谁应该阅读这本书？

这本书的主题是 Shader 开发和管理，是为所有需要在应用程序中使用 shader 的开发人员而准备。如果你对 shader 感兴趣，并希望把它高效集成到程序中，那么你该阅读本书。由于本书既可以作为学习指南，同时也是参考手册，因此无论对普通爱好者还是专业开发者，都会有所帮助。

此外，本书的组织方式同样适合于作为学校在计算机科学中开展实时图形渲染技术教学的教科书来使用。

本书将讨论哪些内容？

The Complete Effect and HLSL Guide 的主题是 shader 开发和管理，这也是所有我们将要关注的内容。由于编写这本书的目的既是作为学习指南，同时也是参考手册，我将会覆盖 HLSL 语言和 effect framework 的大部分内容，但不会讨论关于特定 shader 和渲染技术。以下是本书将要讨论的内容摘要：

- 详细介绍 DirectX SDK 中的 HLSL 着色语言以及 effect framework。
- 深入讨论 HLSL 的语法和原理。
- 覆盖了 effect framework 中所有主要的组件，以及如何运用和组织这些技术，来开发一个着色器管理框架。

技术支持

http://www.ParadoxalPress.com 包含了本书的所有技术支持。网站将会定期发布勘误和必要的更新。

如果你对本书由任何意见或问题，随时欢迎您联系我：Sebastien St-Laurent，sebastien.st.laurent@gmail.com

第一部分 HLSL 着色语言

欢迎阅读 The Complete Effect and HLSL Guide。正如书名所示，贯穿本书的所有章节，我们将探索 DirectX effect framework 和 HLSL（Hight-Level Shading language 高级着色语言）的世界。你将学习如何高效的使用 HLSL 着色语言。此外，我还将教你如何使用 effect framework。作为 DirectX SDK 的一部分，effect framework 帮助你把 shader 集成到应用程序中。无论开发 3D 应用程序或视频游戏，为了适应 shader 日益增加的复杂度，同时保证向后兼容性，effect framework 无疑是集成和管理 shader 的首选。

编写这本书背后的目的有两个。首先，它作为一本学习指南，将带领你深入了解 HLSL 和 effect framework。然而，更重要的是，本书还可以作为一本参考

手册！它包含了所有没有包含在 DirectX 文档中，但对于编写 shader 来说必不可少的信息。哦，让我们不要浪费时间了，快速进入主题吧。

本书的第一部分将着重讲解 HLSL 着色语言，包括它的语法，以及用法。第二部分则聚焦于 effect framework，展示使用 effect framework 管理应用程序中的 shader 是多么简单。好了，接下来我们就深入学习 HLSL 着色语言。

第一章 着色器和 HLSL 语言

过去几年来，shader 技术取得了巨大飞跃。这一章，我将讲解 HLSL 着色语言的基本语法（syntax）和用法。为了让学习过程变得容易一些，我把对 HLSL 语法的描述分散到了多个章节中，每一章讲述着色语言的几个特定部分。本章将介绍基本语法，并且对基础知识进行一些概括。接下来的几章则着重讨论特定部分。

在开始介绍 HLSL 语法之前，先花一点时间来了解学习本书需要做的准备，以及一些关于 HLSL 着色语言和 effect framework 的历史。

准备工作

虽然本书是关于 HLSL 和 effect framework 的，但并不会教你基本的 DirectX 和 Direct3D 知识。因此，最重要的准备条件就是你必须对基本的 DirectX API 有一些了解，此外还需要具备一些 3D 渲染技术的基础知识。即使你对 3D 图形一窍不通，以上两点也很容易学习。

除了知识方面的要求，下面还列出了将会用到的软件和硬件：

- DirectX 9.0 Summer 2004 Update SDK (包含在 CD 中)
- Windows 2000（with service pack 2）或者 Windows XP（家用或专业版）。
- 奔腾 3 或更高的处理器
- 最少 256MB 内存
- 高端的 3D 图形卡。虽然任何图形卡都可以，但如果想尝试 shader 编程的所有方面，最好使用支持 Shader Mode 2.0 或 3.0 的图形卡。
- 最新的显卡驱动程序

准备工作做好之后，就可以开始学习和开发 shader，利用 effect framework 了。现在你已经知道如何来使用本书。那么让我们了解一点关于 shader 和 effect framework 的历史。

一点点历史....

从 1995 年，3Dfx 发布第一块消费级的 3D 硬件加速图形卡开始，计算机图形技术和相关的硬件技术都取得了重大进展。虽然这类图形卡在渲染功能上有诸多限制，但为开发者打开了一片新的天地，终结了只能依靠软件解决方案进行渲染的时代。其结果是让实时 3D 图形和游戏都变得更加真实。

此后，接下来的几代硬件都在性能和功能方面有了重大突破。但是，由于受到硬件固定管线构架（fixed-pipeline architecture）的限制，仍然有很多约束，开发者被强制只能通过使用和改变渲染状态来控制渲染过程，获得最终的输出图形。

固定管线构架功能上的局限性，限制了开发者创建所需效果的能力。总的来说，它所产生的图形都不够真实。另一方面，用于电影 CG 渲染的高端软件渲染构架则发明了一些让渲染更加逼真的方法。Pixar Animation Studios 开发了一门称为 RenderMan 的着色语言。它的目的是让艺术家和开发者使用一门简单但强大的编程语言来完全控制渲染过程。RenderMan 可以创建出高质量的图形，从照片级的真实效果，到卡通风格的非真实渲染效果都可以实现。被广泛用于当今的电影中，包括著名的动画 Toy Story 和 A Bug’s Life。

随着处理器芯片制造技术的革新，和处理能力的增强，RenderMan 的思想逐渐影响并延伸到了消费级图形硬件。DirectX 8 的发布引入了顶点（vertex）和像素着色器（pixel shader）1.0 以及 1.1 版本。虽然这两个版本的着色模型灵活性不高，同时缺乏流程控制等一些功能。但是，这第一步，给予了艺术家和开发者长久以来所梦想的，创造夺目的、真实的图形的能力。消费级图形卡所生产的图形终于能和好莱坞电影工作室所渲染出的图形相比了。

接下来的几年间，图像硬件和 3D API 无论在功能和性能上都取得了巨大飞跃，甚至打破了摩尔定律中的技术进步速率。随着 DirectX 9.0 SDK 以及最新的一代图形卡的发布，比如 Nvidia 的 Geforce FX 系列和 ATI 的 Radeon 9800 系列，顶点和像素着色器发展到了 2.0 和 2.x 版本。以及随后的 3.x 版本。

注意：

摩尔定律是 1965 年，由戈登摩尔（Gordon Moore）——intel 的创建者之一，通过统计得出的结论：集成电路上可容纳的晶体管数目，约每隔一年便会增加一倍。他还预测在以后的几十年中仍然将是这样。至今为止，这条理论依然很正确。另外，由于晶体管数量与集成电路的性能有关，因此，摩尔定律也是硬件性能增长的预测的依据。

这些新的着色模型为实时图像程序开发者带来了前所未有的灵活性。然而，大部分 shader 都通过一种低级的，类似于汇编的语言来编写的。这意味着作为一名开发人员，你必须像多年前使用汇编语言的时代那样，自己管理寄存器，分配变量以及优化。此外，shader model 2.0 和 3.0 增加的复杂性让开发人员更加头疼，因为不同的图形卡寄存器数量不一样，甚至同样的指令执行结果也不一样。

为了简化 shader 开发，同时，给予硬件开发者更多的自由优化性能，微软在 DirectX 9.0 中引入了 High-Level Shading Language（HLSL）。这门语言和其他高级语言，比如 C 或 C++很类似，这样，开发者就能把注意力集中在 shader 所要实现的功能上，而不是把精力放在如何使用寄存器，或对某种硬件如何组合指令才能最优化之类的琐碎问题上。

在讲解 HLSL 能做什么，以及如何来使用它之前，先来看看不同的 shader 版本可以提供哪些功能。需要说明的是，在编写 shader 之前，需要知道硬件都有哪些功能（capable）。使用 HLSL 并不能消除特定硬件平台上的限制，但却可以把这些限制隐藏起来。

顶点和像素着色器管线以及 Capabilities

与随 DirectX 8.0 发布的顶点和像素着色器 1.0 和 1.1 版本相比，shader model 2.0 对语言进行了许多重要改进。由于最新的 DirectX 9.0 所使用的顶点和像素着色器版本为 2.0，同时，已经有大量支持 vertex 和 pixel 2.0 的显卡，所以本书主要讨论基于这一技术的 shader。

注意：

虽然在编写本书时，支持 shader model 3.0 的图形卡已经开始上市，但尚未普及。我们会讨论一些 shader model 3.0 的特性，但大部分例子都是基于 2.0 或更低版本的 shader 技术。

假设你已经有一定的 3D 和 shader 基础知识，我们来看看第二代着色语言和上一代技术相比有哪些比较重要的改变。

顶点着色器 2.0 和 2.x 相对于 1.x 的版本有如下改进：

- 支持整数和布尔数据类型，并分别有相应的设置指令。
- 增加了临时和常量寄存器的数量。
- 对程序所能包含的最大指令数进行了增加，给开发者以更多灵活性（标准所要求的最小指令数从 128 增加到了 256，某些硬件还能支持更多指令）。
- 添加了许多支持复杂运算的宏指令，比如 sine/cosine，absolute，以及 power 等。
- 支持流程控制语句，比如循环和条件测试。

下面列出的则是像素着色器 2.0 和 2.x 相对于 1.x 版本的改进：

- 支持扩展 32-bit 精度的浮点运算。
- 支持对寄存器元素的任意重组（swizzling）和遮罩（masking）。
- 增加了常量和临时寄存器的可用数量。
- 标准所允许的最小指令卡有明显增加。算术指令从 8 条增加到 64 条，同时还允许使用 32 条纹理指令。像素着色器 2.x 默认情况下甚至支持更多指令，允许硬件支持比标准最小要求多的指令数。

- 支持整数和布尔常量，循环计数器以及断言寄存器（predicate register）。
- 支持动态流程控制，包括循环和分支。
- Gradient instrctions allowing a shader to discover the dervate of any input register

通过这一系列强大的改进，如今，开发者可以自由发挥想象力，创造出令人吃惊的效果。到这里，我们应该学习一下两种着色器的构架，以便更好的了解数据是怎样在图形硬件上流动。

当渲染 3D 图形时，几何体信息通过 Direct3D 之类的渲染 API 传递给图形硬件。硬件一旦接收到这些信息，就为 mesh 中的每一个顶点调用顶点程序。图 1.1 描绘了顶点和像素着色器 2.0 标准实现的原理图。

从图 1.1 中可以看到，开发者通过 3D 渲染 API，以数据流的形式，为顶点着色器提供顶点数据。数据流中包含了正确渲染几何体所需的所有信息，包括顶点位置，颜色和纹理坐标等等。当这些信息传递进来时，将分别放到合适的输入寄存器 v0 到 v15 中，以便顶点着色程序使用。顶点程序还需要访问许多其他的寄存器，才能完成自己的工作。常量寄存器都是只读的，通常用来为 shader 储存静态数据，因此，必须预先设置好它们的值。顶点着色器 2.0 标准下，常量寄存器储存的都是矢量，可以保存浮点数，整数，以及布尔类型的值。需要注意，顶点着色器中所有的寄存器都把数据储存为包含 4 个分量的矢量，可以并行访问所有分量，也可以使用重组或遮罩分别访问某个分量。

在图 1.1 的右边部分，是临时寄存器，用来储存顶点着色器计算出的中间结果。显然，由于它们是临时性的，因此，可以对这些寄存器进行写入和读取操作。注意名称为 a0 到 aL 的寄存器，它们是循环时用来索引地址和追踪循环所用的计数寄存器（counter register）。记住，由于 HLSL 是一门高级的着色语言，你不需要关心寄存器是如何分配的。对开发者来说这个过程应该是透明的，并且只有在 shader 被最终编译为机器代码时才发生。

在访问了输入寄存器，临时寄存器和常量寄存器之后，顶点着色器程序才开始以开发者所希望的方式来处理和控制输入的顶点。处理过程结束之后，结果立即被输送到最终的输出寄存器中。其中

最重要的就是 oPos 寄存器，因为它包含了顶点最终被映射到屏幕上的位置。其它的寄存器则包含了颜色和最终的纹理坐标等信息。

顶点着色器的任务完成之后，所有数据就被传递给光栅器（rasterizer）。硬件的这一部分将决定每个多边形所覆盖的像素。它还完成一些其他的渲染工作，比如顶点信息插值和遮挡计算（occlusion）。通过遮挡计算，可以帮助硬件减少部分工作量。光栅器决定了像素所覆盖的位置之后，就对每个需要在屏幕上进行绘制的像素调用像素着色程序。图 1.2 描绘了像素着色器构架的原理图。

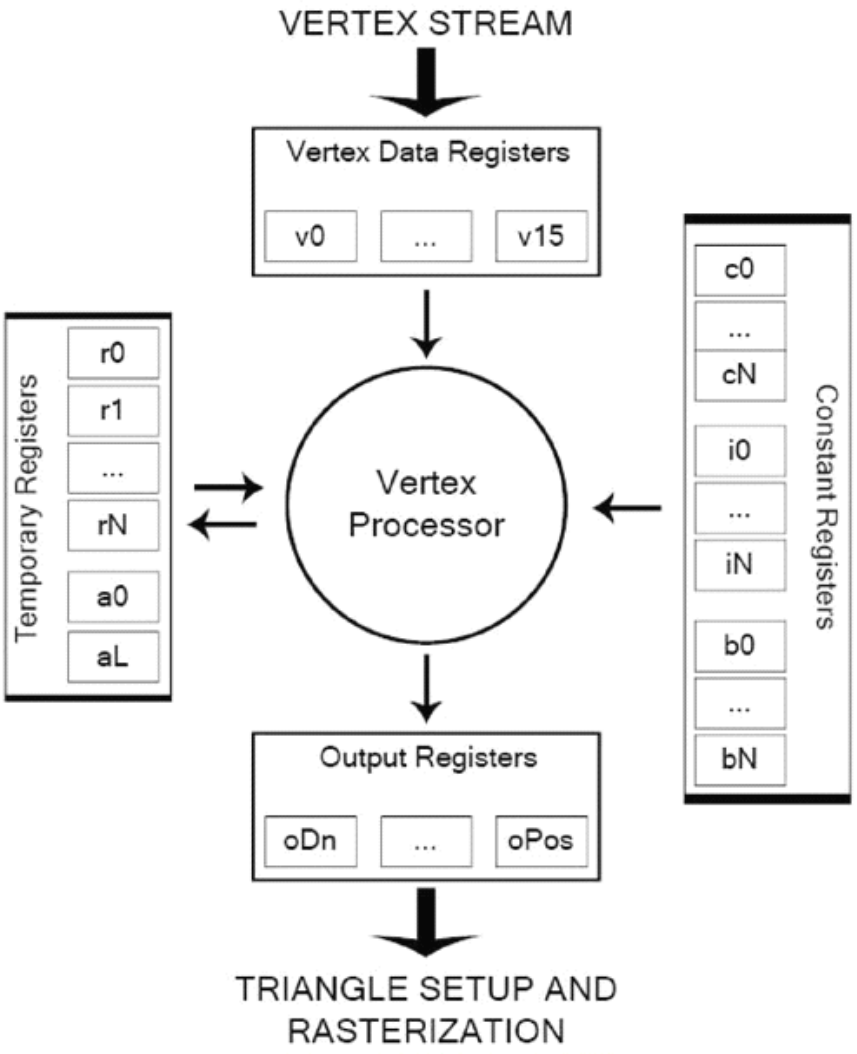


图 1.1 顶点着色器硬件构架原理图

在图 1.2 中可以看到，硬件把计算所得的像素输送到颜色和纹理寄存器中。这些值都是由之前定义在顶点着色器中的数据插值计算而来。寄存器 v0 和 v1 保存了插值过的漫反射和镜面反射元素。寄存器 t0 到 tN 储存了插值过的纹理坐标。此外，寄存器 s0 到 sN 指向在像素处理过程中像素着色器所要采样的纹理。虽然这些寄存器都有明确的语义（semantics），但同样可以储存来自于顶点着色器中的任何数据。

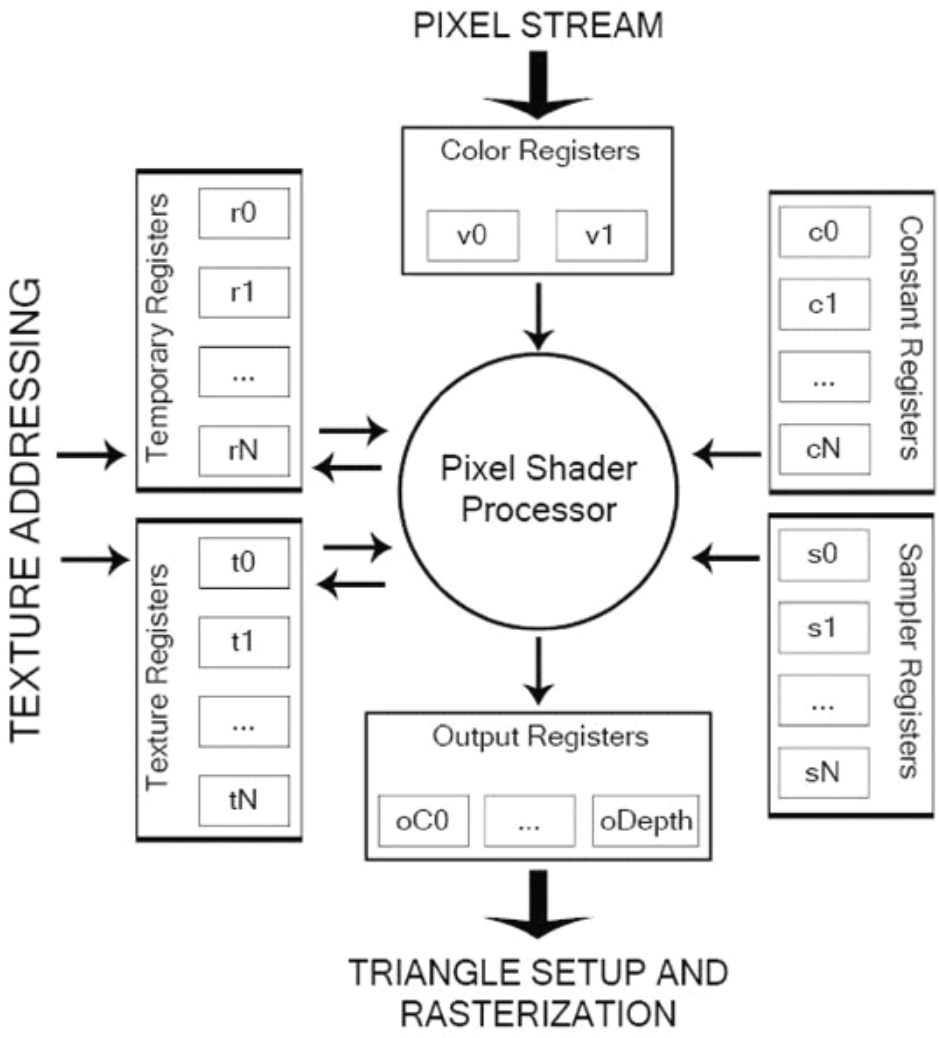
常量寄存器 c0 到 cN 同样是只读的，并且必须预先设置好它们的值。最后，临时寄存器 r0 到 rN

用来储存处理每个像素时所产生的中间值，它们是可读写的。当使用 HLSL 时，所有寄存器的使用方式和分配都是自动完成，并且对用户透明的。

除了某些寻址寄存器和循环计数器以外，像素着色器中绝大部分的寄存器都为包含四个浮点值分量的矢量寄存器。矢量构架的优点是可以同时处理多个值。默认情况下，所有浮点值都按照 32-bit 精度的浮点值来处理。然而，通过一些特别的指令，像素着色器规范也允许处理 16-bit 精度的值。对于某些特定的硬件实现来说，使用 16-bit 精度的浮点值来进行数学运算将会相当快。

在像素着色器处理完像素之后，输出数据在帧缓冲中混合为最终结果，之后便呈现到屏幕上。

需要说明的是，这仅仅是一个简化的渲染构架模型，它背后还有很多复杂的操作。这个构架也可能和某些硬件的具体实现有些细微差别；但是，遵循标准保证了不同的硬件实现将提供相同的功能，最终的输出结果也必须相同。



翻译：clayman
http://blog.csdn.net/soilwork
clayman_joe@yahoo.com.cn

高级着色语言

随着 shader model 2.0 和 3.0 复杂度的增加，对开发者来说，使用着色指令开发高效的 shader 就成了一项相当枯燥的工作。由于着色指令集和通用处理器上的指令集很类似，因此，开发一门高级语言来简化和抽象 shader 开发是很有意义的。于是高级着色语言（HLSL）诞生了。微软开发这门语言的目的就是对 shader 进行抽象，以便让开发者把精力放在如何设计 shader 等更重要的方面，而不是浪费在如何分配寄存器之类的琐碎细节上。

和其他的高级语言一样，HLSL 也由一系列语法，关键字和操作组成。这一部分，我将简要概括它的语法。但不要担心，之后的章节我们会详细讨论它的每一方面，并且给出示例。

关键字

任何编程语言都包含了一系列标识符和关键字，HLSL 也不例外。这意味着关键字定义了语言的某种特定含义，因此，你不能把它们用于其它用途。最重要的语言元素就是关键字。关键字和保留标识符一样，从呈现常量到控制代码行为，都他们由来表示。表 1-1 列出了 HLSL 的所有关键字，以及对它们的简短描述。

表 1-1 HLSL 关键字

关键字	定义
asm	这个关键字对大小写不敏感，用来允许在 shader 中手动插入着色汇编指令。它的使用语法如下：asm{ /* assembly instructions */ }。
asm_fragment	保留用于未来使用。
bool	用来定义布尔类型的值。详见第二章中关于数据类型的部分。
column_major	这关键字和矩阵一起使用，用来表示矩阵按以列为主的顺序排列。
compile	这个关键字用来声明顶点或像素片断（或着色器），指出使用哪一个 profile 来编译片断。当直接把一个着色器分配给顶点或像素着色器时使用这个关键字。
compile_fragment	这个关键字用来声明顶点或像素片断（或着色器），指出使用哪个 profile 来编译片断。
const	这个关键字用来定义常量。
discard	这个关键字只能在片断着色器中使用，用来取消对当前顶点的渲染。
decl	这个关键字对大小写不敏感。它是除 asm 关键字以外，用来以汇编方式定义常量寄存器值的指令。
do	这个关键字和 while 关键字一起使用，用来定义 do-while 循环。
double	这个关键字用来定义双精度的浮点数据类型。详见第二章中关于数据类型的部分。
else	这个关键字和 if 关键字一起使用，用来定义 if-else 语句。
extern	这个关键字在声明变量时使用，表明这个值可以被 effect 以外的代码访问。详见第二章中关于数据类型的部分。
false	这个关键字表示布尔数据类型中的 false 常量。详见第二章中关于数据类型的部分。
float	这个关键字用来定义单精度浮点数据类型。详见第二章中关于数据类型的部分。
for	这个关键字用来定义循环。
half	这个关键字用来定义半精度（通常为 16 位）浮点数据类型。详见第二章中关于数据类型的部分。
if	这个关键字用来定义 if-else 条件语句。
in	这个关键字表示函数的一个参数只能作为输入数据使用。详见第三章中关于函数

关键字	定义
	的部分。
inline	这个关键字提示编译器以内联方式来编译特定函数。
inout	这个关键字表示函数的一个参数既可以作为输入也可以作为输出数据使用。详见第三章中关于函数的部分。
int	这个关键字用来定义整型数据类型。详见第二章中关于数据类型的部分。
matrix	这个关键字用来定义矩阵数据类型。详见第二章中关于数据类型的部分。
out	这个关键字表示函数的一个参数只能作为输出数据使用。详见第三章中关于函数的部分。
pass	这个关键字是大小写不敏感的，用来定义多 pass effect 中的每个 pass。注意这个关键字和 effect 文件有很大关系，详见本书第二部分中的讨论。
pixelfragment	这个关键字用来定义像素片断（或着色器）。
return	这个关键字用来从函数中返回值。详见第三章中关于函数的部分。
register	这个关键字把寄存器预定为常量来使用。
row_major	这个关键字和矩阵一起使用，用来表示矩阵按以行为主的顺序排列。
sampler	这个关键字用来定义采样器数据类型。采样器是纹理和纹理属性（比如过滤器）的组合。详见第二章中关于数据类型的部分。
sampler1D	这个关键字和 sampler 的含义类似，但是特别指定了采样器用于 1D 纹理。详见第二章中关于数据类型的部分。
sampler2D	这个关键字和 sampler 的含义类似，但是特别指定了采样器用于 2D 纹理。详见第二章中关于数据类型的部分。
sampler3D	这个关键字和 sampler 的含义类似，但是特别指定了采样器用于 3D 纹理。详见第二章中关于数据类型的部分。
samplerCUBE	这个关键字和 sampler 的含义类似，但是特别指定了采样器用于立方纹理。详见第二章中关于数据类型的部分。
sampler_state	这个关键字通过定义一系列采样器状态来定义一个采样器。
shared	这个关键字表示一个全局变量被在多个 effect 共享。详见第二章中关于数据类型的部分。
stateblock	这个关键字用来定义 stateblock 类型的变量。Stateblock 变量保存了一系列 effect 状态。注意通常只在 effect 文件中使用这个关键字。详见本书第二部分中关于 effect 文件的内容。
stateblock_state	这个关键字用来定义包含一系列 effect 状态的状态块。
static	这个关键字用来定义静态变量。详见第二章中关于数据类型和变量的内容。
string	这个关键字用来定义字符串数据类型。详见第二章中关于数据类型的部分。
struct	这个关键字用来定义结构。详见第二章中关于数据类型的部分。
technique	这个关键字是大小写不敏感的，用来定义分配给 effect 的不同 technique。注意这个关键字和 effect 文件有很大关系，详见本书第二部分中的讨论。
texture	这个关键字用来表示纹理数据类型。详见第二章中关于数据类型的部分。
texture1D	这个关键字和 texture 关键字类似，但特别制定了所表示的为 1D 纹理。详见第二章中关于数据类型的部分。
texture2D	这个关键字和 texture 关键字类似，但特别制定了所表示的为 2D 纹理。详见第二章中关于数据类型的部分。
texture3D	这个关键字和 texture 关键字类似，但特别制定了所表示的为 3D 纹理。详见第二章中关于数据类型的部分。
textureCUBE	这个关键字和 texture 关键字类似，但特别制定了所表示的为立方纹理。详见第二章中关于数据类型的部分。

关键字	定义
true	这个关键字表示布尔类型中的 true 常量。详见第二章中关于数据类型的部分。
typedef	这个关键字用来定义一个新的数据类型。详见第二章中关于数据类型的部分。
uniform	这个变量用来把变量定义为 uniform 的，这表示在所有着色器运行时，这个变量的初始值都不会改变。详见第二章中关于数据类型的部分。
vector	这个关键字用来表示一个矢量类型的数据。详见第二章中关于数据类型的部分。
vertexfragment	这个关键字用来定义一个顶点片断（或着色器）。
void	这个关键字表示一个 void（或 empty）数据类型。详见第二章中关于数据类型的部分。
volatile	这个关键字用来提示编译器一个变量将会频繁改变。详见第二章中关于数据类型的部分。
while	这个关键字用来定义条件 do-while 循环

上面这个表可能看起来太枯燥了，特别是对于 HLSL 的初学者，甚至会感到有些迷惑。不要担心。这本书既是学习指南也是参考书。表 1-1 中的信息主要作为参考信息来使用。当我们学习 HLSL 语法和示例程序时，你就知道如何来使用它了。这一章里，我将会指出 HLSL 中所有主要的语法和文法（grammar）元素。在以后章节中再逐渐深入。

除了表 1-1 中的关键字外，还有一系列 HLSL 目前没有使用的保留关键字。这些关键字是为今后语言扩充所预留的，表 1-2 列出了这些保留关键字：

表 1-2 HLSL 保留关键字

auto	break	case	catch
default	delete	dynamic_case	enum
explicit	end	goto	long
mutable	namespace	new	operator
private	protected	public	reinterpret_case
short	signed	sizeof	static_cast
switch	template	this	throw
try	typename	union	unsigned
using	virtual		

预处理指令

除了保留关键字以外，HLSL 还定义了一组预处理指令来控制程序的编译。HLSL 所支持的预处理指令包含一下几个：#define, #endif, #else, #elseif,#error, #if, #ifdef, #ifndef, #include, #line, #pragma 和#undef。表 1-3 包含了对这些指令的简单描述

表 1-3 预处理指令

指令	定义
#define	这条指令用来声明一个新的编译器宏。
#if,#elseif,#endif, #ifdef,#ifndef	这是一组用来定义编译器条件的指令
#error	这条指令用来强制编译器生成一个错误信息，通常和其他条件指令一起使用。
#include	这条指令告诉编译器在编译时包含一个外部文件。
#line	这条指令将把它所在文件中的行号提交给编译器。
#pragma	这条指令用来控制编译器的一些特定行为，我们将稍后讨论。

注意，只有通过 ID3DXInclude 接口来使用 HLSL 编译器时，#include 指令才是有效的。详见第 12 章中关于 include manager 接口的部分。

在 HLSL 语言中，#pragma 指令定义了一系列子指令。可以像下面这样来使用 pack_matrix 指令：

```
#pragma pack_matrix ( row_major) // 或者 column_major
```

这条指令告诉编译器如何解释定义在 HLSL 文件中的矩阵（以列为主或以行为主）。在第二章中，当我们讨论在 HLSL 中定义不同类型的数据时，再来做进一步解释。

Warning 指令是另一条比较有意思的#pragma 子指令。它用来帮助我们定制 HLSL 编译器输出的警告消息。它的语法如下：

```
#pragma warning ( type : warning – number)
```

Type 参数定义了如何处理以 warning – number 表示的特定警告。第二个参数是一个以空白符分割的警告代号集合，代表你希望影响的警告。以下是 type 参数可能的值:

- once：指定警告信息只显示一次，忽略后面所有相同的警告。
- default：把指定警告信息的配置恢复为默认状态。
- disable：忽略指定警告的所有提示。
- error：把指定警告当作编译错误来对待。

#pragma 的最后一条预编译子指令是 def。这条子指令用来告诉编译器，特定编译 profile 中应该包含哪些寄存器。记住，对编译器来说，这只是一条优化提示指令，具体如何来使用寄存器还是取决于编译器。

注意

目前为止，只有常量寄存器（c#）支持 #pragma def 指令。

Profile 定义了编译顶点和像素着色器的硬件版本。表 1-4 列出了当前所有可用的 profile：

表 1-4 HLSL 编译 Profile

Profile	描述
vs_1_1	顶点着色器 1.1 版。

vs_2_0	顶点着色器 2.0 版。
vs_2_x	这个 profile 是顶点着色器 2.0 的扩展。它具有额外的能力，支持判断，流程控制和 12 个以上的临时寄存器。
vs_3_0	顶点着色器 3.0 版。
ps_1_1	像素着色器 1.1 版。
ps_1_2	像素着色器 1.2 版。
ps_1_3	像素着色器 1.3 版。
ps_1_4	像素着色器 1.4 版。
ps_2_0	像素着色器 2.0 版。
ps_2_x	这个 profile 是像素着色器 2.0 的扩展，它具有额外的能力，支持判断，流程控制和多于 12 的临时寄存器。
ps_3_0	像素着色器 3.0 版

注意

虽然没有通过 HLSL 呈现，但实际上 Direct3D 中 2.0 的像素着色器 profile 还有两种变体,称为 ps_2_a 和 ps_2_b。他们和 ps_2_0 很类似，但提供了以下额外功能：

- 临时寄存器的数量大于或等于 22（对 ps_2_b 来说是 32）。
- 任意的源数据重组（只对 ps_2_a 有效）。
- Gradient 指令：dsx, dsy（只对 ps_2_a 有效）。
- 读取纹理没有依赖限制（只对 ps_2_a 有效）。
- 对纹理指令数量没有限制。

记住 2.a 和 2.b 之间并不提供向后兼容性，它们只是在 shader model 3.0 发布之前，为了提供新功能，而使用的折中方式。

理解 HLSL 语法

在看完了 HLSL 中所有关键字，大部分预编译指令之后，我们可以开始学习语法了。计算机语言通常用巴科斯诺尔范式（Backus-Naur Form）(BNF)来表示语法，这是一种简单明了的方法。

用 BNF 来定义语法，不但简单明了，解释起来也很方便。现在就来看看如何用 BNF 定义语法，后面很多地方我都将用到它。总的来说，每行语法都代表了一条规则。每条规则定义了它所包含的结果。

if – statement : KW_IF KW_LPAREN expression KW_RPAREN statement

上面这条语法规则定义了 if 条件语句。表示 if 条件语句以 if 为关键字，紧接一个包含在封闭括号中的表达式，接下来还有一个语句块。表达式和语句块则由下面的规则来定义。除了基本规则之外，你还应该知道有许多重要的操作符可以和规则一起使用。

- a | b：竖线用来对规则进行逻辑或操作。
- (a)：括号用来把规则片段（segment）封装为一个块。
- [a]：带方括号的规则表示该片段是可选的。
- a + ：加号表示对一个片段进行一次或多次重复。
- a * ：星号表示对片段进行多次循环。

有了上面这些信息，你可以轻松的阅读语法规则了。在讲解 HLSL 语法之前，还需要了解一些词汇约定。

词汇约定

虽然语法定义了如何把所有语言元素组合到一起，比如，如何定义函数和代码片段，但它只定义了如何把表达式和操作符以及标识符一起使用。这意味着语言的语法并没有定义文法（例如： 什么是标识符）。接下来的几段详细解释了 HLSL 编译器中的词汇约定。

空白字符

HLSL 语言中，下面这些字符都被认为是空白字符：

- 空格
- Tab 字符
- 换行符
- C 风格的注解（/* */）。
- C++风格的注解（ // ）。
- asm 代码块中，汇编风格的注解（；）。

数字

HLSL 中的数字可以为浮点类型，也可以为整型。浮点数通常有以下呈现形式：

Float: (fractional-constant [exponent-part] [float-suffix] ）|

(digit-sequence exponent-part [float-suffix])

Fractnal-const: ([digit – sequence] . digit – sequence) | (digit – squence .)

Sign: + | -

Dight – sequence: digit | (digit – sequence digit)

floating – suffix: h | H | f | F

整型的语法与此类似：

Integer: integer – constant [interger – suffix]

Integer – constant: digit – sequence | (0 digit – sequence) | (0 × digit – sequence)

Digit – sequence: digit | (digit – sequence digit)

Integer – suffix: u | U | l | L

字符

HLSL 允许定义字符和字符串。字符串都是由字符组成。下面是字符的定义：

- ‘ c ‘ (字符)
- ‘ \t ‘ , ‘ \n ‘,…..(转义字符)
- ‘ \### ‘ (八进制换码顺序)
- ‘ \x## ‘ (十六进制换码顺序)

注意

预处理指令中不能包含转义字符。

字符串包含在一对引号中，可以包含前面所述的任意有效字符组合。

标识符

标识符用来表示函数名或变量名之类的语言元素。除了前面所列的关键字以外，标识符可以是字母和数字的任意组合，但必须保证第一个字符为字母。

操作符

HLSL 定义了一组操作符，以便在表达式中使用。表 1-5 列出了所有标准操作符，以及他们的含义。如果你熟悉 C 或 C++，那么这些操作符对你来说应该是一目了然的。

表 1-5 HLSL 操作符

操作符	描述
++	一元加法。
--	一元减法。
&&	逻辑与。
||	逻辑或。
==	等号。
::	成员标识符（用于结构和类）。
<<	二进制左移。
<<=	自赋值（self assigning）二进制左移，a<<= b 等于 a = a<<b。
>>	二进制右移。
>>=	自赋值二进制右移，a>>=b 等于 a = a>>b。
…	省略符（用于可变参数函数）。
<=	小于等于。
>=	大于等于。
!=	不等于。
*=	自赋值乘法， a *= b 等于 a = a * b。
/=	自赋值除法， a /= b 等于 a = a / b。
+=	自赋值加法， a += b 等于 a = a + b。
-=	自赋值减法， a -= b 等于 a = a – b。
%=	自赋值求余， a %= b 等于 a = a % b。

&=　　自赋值逻辑与， a &= b 等于 a = a & b。
|=　　自赋值逻辑或， a |= b 等于 a = a | b。
^=　　自赋值求幂， a ^= b 等于 a = a ^ b。
->　　重定向操作符，用来访问结构成员。

语言语法

HLSL 语言的语法相当简单。初看可能有些复杂，但只要使用它写几个程序，你马上就能掌握要领。目前为止，你不应该对语法太过担心，后面的章节我们将逐步了解语言的每个部分。由于实际的语法表相当长，我决定单独把他放到附录 D 中。另外你也可以参考 DirectX SDK 获取更多信息。

（译注：请参考 DirectX SDK 中 DirectX Graphics--Reference--HLSL Shader Reference--Appendix 中的 Language Syntax 部分）

观察表的第一行，可以看到 HLSL 程序被定义为一个 *program*。每个 program 要么为空，要么包含一系列 *decl*（声明）。最初的两行表示每个 decls 可以由多条其它 decl 组成。你可能已经注意到，声明可以用来定义空白语句，类型声明，变量声明，结构声明，函数声明或 technique 声明。语义定义了不同声明类型等等。

小结以及接下来的内容

在这一章里，我们简要概括了 DirectX 和 shader 技术在过去几年间的发展和历史。随着 shader model 2.0 和 3.0 复杂度的增加，开发者不但需要利用语言的所有新能力，同时，还需要高效的完成任务。由于新着色管道的指令和通用处理器上的指令越来越类似，因此，开发一门高级语言，让开发者把注意力集中在 shader 所要实现的功能上，而不是把精力放在如何使用寄存器，或对某种硬件如何组合指令才能最优化之类的琐碎问题上是很有意义的。

在需求的驱动下，微软开发并通过 DirectX SDK 发布了 HLSL 着色语言，帮助开发者使用最新的图形技术，创建更加真实的图形。本章，我们学习了很多语法背后的基础知识。虽然这章看起来有些枯燥，不要担心，随后的几个章节我们就会讨论一些比较有趣的内容。

第二章 HLSL 着色语言

这一章我们将学习 HLSL 的大部分基本内容，包括丰富的变量类型，如何定义变量，和如何编写 shader 代码。你可能还想知道如何在 shader 中使用函数。虽然它们是使用 HLSL 编写 shader 的必须元素，但我还是决定把所有关于函数的内容作为单独一章来讲解。在第三章中，我们将详细讲解如何定义函数，以及 HLSL 为开发者所编写的内置函数。

在继续学习之前，需要进行一点点说明，读者在前一章学习 HLSL 语法时可能注意到了诸如 techniques，渲染状态，pass 之类的概念。虽然它们也是 HLSL 语言的一部分，但事实上它们只在 effect 文件中使用，而 effect 文件则是 HLSL 的一个超集。

好了，闲谈到此为止，让我们直接进入本章主题，讨论在任何语言中都是最重要的元素，数据类型。

数据类型

数据类型是任何一门编程语言的核心。没有它们你将无法定义变量或者在函数之间传递数据。如果连呈现数据的方法都没有，又怎么能储存数据呢？和其他高级语言一样，HLSL 有一系列内置定义类型，可以把它们分为以下几类：

- 标量类型
- 矢量类型
- 矩阵类型
- 对象类型

除了上面的内置类型外，HLSL 同样允许自定义的复合类型，比如数组和结构。在介绍 HLSL 特定数据类型前，先来仔细看看大部分 3D 硬件上呈现数据的方式。当处理数据时，基本的选项是浮点值，根据硬件构架的不同，这些值可能为 16 或 32 位。记住，虽然可以让硬件同一时间只处理一个值，但这并不是最高效的方法。相反，如果硬件每次处理一个包含四个分量值的单元（或矢量），那么则可以同时对四个分量进行处理。当然，这就需要考虑硬件如何来保存非矢量的值。我稍后会讲解这一点。

除了浮点值以外，较新的硬件也内置支持整型和布尔值，但它们主要用于分支，循环，和条件测试语句。接下来的几页中，我将浏览每一种数据类型，并解释如何来使用它们。

标量类型

标量类型是由 HLSL 标准定义的，他们是最基本的类型。所有复杂类型，比如矢量，矩阵，和结构都由标量组成。表 2-1 列出了所有可用的标量类型以及它们所表示的值。

表 2-1 HLSL 标量类型

标量类型	值
bool	true 或 false
int	32 位有符号的整型
half	16 位浮点数
float	32 位浮点数
double	64 位浮点数

需要注意并不是所有 shader 目标都天生支持整型，half 和 double 值。如果要将 shader 编译为不支持指定类型数据的目标，那么将模拟处理 float 值的方式来处理它们。与使用原生（native）类型相比，结果可能不正确。在确定目标平台支持某个特定类型之前，为了保证一致性和可移植性，最好坚持使用标准浮点数。

注意

我们在上一章已经简要讨论过 profile，它用来告诉 HLSL 编译器为哪种构架的 shader 产生代码。通常 Profile 的值直接对应于可用的顶点和像素着色器版本，但对于指令集没有改变的情况可能会有例外。

矢量和矩阵类型

矢量和矩阵是把标量组织为 1 维或 2 为数组的标准方式。它们通常用来表示 3D 数据，比如法线和变换矩阵。

矢量是 HLSL 标准所定义的，由特定标量类型组成的一维数组。默认情况下，一个矢量由 4 个浮点值组成。如表 2-2 所示，可以手动定义任何类型的矢量。

表 2 -2 HLSL 矢量类型

矢量类型	值
vector	一个包含四个浮点分量的矢量
vector<type, size>	包含 size 个 type 类型的矢量

另外，为了简便，HLSL 语言预定义了一组标准矢量类型，方便开发者使用。下面就是这些预定义的矢量类型

```
typedef vector <float, 4> VECTOR;
typedef vector <bool, #> bool# ;
typedef vector <int, #>  int# ;
typedef vector <half, #> half# ;
typedef vector <float, #> float# ;
typedef vector <double, #> double# ;
```

（译注：“#”表示分量个数。）

你可能还记得我之前提到过图形硬件是基于矢量的。你看，这里所定义的矢量数据类型恰好与硬件相对应，编译器将把这些数据直接映射为矢量。当你使用少于四个分量的矢量时，编译器将在空闲分量中添加其他变量，以便压缩数据。

简要讨论了矢量之后，你可能想知道如何才能单独访问矢量中的某个分量。有许多方式允许我们单独访问矢量中的分量。下面列出了访问数组分量的几种不同方式。注意对多于四个分量的数组来说，额外的分量只能通过索引访问。

- 使用分量符访问：vector.x , vector.y , vector.z , vector.w
- 使用颜色符访问：vector.r , vector.g , vector.b , vector.a
- 使用索引访问：vector[0], vector[1], vector[2], vector[3]

接下来看看矩阵类型。矩阵是 HLSL 标准所定义的，由特定标量类型组成的二维数组。默认情况下

matrix 是一个四乘四的浮点数据。如表 2-3 所示，可以手动定义任何类型的矢量。

表 2-3 HLSL 矢量类型

矢量类型	值
matrix	一个 4x4 的浮点矩阵
matrix<type, rows,cols>	包含 rows 行 cols 列 type 类型的矢量

同样，为了简便，HLSL 语言预定义了一组标准矩阵类型：

```
typedef matrix<float, 4 , 4>    Matrix;
typedef matrix <bool, #, #> bool#x# ;
typedef matrix <int, #, #>    int#x# ;
typedef matrix <half, #, #> half#x# ;
typedef matrix <float, #, #> float#x# ;
typedef matrix <double, #, #> double#x# ;
```

和矢量类似，硬件将在内存中使用一系列连续的矢量来储存矩阵，每个矢量表示矩阵中的一行数据。可以使用数组风格的寻址方式来单独访问矩阵中的某一行。举个例子，使用索引值，比如 Matrix[3]，就能获得一个大小正确的矢量，从而单独访问矩阵中的一行。你甚至可以进一步通过行访问矩阵中的单个分量，比如 Matrix[2].x 或 Matrix[3][2]。

此外，还可以使用预置分量名规则来访问矩阵中的单个分量，预置分量名有面两种形式。

基于 1 的分量名

_11	_12	_13	_14
_21	_22	_23	_24
_31	_32	_33	_34
_41	_42	_43	_44

基于 0 的分量名

_m00	_m01	_m03	_m04
_m10	_m11	_m12	_m13
_m20	_m21	_m22	_m23
_m30	_m31	_m32	_m33

你可能还记得第一章中的#pragma pack 预处理指令。这个指令使用 row_major 或 col_majoe 参数来控制矩阵数据在 shader 中的呈现方式。默认情况下，数据被打包为以列为主的形式。这表示矩阵中的每一列都保存在一个常量寄存器中。相反，以行为主的打包方式则把矩阵中的每一行放到一个常量寄存器当中。

一般来说，列为主的矩阵要比行为主的矩阵高效，因为他们和 Direct3D 所期望的数据所匹配， shader 只需用最少的指令就能完成操作。

注意

注意，#pragma pack 只会影响作为全局变量或函数参数传递的矩阵。在 shader 中定义的矩阵将忽略这条指令，并总是把矩阵按列为主的方式对待。

分量访问和重组

如前所述，可以像访问结构成员一样来访问矢量和矩阵中的单个分量。这里对矢量和矩阵的访问方式做个小结。

_11, x, r	_12, y, g	_13, z, b	_14, w, a
_21	_22	_23	_24
_31	_32	_33	_34
_41	_42	_43	_44

此外，通过把二到四个分量名串连到一起作为下标来使用，还可以让矢量只包含某几个指定分量，HLSL 把这门技术称为重组。这里就是一些有效的重组实例：

bgr， yyzw， _12_22_32_42

对矩阵来说也是如此。

```
tempMatrix._m00_11 = worldMatrix._m00_m11;
tempmatrix._11_22_33 = worldMatrix._24_23_22;
temp = fMatrix._m00_m11;
temp = fMatrix._11_22;
```

注意，所有下标都必须来自于同一下标集，（比如 xyzw，rgba，或者_11 到_44），不同下标集之间的元素不能进行混合，比如不能出现 xyrg 这样的组合。同一个分量可以出现多次。但对赋值目标来说，则不允许分量重复出现。

注意

并不是所有硬件都支持所有类型的重组操作。对像素着色器 1.x 的某些版本来说，部分重组操作是不允许的。编译器可以通过模拟来补偿这些限制，但将带来一定性能损失。

对象类型

HLSL 还定义了一系列范围广泛的对象数据类型。这些类型通常用来呈现非数字数据与复合类型的句柄，典型的例子就是纹理和结构体。以下是 HLSL 中所定义的对象类型：

- 采样器
- 纹理
- 结构体
- 顶点和像素着色器
- 字符串

结构的内容将在下一节关于自定义类型的部分详细讨论。

字符串指使用 ASCII 码定义的字符串，它们除了用作注解以外，用处不是太多，这里也不对它进行详细讨论。采样器，着色器和纹理才是我们所关心的重点部分。

采样器和纹理

HLSL 定义了两种数据类型用于在 shader 中获取纹理信息。纹理本质上就是一个指向硬盘上一系列物理像素信息的句柄。采样器则是一组纹理采样参数的组合，比如 warp mode 或者 mipmap 属性等。

顶点和像素着色器

HLSL 定义了两种数据类型用来保存顶点和像素着色器，它们分别是 vertexshader 和 pixelshader。如果使用 asm 关键字以汇编方式来编写着色程序，则可以把这些程序直接分配给着色器：

```
vertexshader vs =
{
    vs_2_0
     decl_position v0
     mov oPos, v0
};
```

如果使用 HLSL，那么则必须通过函数来定义着色器。下面就是使用高级指令定义像素着色器的例子：

```
pixelshader ps = compile ps_2_0 psmain();
```

结构及自定义类型

除了前面看到的大量预定义数据类型以外，HLSL 也允许开发者创建新类型。自定义类型通常都是结构，它们是由一组其他（内置或自定义）数据类型，或基于现有类型声明的新类型，组成的对象。

使用关键字 struct 来定义结构。结构是复合类型，用来把多个数据组合为一个整体。用下面的语法来定义结构：

```
struct [ ID ] { members }
```

这里是创建结构的一个例子：

```
struct Circle
{
    float4 Position;
    float Radius;
```

};

另外，HLSL 还允许使用 typedef 关键字，为现有类型声明一个新名称。它的语法如下:

typedef [const] type id [array_suffix] [, id…];

数组后缀可以跟随在 ID 之后，允许把数组作为新类型。当声明了一个类型之后，就可以通过 ID 来对它进行引用。注意 array_suffix 由一个或多个 literal_integer_expression 组成，用来表示数组维度。

类型转换

在程序设计中，术语类型转换表示把一种数据转变为另一种数据的能力。HLSL 支持多种内置类型间的转换。表 2-4 总结了内置数据间可能的转换。

表 2-4 HLSL 中的类型转换

转换类型	描述
标量——标量	这类转换总是有效的。当把布尔值转换为整型或浮点类型时，false 值表示 0，true 表示 1。同样，当把整型或浮点类型转换为布尔值时，0 表示 false。当把浮点类型转换为整型时，将四舍五入为最接近的整数。
标量——矢量	这类转换总是有效的。转换将把标量复制并填充到矢量中。
标量——矩阵	这类转换总是有效的。转换将把标量复制并填充到矩阵中。
标量——对象	这类转换是是无效的。
标量——结构	这类转换仅当结构中的成员都为数字时才是有效的。转换将把标量复制并填充到结构中。
矢量——标量	这类转换总是有效的。转换将复制矢量中的第一个分量，并填充到标量中。
矢量——矢量	目标矢量容量不大于源矢量时才是有效的。转换只保留最左边（left-most）的分量，截去剩下的分量。
矢量——矩阵	只有当矢量和矩阵一样大时，转换才是有效的。
矢量——对象	这类转换总是无效的。
矢量——结构	这类转换只有当结构容量不大于矢量，且所有成员都为数字时才是有效的。
矩阵——标量	这类转换总是有效的。转换将把矩阵左上角的值填充到标量中。
矩阵——矢量	只有当矢量和矩阵一样大时转换才是有效的。
矩阵——矩阵	只有当目标矩阵维度不大于源矩阵时，转换才是有效的。转换将把源矩阵填充到目标矩阵的左上部分，并且丢弃余下数据。
矩阵——对象	这类转换总是无效的。
矩阵——结构	只有当结构容量和矩阵一样大，且所有成员都为数字时，转换才是有效的。
对象——标量	这类转换总是无效的。
对象——矢量	这类转换总是无效的。
对象——矩阵	这类转换总是无效的。
对象——对象	只有当两个对象都是同一类型时，转换才是有效的。
对象——结构	只有当结构包含一个以上的成员时，转换才是有效的。结构中成员的类型必须和对象的类型一样。

结构——标量	只有当结构包含一个以上的成员时，转换才是有效的。这个成员必须为数字。
结构——矢量	只有当结构容量不小于矢量时，转换才是有效的。它的第一个成员必须为数字，并且等于矢量的大小。
结构——矩阵	只有当结构容量不小于矩阵时，转换才是有效的。它的第一个成员必须是数字，并且等于矩阵大小。
结构——结构	只有当目标结构容量不大于源结构容量时，转换才是有效的。目标结构和源结构间各自成员的转换也必须是有效的。

定义变量

HLSL 允许把变量定义为常量，输入，输出和临时变量。变量的标准定义语法如下：

[static uniform vloatile extern shared] [const] type id [array_suffix]
[:semantics] [= initializers] [annotations] [, id …];

从语法定义规范可以看出，对变量可以使用多个变量关键字前缀，来告诉编译器如何对待变量。表 2-5 列出了不同前缀所表示的含义。

表 2－5 变量前缀

前缀	描述
static	static 用于全局变量，表示值为一个内部变量，仅能被当前 shader 中的代码访问。用于局部变量则表示这个值将在多次调用间驻留。静态变量只能进行一次初始化操作，如果没有使用特定值进行初始化，则默认为 0。
uniform	使用 uniform 声明的全局变量表示对整个 shader 来说，它是一个统一的输入值，即不能在 shader 运行期间改变。所有非静态全局变量都被认为是统一的。
extern	使用 extern 声明的全局变量表示对 shader 来说，它是一个外部输入的值。所有非静态全局变量都被认为是外部的。
volatile	这个关键字提示编译器变量将频繁改变。
shared	这个关键字用来修饰非全局变量，告诉编译器这个值将被多个 effect 共享。
const	声明为 const 的变量表示在初始化之后，就不能改变它的值。

声明变量的语法中，需要注意的是 semantics 部分。语义（semantixs）对 HLSL 语言来说并没有什么特别含义，它是用来定义如何把 shader 中的变量及变量的含义映射到 effect framewrok 中的。

语义通常用于顶点和像素程序的输入和输出变量，把他们映射为特定含义，比如顶点位置或纹理坐标。举例来说，COLOR0 语义用来告诉编译器，指定变量表示第一种漫反射颜色，并且在大多数 shader 版本中，将把它放到 d0 寄存器中。

如语法定义规范所示，允许全局变量包含一个注解（annotation）。注解以｛member_list｝的形式出现。Member_list 是一个程序声明的集合，其中每个成员都被初始化为指定字面值。注解只能用来为 effect 传递元数据，不能在程序中对它进行引用。

我们将在第六章详细讨论语义和注解。

语句和表达式

与其它高级语言一样，HLSL 也包含语句和表达式。虽然这些元素通常作为函数的一部分来使用，而我们要在下一章才讲解函数。但由于他们是构建 shader 的主要元素，因此，现在就来看看这些元素。

语句

语句用来控制程序流的执行顺序。HLSL 定义了多种类型的语句供开发者使用。下面按功能的不同，对这些语句分为了四类：

- 表达式
- 语句块
- 返回语句
- 流程控制语句

接下来将详细讨论上面的每种元素，第一项是表达式，但我想把它们放到这节的后面一点，先来看看第二项，语句块。

简单来说，语句块就是包含在一对大括号中的语句集合。它把语句组织为一个群组，同时定义了一个子范围，块中所定义的变量只存在于当前语句块范围中。

{ [statements] }

接下来看返回语句。返回语句用于把函数执行的结果返回给调用者。它的语法如下：

return [expresstion] ;

上面可以看出，使用 return 关键字和一个紧随的表达式来定义返回语句。记住，为了让程序通过编译，表达式类型必须和函数所定义的返回值类型相匹配。

最后一项流程控制语句，它们用来控制程序的执行顺序：

if （expression） statement [else statement]

do statement while (expression)

```
while (expression) do statement

for ( [ expression | variable_decleration ] ; [ expression ] ; [exxpression ] )
statement
```

可以看到，HLSL 中的流程控制语句与 C 或 C++中的基本相同。但与 C 或 C++不同的是，在 shader 中使用流程控制语句，必须进行一些性能上的考虑。

流程控制性能考虑

目前，大多数顶点和像素着色器硬件都以线性方式执行 shader，每条指令执行一次。HLSL 支持的流程控制形式包括静态分支，断言指令，静态循环，动态分支和动态循环。由于某些着色器实现的限制，部分流程控制指令可能会带来重大的性能损失。

举例来说，顶点着色器 1.1 版的构架并不支持动态分支，因此使用 if 语句，产生的汇编代码将同时实现 if 语句中所有代码。Shader 将顺序执行完这些代码，但只使用 if 语句中某一块代码的输出作为结果。这里是一段将使用 vs_1_1 编译的程序：

```
if ( Value > 0 )
    Position = Value1;
else
    Position = Value2;
```

下面是编译之后的汇编代码：

```
// 在 r0.w 中计算线性插值量
mov     r1.w, c2.x
slt     r0.w, c3.x, r1.w

//根据比较结果对 Value1 和 Value2 进行插值
move    r7, -c1
add     r2, r7, c0
mad     oPas, r0.w, r2, c1
```

从上面的代码可以看到，在顶点着色器模型 1.1 中使用 if 语句，将导致 if 语句中的所有表达式都被执行，之后通过插值来计算最终输出。在真正支持动态分支的情况下，这个语句只会产生一条指令，但这里确需要五条指令。

除 if 语句外，一些硬件也允许使用动态或静态循环，但多数情况下他们都是线性执行的。

除像素着色器 1.1 以外，所有着色模型都支持流程控制语句，但只有支持顶点和像素着色器 3.0 的硬件才是真正使用 18 条流程控制语句来支持流程控制的。这意味着所有非 3.0 的 shader 都将把流程控制转换为一系列代码，执行分支的所有部分，或者把循环展开来使用。对图形硬件

来说，硬件流程控制还处于起步阶段，所以性能欠佳。编写代码时应该注意如何才能让程序合理的执行。在第七章中，我将深入讨论关于动态分支的性能。

流程控制可以分为静态或动态的。对静态流程控制来说，语句块中的表达式实际上是常量，并且在 shader 执行前就确定了。举例来说，静态分支允许根据 shader 中的一个布尔常量来决定是否执行一块代码。这是一个很方便的功能，我们可以根据当前所渲染的对象类型来控制代码执行的路径。在调用渲染函数之前，你可以选择让当前 shader 支持哪些特性，之后，为相应代码块设置布尔标志。

相反，大部分开发者最熟悉的还是动态分支。对于动态分支来说，条件表达式的值是一个变量，只有在运行时才能确定。考虑动态分支的性能时，应该包括分支语句本身的代价以及分支中指令的代价。目前只有在硬件支持动态流程控制的顶点着色器中动态分支才是可用的。

表达式

在讨论了语句之后，我们来看看表达式。表达式定义为字面值，变量或通过运算符对两者的组合。表 2-6 列出了所有可用的运算符以及以及它们的含义。

表 2－6 运算符

运算符	用法	定义	结合方向
()	(value)	子表达式	左到右
()	id(arg)	函数调用	左到右
()	type(arg)	类型构器	左到右
[]	array[int]	数组下标	左到右
.	structure.id	选择成员	左到右
.	value.swizzle	分量重组	左到右
++	variable++	递增后缀（作用于所有分量）	左到右
--	variable--	递减后缀（作用于所有分量）	左到右
++	++variable	递增前缀（作用于所有分量）	右到左
--	--variable	递减前缀（作用于所有分量）	右到左
!	!value	逻辑非（作用于所有分量）	右到左
-	-value	一元减法（作用于所有分量）	右到左
+	+value	一元加法（作用于所有分量）	右到左
()	(type)value	类型转换	右到左
*	value * value	乘法（作用于所有分量）	左到右
/	value / value	除法（作用于所有分量）	左到右
%	value % value	模运算（作用于所有分量）	左到右
+	value + value	加法（作用于所有分量）	左到右
-	value – value	减法（作用于所有分量）	左到右
<	value < value	小于（作用于所有分量）	左到右
>	value > value	大于（作用于所有分量）	左到右
<=	value <= value	小于等于（作用于所有分量）	左到右

>=	value >= value	大于等于（作用于所有分量）	左到右
==	value == value	等于（作用于所有分量）	左到右
!=	value != value	不等于（作用于所有分量）	左到右
&&	value && value	逻辑与（作用于所有分量）	左到右
||	value || value	逻辑或（作用于所有分量）	左到右
?:	float ? value : value	或条件	右到左
=	value = value	赋值（作用于所有分量）	右到左
*=	variable *= variable	乘法赋值（作用于所有分量）	右到左
/=	variable /= value	除法赋值（作用于所有分量）	右到左
%	variable %= value	取模赋值（作用于所有分量）	右到左
+=	variable += value	加法赋值（作用于所有分量）	右到左
-=	variable -= value	减法赋值（作用于所有分量）	右到左
,	value , value	逗号	左到右

由于硬件求值方式的差异，与C语言不同，&&、||和?:三个短路求值表达式并不是短路的（译注：对多数现代语言来说，布尔表达式中，只需要部分进行求值。比如逻辑与，如果第一个表达式结果为False，则会结束这个表达式的求值，并生成一个False结果，这种方式称为短路。）。此外，你可能已经注意到了很多运算符都标注为“作用于所有分量”。这表示将对输入值（通常是4D向量）中的每一个分量进行独立运算。运算结果将保存到输出向量的相应分量中。

小结以及接下来的内容

这一章，我们学习了HLSL语言的基本内容。此时，你应该对HLSL中的数据类型有了相当了解，能定义变量，构造语句和表达式。你看，HLSL的语法和C或C++是很类似的。

需要告诫的是，在使用高级语言编写shader时，必须时时记住目标硬件所支持的功能。特别是编写流程控制语句时。早期的硬件着色器并不支持任何形式的流程控制，因此，需要做性能上的考虑。

继续学习，下一章，我们将讨论函数，并完成对HLSL语言部分的学习。我将教授你如何使用HLSL中丰富的预置函数库，以及如何编写你自己的函数。好了，不要浪费时间，马上进入下一章.....

第三章 函数，只讨论函数

上一章里，我们学习了HLSL主要的语法元素。现在，唯一没有讲解的只剩下函数了，包括如何声明与定义函数，如何在shader中使用函数。函数是高级语言中的重要组成部分，它在

shader 中也同样扮演了重要角色。HLSL 语法允许使用两种类型的函数。内置（或固有）函数为 shader 提供了一个预定义函数库，同时也为特定着色构架提供了某些特殊指令。

当然，你可以创建自定义函数。自定义函数可以用来把 shader 组织为一个整体，也可以用来打包部分希望重用的功能。

接下来的几节里，我将会讨论两种类型的函数，在最后还会讲解如何用函数定义 shader。先来看看 HLSL 提供的丰富内置函数库吧。

内置函数

HLSL 着色语言包含了一系列广泛的，内置，或固有函数。这些函数在开发 shader 时相当有用。它们提供了从数学计算到纹理采样等广泛的功能。先依次浏览一下这些函数。

表 3-1 HLSL 内置函数

函数名	用法
abs	计算输入值的绝对值。
acos	返回输入值反余弦值。
all	测试非 0 值。
any	测试输入值中的任何非零值。
asin	返回输入值的反正弦值。
atan	返回输入值的反正切值。
atan2	返回 y/x 的反正切值。
ceil	返回大于或等于输入值的最小整数。
clamp	把输入值限制在[min, max]范围内。
clip	如果输入向量中的任何元素小于 0，则丢弃当前像素。
cos	返回输入值的余弦。
cosh	返回输入值的双曲余弦。
cross	返回两个 3D 向量的叉积。
ddx	返回关于屏幕坐标 x 轴的偏导数。
ddy	返回关于屏幕坐标 y 轴的偏导数。
degrees	弧度到角度的转换
determinant	返回输入矩阵的值。
distance	返回两个输入点间的距离。
dot	返回两个向量的点积。
exp	返回以 e 为底数，输入值为指数的指数函数值。
exp2	返回以 2 为底数，输入值为指数的指数函数值。
faceforward	检测多边形是否位于正面。
floor	返回小于等于 x 的最大整数。
fmod	返回 a / b 的浮点余数。
frac	返回输入值的小数部分。
frexp	返回输入值的尾数和指数
fwidth	返回 abs (ddx (x) + abs (ddy(x))。
isfinite	如果输入值为有限值则返回 true，否则返回 false。
isinf	如何输入值为无限的则返回 true。

isnan	如果输入值为 NAN 或 QNAN 则返回 true。
ldexp	frexp 的逆运算，返回 x * 2 ^ exp。
len / lenth	返回输入向量的长度。
lerp	对输入值进行插值计算。
lit	返回光照向量（环境光，漫反射光，镜面高光，1）。
log	返回以 e 为底的对数。
log10	返回以 10 为底的对数。
log2	返回以 2 为底的对数。
max	返回两个输入值中较大的一个。
min	返回两个输入值中较小的一个。
modf	把输入值分解为整数和小数部分。
mul	返回输入矩阵相乘的积。
normalize	返回规范化的向量，定义为 x / length(x)。
pow	返回输入值的指定次幂。
radians	角度到弧度的转换。
reflect	返回入射光线 i 对表面法线 n 的反射光线。
refract	返回在入射光线 i，表面法线 n，折射率为 eta 下的折射光线 v。
round	返回最接近于输入值的整数。
rsqrt	返回输入值平方根的倒数。
saturate	把输入值限制到[0, 1]之间。
sign	计算输入值的符号。
sin	计算输入值的正弦值。
sincos	返回输入值的正弦和余弦值。
sinh	返回 x 的双曲正弦。
smoothstep	返回一个在输入值之间平稳变化的插值。
sqrt	返回输入值的平方根。
step	返回（x >= a）? 1 : 0。
tan	返回输入值的正切值。
fanh	返回输入值的双曲线切线。
transpose	返回输入矩阵的转置。
tex1D*	1D 纹理查询。
tex2D*	2D 纹理查询。
tex3D*	3D 纹理查询。
texCUBE*	立方纹理查询。

为 了贴近实际，举个例子来展示如何使用这些函数吧。假设你需要把纹理映射到一个像素上，并且使用方向光来照亮这个像素。要完成这个任务，必须先计算光源对像 素颜色的贡献，然后查找纹理颜色，最后把这两个颜色混合起来。首先，为了计算方向光的贡献，需要计算像素法线和光源方向的点积。使用 dot 函数可以很方便的完成这一步计算：

```
LightIntensity = dot ( LightDirection , PixelNormal);
```

这里可能会出现一点小小的问题，如果像素法线背对着光源方向，那么得到的亮度将为负值。必须保证亮度在 0 和 1 之间，以避免这种效果。怎么做呢？很幸运，saturate 函数能完成这个任务，修改上面的代码：

LightIntensity = saturate (dot (LightDirection , PixelNormal));

在把灯光颜色添加到物体上之前，还需要从纹理中获得像素的颜色。假设像素已经包含了适当的纹理坐标并且有一张简单的 2D 纹理，那么纹理采样的代码如下：

PixelColor = tex2D (objectTexture , TextureCoord) ;

最后一步，就是对灯光和像素颜色进行混合。这里，我们需要灯光颜色，灯光亮度以及像素颜色作为参数。计算很简单，但你应该注意 shader 构架的矢量天性是如何对颜色中的所有分量同时起作用的。

FinalColor = (LightColor * LightIntensity) * PixelColor ;

这个例子虽然简单，但是它展示了 HLSL 的强大威力，仅仅使用三行代码就能完成简单的光照。在进入下一个主题之前，需要指出根据完成功能的不同，内建函数可以接收不同的参数。另外，由于硬件性能的不同，部分内建函数并不是在所有顶点和像素着色器版本上都可用。

自定义函数

除了 HLSL 提供的大量内建函数以外，同样可以使用类似于 C 语言的方式定义自定义函数。下面就是声明函数的语法：

[static inline target] [const] return_type id ([parameter_list]) { [statement] }

接下来的是定义函数原型的语法：

[static inline target] [const] return_type id ([paramter_list]);

你看，可以使用一系列修饰符作为函数的前缀，来控制编译器对待这些函数的行为。表 3－2 列出了可能的自定义函数前缀，以及它们的含义。

表 3-2 自定义函数前缀

前缀	定义
static	这个前缀表示函数只存在于当前 shader 程序的作用域中，不能被多个 shader 共享。本书中大多数情况都不使用这个关键字。
inline	这个前缀表示把函数代码复制到调用代码之后执行，而不是真正按照函数调用的方法来执行代码。注意，对编译器来说这个前缀只起提示作用，并不能保证函数是内联的。还需要注意，这个前缀是当前 HLSL 编译器的默认行为。

target	这个前缀表示希望使用哪一个版本的顶点或像素着色器版本来编译代码。允许编译器对特定着色器版本进行优化。
const	这个前缀表示参数值在函数中不能改变。

记住，默认情况下所有函数都是内联的，因此不能递归调用。这是因为函数的处理，编译和执行都是由顶点和像素着色器硬件来完成的。着色器硬件只能以线性方式执行代码，不能跳转到代码的其他位置。这表示函数总是被内联到调用代码中。递归将导致代码路径是不确定的，所以被禁止。

此外，在 parameter_list 中定义的参数，也必须符合特定的声明语法：

[uniform in out inout] type id [: semantic] [= default]

同样可以使用修饰符和关键字作为参数前缀，控制编译器对待参数的行为。表3－3 列出了参数前缀和它所表示的含义。

表 3-3 函数参数前缀

转换类型	描述
in	这是默认情况下的参数行为，对函数来说，这是个只读参数。
out	这个前缀表示参数是一个返回值，任何对它所做的改变都将返回给调用者。
inout	这个前缀是 in 和 out 行为的组合。
uniform	这个前缀和 in 前缀具有相同含义，但是特别指明参数来自于 shader 中的常量。

当为参数指定了语义标识符之后，他将会告诉编译器去哪里找输入数据源。举例来说，TEXCOORD0 标识符将会告诉编译器把第一组纹理坐标作为这个参数的输入值。注意，标识符只对 shader 中的顶级函数才有意义，也就是顶点着色器或像素着色器的入口函数才能使用语义标识符。

目前为止，只剩下 return_type 参数没有讨论了，它用来定义函数的返回值类型。当函数没有返回值，或者通过 out 参数返回数据时，应该把返回值类型设置为 void。

函数返回值可以是 HLSL 中定义的任何基本数据类型。此外，也可以是结构，允许函数同时返回一系列值。下面就是把结构作为返回值的例子：

```
struct VS_OUTPUT
{
    float4 vPosition    : POSITION;
    float4 vDiffuse     : COLOR;
};

VS_OUTPUT   VertexShader_Tutorial ( flaot4 inPos : POSITION )
{
    VS_OUTPUT Result;
```

```
    //Do something…
    return Result;
}
```

使用 return 关键字加变量名来从函数中返回值，这里变量类型必须和函数返回值类型一致。

这里，你可能在想如何使用带 out 前缀的参数。实际上，可以使用 out 参数来代替函数返回值。下面的代码展示了如何使用 out 参数来代替函数返回值。

```
struct VS_OUTPUT
{
    float4 vPosition      : POSITION;
    float4 vDiffuse       : COLOR;
};

void VertexShader_Tutorial ( float4 inPos : POSITION,
                             out VS_OUTPUT outReturn)
{
    VS_OUTPUT   Result;
    //Do something
    outReturn = Result;
}
```

你看，在 HLSL 中编写函数和使用其他高级语言几乎是一样的。在学习复杂函数和 shader 之前，先来看看如何用函数定义 shader。

通过函数创建 Shader

编写自定义函数的主要目的之一就是通过它们定义 shader。虽然讨论 effect framework 时我们才会详细学习如何声明 shader，但是我希望先透露一点点内容给你。这里是使用自定义函数声明 shader 的例子。

```
Shader = compile shaderProfile FunctionName();
```

上面的语法中，shaderProfile 可以是第一章中提到过的任意一个 profile 值，FunctionName 元素则将被编译为 shader。定义 shader 的过程实际上很简单，把希望的 shader 代码编写为一个函数，之后使用上面的语法把他编译并声明为 shader。

为了让这个例子更具体一些，我们来看看如果如何编写一个简单的像素光照函数，并把它声明并编译为像素着色器。

```
flaot4 lighting ( in float3 normal, in float3 light, in float3 halfvector, in float4 color)
{
```

```
        float4 color;
        color = dot ( normal, light) * color;
        color += dot ( light , halfvector) * color;
        return color;
}

float4 myShader ( in float2 tex:TEXCOORD0, in float3 normal : TEXCOORD1,
            in float3 light: TEXCOORD2, in float3 halfvector:TEXCOORD3,in float4
           color:COLOR0)
{
    //compute the lighting color
    Float4 lightColor = lighting(normal,light,halfvector,color);
    //Fetch the texture color
    Float4 terColor = tex2D( texture_sampler, tex );
    //Modulate the final color
    Return lightColor * terColro;
}

PixelShader = compile ps_2_0 myShader();
```

小节以及接下来的内容

这一章，我们讨论了 HLSL 中的函数。HLSL 语言本身提供了一个丰富的内置函数库，有大约 70 个内建函数供开发者调用。但更重要的是你可以通过编写自定函数定义 shader，或把完成特定功能的代码打包到一起，以便复用。实际上，函数也许是 HLSL 中最重要的元素，每次定义 shader 时都必须用它。

现在我们已经学习了 HLSL 的大部分语法，可以开始学习 effect framework 了吗？暂时还不行！下一章我们将学习编写一系列基本的 shader 例子。这些例子不单用来告诉你 HLSL 能做什么，同时，也是我们继续学习所要用的基础代码。

第四章 Shader 示例

上一章里，我们详细讨论了 HLSL 着色语言的各方面。但并没有实际展示如何编写 shader。虽然本书不是关于如何编写 shader 的，但还是有必要编写几个简单的 shader，帮你深入了解 HLSL。此外，在学习 effect framework 时，我们还会用到这些例子来阐述一些核心概念。

记住，对创建一个完整的 shader 来说，不仅仅是编写 shader 代码，还包括用适当的语义符设置一系列渲染状态和变量。当然，由于目前你还缺乏编写完整 shader 的一些知识，所以，这里只讨论前者:也就是顶点和像素着色程序代码。本书的后面会对这些代码进行扩展。

最简单的 Shader

对于把物体渲染到屏幕上来说，有几个基本的步骤是必须完成的。首先，需要接收输入顶点的位置（顶点在世界坐标中的位置）并把它们转变为屏幕坐标。通常使用 world-view-projection 矩阵来完成这个任务，它包含了把顶点从局部坐标映射为最终屏幕坐标的所有信息。现在开始，我们假设已经有这样一个矩阵变量，并且名称为 view_proj_matrix。

先来定义一个把数据从顶点着色器传递给像素着色器的结构。我们把这个结构称为 VS_OUTPUT,当然，也可以是任何你喜欢的名字。目前，只需要把顶点位置数据添加到这个结构中。

```
struct VS_OUTPUT
{
    float4 Pos:   POSITION;
};
```

你应该注意到我们把 POSITION 语义连接到了 Pos 变量上，它将告诉 effect 系统如何把这个变量传递到像素着色器中。我会在下一章讲解语义。现在只差顶点着色器代码了。顶点着色器接收顶点位置，并使用 view_proj_matrix 矩阵对它进行变换，可以用内建的 mul 函数来完成这一步计算。我们把顶点着色器代码放到一个名为 vs_main 的函数中：

```
VS_OUTPUT   vs_main ( float4 inPos : POSITION)
{
    VS_OUTPUT Out;
    // output a transformed and projected vertex position
   Out.Pos = mul ( view_proj_matrix , inPos);
    return Out;
}
```

这里同样使用了 POSITION 语义修饰输入参数 inPos。它将告诉顶点着色器把几何体数据流信息映射为这个参数的输入值。接下来进入完成这个简单 shader 的第二步。现在你知道了顶点在屏幕上的位置，可以定义顶点的颜色了。最简单的方法就是把所有顶点的颜色都设置为一个常量。通常像素着色器将返回一个 float4 类型的值来表示当前像素在屏幕上的颜色值，float4 分量分别表示红色，绿色，蓝色和 alpha 值。我们把像素着色器代码放到一个名为 ps_main 的函数中：

```
float4 ps_main ( void ) : COLOR
{
    //Output constant color
    float4 Color;
    color[0] = color[3] = 1.0; // red and alpha on
    color[1] = color[2] = 0.0;// Green and Blue off
    return color;
}
```

这几乎是最简单的代码了，注意我们用 COLOR 语义修饰了函数的返回值，它将告诉编译器把函数返回值作为当前像素的颜色值。

着色

我们已经有渲染物体所需的最基本代码了，如何把纹理映射到几何体上，让物体看起来更加真实呢？对于需要使用纹理的 shader 来说，需要有一个 sampler 类型的全局变量。在后面的章节中，我会教你如何使用语义和 effect framework 来设置纹理状态。目前我们假设已经设置了好了纹理状态：

```
sampler Texture0;
```

使用纹理之前，还需要知道知道对纹理的哪一部份进行采样映射，因此，每个像素都必须有相应的纹理坐标。一般情况下，纹理坐标将作为几何体信息的一部分输入到顶点着色器中，经由顶点着色器计算处理之后，传入到像素着色器中。通常使用 TEXCOORDx 语义来修饰作为参数传递的纹理坐标。这个语义将会告诉硬件如何在顶点和像素着色器之间交换数据。以下是修改之后的顶点着色器代码:

```
struct VS_OUTPUT
{
    float4 Pos :    POSITION;
    float2 Txr1:    TEXCOORD0;
}

VS_OUTPUT vs_main(
    float4 inPos : POSITION;
    float2 Txr1 : TEXCOORD0)
{
    VS_OUTPUT Out;
    //Output our transformed and projected vertex position and texture coordinate
    Out.Pos = mul ( view_proj_matrix, inPos);
    Out.Txr1 = Txr1;
    returen Out;
}
```

像素着色器也同样简单。在创建了 sampler 变量之后，可以使用 HLSL 的内建函数 tex2D 来对纹理进行采样，代码如下：

```
sampler Texture0;
float4 ps_main(
    float4 inDiffuse : COLOR0,
    float2 inTxr1 : TEXCOORD0) : COLOR0
{
```

```
    //Output the color taken from our texture
    return tex2D ( Texture0, inTxr1);
}
```

添加光照

虽然添加了纹理的对象看起来不错，但显然还不够真实。在增加场景真实度的过程中，很重要的一步就是为对象添加光照。真实世界中，从太阳到灯泡，充满了各种光。没有了光线，就什么都看不到了。

虽然光照本身是一个相当复杂的主题，但在计算机图形领域中，光通常被简化为几种基本类型：

- **环境光（Ambient lighting）**：场景中所有光源经过多次放射和折射之后，对场景总亮度贡献的近似模拟。通常用它来减少场景中所需光源的数量，模拟出多光源下的照明效果。环境光通常是一个常量，对所有物体的作用效果都一样。
- **漫反射光（Diffuse lighting）**：材质的微观粗糙表面将导致在有所方向上均匀的反射入射光线。在任何角度接收到的反射光线强度都是相同的。
- **镜面高光（ Specular lighting）**：当材质表面相当光滑，粗糙度很低时，将以一种非均匀的方式反射光线。对镜面高光来说，光线强度不但与入射光角度有关，和观察者的角度也有关。

除了知道光线如何影响物体之外，你还需要如何对光源本身分类。虽然光总是由某个表面发出，比如太阳或灯泡表面，但你也可以把它们看作来自某个方向或某个点。

光照技术中，方向光是最简单的类型。它们没有位置信息，并且假设所有光线之间都是平行的，指向同一个方向。哪一种光源是这样的呢？现实中并没有这样的光源。方向光是假设光源离物体无限远时，照射到物体上的光线将近似于平行而得出的。

方向光最好的例子就是阳光。如果把太阳看作一个离地球上亿千米的点光源，那么当阳光到达地球表面时已经近似于平行了，完全可以看作是方向光。

此外没有位置信息表示方向光不随距离而衰减。对方向光来说，要考虑的因素只有两个：方向和光的颜色。看到这里你可能会问光线是如何影响物体表面的。如图所示，光线照射到物体表面的强度只与入射光线和表面法线的角度有关。

知道了这些基础知识，就可以用入射光的方向矢量和表面法线的点积以及灯光的颜色因子计算出物体表面上任意一点的光照强度和颜色。这让我们得出了以下代码：

```
Color = Light_Color * saturate ( dot ( Light_Direction, inNormal ) );
```

注意在上面的代码中我们使用了 saturate 函数。它保证对于背对光线的面来说，获得的光照强度不会为负值。当然，你也可以使用 clamp 函数，但是对于把值限制在 0 到 1 之间来说，saturate 函数要更加高效。

一般来说，场景中大多数的光都来自于灯泡，火炬或类似的光源。仔细观察一下这类光源，它们通常由一个很小的有限点发出，并且位于场景中的某个特定位置。简化一下，你可以把这些光源都看作场景中的一个点，这就是点光源。

对 这类光源来说，光线呈放射状发出。这意味着只要物体和光源的距离相等，那么无论在哪个方向，所受到的影响都相同。由于表面的光照强度与光线和物体表面法线 之间的关系有关，因此我们所要做的第一步就是计算出光线的方向。显然，对于表面上的任意点来说，光线方向就等于从当前点的位置指向光源位置的矢量。对点光 源来说，随角度的衰减值如下：

```
//compute the normalized light direction vector and use it to determine the angular light attenuation
float3 Light_Direction = normal ( inPos – Light_Position);
float AngleAttn = saturate ( dot ( inNormal, Light_Direction) );
```

此外对于点光源来说，还需要考虑它在距离上的衰减。自然，需要计算光源到当前点的距离，使用如下代码：

```
float Distance = length ( inPos – Light_Position);
```

通常情况下，点光源的衰减因子随距离的平方成反比。但是为了获得很多的可控性，可以调整公式，让衰减和距离的二次多项式成反比，代码如下：

```
//compute distance based attenuation. this is defined as
// attenuatin = 1 / ( a + b*distance + c * disctance * distance)
float DistAttn = saturate( 1 / ( LightAttenuation.x + LightAttenuation.y * Dist +
                                 LightAttenuation.z * Dist));
```

现在把前面的代码集成到顶点着色器中吧：

```
struct VS_OUTPUT
{
    float4 Pos:      POSITION;
    float2 TexCoord:  TEXCOORD0;
    float2 Color:    COLOR0;
};

float4 Light_PointDiffuse( float3 VertPos, float3 VertNorm, float3 LightPos, float4 LightColor,
                           float4 LightAttenuation)
{
    //determine the distance from the light o the vertex and the direction
    float3 LightDir = LightPos – VertPos;
```

```
        float Dist = length(LightDir);
        LightDir = LightDir / Dist;
        //Compute distance based attenuation.
        float DistAttn = saturate( 1 / ( LightAttenuation.x + LightAttenuation.y * Dist +
            LightAttenuation.y * Dist*Dist));
        //comopute angle based attenuation
        float AngleAttn = saturate ( dot (VertNorm, LightDir));
        // Computer the final lighting
        return LightColor * DistAttn * AngleAttn;
}

VS_OUTPUT vs_main( float4 inPos: POSITION,
        float3 inNormal: NORMAL,
        float2 inTxr : TEXCOORD0)
{
        VS_OUTPUT Out;
        Out.Pos = mul ( view_proj_matrix, inPos);
        Out.TexCoord = inTxr;
        float4 Color = Light_PointDiffuse ( inPos, inNormal, Light_Position, Light1_Color,
Light_Attenuation)
        Out.Color = Color;
        return Our;
}
```

我把计算点光源光照的代码单独放到了 Light_PointDiffuse 函数中，因此，当场景中有多个点光源时，你可以复用这段代码。当然，我们在后面的章节会有这样的例子。

我们已经有了个一个漫反射点光源着色器，现在应该考虑光照方程中的高光部分了。和漫反射相比，镜面高光最大的区别就是光照亮度不但与光线到表面的角度有关，还和观察者的角度有关。

为了计算高光，我们需要一个称为中间（half）矢量的值。这个矢量其实是观察矢量和灯光矢量的中间值。为了计算观察矢量需要把观察点变换到模型空间。我们假设观察点的位置位于(0,0,10)。把它变换到模型空间之后，加上顶点位置，就是最终的观察矢量：

```
EyeVector = -normal（mul ( inv_view_matrix, float4(0,0,10,1) + inPos);
```

把 EyeVector 与光源矢量混合到一起，然后进行标准化，就是中间矢量。由于 EyeVector 和 LightDir 都是标准矢量，所以中间矢量相当于二者的均值（A + B）/ 2。

```
HalfVect = normalize ( LightDir – EyeVetor);
```

(译注：更直观的做法是在世界坐标下计算 EyeVector，然后用 LightDir ＋ EyeVetor 计算 HalfVect)

有了中间矢量，把它和表面法线的点积进行 m 次幂运算，就能算出光源基于角度的光线强度。幂运算时的指数相当于物体的镜面指数，值越大，高光区域就越小也越明亮。下面的代码假设镜面指数为 32。

```
struct VS_OUTPUT
{
    float4 Pos:      POSITION;
    float2 texCoord: TEXCOORD0;
    float2 Color: COLOR;
};

float4 Light_PointSpecular( float3 VertPos, float3 VertNorm,float3 LightPos,
        float4 LightColor, float4 Lightattenuation, float3 EyeDir)
{
    //Determine the Distance from the light to the vertex and the direction
    float3 LightDir = LightPos - VertPos;
    float Dis = length(lightDir);
    LightDir = LightDir / Dist;
    //Computer half vector
    float3 HalfVect = normalize( LightDir = EyeDir);
    //Compute distance based attenuation. this is defined as:
    //Attenuation = 1/(LA.x + LA.y * Dist + LA * Dist * Dist)
    float DistAttn = saturate( 1 / ( LightAttenuation.x + LightAttenuation.y * Dist +
            LightAttenuation.z * Dist * Dist));
    float SpecularAttn = pow(saturate ( dot(VertNorm,HalfVect)),32);
    //Compute finfal lighting
    return LightColor * DistAttn * SpecularAttn;
}

VS_OUTPUT vs_main( float4 inPos:POSITION,float3 inNormal:NORMAL,float2
inTxr:TEXCOORD0)
{
    VS_OUTPUT Out;
    //compute the projected positon and send out the texture coordinates
    Out.Pos = mul(View_proj_matrix,inPos);
    Out.TexCoord = inTxr;
    //Output the ambient Color
    float4 Color = Light_Ambient;
    //Determine the eye vector
    float3 EyeVector = -normlaize(mul(inv_view_matrix,float4(0,0,10,1)) + inPos);
    //computer the light contribution
    Color = Light_PointSpecular(inPos,inNormal,Light1_Position,Light1_Color,Lihgt_Attenuation,
        EyeVector);
    //Output Final Color
```

```
    Out.Color = Color;
    return Out
}
```

当考虑光照时，大部分人都认为逐顶点的光照足够好了。对于镶嵌度较高的模型来说是这样，但对某些复杂或的精度模型来说却不一定。出现这种效果的原因是顶点间颜色插值的方式。

当 逐顶点照亮对象时，将为每个顶点计算一次光照颜色，然后在通过顶点在多边形所覆盖的区域对像素颜色进行线形插值。现实中，光照值取决于光线角度，表面法 线，和观察点（对于镜面高光来说）。与逐像素对所有光照元素进行单独插值，再计算光照相比，直接对顶点颜色进行插值所得的结果通常不够精确，特别是对面积 较大的多边形来说。

（左图为 per-pixel lighting,右图为 per-vertex）

上图显示了逐像素和逐顶点光照的差别。当处理高精度多边形模型时，由于每个多边形所覆盖的区域很小，因此插值之后每个像素的误差也很小，所以逐顶点光照可以工作的很好。而当处理低模时，这种误差就变的很大了。

使用逐像素光照的另一个好处是可以在渲染时添加并不存在的表面细节。通过 bump map 或 normal map，可以在像素级别让原本平坦的表面表现出近似的凹凸效果。

再回过头来看看漫反射光线，你需要决定哪些光照元素可以插值之后传送到像素着色器，哪些元素必须逐像素计算。从头到尾，决定漫反射光照的就是表面法线和光线矢量的点积。

这个点积定义了光照的强度，但是对它进行插值的结果通常不正确。因此这一步计算应该移动到像素着色器中，进行逐像素计算。

表面法线和光矢量是计算点积的要素。通常矢量间的插值是正确的。因此可以在顶点着色器计算他们的值，并传递到像素着色器中，然后进行最终的点积计算。

注意

虽然对法线的插值是正确的，但是插值之后的向量将不再是标准向量。为了对此进行校正，需要在像素着色器中对法线重新进行标准化，可以使用内建的 normalize 函数。

为了把光照计算移动到像素着色器中，需要先添加两个变量到顶点着色器的输出结构中。可以使用 TEXCOORD1 和 TEXCOORD2 语义把这些值传递到像素着色器。代码如下：

```
struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float2 TexCoord: TEXCOORD0;
    float3 Normal: TEXCOORD1;
    float3 LightDir: TEXCOORD2;
}
```

接下来修改顶点着色器代码。目前我们只考虑单光源光照的情况，所以先删除原先的光照函数把其中的代码复制到 vs_main 函数中。记住，我们后面回讨论多光源的情况。

你可能已经注意到我们还没有讨论基于距离的衰减值应该在哪里计算。虽然对距离的插值结果不是完全正确，但由于光线的变化有足够容差范围，距离上的微小差别并不会给结果带来明显变化，所以不需要逐像素计算。

衰减值只是一个标量，为了把它传递给像素着色器，而又不浪费一个额外的寄存器来传值，可以把 LightDir 改为一个 float4 的矢量，把衰减值作为这个矢量的 w 分量。修改后的顶点着色器如下：

```
VS_OUTPUT vs_main(float4 inPos:POSITION,float3 inNormal: NORMAL,
    float2 inTxr: TEXCOORD0)
{
    VS_OUTPUT Out;
    Out.Pos = mul( view_proj_matrix, inPos);
    Out.TexCoord = inTxr;
    Out.Normal = inNormal;
    //Computer and move the light direction to the pixel shader
    float4 LightDir;
    LightDir.xyz = Light1_Position - inPosition;
    float Dist = length(LightDir.xyz);
    LightDir.xyz = LightDir.xyz / Dist;
    LightDir.w = saturate( 1 / ( LightAttenuation.x + LightAttenuation.y * Dist +
            LightAttenuation.y * Dist*Dist));
    //Output the light direction
    Out.LightDir = lightDir;
    return Out;
}
```

在像素着色器中，只需要接收参数，计算点击和颜色就可以了。为了方便，把光照计算单独放到一个名为 Light_PointDiffuse 的函数中。本质上来说，这里计算光照的方法和在顶点着色器中是一样的。

```
float4 Light_PointDiffuse(float4 LightDir, float3 Normal,Float4 LightColor)
{
    //compute dot product of N and L
    float AngleAttn = saturate(dot(Normal,LightDir.xyz));
    //compute final lighting
    return LightColor * LightDir.w * AngleAttn;
}

float4 ps_main(float3 inNormal:TEXCOORD1,
    float4 inLightDir : TEXCOORD2) : COLOR
{
    //Compter the lighting contribution for this single light
    return Light_PointDiffuse(inLightDir,inNormal,Light_Color);
}
```

很简单，对吧！接下来编写镜面高光像素着色器。整个步骤基本上和我们刚才编写漫反射像素着色器的方法差不多。这一个 shader 的核心是法线和中间矢量的点积，需要把它们移动到像素着色器中。这意味着需要在顶点着色器中计算光照矢量，中间矢量。先修改顶点着色器的输出结构。

```
struct VS_OUTPUT
{
    float4 pos: POSITION;
    float2 TexCoord: TEXCOORD0;
    float3 Normal : TEXCOORD1;
    float4 LightDir: TEXCOORD2;
    float3 HalfVect: TEXCOORD3;
}
```

在顶点着色器中，需要计算以下值：表面法线，光照矢量，中间矢量和基于距离的衰减值。只需要把先前编写的镜面高光着色器代码稍作修改就行了。

```
VS_OUTPUT vs_main(float4 inPos:POSITION,float3 inNormal:NORMAL,
        float2 inTxr: TEXCOORD0)
{
    VS_OUT Out;
     Out.Pos = mul(view_proj_matrix, inPos);
    Out.TexCoord = inTxr;
    float4 LightDir;



    LightDir.xyz = Light1_Position - inPos;
    flaot Dist = length(LightDir.xyz);
    LightDir.xyz = LightDir.xyz / Dist;

    LightDir.w = saturate( 1 / ( LightAttenuation.x + LightAttenuation.y * Dist +  LightAttenuation.y
* Dist*Dist));
    //computer eye vector and the half vector
    float3 EyeVector = -normalize(mul(inv_view_matrix,float4(0,0,10,1))+inPos);
    Out.HalfVect = normalize(LightDir - EyeVector);
    //Output normal and light direction
    Out.Normal = inNormal;
    Out.LightDir = LightDir;
    return Out;
}
```

对像素着色器来说，同样把计算高光的代码作为一个单独的函数：

```
float4 Light_PointSpecular(float3 Normal,flaot3 HalfVect,float4 LightDir,float4 LightColor)
```

```
{
    float SpecularAttn = pow(saturate(dot(Normal,HalfVect)),32);
    return LightColor * LightDir.w * SpecularAttn;
}

float4 ps_main(float3 inNormal:TEXCOORD1,float4 LightDir:TEXCOORD2,
        float3 HalfVect:TEXCOORD3):COLOR
{
    //simply route teh vertex color to the output
    return Light_PointSpecular(inNormal,HalfVect,LightDir,Light1_Color);
}
```

介绍 HLSL 或者 DirectX \ XNA 的书中都很少会详细讲解 Semantics，DirectX SDK 对此也是草草带过，加上这个单词本身对汉语来说含义比较难懂，因此 Semantics 常常成了 HLSL 中鲜为人知的一部分。然而，对于构建复杂 Effect 系统来说，Semantics 作为 DirectX \ XNA 和 shader 之间传输数据的纽带，有相当重要的作用。

Semantics 究竟是什么呢？简单来说，它是描述 shader 中变量的原数据，用来表示某个变量的语义。它与 DirectX 中 D3DDECLUSAGE 或者 XNA 中 VertexElementUsage 枚举的作用相当类似。

Semantics 有什么用呢？Shader 程序使用 Semantics 作为标记，实现数据从 CPU 到 GPU 的绑定。

为什么使用 Semantics 呢？ 在 Semantics 出现之前，shader 程序员常常直接使用寄存器名来绑定数据。这样做有两个缺点，首先，编译器无法对代码进行优化，其次，维护移植起来也相当不方便。Semantics 隐藏了寄存器的复杂性，所有积存器的分配都由编译器来为你处理，同时，对外提供了一个统一的接口。

如何使用 Semantics 呢？先来看 Semantics 的语法：

[Modifiers] ParameterType ParameterName [: Semantic] = [Initializers]

比如：

float4x4 WorldMatrix : WORLD;

冒号前面的部分，声明了一个变量 myWorldMatrix，你也许希望用它来表示世界坐标变换。但是，编译器对此一无所知，只会把它当作一个普通的 float4x4 变量来对待。然而，添加了冒号以及标准 Semantics WORLD 之后，无论变量叫什么名字，编译器都会知道这个变量将被当作坐标变换矩阵来使用。注意，Semantics 本身并不区分大小写，只是习惯上用大写表示。

从类型上来看，Semantics 可以分为 vertex\pixel semantics 和普通变量 semantics 两种。

对于标记了 vertex\pixel semantics 值的变量来说，顶点会自动把 vertex shader 输入顶点或输出数据中相应的值绑定到变量上。

比如：

vs_main(float4 pos : Position, our float3 oColor : COLOR0){…}

Shader 会自动把输入顶点中的位置信息绑定到 pos 变量上，此外

ps_main(float3 diffColor : COLOR0){…}

则会把 vertex shader 的输出颜色 COLOR0 绑定到 diffColor 变量。

对于普通变量 semantics 来说，则主要用来与外部数据进行绑定。在开发复杂 Effect 系统时，除非在开发先就有非常详细的变量命名规则，否则，程序将淹没在各种不同的 Effect 变量名间。每次调用 Effect.SetValue()方法时，你都不得不查看 HLSL 代码中所用的具体变量名，这无疑是开发者的噩梦。然而，只要使用了 semantics，这个问题就变的简单了。通过 DirectX 中的 ID3DXBaseEffect::GetParameterBySemantic()或者 XNA 中的 EffectParameterCollection.GetParameterBySemantic()方法，能轻易得到对应某个 semantics 的变量名，从而为它赋值，大大节约了开发时间。这也是 ms 推荐使用标准 semantics 的原因。

需要说明的是在 DirectX 中，似乎只列出了标准的 vertex\pixel semantics，此外，还有相当数量的标准普通变量 semantics，他们几乎涵盖了 shader 中所有可能用到的变量用法。