cone, as shown in Figure 5-20. This commonly used approach creates the more sophisticated effect demonstrated in the right side of Figure 5-21.

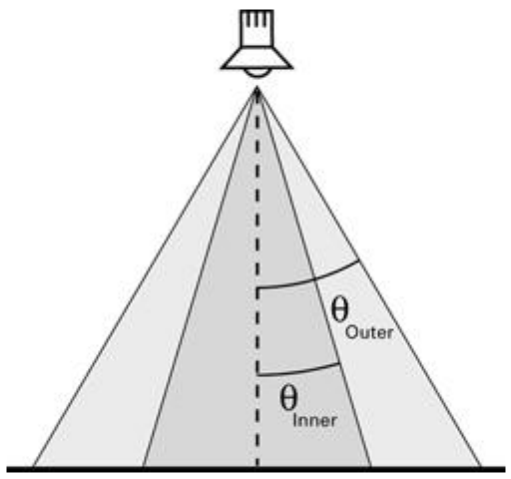


Figure 5-20 Specifying Inner and Outer C ones for a Spotlight

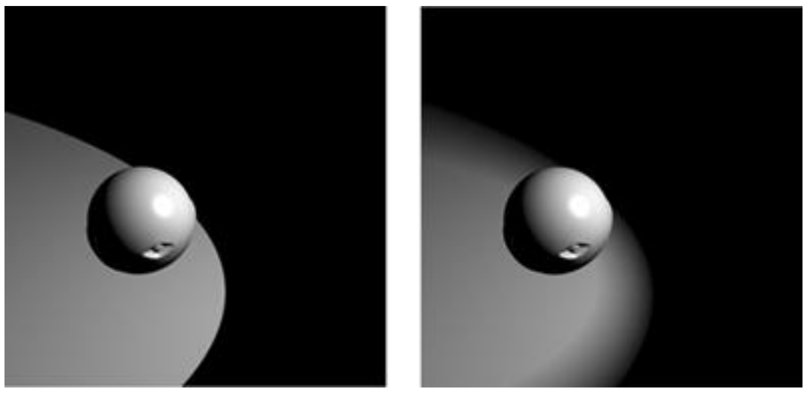


Figure 5-21 The Effect of Adding Inner and Outer Spotlight Cones

找出一个特定的店 P 是在内部圆锥中还是在外部圆锥中并不是很难。与基本聚光灯的唯一区别是，你需要根据 P 所在的圆锥体来变化光强度的计算。

It's not hard to find out if a particular point *P* is in the inner cone or the outer cone. The only difference from the basic spotlight is that you now need to vary the intensity calculation based on which cone *P* lies.

如果 P 在内部圆锥中，你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 lcrp 函数非常适合这种类型的国度，但是使用高级的 profile 你能够做得更好。

If *P* lies in the inner cone, it receives the spotlight's full intensity. If *P* lies in between the inner and outer cones, you need to lower the intensity gradually based on how far *P* is from the inner cone. C g's lerp function works well for this type of transition, but with advanced profiles, you can do better.

Cg 有一个 smoothstep 函数可以在视觉上生成一个比简单的 lerp 更加吸引人的效果。不幸的事，smoothstep 函数也许在一些基本 profile 上无法工作，因为他们功能被限制。我们将在这个例子中使用 smoothstep，虽然你能够用你选择得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进行插值。

Cg has a smoothstep function that produces more visually appealing results than a simple lerp . Unfortunately, the smoothstep function might not work with some basic profiles, because of their limited capabilities. We'll use smoothstep in this example, though you could replace it with another function of your choice.

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的时候，可以使用 smoothstep。smoothstep 的另外一个方便的特点是它把值映射到[0，1]的范围内。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 在外部圆锥的时候返回 0.0.

Figure 5-22 graphs what the smoothstep function looks like. Use smoothstep when you want to create a visually pleasing transition between two values. Another convenient feature of smoothstep is that it clamps to the [0, 1] range. If you set up your parameters to smoothstep correctly, it will return 1.0 when *P* is in the inner cone, and 0.0 when *P* is outside the outer cone.

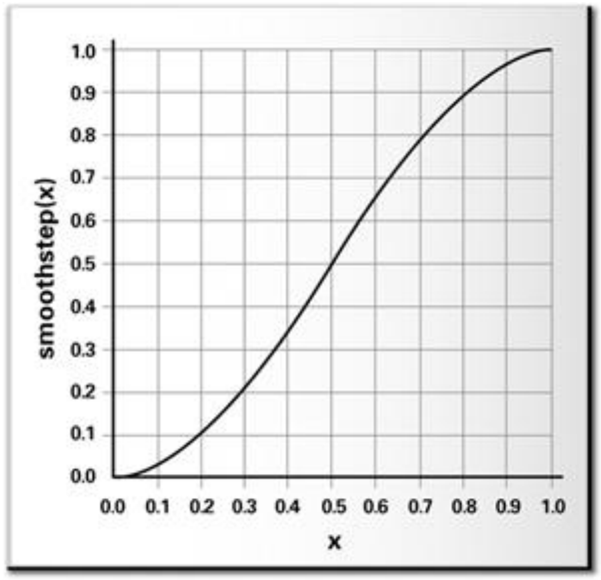


Figure 5-22 A Graph of the Function

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代替了。

Once again, we can extend the Light structure to include the new spotlight parameters. The single cut-off angle is now replaced by an inner angle cosine and an outer angle cosine.

下面是 Light 结构进一步更新的版本：

Here is how our further updated version of the Light structure looks:

```
struct Light {
float4 position;
float3 color;
float kC;
float kL;
float kQ;
float3 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。

The C5E9_dualConeSpotlight internal function shown in Example 5-9 combines all this to create a spotlight with a hotspot.

```
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结合了衰减和聚光灯项。

The C5E10_spotAttenLighting internal function combines both the attenuation and the spotlight terms with specular and diffuse lighting, as shown in Example 5-10.

Example 5-10. The C5E10_spotAttenLighting Internal Function

```
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

（**5.5.3 Directional Lights**）

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，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。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觉察结果的计算。

Although computations such as the spotlight effect and attenuation add to the visual complexity of your scenes, the results are not always noticeable. C onsider rays from the Sun that light objects on Earth. All the rays seem to come from the same direction because the Sun is so far away. In such a situation, it does not make sense to calculate a spotlight effect or to add attenuation, because all objects receive essentially the same amount of light. A light with this property is called a *directional light*. Directional lights do not exist in reality, but in computer graphics, it is often worthwhile to identify situations in which a directional light is sufficient. This allows you to avoid computations that will not produce perceptible results.

第 6 章 动画

（Chapter 6. Animation）

本章将描述使用 Cg 驱动物体的各种方法。它包括一下 5 个部分：

1） “随时间运动”介绍了动画的基本概念。

2） “一个有规律搏动的物体”显示了一个顶点程序，可以周期性地朝着它的法向量的方向移动一个物体的表面

3） “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，用到的所有就说那都是由 GPU 完成的。

4） “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子条件插值来使物体运动。

5） “顶点混合”解释了在任务动画中如何给予多权重控制矩阵移动顶点来实现更多的动态控制。

This chapter describes various ways to animate objects using C g. It has the following five sections:

• **"Movement in Time"** introduces the concept of animation.

• **"A Pulsating Object"** shows a vertex program that periodically displaces an object's surface outward, in the direction of its normal vectors.

• **"Particle Systems"** describes how to use a physically based simulation to create a particle system, with all the calculations done by the GPU.

• **"Key-Frame Interpolation"** explains key-frame animation, in which a vertex program animates an object by interpolating between different object poses.

• **"Vertex Skinning"** explains how to displace vertices based on multiple weighted control matrices for more dynamic control in character animation.

6.1 随时间运动

（6.1 Movement in Time）

动画是随时间发生的一个动作的结果——例如，一个物体不停的波动，一个光的淡出，或一个跑动的任务。你的应用程序可以使用以 Cg 编写的顶点程序来实现这些类型的动画。动画的来源是在你应用程序中的一个或多个随时间变化的程序参数。

Animation is the result of an action that happens over time—for example, an object that pulsates, a light that fades, or a character that runs. Your application can create these types of animation using vertex programs written in C g. The source of the animation is one or more program parameters that vary with the passing of time in your application.

为了创建动画的渲染，你的应用程序必须在 Cg 以上甚至在 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。这个变量将随着时间的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。

To create animated rendering, your application must keep track of time at a level above Cg and even above OpenGL or Direct3D. Applications typically represent time with a global variable that is regularly incremented as your application's sense of time advances. Applications then update other variables as a function of time.

你可以在 CPU 上计算动画更新，然后把这些动画数据传给 GPU。但是，更有效的方法是用一个顶点程序在 GPU 上执行尽可能多的动画计算。而不需要 CPU 来做所有的数字处理。把动画工作从 CPU 上卸载下来可以帮助平衡 CPU 和 GPU 的资源，可以释放出 CPU 来进行其他计算，例如碰撞检测，人工智能和博弈等。

You could compute animation updates on the CPU and pass the animated data to the GPU. However, a more efficient approach is to perform as much of the animation computation as possible on the GPU with a vertex program, rather than require the C PU to do all the number-crunching. Offloading animation work from the CPU can help balance the C PU and GPU resources and free up the C PU for more involved computations, such as collision detection, artificial intelligence, and game play.

6.2 一个有规律波动的物体

（**6.2 A Pulsating Object**）

在第一个示例中，你将学习如何使得一个物体周期性地变形，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参数，然后基于时间修改集合物体的顶点位置。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。

In this first example, you will learn how to make an object deform periodically so that it appears to bulge. The goal is to take a time parameter as input and then modify the vertex positions of the object geometry based on the time. More specifically, you need to displace the surface position in the direction of the surface normal, as shown in Figure 6-1.

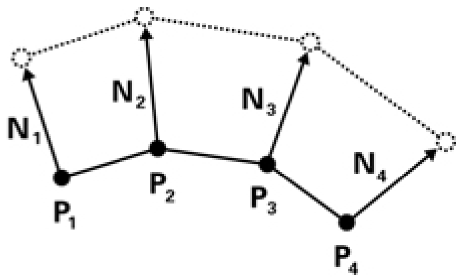


Figure 6-1 Making an Object Bulge

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。这种搏动动画在一个顶点程序里

By varying the magnitude of the displacement over time, you create a bulging or pulsing effect. Figure 6-2 shows renderings of this effect as it is applied to a character. The pulsating animation takes place within a vertex program.

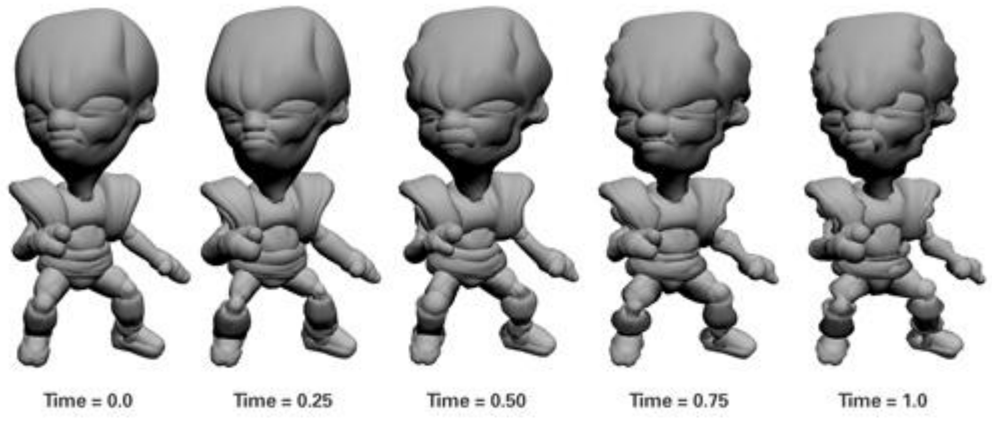


Figure 6-2 A Pulsating Alien

6.2.1 顶点程序

（**6.2.1 The Vertex Program**）

示例 6-1 显示了 C6E1v_bulge 顶点程序的全部源代码，他计划于第 2 章的 C2E2f_passthrough 片段程序一起使用。而膨胀效果真正需要的是顶点位置和法向量。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 的辅助函数只是计算漫反射和镜面光照效果（为了简单镜面反射材质被假定为白色）。

Example 6-1 shows the complete source code for the C6E1v_bulge vertex program, which is intended to be used with the C2E2f_passthrough fragment program from Chapter 2. Only the vertex position and normal are really needed for the bulging effect. However, lighting makes the effect look more interesting, so we have included material and light information as well. A helper function called computeLighting calculates just the diffuse and specular lighting (the specular material is assumed to be white for simplicity).

Example 6-1. The C6E1v_bulge **Vertex Program**

```
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

（**6.2.2 Displacement Calculation**）

6.2.2.1 创建一个基于时间的函数

（**Creating a Time-Based Function**）

这个想法是计算一个被称为 displacement 的变量，他沿表面法向量的方向向上或向下移动顶点的位置。要事程序的效果运动起来，displacement 必须随着时间改变。你可以为他选择任

何你喜欢的方式进行变换。例如，你可以选择如下的方式。

The idea here is to calculate a quantity called displacement that moves the vertex position up or down in the direction of the surface normal. To animate the program's effect, displacement has to change over time. You can choose any function you like for this. For example, you could pick something like this:

```
float displacement = time;
```

当然，这个行为没有什么意义，因为 displacement 总是在增加，使得物体随着时间的增长而变得越来越大。我们其实想要一种搏动的效果，使物体在膨胀变大和返回他原来正常的形状之间振荡。正弦函数提供了这样一种平滑的振荡行为。

Of course, this behavior doesn't make a lot of sense, because displacement would always increase, causing the object to get larger and larger endlessly over time. Instead, we want a pulsating effect in which the object oscillates between bulging larger and returning to its normal shape. The sine function provides such a smoothly oscillating behavior.

正弦函数的一个有用的性质是他结果总是在-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给一个更方便的范围，例如 0 到 1。

A useful property of the sine function is that its result is always between –1 and 1. In some cases, such as in this example, you don't want any negative numbers, so you can scale and bias the results into a more convenient range, such as from 0 to 1:

```
float displacement = 0.5 * (sin(time) + 1);
```

Note
你是否知道正弦函数在 CineFX 架构中与加法和乘法一样有效？实际上，计算余弦的的函数也一样快。利用这些性质可以给你的程序增加视觉的复杂度而不减慢他们的执行效率。

Did you know that the sin function is just as efficient as addition or multiplication in the C ineFX architecture? In fact, the cos function, which calculates the cosine function, is equally fast. Take advantage of these features to add visual complexity to your programs without slowing down their execution.

6.2.2.2 为程序增加控制

（**Adding Controls to the Program**）

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：

The idea here is to calculate a quantity called displacement that moves the vertex position up or down in the direction of the surface normal. To animate the program's effect, displacement has to change over time. You can choose any function you like for this. For example, you could pick something like this:

```
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，我们可以得到：

You may also want to control the amplitude of the bulging, so it's useful to have a uniform parameter for that as well. Throwing that factor into the mix, here's what we get:

```
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。或者，你提供附加的统一参数来指明人物的呼吸的频率。经过一段时间之后，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，并且能够帮助玩家增加游戏的沉浸感。

As it is now, this equation produces the same amount of protrusion all over the model. You might use it to show a character catching his breath after a long chase. To do this, you would apply the program to the character's chest. Alternatively, you could provide additional uniform parameters to indicate how rapidly the character is breathing, so that over time, the breathing could return to normal. These animation effects are inexpensive to implement in a game, and they help to immerse players in the game's universe.

6.2.2.3 改变膨胀的大小

（**Varying the Magnitude of Bulging**）

但是如果你想要膨胀的大小在模型上不同的部位不一样怎么办呢？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。在这里我们想你显示了一个更加简单的,基于顶点位置的方法来给膨胀增加一些变化:

But what if you want the magnitude of bulging to vary at different locations on the model? To do this, you have to add a dependency on a per-vertex varying parameter. One idea might be to pass in scaleFactor as a varying parameter, rather than as a uniform parameter. Here we show you an even easier way to add some variation to the pulsing, based on the vertex position:

```
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。

This code uses the *y* coordinate of the position to vary the bulging, but you could use a combination of coordinates, if you prefer. It all depends on the type of effect you are after.

6.2.2.4 更新顶点位置

（**Updating the Vertex Position**）

在我们的例子中，我们先用位置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：

In our example, the displacement scales the object-space surface normal. Then, by adding the result to the object-space vertex position, you get a displaced object-space vertex position:

```
float4 displacementDirection = float4(normal.x, normal.y, normal.z, 0);
float4 newPosition = position + displacement * displacementDirection;
```

6.2.2.5 当可能的时候预先算统一参数

（**Precompute Uniform Parameters When Possible**）

前面的例子证明了很重要的一点。再仔细看看下面这行来自示例 6-1 的代码：

The preceding example demonstrates an important point. Take another look at this line of code from Example 6-1:

```
float displacement = scaleFactor * 0.5 * sin(position.y * frequency * time) + 1;
```

如果你想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，因为他们都依赖于统一数。这意味着你将在 GPU 上为每个顶点计算这个唯一，但实际上你能够在 CPU 上为整个网格值只计算位移一次，然后把位移当作一个统一参数传给顶点程序。但是，当顶点的位置是唯一公式的一部分时，sine 函数必须为每个顶点求值。并且如你所预期的那样，如果唯一的值像这样在每个顶点都发生变化,类似这样的一个逐顶点计算在 GPU 上执行就要比在 CPU 上执行效率高很多。

If you were to use this equation for the displacement, all the terms would be the same for each vertex, because they all depend only on uniform parameters. This means that you would be computing this displacement on the GPU for *each vertex,* when in fact you could simply calculate the displacement on the CPU just once for the entire mesh and pass the displacement as a uniform parameter. However, when the vertex position is part of the displacement equation, the sine function must be evaluated for each vertex. And as you might expect, if the value of the displacement varies for every vertex like this, such a per-vertex computation can be performed far more efficiently on the GPU than on the CPU.

Note

如果一个被计算的值对整个物体是一个常数，你可以通过用 CPU 在每个物体的基础上余弦计算这个值来优化你的程序。然后把整个预先计算的值当做一个统一参数传递给 Cg 程序。这个方法比为每个被处理的片段或定点单独计算这个值要有效的多。

If a computed value is a constant value for an entire object, optimize your program by precomputing that value on a per-object basis with the C PU. Then pass the precomputed value to your C g program as a uniform parameter. This approach is more efficient than recomputing the value for every fragment or vertex processed.

6.3 粒子系统

（**6.3 Particle Systems**）

有时候，你需要把每个顶点当作一个小物体或者粒子（particle），替代网格中的顶点动画。根据某个特定运动规律的大量的粒子被称为一个粒子系统。下面这个和示例在一个顶顶啊程序中实现了一个简单的粒子系统。现在，我们只关心这个系统是如何工作的，而不必担心它简单的外表。图 6-3 显示了饿一个随时间变化的粒子系统。

Sometimes, instead of animating vertices in a mesh, you want to treat each vertex as a small object, or *particle*. A collection of particles that behave according to specific rules is known as a *particle system*. This example implements a simple particle system in a vertex program. For now, focus on how the system works; don't worry about its simplistic appearance. At the end of this section, we will mention one easy method to enhance your particle system's appearance. Figure 6-3 shows the particle system example progressing in time.

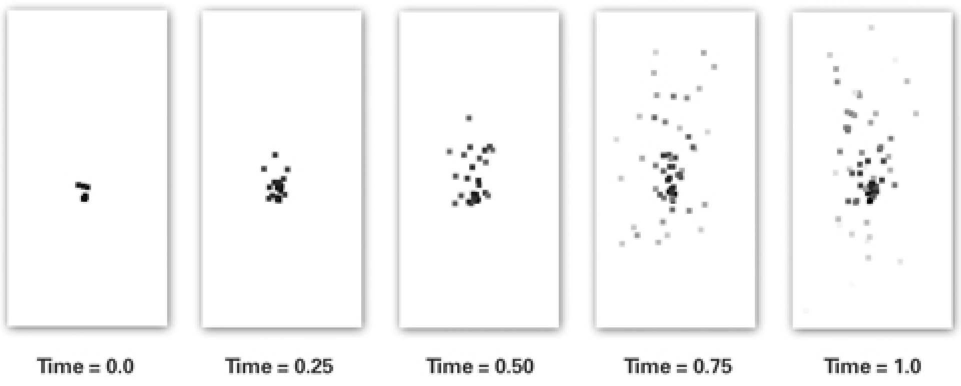


Figure 6-3 A Particle System

这个示例的粒子系统根据一个来自物理学最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例子的 x、y 和 z 的位置。你将使用如 6-1 所示的这个简单公式。

The example particle system behaves according to a simple vector kinematic equation from physics. The equation gives the *x, y,* and *z* positions of each particle for any time. The basic equation from which you will start is shown in Equation 6-1.

Equation 6-1 Particle Trajectory

$$p_{final} = p_{initial} + vt + \frac{1}{2}at^2$$

其中

1) P_{final} 是粒子的最终位置。
2) $P_{initial}$ 是粒子的初始位置。
3) v 是粒子的初始速度。
4) a 是粒子的加速度。
5) t 是所经过的时间。

where:

- p_{final} is the particle's final position,
- $p_{initial}$ is the particle's initial position,

- *v* is the particle's initial velocity,
- *a* is the particle's acceleration, and
- *t* is the time taken.

这个公式建立了在初始速度和重力影响下的、没有与其他粒子发生相互作用的粒子轨道集合。加入你提供了一个粒子的初始位置、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。

The equation models the trajectory of a particle set in initial motion and under the influence of gravity, but not otherwise interacting with other particles. This equation gives the position of a particle for any value of time, assuming that you provide its initial position, initial velocity, and constant acceleration, such as gravity.

6.3.1 初始化条件

（**6.3.1 Initial Conditions**）

应用程序必须用可变参数的形式为每个粒子提供初速度和初始位置。这两个参数被称为初始化条件，因为它们描述了粒子在模拟开始时的情况。

The application must supply the initial position and initial velocity of each particle as varying parameters. These two parameter values are known as *initial conditions* because they describe the particle at the beginning of the simulation.

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，重力加速度可以是一个统一参数。

In this particular simulation, the acceleration due to gravity is the same for every particle. Therefore, gravity is a uniform parameter.

为了使这个模拟更加精确，你能够增加一些效果例如拖拽甚至是旋转——我们把这个作为练习留给你。

To make the simulation more accurate, you could factor in effects such as drag and even spin—we leave that as an exercise for you.

6.3.2 向量化计算

（**6.3.2 Vectorized Computations**）

现代 GPU 拥有足够强大的向量处理能力，特别是对加法和乘法，他们特别适合处理做多到四元的向量。因此，使用向量计算与使用标量计算通常是一样有效的。

Modern GPUs have powerful vector-processing capabilities, particularly for addition and multiplication; they are well suited for processing vectors with up to four components. Therefore, it is often just as efficient to work with such vector quantities as it is to work with scalar (single-component) quantities.

公式 6-1 是一个向量公式，因为初始位置、初速度、常熟加速度，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后，你能够协助编译器把你的程序翻译成能够在你的 GPU 上有效执行的形式。

Equation 6-1 is a vector equation because the initial position, initial velocity, constant acceleration, and computed position are all three-component vectors. By implementing the particle system equation as a vector expression when writing the C g vertex program, you help the compiler translate your program to a form that executes efficiently on your GPU.

Note
尽可能是你的计算向量化，来充分利用 GPU 强大的向量处理能力。

Vectorized your calculations whenever possible, to take full advantage of the GPU's powerful vector-processing capabilities.

6.3.3 粒子系统的参数

（6.3.3 The Particle System Parameters）

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。

Table 6-1 lists the variables used by the vertex program presented in the next section.

Table 6-1. Variables in the Particle Equation

Variable	**Type**	**Description**	**Source (Type)**
pInitial	float3	Initial position	Application (Varying)
vInitial	float3	Initial velocity	Application (Varying)
tInitial	float3	Time at which particle was created	Application (Varying)
acceleration	float3	Acceleration (0.0, -9.8, 0.0)	Application (Uniform)
globalTime	float	Global time	Application (Uniform)
pFinal	float3	Current position	Internal
t	float	Relative time	Internal

除了在顶点程序内部计算的相对时间（t）和最后位置（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 米/平方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的空间向量。

Each variable is a parameter to the vertex program, except for the relative time (t) and final position (pFinal), which are calculated inside the vertex program. Note that the *y* component of the acceleration is negative—because gravity acts downward, in the negative *y* direction. The constant 9.8 meters per second squared is the acceleration of gravity on Earth. The initial position, initial velocity, and uniform acceleration are object-space vectors.

6.3.4 顶点程序

（**6.3.4 The Vertex Program**）

示例 6-2 为 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。

Example 6-2 shows the source code for the C6E2v_particle vertex program. This program is meant to work in conjunction with the C2E2f_passthrough fragment program.

Example 6-2. The C6E2v_particle **Vertex Program**

```
void C6E2v_particle(float4 pInitial : POSITION,
float4 vInitial : TEXCOORD0,
float tInitial : TEXCOORD1,
out float4 oPosition : POSITION,
out float4 color : COLOR,
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;
}
```

6.3.4.1 计算粒子的位置

（**Computing the Particle Positions**）

在该程序中，应用程序记录了一个全局时间并把它当做统一参数 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。该粒子的创建时间使用可变参数 tInitial 传递给顶点程序。想要知道一个例子已经活动了多久，你只需要用 tInitial 减去 globalTime:

In this program, the application keeps track of a "global time" and passes it to the vertex program as the uniform parameter globalTime . The global time starts at zero when the application initializes and is continuously incremented. As each particle is created, the particle's time of creation is passed to the vertex program as the varying parameter tInitial . To find out how long a particle has been active, you simply have to subtract tInitial from globalTime :

```
float t = globalTime - tInitial;
```

现在，你可以把 t 放到公式 6-1 中来计算粒子的当前位置：

This position is in object space, so it needs to be transformed into clip space, as usual:

```
float4 pFinal = pInitial + vInitial * t + 0.5 * acceleration * t * t;
```

6.3.4.2 计算粒子的颜色

（**Computing the Particle Color**）

在这个例子中，时间控制了粒子的颜色。

In this example, time controls the particle color:

```
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增加。注意颜色增加到纯白色(1, 1, 1, 1)就饱和了。你可以尝试自己的选择，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。

This is a simple idea, but it produces an interesting visual variation. The color increases with time linearly. Note that colors saturate to pure white (1, 1, 1, 1). You can try your own alternatives, such as varying the color based on the particle's position, or varying the color based on a combination of position and time.

6.3.4.3 计算粒子的大小

（**Computing the Particle Size**）

C6E2v_particle 使用了一个新的被命名为 PSIZE 的顶点程序输出语义词。当你渲染一个点到屏幕上的时候，一个带有该语义词的输出指定了这个点每像素所表示的真是的宽度（和高度）。这是你的顶点程序能够程序化控制光栅器使用的点的大小。

C6E2v_particle uses a new vertex program output semantic called PSIZE . When you render a point to the screen, an output parameter with this semantic specifies the width (and height) of the point in pixels. This gives your vertex program programmatic control of the point size used by the rasterizer.

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那样的效果。作为一个额外的修士，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得大一点。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：

The point size of each particle varies as time passes. The particles start out small, increase in size, and then gradually shrink. This variation adds to the fireworks-like effect. As an extra touch, we added a slight dependence on the particles' height, so that they get a little larger on their way up. To accomplish all this, we use the following function for the point size:

```
pointSize = -8.0 * t * t + 8.0 * t + 0.1 * pFinal.y + 1;
```

图 6-4 显示了这个函数的曲线

Figure 6-4 shows what the function looks like.

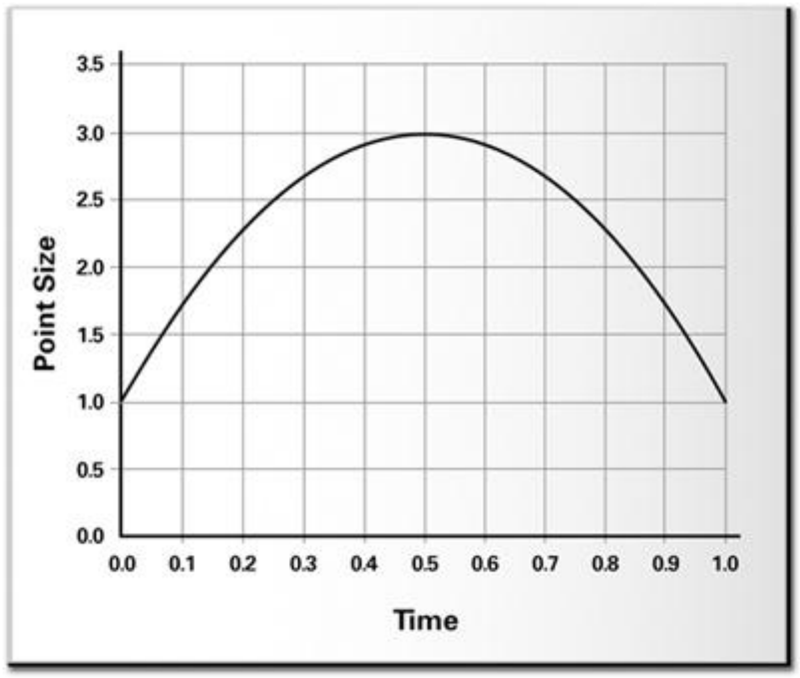


Figure 6-4 A Point Size Function

这个函数没有什么特殊的地方——我们只不过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。

This function is nothing special—we merely created the formula to achieve the effect that we wanted. In other words, the formula does not have any real physical meaning, aside from attempting to mimic the effect we had in mind.

6.3.5 修饰你的粒子系统

（6.3.5 Dressing Up Your Particle System）

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正方形而已。

Although the C6E2v_particle program produces interesting particle motion, the particles themselves do not look very appealing—they are just solid-colored squares of different sizes.

但是，你能够通过使用粒子图来改善粒子的外表。通过使用粒子图，硬件能够取得每个被渲染的点，并画出一个如图 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，这使得你能够得到从一个正方形到任何你想要的纹理图像来改变例子的外表。

However, you can improve the particle appearance by using *point sprites*. With point sprites, the hardware takes each rendered point and, instead of drawing it as a single vertex, draws it as a square made up of four vertices, as shown in Figure 6-5. Point sprites are automatically assigned texture coordinates for each corner vertex. This allows you to alter the appearance of the particles from a square to any texture image you want.

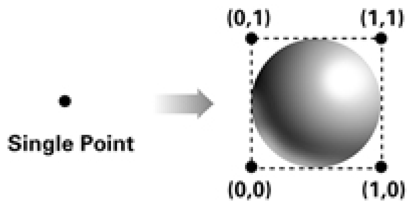


Figure 6-5 Converting Points to Point Sprites

通过把点当作例子图来渲染，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几何复杂性的隐形，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，OpenGL 和 Direct3D 都有渲染粒子图的标准接口。

By rendering the points as point sprites, you can use the assigned texture coordinates to sample a texture that supplies the shape and appearance of each point vertex, instead of simply rendering each point vertex as a square point. Point sprites can create the impression of added geometric complexity without actually drawing extra triangles. Figure 6-6 shows a more visually interesting example of a particle system, using point sprites. Both OpenGL and Direct3D have standard interfaces for rendering point sprites.

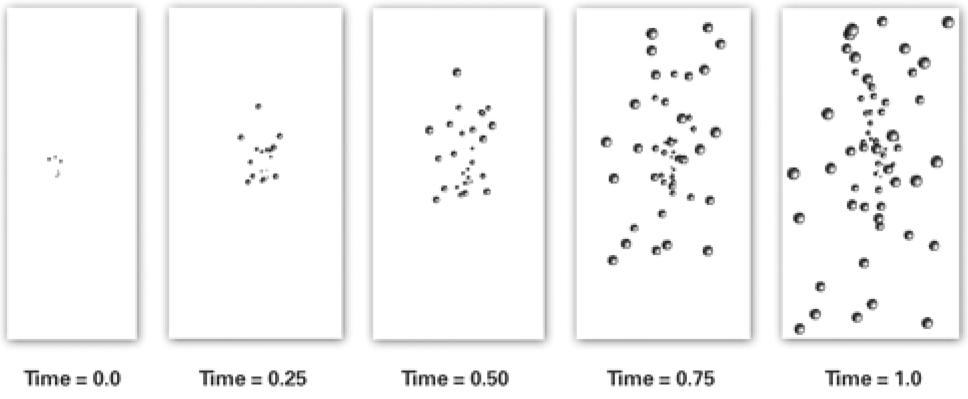


Figure 6-6 A Particle System with Point Sprites

6.4 关键帧插值
（**6.4 Key-Frame Interpolation**）

3D 游戏通常使用一系列的关键帧来表示一个动画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姿态称为一个关键帧。

3D games often use a sequence of key frames to represent an animated human or creature in various poses. For example, a creature may have animation sequences for standing, running, kneeling, ducking, attacking, and dying. Artists call each particular pose that they create for a given 3D model a key frame.

6.4.1 关键帧的背景知识
（**6.4.1 Key-Framing Background**）

关键帧这个术语来自卡通动画。为了生成一个卡通动画片，一名美工人员首先迅速地描述一个粗略的帧序列来动画化一个任务，而不是话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充当前后的参考。

The term *key frame* comes from cartoon animation. To produce a cartoon, an artist first quickly sketches a rough sequence of frames for animating a character. Rather than draw every frame required for the final animation, the artist draws only the important, or "key," frames. Later, the artist goes back and fills in the missing frames. These in-between frames are then easier to draw, because the prior and subsequent key frames serve as before-and-after references.