《Real Time Rendering》第三章 图形处理单元

历史上，硬件图加速器出现于管线的末端，首先是运行三角面扫描线的光栅化。紧接着的下一代硬件沿管线上溯，到一些更上级的层次，一些应用程序阶段的算法亦被囊括在硬件加速器的范围内。致力于使用硬件唯一的好处是其超过软件实现的速度，速度是关键的。
在过去的十年，图形硬件经历了一个不可想象变革。第一块包括硬件顶点处理的消费性图形芯片（NIDIA的Geforce256）在1999年问世。NVIDIA杜撰了图形处理单元（GPU）这一术语去区分Geforce256和之前其它只处理光栅化的芯片，（and it stuck）。在此后的几年，GPU从原来复杂的可配置固定功能管线到高度可编程的实现，就像白纸一样，开发者可在其上实现他们自己算法。各种可编程着色器是GPU控制的主要手段。顶点着色器能实现多种运行在每个顶点上的运算（包括坐标变换和变形）。相似的，像素着色器处理每个像素个体，允许在逐素上使用复杂的方程求值。几何着色器允许GPU在工作的间隙创建和销毁几何图元（点，线，三角面）。计算得出的数值可写入高数度缓存中作为顶点纹理数据重用。基于效率，管线的一部分仍然是可配置，而非可编程，但趋势是朝向可编程性和灵活性发展。

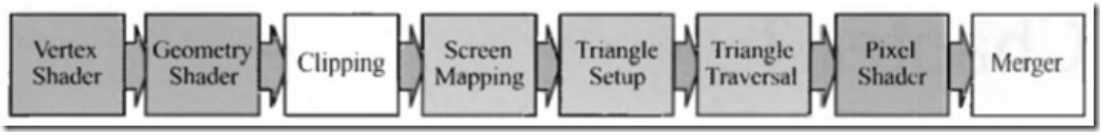

图3.1 GPU实现的渲染管线。图根据使用者可控制的程度使用不同颜色标识。绿色的阶段为完全可编程。黄色的阶段为可配置而非可编程，例如裁剪阶段是可选择运行的裁剪和添加用户制定的裁剪平面。蓝色的阶段的功能是完全固定。

3.1 GPU管线概述

GPU实现了第二章所描述的几何和光栅化的概念管线。这些被分割成数个硬件实现的阶段，这些阶段支持不同程度的可配置性或可编程性的。图3-1通过不同的颜色展示了其可编程性和可配置性。需要注意的是这些物理的阶段划分相对第二章中的功能性阶段有轻微的不同。
顶点着色器是一完全可编程的阶段，通常用于实现“模型和视图变换”，“顶点着色”和“投影”等功能性阶段。几何着色器则是可选的，完全可编程的阶段，对图元（点、线、三角面）的顶点进行操作。它可以用于每个图元的着色操作，销毁或创建新图元。而裁剪，屏幕映射，三角形建立和三角形遍历阶段是固定功能阶段，实现其同名的功能阶段。像顶点和几何着色器，像素着色器是一完全可编程且运行像素着色的功能阶段。最后，融合阶段处于完全可编程的着色器阶段和其它阶段的固定操作之间。虽然它不是可编程的，但它是高度可配置并且可在之上运行多种操作。当然，它实现了融合这一功能阶段，负责修改颜色、深度、混合、模板及其它相关缓存。
随着时间的过去，GPU管线由硬编码的操作向越来越灵活和可编程操控的方向发展。可编程着色器阶段的引入是管线进化的重要一步。下节将描述各种可编程阶段共有的特点。

3.2 可编程着色器阶段

现代的着色器阶段（例如在Vista上支持Shaker Model 4.0, DirectX10及后面出现的）使用一个共同的着色器内核。这意味着顶点、像素和几何着色共享同一个编程模型。我们在书里需要区分着色器公共核心（一个为应用程序员所见的功能说明）和统一着色器（一个与这个核心相吻合的GPU架构）。见18.4节。着色器公共核心是API；统一着色器是GPU的一特征。早期的GPU中的顶点和像素着色器并无太多共同的地方，且其没有几何着色器。但这个模型大多数的设计元素是共享使用了旧的硬件；对于大部分的修改，旧版本的设计不是太简单就是缺少这个功能，而非根本的不同。所以现在我们反焦点放在Shader Model 4.0而将旧GPU着色器模型放在后面讨论。
描述整个编程模型是已经超出本书的范围，且有很多的文件、书籍和网页已经做了这方面的工作。但很少关于这些文献的良好的评论。着色器使用与C语法相似的着色语言如HLSL，Cg和GLSL等进行编程。这些代码被编译成独立于计算机硬件的汇编语言，亦叫作中间语言。之前的着色器模型允许直接使用汇编编程，但DirectX10,这种语言的编程仅作为调试版本的输出。这些汇编语言会在一单独的步骤被转换成机器码，这通常在驱动程序中。这种设计能兼容不同的硬件实现。这种汇编语言可被视为定义了一以着色语言编译器为目标的虚拟机。这虚拟机是一个带有各种寄存器和数据源的处理器，它使用一套指令进行编程。因为很多的图形操作是使用短整形的向量（长度为4），处理器有四道的SIMD（单指令多数据）能力。每个寄存器包含四个独立的数值。32位单精度浮点标量和向量是基本的数据类型；最近加入了对32位整形数的支持。浮点数何量通常包含了位置（XYZW），法线，矩阵行，颜色（rgba）和纹理坐标（uvwq）等数据。整形数通常用于计数器，索引或位掩码。集合数据类型例如结构件，数组和矩阵等亦被支持。为了方便向量的使用，拌和技术（swizzling）——向量成员的复现，亦被支持。这是何量的单元可以被任意地重排或重复。相似地，掩蔽技术（masking），即使用所指定的向量单元，亦被支持。
绘制调用到图形API去绘制一组图元，因此引起图形管线的执行。每个可编程着色器阶段有两种输入：一致输入（uniform input），其数值在整个绘制调用中保持不变（但在不同的绘制调用间是可变的）；而可变输入，对于每个顶点或像素着色器是不同的。纹理贴图是一种特殊的一致输入，在以前经常被认为是应用于表面的一幅彩色图片，但现在也可以认为是一任意大小的数据。需要注意的是虽然着色器可以不同方式片理各种输入，但其输出是极度固定的。这是着色器与运行在通用处理器上的程序最显注的区别。虚拟机的底层提供了用于不同输入输出的寄存器。一致输入能被常量寄存器或常量缓存访问，如此称谓是因为它们的内容在一次绘制调用中保持不变。常量寄存器的数目远远大于可变的输入输出寄存器的数目。这是因为不同的顶点和像素输入和输出需要分别存储，而一致输入存储一次然后在绘图调用的所有顶点或像素中重用。虚拟机亦具有通用的临时变量，是用于暂存空间。所有类型的寄存器可使用临时寄存器中的整数值进行数组索引。着色器虚拟机的输入输出如图3.2所示。

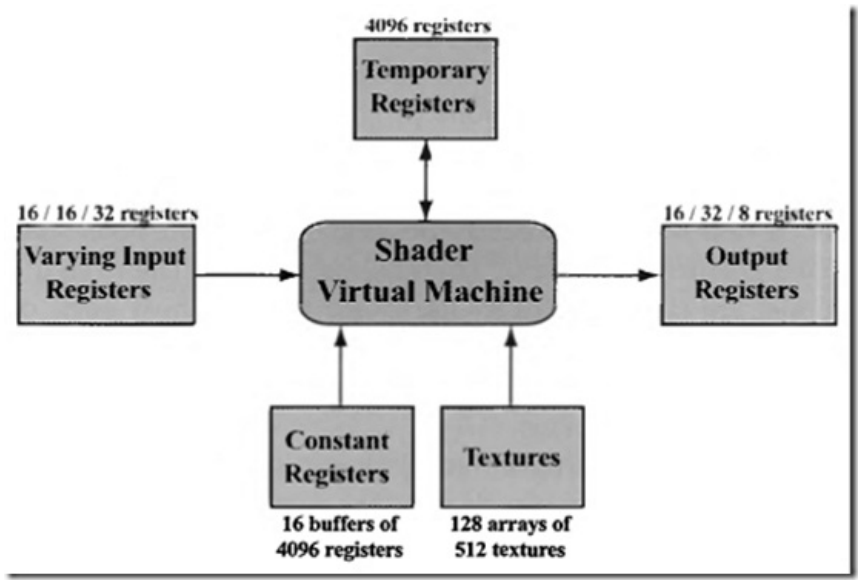

图3.2 在Directx10下的公共着色器核心虚拟机的架构和寄存器的布置。每个资源的最大的数值在其旁边显示。三个使用斜杠分隔的数值分别指代其顶点上，几何，像素着色器的最大值（从左到右）。

在图形计算中常用的操作在当代的GPU上是高效地执行的。例如，最快的操作是标量和向量的乘法、加法和它们的混合，例如乘加和点积。另一些操作，例如倒数，平方根，正弦，余弦，指数和对数，相较而言会消耗更多，但仍然是相当快速的。纹理操作（见第六章）也是高效率的，但它们的运行表现受限于一些因素，例如花费时间在等待访问资源的结果。着色语言通过操作符提供了这些普遍的操作（例如通过操作符*和+提供加法和乘法）。其余则通过内置函数提供，例如atan(),dot()log（）及其它。内置函数亦处理更复杂情况，例如向量的单位化和反射,叉积，矩阵转置和行列式等。

流程控制这个术语指的是使用分支指令改变代码执行的流程。这些指令用于实现高层语言的“if”和“case”声明语句，以及各种循环。着色器支持的两种流程控制。静态流程控制分支是基于一致输入的值。这意味着流程控制在绘制调用中保持不变。静态流程控制的主要好处在于允许同一个着色器应用在各种不同的情况（例如，光线数目的变化）。动态流程控制则是基于可变输入的值。这比起静态流程、控制的功能更为强大但亦耗费更多，特别是在着色器调用之间不规律地改变。在18.4.2所论及的，着色器每次都对若干的顶点或像素进行赋值。虽然在流程选择中的if分支用于某些单元，而else分支用于其它，但对于所有单元所有的分支都会被赋值（而在每个单元中没有使用的分支将被丢弃）。

着色器程序可在程序被装载前离线编译或可以运行时编译。像其它任何的编译器一样，这有关于不同输出文件和不同代码优化水平的设置。编译好的着色器被以文字字符串形式保存，通过驱动程序传递到GPU。

3.3 可编程着色器的衍化历程

关于可编程着器框架的想法可追溯到1984所Cook的着色树。一简单的着色器和他对应的着色树如图3.3所示。Render Man着色语言就是建基于这种想法，它产生于80年代末而时至今天仍广泛用于电影场景的渲染。在GPU原生地支持可编程语言之前，已有一些以多渲染路径实现实时可编程着色操作的尝试。1999年的《雷神之锤III：竞技场》的脚本语言是第一个在这方面取得普遍的商业化成功的例子。在2000年，Peercy等人描述一个将Render Man着色器翻译并运行在图形硬件的多个渲染路径的系统。他们发现GPU缺少两项能使这个方法变得非常普遍的特性：像纹理座标那样使用计算结果（附属纹理贴图读取），以及支持扩展和精度的纹理和颜色缓冲数据类型。所提出的新型数据（在当时）是16位的浮点数。在那个时候，没有任何商业化的GPU支持可编程着色，虽然大部分已经有高度可配置的管线。

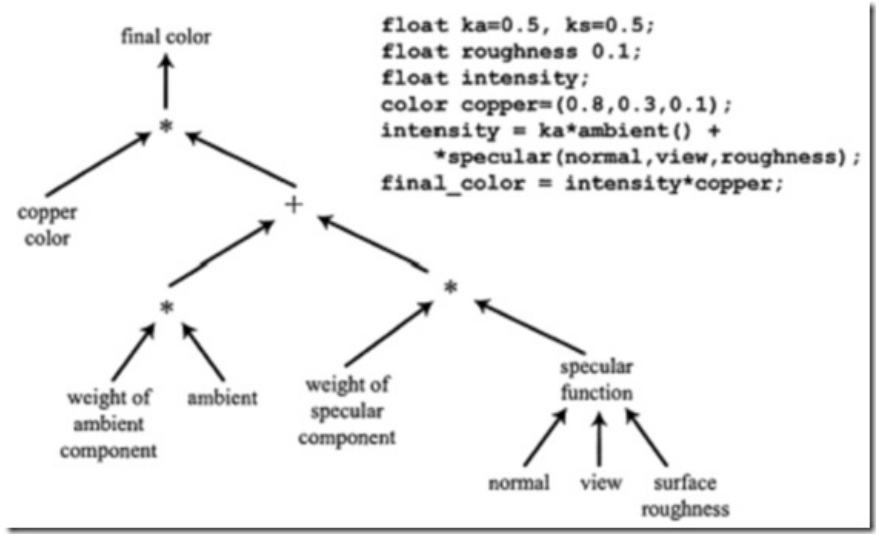

图3.3铜材质的着色树，和它相应的着色语言程序

在2001年的上半年，NVIDIA的GeForce3是第一颗支持可编程顶点着色器的GPU，可通过DirectX8.0和Open GL扩展去使用。这些着色器使用类似汇编的语言进行编程并由驱动程序转换为机器码。像素着色器也包含在DirectX8.0中，但SM1.1中的像素着色器并未达到真正意义上的可编程性——十分有限的“编程”的支持，即通过驱动程序，与硬件的“寄存器组合器”连接，去转换纹理贴图的混合状态。这些程序不仅在长度上有限制（小于等于12个指令），而且缺少两个Peercy等人指出的两个关于可编程性极为关键的元素（附属纹理读取和浮点数据）。

这个时期的着色器并不允许流程控制（分支），因此条件句可以以通过计算所有的条件然后选择或对结果进行插值的方式进行模拟。

DirectX定义了着色器模型这样的一个概念去区分拥有不同着色器编程能力的硬件。GeForce3支持顶点着色器模型1.1和像素着色器模型

1.1（着色器模型1.0针对的是硬件故并未公布出来）。在2001年，GPU在通用的像素着色器模型上取得进步。DirectX8.1增加了像素着色器模型1.2和1.4（两者意味着不同的硬件），它们扩展像素着色器的能力，增加了额外的指令和对附属纹理读取更综合的支持。
2002年，DirectX9.0对外公布，其包含着色器模型2.0（并扩展到2.X版本）。这个模型是真正的可编程顶点着色器及像素着色器。相同的功能出现在OpenGL的各种扩展库中。支持任意的附属纹理读取和16位浮点数储存，最终完成了所有Peercy等人在2000年提出的一系列需求。有限的着色器资源例如指令，纹理和寄存器有所增加，使得着色器有能力处理更复杂的效果。增加了流程控制。增长中的着色器代码长度和复杂度使得汇编编程模型变复越来越笨拙和难以处理。幸好，DirectX9.0亦包括了一种新的着色器编程语言，叫HLSL（变级着色语言），HLSL由Microsoft与Nvidia合作研发的，Nvidia亦公开发布了其另一种跨平台的变体，叫Cg。与此同时，OpenGL架构检查委员会（Architecture Review Board）公布与之类似OpenGL版本的语言，叫GLSL（亦被称为GLslang）。这些语言受到了C编程语言的语法和设计理念以及Render Man着色语言的重大影响。
Shader Model 3.0在2004年被引入并逐步改进，将可选的特性转化为模型的需求，更进一步的是增加资源的限值和对纹理读取和顶点着色的有限支持。当新的一代游戏主机在2005年下半年（微软的Xbox360）和2006年（索尼的PS3）被引入时，它们都配置了支持Shader Model 3.0的GPU。固定功能管线并非完全废除：2006年问世的任天堂的Wii游戏主机则使用了固定功能GPU，但几乎可以肯定的是这是最后一款这种类型的主机了，因为甚至移动设备，例如手机亦可使用可编程的着色器了。
其它语言和着色器开发环境也出现了。例如Sh语言允许通过C++库生成和混合GPU着色器。这开源项目可在多个平台上运行。在另一方面，一些可视化编程工具被引入，使得艺术家们（其中大部分的人对类C语言编程不熟悉）可以设计着色器。这样工具包含用于联接预定义着色器代码块的可视图形编辑器，以及将结果图形转换为着色语言（如HISL）的编译器。图3.4展示了一个这类型工具的截图（melatal mill,包含在NVIDIA的FX Cowposer2中）。Mc Guire等人调研了可视着色器编程系统并提出了关于高层次，抽像可扩展性的概念。

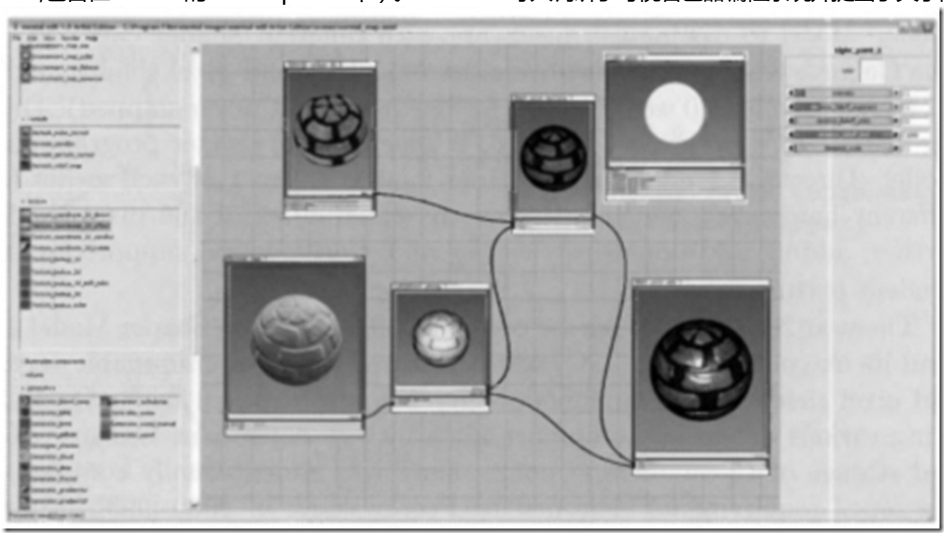

图3.4 一用于着色器设计的可视图形系统。各种操作被封装进左边可选的功能盒中。当选中时，功能盒的可调整参数将展示在右边。每个功能盒的输入和输出将其它联接在一起，共同形成最终结果，即中间窗体右下角的那幅图。（截图来自mental image公司的“mental mill”）
2007年，业界又向可编程方向迈出又一大步。着色器模型4.0（包括DirectX10.0和通过扩展的OpenGL）引入了一些主要特征，例如几何着色器和流输出。
着色器模型4.0包括了一个针对所有着色器（顶点、像素和几何）的统一的编程模型，其公共着色器核心在关面已经描述过了。资源上限进一步增加，并加入了对整形数据类型的支持（包括位运算）。值得注意的是着色器模型4.0仅支持高层语言着色器（DirectX的HLSL和OpenGL的GLSL）——没有像之前的版本那样，提供用户可写的汇编语言接口。
GPU制造商，微软和OpenGL架构委员会一直不断在提升和扩展可编程着色的能力。除了新版本的扩展API,新的编程模型，例如NVIDIA的CUDA和AMD的CTM以非图形应用程序为目标。关于这个运行在GPU上的通用计算，在18.3.1节中有概括的介绍。
3.3.1 着色模型的比较
虽然这章的内容关注在着色模型4.0（写作时的最新版本），开发者经常需要支持使用旧着色模型的硬件。因此，我们对一些最后的着色模型的能力进行一个简单的比较：2.0（和它的扩展版本2.X），3.0和4.0。列出所有的不同点则超越本书的范围；具体的信息可以MSDN和DirectX SDK中获行。
我们在这里集中关注DirectX，因为其清楚明了的版本发布，相比这下OpenGL扩展库的演化，一些是由OpenGL架构检查委员会批准通过的，而另一些则是生产商指定的。这样的扩展系统有一个优势，就是来自指定的独立硬件厂商（Independent Hardware Vendor）的最尖端的特性可以立刻使用。DirectX9以及更早的支持独立硬件厂商的变体版本通过检查“能力比特”去判断GPU是否支持该特性。到了DirectX10，微软急速地从这些做法转向所有独立硬件厂商都必须支持的标准模型。虽然这里集中关注DirectX，但接下来的讨论亦适用于OpenGL,因为相关联的底层GPU在同一时期都具有相同特性。
表3.1比较了各种着色模型的能力。在这个表格中，“VS”代表顶点着色器和“PS”代表像素着色器（着色模型4.0 引入了几何着色器，其能力与顶点着色相似）。如果没有出现“VS”和“PS”，那么该行适用于顶点和像素着色器。因为虚拟机是四路SIMD，每个寄存器可以储存一到四个独立的值。“指令槽位”指代着色器可包含的最大指令数目。“最大执行步长”指代执行指令的最大数目，考虑上分支和循环语句。“临时寄存器”展示了用于储存中间结果的通用寄存器的数目。“常量寄存器”指示了可输入着色器的常量数目。“流程控制，判断”涉及通过分枝指令和判断（条件执行或跳过命令等的能力）去计算条件表达式和执行循环。“纹理贴图”展示了可被着色器访问（每张纹理贴图匀可被多次访问）的distinct纹理（见第六章）。“整数支持”指代其使用位运算操作符和整数算术进行整数类型运算的能力。“VS输入寄存器”展示可被顶点着色器访问的各种输入寄存器的数目。“插值寄存器”是顶点着色器的输出寄存器和像素着色器的输入寄存器。他们被如此称谓因为在传送到像素着色器前顶点着色器的输出值在整个三角形片上进行插值。最后，“PS输出寄存器”展示了可输出像素着色器的寄存器数目——每个均与不同的缓冲区或绘制目标相关联。

	SM 2.0/2.X	SM 3.0	SM 4.0	备注
引入版本	DX 9.0, 2002	DX 9.0c, 2004	DX 10, 2007	a. 最小需求量（或许可使用更多）； b. 最少32纹理和64算术指令； c. 14个常量缓存可供使用（+2私有，预留给微软或硬件厂商），每个都可以包含最大4096个常量； d. 顶点着色器须支持静态流程控制（基于常量）； e. 对于顶点纹理，SM3.0硬件通常的格式种类非常有限并且没有过滤器； f. 高达128个纹理队列，每个都能包含最多512个纹理； g. 不包括两种颜色在有限精度和范围的差值； h. 顶点着色器输出16个插值器，而几何着色器可以扩展至32个
VS指令槽位	256	≥512 a	4096	
VS最大执行步长	65536	65536	∞	
PS指令槽位	≥96 b	≥512 a	≥65536 a	
PS最大执行步长	≥96 b	65536	∞	
临时寄存器	≥12 a	32	4096	
VS常量寄存器	≥256 a	≥256 a	14×4096 c	
PS常量寄存器	32	224	14×4096 c	
流程控制，判断	Optional d	Yes	Yes	
VS纹理贴图	None	4 e	128×512 f	
PS纹理贴图	16	16	128×512 f	
整数支持	No	No	Yes	
VS输入寄存器	16	16	16	
插值寄存器	8 g	10	16 / 32 h	
PS输出寄存器	4	4	8	

表3.1 着色器能力，按DirectX着色模型版本列出。

3.4 顶点着色器

顶点着色器是图3.2中展示的功能管线的第一阶段。因为这是所有图形处理的第一个阶段，在此阶段前的数据处理是毫无意义的。这在DirectX中称为输入装配，许多的数据流组合在一起形成一个顶点和图元组成的集合，发送到管线中。例如，一对像可由一位置坐标数组和一个颜色数组所代表。输入装配会基于位置和颜色生成该对像的三角形面（或点、线）。另一个对像可以由同一个位置数组（随不同的模型转换矩阵）和不同的颜色数组去代表。数据表现的细节将在12.4.5节讨论。在输入装配中亦支持实例化。这允许通过一个单独的绘制调用，对一个对像的不同实例使用不同的数据进行多次绘制。关于实例化的使用参见15.4.2节。DirectX10的输入装配功使用标识数标记了每个实例、图元和顶点，使得后续的任何着色器阶段都可以访问。对于较早期的着色器模型，这种数据需要明确地加入模型。

一个三角片网格是由一个顶点集合和描述顶点所属三角形的付加信息所组成。顶点着色器是处理三角形网格的第一个阶段。顶点着色器未能得到关于三角形所组成什么的信息；像名字所暗示的，它只处理输入进来的顶点。从总体上说，顶点着色器提供一个修改、创建或忽略那些与每个多边形顶点的值（例如其颜色，法向量，纹理坐标和位置）的方法。通常顶点着色程序一般将顶点从模型空间转换到齐次裁空间；一个顶点着色器至少也必须输出它的位置。

这个功能第一次在2001年的DirectX8中引入。因为它是管线的第一个阶段而且调用的频率不高，它可以实现在GPU或CPU上，然后将数据发送到GPU做光栅处理。这使得由旧硬件向新硬件的转换仅是速度的问题，而非功能性的问题。现在所有的GPU产品都支持顶点着色器。

顶点着色器本身与3.2节中描述的公共核心虚拟机十分相似。每个传入的顶点会被顶点着色器程序处理，然后输出许多在三角形或线上插值得出的数值。顶点着色器既不能创造出不能消毁顶点。因为每个顶点是被独立处理的，任何数目的着色器处理器在GPU上均可并行处理输入的顶点流。

下面列出解释顶点着色器效果的章节，例如阴影体的创建，用于动画连接的顶点混合和轮廓的渲染。其它关于顶点着色器的用法包括：

·棱镜效果，使得屏幕出现鱼眼，水下或其它扭曲效果。

·通过仅一次的网格创建并由顶点着色器变形定义对象。

·对像的扭曲，弯曲和雏化操作。

·程序化的变形，例如旗帜、服装或水的移动。

·图元创建，by sending degenerate meshes down the pipeline and having these be given an area as needed. .这个功能在新的GPU中由几何着色器所到代。

·页的卷曲，热天的雾气，水的涟漪以及其特效可以通过将整个帧缓存的内容当做屏幕的纹理与正在进行程序化变形的网对齐。

·顶点纹理获取（在SM3.0之后可获取）可用于将纹理贴图应用在顶点网格，使得海洋表面和地形高度场的应用开销不再昂贵。

图3.5展示了一些使用顶点着色器的变形效果。

图3.5在左边，是一个正常的茶壶。顶点着色器运行一简单的剪切操作产生了中间的图片。在右边，噪声函数产生一场扭曲了模型。（图片由FX Composer 2产生）

顶点着色器的输出数据可以有多种不同的使用方式。对于每个实例的三角形通常的路径是生成和光珊化，并将每个独立的像素片段发送到像素着色程序去继续处理。在着色模型4.0引入后，数据亦可发送到几何着色器中，以流输出，或两种楷可。这些选项是下一节的内容。

3.5 几何着色器

2006年末发布的DirectX10将几何着色器加入了硬件加速图形管线。在管线中，它紧随顶点着色器之后，并且是可选。它是着色模型4.0的一部分，但在之前的着色模型并不存在。

几何着色器的输入是一单独的对像和对像所关联的顶点。对像通常是一网格中的三角形，一直线的线段或简单的一个点。此外，扩展的图

元可以被几何着色器定义和处理。特别的是，三个额外的在三角形外面的顶被传入，并且两个相邻的在多边形上的顶亦被使用。见图3.6。

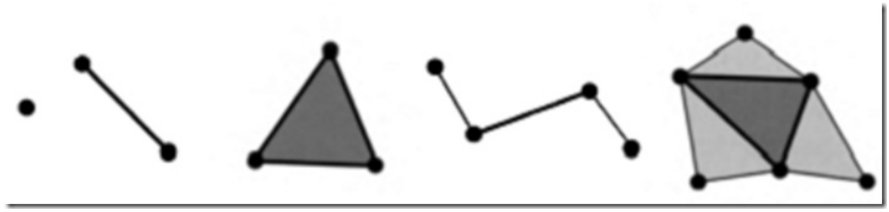

图3.6 几何着色器程序的输入是一个单独的类型：点，线段，三角形。两个最右边的图元，包括与线和三角形对象相邻的顶点也可被使用。

几何着色器处理这些图元并输出零个或更多的图元。输出的形式为点，多边形和三角形条带。例如一次单独调用几何着色程序的输出可以输出多个三角形条带。同样重要的，几何着色器亦可生成没有输出的结果。通过这个方法，网格可以被选择性地通过编辑顶点，增加新图元和移除其它单元等方法修改。

几何着色器程序输入一类对象然后输出另一类对像，这两类对像并不需要相匹配。例如，输入可以是三角形而它们的形心可以以点的形式，对应每一个三角形地输出。即便输入和输出的对像类型是相匹配的，其每个顶点所携带的数据亦有可能被忽略或扩展。例如，三角形平面的法向量可以被计算，并加入到每个顶点数据中。与顶点着色器相似的是，几何着色器必须将每一个顶点输出到齐次裁剪空间的位置上。

几何着色器保证其输出的结果图元的顺序与输入的是一样的。这影响到其运行效率，因为许多的着色器单元是并行运行的，其结果必须保存并排序。为了兼顾功能和效率，着色模型4.0中存在一个每次执行只能生成1024个32位数值的限制。所以以单个叶子作为输入去生成成千上万的灌木丛叶子是不可行，也不推荐的几何着色器使用方式。使用曲面细分将简单的平面细分为更精细的三角形网格在这里也是不被推荐的。这个阶段更多是关于以程序方式修改输入数据或进行数量有限的复制，而非大量地复制或放大细化它。例如，一个使用是生成六个经座标变换的数据副本似便同时渲染六个面的立方体贴图，详见8.4.3节。附加的算法可利用几何着色器实现包括从点数据生成各种粒子，沿轮廓挤压出鳍用于毛发渲染，以及用于阴影算法的对象边界查找。图3.7有更多这方面的应用。这些和其它的应用将在书本剩余部分讨论

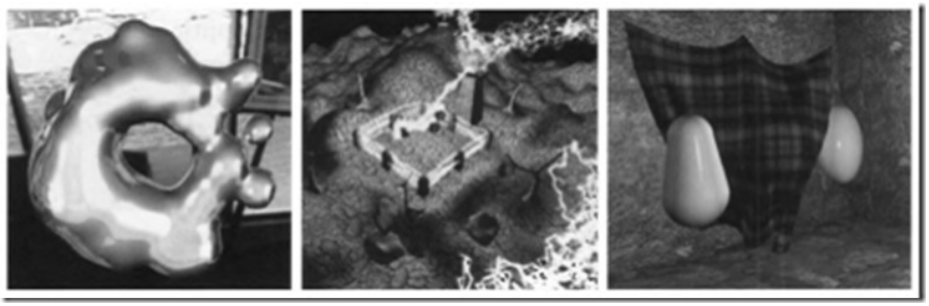

图3.7 一些几何着色器的应用。在左边，元球的等值面曲面细分运行在几何着色器上。在中间，线段细分的分形使用了几何着色器和流输出，并且几何着色器生成用于展示的广告板。在右边，布料模拟使用了顶点和几何着色器的流输出。（图片来自NVIDIA SDK 10的例子）

3.5.1 流输出

GPU管线的标准使用方式是发送数据到顶点着色器，然后对结果的三角形进行光栅化并在像素着色器中处理它们。数据总是穿过整个管线而中间结果不能被访问。流输出是在着色模型4.0中引入。在顶点经过顶点着色器的处理后（及选择的几何着色器），除发送光栅阶段外，可以输出到流，也就是一个有序数组。实际上，光栅化可以整个被关闭，而管线可被当作非图形的流处理器被应用，数据以这种方式处理可以通过管道回传，那么可以迭代地处理。这种操作对模拟流动的水体或粒子效果特别有用，将在10.7节中讨论。

3.6 像素着色器

在顶点和几何着色器运行完他们的操作后，图元像上一章所说明的那样被裁剪并设置（set up）为光栅化准备。管线这部分的处理步骤是相对固定的，不可编程的。每个三角形遍历而顶点上的数值将在三角形上插着。像素着色器是另外一个可编程阶段。在OpenGL，这个阶段称为片段着色器（fragment shader），一个某程度上更好的名字。其思想是一个三角形完全或部分覆盖每个像素单元，而材质绘制成透明或不透明的。光栅器生成一些数据（数据在一定程度上描述三角形如何覆盖像素单元），而不会直接影响像素所储存的颜色。在接下来的融合阶段，这些片段数据用于修改像素里面储存的数据。

顶点着色程序的输出实际上变成了像素着色程序的输入。在Shader Model 4.0中，总计16个向量（每个有四个值）可以由顶点着色器传送到像素着色器。当使用几何着色器后，他可以向像素着色器输出32个向量。

像素着色器的追加输入是在Shader Model 3.0中引入的。例如，三角形的那一面是可视的是通过增加输入标志。这个知识对于在一条独立路径中对三角形的前后面渲染不同材质是十分重要。像素着色器也可以获得片段的屏幕位置。

像素着色器的局限是它只能影响它所处理的片段。就是说，当像素着色器运行时，它不能将结果直接发送给相邻的像素。恰恰相反，它使用经顶点插着而得的数据，以及任何储存的常量和纹理数据，计算得出的结果仅会影响一个单独的像素。然和，这个限制并不像它听起来那么严重。邻近的像素可以通过图像处理技术最终被影响，在10.9节有相应描述。

一个像素着色器可以访问邻近像素（虽然不是直接）的例子是计算梯度或派生信息。像素着色器可以取任何计算值并计算其数值，该数值可以根据每个像素在屏幕上的x和y坐标变化。这对各种计算和纹理寻址都十分有用。这些梯度是十分重要的操作，例如过滤（详见6.2.2）。大多数GPU通过一2×2或更多的分组处理像素实现这个功能。当像素着色器请求一个梯度值，相邻像素之间的差值被返回。这样实现的一个结果是梯度信息不能访问，因为着色器被动态流程控制影响——所有在同一组的像素必须被同样的指令处理。这是存在于离线渲染系统的基础性的局限。访问梯度信息是像素着色器独有的能力，是其他着色器所不具备的。

像素着色程序通常设置用于最后融合阶段的片段颜色。由光栅阶段生成的深度值同样可以被像素着色器修改。模板缓存的值是不可修改，而是直接送到融合阶段。在SM2.0以及后续版本，像素着色器可以丢弃输入的片段数据，也就是没有输出。这样的操作会花费性能，因为通常由GPU运行的优化不能够使用。详见18.3.7节。一些操作，例如雾计算和α测试，在SM4.0中已经由属于融合的操作转到属于像素着色器的计算。

现在的像素着色器可以做大量的处理。计算在一条渲染路径上的多个数值的能力引出了多渲染目标（Multiple Render Targets）的概念。不是将像素着色器程序的计算结果保存到单独的颜色缓存，而是可以为每个片段生成多个向量并保存到不同的缓存中。这些缓存必须是维度相同，并且一些架构还要求它们每一个都有相同的位深度（虽然根据需要为不同格式）。在表3.1中的PS输出寄存器数目指的是单

独缓存访问数量，即4或8。与可显示的颜色缓存不同，对于任何附加目标都有一定的限制。例如，普遍不能运行反锯齿。即使存在这些限制，MRT功能仍旧是为高效运行渲染算法提供强大帮助。如果若干中间结果图像从同一数据集合中计算出来，只需要一个单独的渲染路径，而不是一个路径对应每个输出缓存。另一项与MRT相关的关键能力是以纹理的方式读取结果图片的能力。

3.7 融合阶段

像2.4.4节讨论的，融合阶段是个体片段的深度和颜色（在像素着色器中生成）与帧缓存结合。这个阶段发生模板缓存与深度缓存运算。在这个阶段发生的另一个操作是颜色的混合，通常用于透明和影像合成（见5.7节）。

融合阶段处于固定功能阶段，例如裁剪，和完全可编程的着色器阶段之间一个非常有趣的中间点。虽然它不是可编程的，但他的操作是高度可配置的。尤其是颜色混合可以运行大量不同的操作。最普遍的是乘法，加法和减法的结合，并涉及颜色值和α值，但其他操作也是可能的，例如最小和最大值，以及位的逻辑运算。DirectX 10增加了将两种来自像素着色器的颜色与帧缓存的颜色混合的功能——叫做二重颜色混合（dual-color blending）。

如果使用了MRT功能，那么混合可以运行在多个缓存上。DirectX 10.1引入了在每个MRT缓存上运行不同混合操作的机制。在之前的版本，所有的缓存总是运行相同的混合操作（注意的是二重颜色混合与MRT不相容）。

3.8 效果

这趟管线之旅到现在为止主要关注各种可编程阶段。顶点、几何和像素着色器控制着这些阶段，他们并不是存在于真空中。首先，一个单独的着色器程序在孤立状态下作用不大：顶点着色器程序将它的结果传送给像素着色器。任何工作的完成都需要加载所有的程序。程序员必须确保顶点着色器的输出与像素着色器的输入相吻合。一个特定的渲染效果可能由多个着色器在数个渲染路径上执行而产生。在着色器本身之外，状态变量有时在一些特殊的配置也必须设置，以使得这些着色器能够正常工作。例如，渲染器的状态包括每次渲染Z缓存和模板缓存是否和如何使用，一片断如何影响已存在的像素的值（例如替换，增加或混合）。

因为这些原因，不同的团体已经研发出不同的效果语言，例如HLSL FX，CgFX，以及COLLADA FX。一个效果文件通常试图包含所有执行一种特定图形算法的所有相关信息。它通常定义一些可被应用程序赋值的全局参数。例如，一个独立的效果文件可以定义用于渲染拟真塑料材质的顶点着色器和像素着色器。它可以提供一些参数，例如塑料的颜色和粗糙程度，使得虽然使用同一个效果文件，但这些参数在每个模型渲染时都可以变化。

为了展示效果文件的受欢迎程度，我们将会由头到尾看一遍一个来自NVIDIA FX Composer 2效果系统的简洁例子。这个DirectX 9 HLSL效果文件实现了一个非常简单的Gooch着色。Gooch着色中，表面的法线与光源位置被使用。如果法向量指向光源，表面被涂上暖和的色调；如果它指向背离光源，冷色调将被使用。两者间的角度在这两种用户自定义颜色间插值。这种着色技术是一种非照片级真实的渲染方式，这是第11章的内容。图3.8展示了这个效果运行的例子。

图3.8 Gooch着色，从暖和的橙色变化为冷的蓝色。

效果变量在效果文件的开始处定义。最初几个变量是不能宁在一起的，这些参数都与因为效果而被自动跟踪的摄像机位置有关：

```
float4x4 WorldXf   : World;
float4x4 WorldITXf : WorldInverseTranspose;
float4x4 WvpXf     : WorldViewProjection;
```

其语法是“类型 变量标示符 ： 语意”。类型floa4x4是用于矩阵，名称是用户定义的，而语意是一个内置的名称。正如名字所暗示的，WorldXf是模型到世界的转换矩阵，WorldITXf是逆转置矩阵，而WvpXf是模型到摄像机裁剪空间的转换矩阵。这些带有公认语意的数值将由应用程序提供，而不是显示在用户接口。

接下来，指定用户定义变量：

```
float3 Lamp0Pos : Position <
    string Object =  “PointLight0”;
    string UIName =  “Lamp 0 Position”;
    string Space =  “World”;
> = {-0.5f, 2.0f, 0.15f};

float3 WarmColor <
    string UIName =  “Gooch Warm Tone”;
    string UIWidget =  “Color”;
> = {-1.3f, 0.9f, 0.15f};

float3 CoolColor <
    string UIName =  “Gooch Cool Tone”;
    string UIWidget =  “Color”;
```

```
> = {0.05f, 0.05f, 0.6f};
```

在尖括号“<>”里面还提供了一些附加的注释，然后赋予默认值。注释是特定于应用程序的，对效果或着色器编译器没有任何意义。这些注释可以被应用程序查询。在此情况下，注释描述如何在用户接口中显示这些变量。
接下来定义着色器输入输出的数据结构：

```
struct appdata {
    float3 Position : POSITION;
    float3 Normal : NORMAL;
};

struct vertexOutput {
    float4 HPosition   : POSITION;
    float3 LightVec    : TEXCOORD1;
    float3 WorldNormal : TEXCOORD2;
};
```

appdata定义了在模型中每个顶点的数据并由此定义了顶点着色器程序的输入数据。vertexOutput则是顶点着色器的产物和像素着色器所需的消耗。将“TEXCOORD*”用作输出名字是工件进化的管道（The use of TEXCOORD* as the output names is an artifact of the evolution of the pipeline.）首先，多重纹理可以被附加到表面上，因此这些附加的数据字段被叫做纹理坐标。在现在中，这些字段持有任何从顶点着色器传递到像素着色器的数据。
接下来将定义各种着色器程序代码的元素。我们只有一个顶点着色器程序：

```
vertexOutput std_VS(appdata IN){
    vertexOutput OUT;
    float4 No = float4(IN.Normal, 0);
    OUT.WorldNormal = mul(No, WorldITXf).xyz;
    float4 Po = float4(IN.Position, 1);
    float4 Pw = mul(Po, WorldXf);
    OUT.LightVec = (Lamp0Pos - Pw.xyz);
    OUT.HPosition = mul(Po, WvpXf);
    return OUT;
};
```

这个程序首先使用矩阵相乘计算了世界空间下表面的法线。转换是下一章的主题，所以我们不会解析为什么这里使用了逆转置矩阵。世界空间下的位置也是通过离屏转换计算的。这个定位是从光源位置中减去从表面到光源的方向向量而得。最后，对象的位置被转换到裁剪空间，用于光栅器。这是一个任何顶点着色器程序所必须的输出。
在世界空间上给出光的方向和表面法线，像素着色器程序计算表面颜色：

```
float4 gooch_PS(vertexOutput IN) : COLOR
{
    float3 Ln = normalize(IN.LightVec);
    float3 Nn = normalize(IN.WorldNormal);
    float ldn = dot(Ln, Nn);
    float mixer = 0.5 * (ldn + 1.0);
    float4 result = lerp(CoolColor, WarmColor, mixer);
    Return result;
}
```

Ln向量是单位化的光照方向，而Nn是单位化的表面法向量。通过单位化，这两个向量的点积ldn表示它们之间夹角的余弦。我们希望在冷和暖色调之间的差值中使用这个数值。函数lerp()产生一个介于0和1之间的值，0代表使用CoolColor，1代表WarmColor，而之间的数值由它们俩混合而成。因为角度余弦的取值范围是[-1,1]，其混合数值转换至[0,1]的范围。这个数值用于混合色调并产生带有合适颜色的片段。这些着色器都是函数。一个效果文件可以由任意函数组成并且可以包含来自其他效果文件的公共函数。
一个路径（pass）通常由定点和像素（和几何）着色器组成，以及任何路径需要的状态设置。一个手法（technique）是一个包含一个或多个路径，用于产生预期效果的集合。这个简单的效果文件拥有一个有一条路径的手法：

```
technique Gooch< string Script =  “Pass=p0;” ; > {
  pass p0 < string Script =  “Draw=geometry;” ; > {
    VertexShader = compile vs_2_0 std_VS();
    PixelShader = compile ps_2_a gooch_PS();
    ZEnable = true;
    ZWriteEnable = true;
    ZFunc = LessEqual;
    AlphaBlendEnable = false;
  }
```

}

这些状态设置使得Z缓存能够以正常的方式（可读可写）被使用，并且通过深度小于或等于所存储深度值的片段。Alpha混合则是关闭的，因为使用这种手法的模型是假定不透明的。这些规定意味着如果片段深度是与Z缓存所存数值相等或者接近，那么计算后的片段颜色将用于替换相应像素的颜色。用另一个说法，使用的是Z缓存的标准用法。

同一份效果文件可以存储若干个手法。这些手法通常是同一个效果的变体，每个都以不同的着色器模型为目标（例如SM2.0与SM3.0）。效果可以是范围极广。图3.9给出了现代可编程着色器管线强大功能的体验。一个效果通常封装了相关的手法。多种管理着色器集合的方法亦以被开发出来。

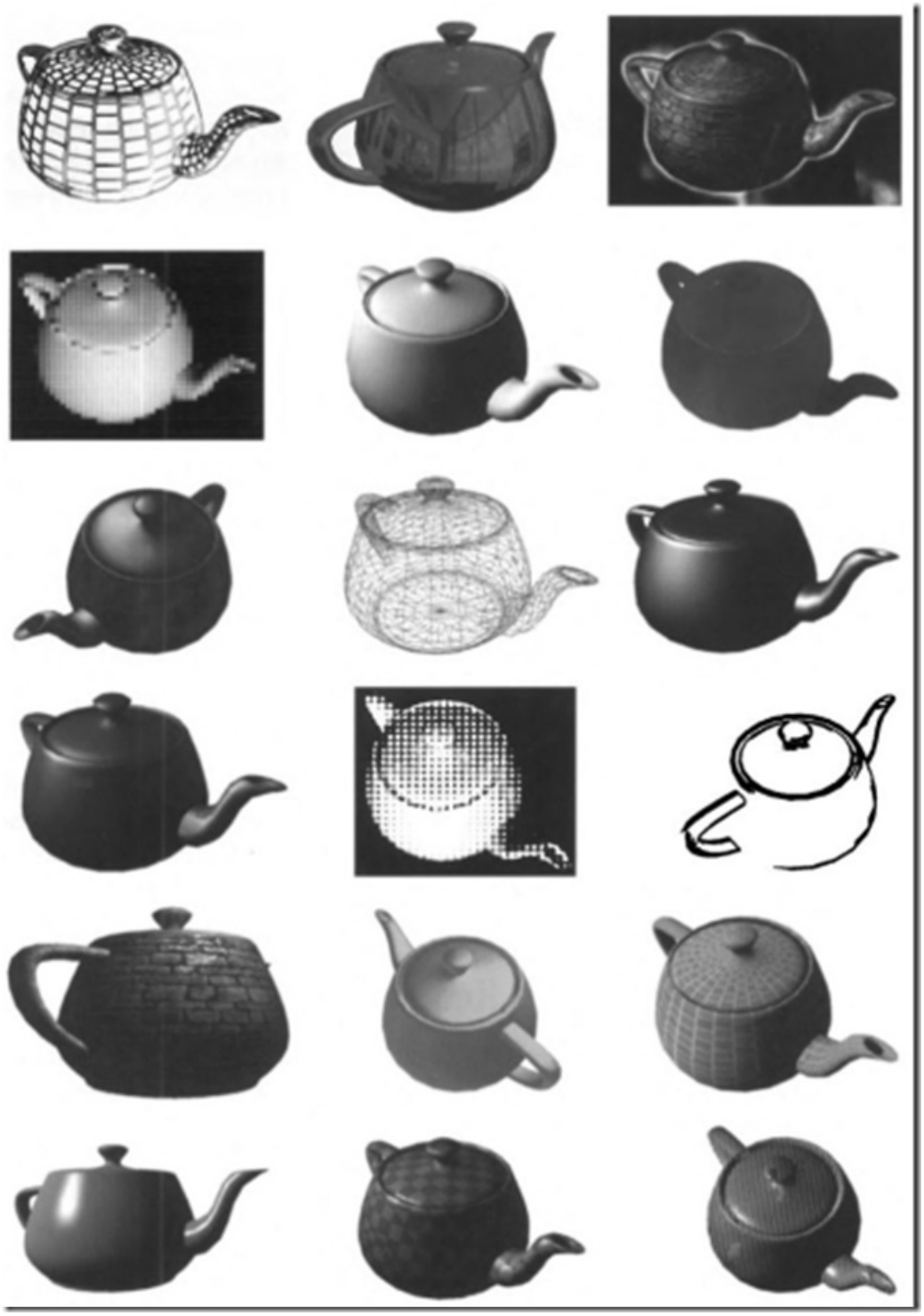

图3.9 可编程着色器使各种各样的材料和后处理效果变得可能。

我们已经在GPU旅程的终点。GPU还可以做的其他很多的，并且有很多使用和结合它功能的方法。相应的利用这些能力的理论和算法是本书的中心主题。有了这些基础知识，关注点将转移到给出对转换和视觉呈现，这些管线关键单元的，深入的理解上。

进一步阅读及资源

David Blythe的关于DirectX 10的论文对现代GPU管线和其设计背后原理有一个非常好的概述，以及相应的参考文献。

单独关于顶点和像素着色器的信息就可以很容易地写出一本书。我们的最佳建议是投入其中：访问ATI和NVIDIA开发者的网站获取最新技术的信息。它们免费的FX Composer 2和RenderMonkey互动的着色器设计工具套件提供了一个优秀的方法去尝试、修改着色器，去看是什么让它们改变。Sander提供了一个固定功能管线在HLSL（用于具有SM2.0能力硬件）上的实现。

正式地学习着色器编程的方方面面需要做更多工作。《OpenGL Shading Language》将红宝书遗漏的部分补上，描述了GLSL——OpenGL的可编程着色语言。为了适应HLSL，DirectX API的每个新版本都不断进化；在它们SDK之上的相关的链接和书籍可见于本书的网站（http://www.realtimerendering.com）。O’Rorke的文章提供了关于效果的易读的介绍和管理着色器的有效办法。Gg语言提供

了一层抽象，输出到很多主要的API和平台，并且也为主要的建模和动画程序提供了插件工具。Sh元程序语言是更为抽象，本质上作为一个C++库那样工作，去将相应的图形代码映射到GPU上。
对于更高级的着色器技术，可以从阅读《GPU精粹》和《ShaderX》系列书籍作为开始。《游戏编程精粹》上也有一些相关的文章。
DirectX SDK也有很多重要的着色器和算法例子。