3.5几何着色器

几何着色器随着DirectX10在2006年后半叶的发布而加入到硬件加速的图形管线中。它在管线中紧接着顶点着色器之后，并且是否启用它是可选的。虽然它是SM4.0所要求的部分之一，但在之前的SM中并没有使用过。

几何着色器的输入是一个独立的对象以及与它相关的顶点数据。这个对象一般是网格中的一个三角形，一条线段或一个简单的点。另外，几何着色器也可以定义和处理一些额外的片元。比如，可以传入三角形以外的三个附加顶点，或者折线上的两个相邻顶点也是可以使用的。见图3.6。

几何着色器会处理这个片元并且输出0个或0个以上片元。输出的格式是点，折线以及三角形带。举例来说，一个单独的几何着色器程序的指令，可以输出不止一个三角形条带。同样重要的是，几何着色器也可以完全没有任何输出。通过这种方式，一个网格可以选择性地被修改，通过编辑顶点，增加新的片元，以及移除一些东西。

几何着色器被设定为输入一个类型的对象并且输出一个类型的对象，但这些对象并不需要有所联系。比如，可以输入一些三角形然后每个输入三角形的重心当作点被输出出来。即使输入输出类型是绑定的，关联在每个顶点上的数据也可以被舍弃或拓展。比如，一个三角面的法线可以被计算出来然后附加到每个输出顶点的数据中。类似于顶点着色器，几何着色器必须为每个产出的顶点输出它们在齐次裁剪空间下的位置。

几何着色器保证来自片元的输出结果的顺序与输入时的顺序相同。这非常影响效率，因为如果一大堆着色器单元是并发执行的，那它们的结果就必须被储存并排序。作为效果和性能的折中，在SM4.0中添加了一个限制，每次计算最多只能生成总共1024个32bit的值。因此，在几何着色器阶段由一个输入叶片生成一千丛树叶的做法是不可行也不被推荐的。使用简单的图形棋盘式镶嵌生成复杂网格也同样是不推荐的。这个阶段更多地是程序化操作修改输入进来的数据或是有所节制地做一些复制，而不是大量地复制和扩大。举个例子，一种用法是按顺序生成六个转换后的数据副本，来同时渲染cube map的六个表面，见8.4.3节。另外有一些算法可以运用几何着色器的优势，包括从一个点数据生成各种尺寸的粒子，沿着轮廓线挤压表面来渲染毛皮，以及为阴影算法寻找对象的边缘。通过图3.7了解更多。这些以及一些额外的应用将会在书中的其他部分得到详细讲解。

3.5.1 流式输出

GPU管线的标准应用就是通过顶点着色器发送数据，然后光栅化得到的结果三角形并在像素着色器中进行处理。这些数据往往贯穿整个管线并且中间结果无法被存取。“流式输出”的概念在SM4.0被引入。当顶点被顶点着色器处理完毕之后（或者，可选的，几何着色器），它们就可以被输出进一个流，也就是，一个有序数组，在被传送进光栅化阶段的基础上。实际上，光栅化可以被完全关闭，从而管线就可以完全变成一个非图形学相关的流处理器。通过这种方式处理的数据可以通过管道返回，因此允许迭代处理。这种类型的运算在模拟流动水体或者其他粒子特效师异常有效，详见10.7节。

3.6 像素着色器

在顶点和几何着色器执行完他们的运算之后，片元被裁剪并为光栅化做准备，就像上一章所述的那样。管线中的这一节在它的处理步骤中是相对固定的，不可编程的。每个三角形都被遍历，并且顶点处的值在三角形的范围内进行插值。像素着色器是下一个可编程的部分。在OpenGL中这一环节被称为片元着色器(fragment shader)，这在某些角度来看是一个更好的名字。这个定义是说一个三角形完全或部分覆盖的每个像素单元，以及材料被描述为透明还是非透明的。光栅处理器并不直接影响像素所存储的颜色，而是生成数据，这些数据或多或少地反映了三角形覆盖像素单元的范围。然后在合并阶段，这些片元数据被用来修改储存在像素中的值。

顶点着色器的输出很快就变为像素着色器的输入。在SM4.0中从顶点着色器总共可以传递16个向量（每个包含4个值）到像素着色器。当几何着色器启用时，它可以输出32个向量到像素着色器。

额外的输入是在SM3.0时为像素着色器特别加入的。比如，三角形的哪一条边是可见的作为一个输入标记而被加入。这个知识在处理只用一个pass渲染每个三角形正面材质不同于背面时非常重要。片元在屏幕中的位置对于像素着色器来说也是可获得的。

像素着色器的限制是它只能影响传递给它的片元。这也就是说，当一个片元着色器程序运行时，它无法直接传递它的结果给它相邻的像素。相反，它使用顶点插值得来的数据，以及所有存储着得常量和图片数据，来计算只会影响一个像素的结果。然而，这个限制并不像它听起来那么严重。相邻的像素可以通过图像处理技术最终得到改变，详见10.9节。

像素着色器可以获得相邻像素信息的一种情况（尽管是间接的）是通过梯度或导数信息进行计算。像素着色器可以获得任意的值并计算它在沿屏幕x轴和y轴每个像素的变化量。这对于各种计算和纹理寻址都非常有用。这些梯度对于如过滤（见6.2.2节）这样的计算特别重要。大部分GPU通过在2x2或更多的像素组成的分组中计算来实现这一特性。当像素着色器请求一个梯度值时，相邻像素之前的差将被返回。这种实现的一个结果是梯度信息无法在受到动态流控制影响的着色器中使用——一个组中的所有像素必须使用相同的指令处理。这是一个根本的限制，它甚至也存在于离线渲染系统中。使用梯度信息是像素着色器独有的功能，并不是所有可编程着色器都有的。

像素着色器程序一般会为进行最后融合阶段的融合操作而设置片元颜色。光栅化阶段生成得深度值同样也可以在像素着色器中进行修改。模板缓存的值是不可修改的，而是传递给融合阶段。在SM2.0及之后的版本中，像素着色器还可以取缔传入得片元数据，也就是说，取消输出。这些运算会降低效率，因为可能会导致在之后通常的GPU优化无法使用。详见18.3.7节。诸如雾效计算和透明度测试等计算在SM4.0中已经从作为合并的可选项转变为像素着色器计算。

如今的像素着色器已经可以进行非常大量的运算。在一个单独的渲染pass中计算任意数量的值的能力使得“多重渲染目标”（multiple render targets-MRT）的想法产生了。每个片元都可以产生多个向量并存储在不同的缓存中，而不再是把一个像素着色器程序的结果存储在一个单独的颜色缓存中。这些缓存必须具有相同的维数，有些架构要求它们还要有相同的位深度（bit depth）（尽管根据需要可以使用不同的格式）。表3.1中像素着色器的输出寄存器的数量就是指可使用的独立缓存的数量，也就是4个或8个。不像可展示的颜色缓存，这些附加的目标都有一些不同的限制。比如，一般来说无法进行抗锯齿。然而即使有着这些限制，MRT根本上在使渲染算法更有效率方面还是一个非常强有力的帮手。如果要从相同的数据设置中计算产生多个中间结果图像，就只需要一个渲染pass即可，而不再是为每个输出缓存对应一个pass。MRT带来的另一个关键功能是从这些结果图像中将数据作为纹理读取出来的能力。

3.7 融合阶段

如2.4.4节所述，融合阶段是独立片元深度值和颜色值（像素着色器所产生的）通过帧缓存合并的地方。这是模板缓存和z缓存运算所存在的阶段。另外一个在此阶段占有一席之地的运算是颜色混合，也就是常用于透明和合成操作的计算（见5.7节）。

融合阶段存在于固定程序阶段，比如裁剪，以及完全可编程着色器阶段的奇妙中介点。虽然它是不可编程的，但是它的运算确实高度可自定义的。颜色混合尤其适合用于大量不同的运算。最常见的是包含颜色值和alpha值的乘法、加法以及减法的混合运算，但是其他运算也是可以的，比如最小值和最大值，以及按位的逻辑运算。DirectX加入了在像素着色器中运用帧缓存中的颜色值混合两种颜色的能力——这个能力被称为“双色混合”。

如果使用了MRT功能，那么混合就可以使用在多个缓存之间。DirectX 10.1 引入了可以在任意不同的MRT缓存之间进行混合的能力。在之前的版本中，总是在所有的缓冲区中执行相同的混合操作（注意，双色混合与MRT不能兼容）。

3.8 特效

这趟管线之旅的重点一定是聚焦于这些个可编程阶段。虽然顶点、几何和像素着色器程序对于这些阶段的控制是必须的，但它们并非处于真空之中。首先，一个处于孤立状态下的单独的着色器程序并不是特别有用：一个顶点着色器程序把它的结果喂给一个像素着色器。两边的程序都必须为了所做的每个工作而被加载。程序员必须设计一些从顶点着色器的输出到像素着色器输入的匹配。除了这些着色器程序本身，有时为了这些程序的正确运行，状态变量必须被配置在一些特别的配置中。举个例子，渲染的状态，包括z缓存和模板缓存是否以及如何被使用，以及一个片元如何影响当下存在的像素值（例如，替换，相加或是混合）。

出于这些原因，各大组织就开发出了特效语言，比如HLSL FX, CgFX, 以及COLLADA FX。一个特效文件试图封装所有运行一个特殊渲染算法所需要的相关信息。它一般定义了一些可以由应用进行赋值的全局参数。举个例子，一个特效文件可能定义了用于渲染逼真塑料材质所需要的顶点和像素着色器。它可能暴露了如塑料颜色和粗糙度这样的可以为渲染每个不同模型而改变的参数，只是使用相同的特效文件。

为了展示特效文件的风味，我们会体验一个取自NVIDIA’s FX Composer 2特效系统的精简过的实例。这个DirectX 9 HLSL特效文件实现了一个非常精简格式的“古氏渲染”（Gooch shading）。古氏渲染中的一部分是使用表面法线与光源位置进行比较。如果法线方向指向光源，那么表面的颜色就会使用一个温暖的色调；如果它指飞了，那么就会使用一个冷色调。中间的角度则会在用户定义的两种颜色间进行插值。这种着色技术是非真实渲染的一种，即第11章的主题。图3.8展示了一个这个特效的实际效果案例。

特效变量被定义在特效文件的开头。前几个变量是“不可调整的”，这些参数与相机位置对特效的自动跟随相关：

float4x4 WorldXf : World;

float4x4 WorldITXf : WorldInverseTranspose;

float4x4 WvpXf : WorldViewProjection;

这里的语法格式是“type id : semantic”。类型float4x4用于矩阵，命名是用户定义的，然后语义是一个内置的命名。就像语义名所暗示的，“WorldXf”是模型到世界的转换矩阵，“WorldITXf”是这个矩阵的逆矩阵，然后“WvpXf”是从模型空间到相机的裁剪空间的转换矩阵。这些具有可识别语义的值应该由程序提供而不被展示在用户界面中。

紧接着，用户定义的变量被指定：

float3 Lamp0Pos : Position <

string Object = "PointLight0";

string UIName = "Lamp 0 Position";

string Space = "World";

> = {-0.5f, 2.0f, 1.25f};

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};

这里有一些额外的注解在尖括号“<>”中接着是指定的默认值。这些注解是应用程序专用的而对于特效和着色器的计算没有任何意义。这些注解可以被应用程序所查询。因此这些个注解描述了如何通过用户界面暴露这些变量。

着色器输入和输出的数据结构体在这之后被定义：

struct appdata {

float3 Position : POSITION;

float3 Normal : NORMAL;

};

struct vertexOutput {

float4 HPosition : POSITION;

float3 LightVec : TEXCOORD1;

float3 WorldNormal : TEXCOORD2;

};

Appdata定义了模型上的每个顶点所具有的数据也就定义了顶点着色器程序的输入数据。vertexOutput是顶点着色器产出然后像素着色器消费用的。TEXCOORD\*作为输出名的用法是管线进化的人工产物。首先，多个纹理可以被贴在一个表面上，这些额外的数据域被称为“纹理坐标”（texture coordinate）。事实上，这些区间存储着从顶点到像素着色器传递的所有数据。

接着，这些个着色器程序的代码成员们被定义。我们此处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;

}

这个程序首先用一个矩阵乘法计算了表面法线在世界空间下的表示。转换是下一章的主题，所以我们不会展开解释为什么此处会使用逆转换矩阵。世界空间的位置也通过屏幕外转换来计算。由光源位置减去这个坐标来获得从表面到光源的方向向量。最后，对象的位置被转换进裁剪空间，以用于光栅化器。这是每个顶点着色器都需要的一个输出。

提供了世界空间下的光的方向和表面法线，像素着色器程序计算了表面颜色：

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;

}

向量Ln是归一化后的光照方向Nn是归一化后的表面法线。通过归一化，这两个向量的点成结果ldn就代表了两者夹角的余弦值。我们希望可以线性地在冷暖色调间用这个值进行插值。函数lerp()期望拿到一个0到1之间的mixer值，当它取到0时就使用CoolColor，取到1时使用WarmColor，在两者之间时则混合两者。尽管一个角度的余弦值的取值范围是[-1, 1]，但mixer的值被转换到区间[0, 1]。这个值紧接着被用于混合色调并且产生适当颜色的片元。这些着色器是函数。一个特效文件可以由任意数量的函数组成并且可以包含其他特效文件中的常用函数。

一个pass一般由一个顶点和一个像素（和一个几何）着色器组成，连同所有这个pass需要的状态设置一起。一个technique是一组由一个或多个pass组成的用于产生需求的效果的东西。这个简单的文件拥有一个technique，而这个technique拥有一个pass：

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;

}

}

这些状态设置强制z缓存按正常的方式来被使用——可以被读写，并且当片元的深度值小于或等于储存着的z深度值时被才被传递。Alpha混合被关闭，因为使用这个technique的模型被假设为不透明的。这些规定意味着一旦片元的z深度相对于存储值等于或更接近时，运算得到的片元颜色将被用于替换掉相应的像素颜色。换句话说，使用标准的z缓存用法。

多个technique可以被存储在同一个特效文件中。这些technique往往是相同效果的变体，每一个对应不同的目标着色模型（比如SM2.0对SM3.0）。很多很多特效是可实现的。图3.9只是给出了一个对于现代可编程着色管线能力的尝试。一个特效通常封装了相关的techniques。各种各样的方法被开发出来管理着色器的设置。

我们已经到了GPU之旅的尾声。GPU可以做许多其他的事情，以及许多使用和合并它的函数的方法。协调相关的原理和算法来利用这些功能的优势是本书的中心主题。有了这些基础知识，接下来的重心会被转移到提供一个对转换和视觉效果的深入理解上，这也是管线中的关键要素。

拓展阅读和资源

略

第四章 转换

“What if angry vectors veer

Round your sleeping head, and form.

There’s never need to fear

Violence of the poor world’s abstract storm.”

—Robert Penn Warren

一个转换是将诸如点、向量或颜色等实体通过某种方式转变他们的一种运算（废话）。对于计算机图形学的从业者来说，主要的转换是极度重要的。通过他们，你可以对对象，光照和相机进行定位，变形，以及播放动画。你也可以确保所有计算都在同一个坐标系下执行，并且通过不同的方式将对象投影到一个平面上。这些只是转换可以执行的很小一部分运算，但是它们足够用来展现转换在实时渲染，或者以至于任何计算机图形中的重要角色。

一个“线性转换”（linear transform）是一个满足向量加法和标量乘法的转换运算。特别的，

f(x) + f(y) = f(x + y),

kf(x) = f(kx).

作为一个例子，f(x) = 5x是一个将一个向量的每一个元素乘以5的转换运算。这种转换类型是线性的，任意两个向量乘以5然后相加的结果一定等于先相加后的向量乘以5所得到的结果。标量乘法的条件显然也是满足的。这种函数被叫做缩放转换，因为它改变了一个对象的缩放（尺寸）。旋转转换是另一种线性转换，它使一个向量关于原点旋转。缩放和旋转转换，实际上所有对于三阶向量的线性转换，都可以表示为一个三阶（3x3）矩阵。

然而，这个尺寸的矩阵往往不够大。一个作用于三阶向量x的函数比如f(x) = x + (7, 3, 2)并非线性的。对两个向量分别执行这个函数会给最终结果增加两次(7, 3, 2)的每个值。对一个向量增加一个固定的向量执行了一个转换，例如，它使所有的坐标移动了一个相同的值。这是一种非常有用的转换，而且我们可能会希望合并多种转换，比如，缩放一个对象到一半大小，然后把它挪到一个不同的位置。将函数保持现在简单的形式（三阶矩阵）使他们很难简单的组合在一起。

组合线性转换和平移可以使用仿射变换（affine transform），一般存储在一个四阶（4x4）矩阵中。一个仿射变换实际上是先执行了一个线性变换然后进行了一次平移。我们使用齐次记法（homogeneous notation）来表示四阶向量，使用相同的方式（粗体小写字母）来表示点和方向。一个方向矢量被表示为v = (vx vy vz 0)T一个点被表示为v = (vx vy vz 1)T。在本章中，我们将会大量使用附录A中的术语。你可能会希望现在先康康这个附录，特别是附录A.4，第905页，关于齐次记法。

所有平移，旋转，缩放，反射以及裁剪的矩阵都是仿射的。仿射矩阵最主要的特点就是它保持了直线的平行关系，但长度和角度不是必须的。一个仿射变换也可能是许多独立仿射变换的任意串联序列。

这一章会从一些最必要的，基本的仿射变换开始。这些的确是“非常”基础，并且这一节可以看做是简单转换的“基础手册”。更多特殊的矩阵在之后会被详细介绍，然后回讨论和描述四元数，一个强大的转换工具。接着是向量混合和变形，这是两个简单但更有效的表达网格动画的方式。最后，投影矩阵会被介绍。这些转换中的大部分，他们的符号，函数以及特点都被总结在表4.1中。

转换是操作几何中一个非常基础的工具。大部分图形学应用程序接口（APIs）都包含矩阵运算实现了许多本章中叙述的转换。然而，我们仍有必要去理解藏在函数调用背后的真实矩阵以及它们之间的相互作用。了解矩阵在函数调用之后的工作是一个开始，但理解矩阵本身的各种特性会让你更进一步。比如，这种理解会让你能够意识到你什么时候是在处理一个正交矩阵，也就是逆等于转置的矩阵（见904页），以更快地获得矩阵的逆。类似于这样的知识可以带来更加高效的代码。

4.1 基本变换

这一节描述了最基本的变换，比如平移，旋转，缩放，剪切，变换链接，刚体变换，法线变换（不太常用），以及逆运算。对于有经验的读者，本节可以被当做是简单变换的参考手册使用，而对于新手，本节可以看做是本主题的一个入门介绍。这些材料是本章其他部分以及本书其他章节的必要背景。让我们从最简单的变换开始——平移。

4.1.1 平移

从一个坐标到另外一个坐标的变化被代表为一个平移矩阵，T。这个矩阵通过一个向量t = (tx, ty, tz)来平移一个实体。T由下方等式4.2给出：

（等式略，见原文56页）

一个平移变换效果的例子展示在图4.1。易见点p = (px, py, pz, 1)与T(t)相乘得到一个新的点p’ = (px + tx, py + ty, pz + tz, 1)，这显然是一次平移。注意向量v = (vx, vy, vz, 0)不受与T相乘的影响，因为方向向量无法平移。与此相比，点和向量都可以受到其他仿射变换的影响。平移矩阵的逆矩阵是T-1(t) = T(-t)，也就是说，取向量t的负值。

4.1.2 旋转

一个旋转变换使一个向量（点或方向）围绕一个给定的穿过原点的轴旋转给定角度。像平移矩阵一样，它是一个刚体变换，也就是说，它维持变换后点与点之间的距离，并且维持手系（即它不会导致左右互换）。这两种变换显然在计算机图形学中对定位和定向对象来说是很有用的。一个“方向矩阵”（orientation matrix）是一个与定义其空间内方向的相机视图或对象相关联的旋转矩阵，也就是说，它指向上或者前。常用的旋转矩阵有Rx(φ), Ry(φ)和Rz(φ)分别是使实体围绕x,y,z轴旋转弧度φ。它们在等式4.3-4.5中被给出。

（等式略，见原文57页）

对于所有三阶旋转矩阵，R，绕任意轴旋转弧度φ，迹（trace详见898页定义）是独立于轴之外的常数，并且这样来计算：

tr(R) = 1 + 2 cos φ.

一个旋转矩阵的影响可以参考62页的图4.4.一个旋转矩阵Ri(φ)的特征是什么，除了它绕着轴i旋转了弧度φ这个事实以外，是它所有在旋转轴i上的点没有任何变化。注意R也会被用于表示一个围绕任意轴的旋转矩阵。上面给出的三个旋转矩阵可以通过使用在一系列的三个变换中来实现计算绕任意轴的旋转。这个程序将在4.2.1节中描述。4.2.4节则直接覆盖了运算一个绕任意轴的旋转。所有的旋转矩阵的行列式都是1，并且都是正交矩阵，这一点使用904页附录A中关于正交矩阵的定义非常容易得到证实。这也适用于任意数量的这类转换矩阵的链接。这是另外一种获得逆矩阵的方式：R i −1 (φ) =Ri(−φ)，也就是说，围绕同一个轴反方向旋转。同样，旋转矩阵的行列式总是1，因为它是正交的。

举例：围绕一个点旋转。假设我们想要围绕z轴使对象旋转弧度φ，并且旋转的中心是一个定点p。那么如何变换呢？这个过程描述在图4.2中。因为绕一点的旋转是以这个点本身不会受到旋转影响这一事实为特点的，所以这个变换开始于平移对象使p点位于原点，也就是执行T(-p)。在其后接着就是真正的旋转：Rz(φ)。最后，通过T(p)使对象被平移回它初始的位置。最终结果变换X，如下所示

X = T(p)Rz(φ)T(−p).

4.1.3 缩放

一个缩放矩阵S(s) = S(sx, sy, sz)通过因子sx, sy和sz分别沿着x,y和z轴缩放一个实体。