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。这意味着一个缩放矩阵可以用来放大或缩小一个对象。si（i ∈ {x, y, z}）越大，被缩放的实体在该方向上也就越大。设置s的任意一个分量为1，自然就可以避免在这个方向上缩放的变化。等式4.8展示了S：

（等式略，见原文58页）

62页的图4.4阐明了缩放矩阵的影响。缩放运算在sx = sy = sz时，被称作“均衡的”（uniform），而其他情况被称作“非均衡的”（nonuniform）。有时术语“各向同性的”（isotropic）和“各向异性的”（anisotropic）缩放被用于代替均衡和非均衡。逆矩阵是S−1(s) = S(1/sx, 1/sy, 1/sz)。

在使用齐次坐标的时候，另外一种可行的创建均衡缩放矩阵的方法是操作矩阵（3, 3）位置的元素，也就是右下角角落的元素。这个值影响齐次坐标的w元件，也因此可以通过这个矩阵变换来缩放每一个坐标。举个例子，想要均匀地用系数5来进行缩放，既可以设置缩放矩阵（0, 0），（1, 1）和（2, 2）的元素为5，也可以设置（3, 3）位置的元素为1/5.上述过程的两个矩阵如下所示：

（等式略，见原文59页）

与使用S进行均匀缩放不同，使用S’的话就必须在接下来的运算中一直使用齐次坐标运算。这也许会效率低下，因为它会导致齐次运算过程中的分离运算；如果右下角（坐标（3, 3））的元素是1，就没必要进行分离运算了。当然，如果系统总是会执行分离运算而不检查是否为1，那么就没有任何额外开销了。

拥有一个或三个负元素的s提供了一个“反射矩阵”（reflection matrix），亦称“镜矩阵”（mirror matrix）。如果只有两个缩放因子是-1，那我们就会旋转弧度π。当检测到反射矩阵时往往需要特殊对待。举个例子，一个顶点顺序为逆时针方向的三角形在经过反射矩阵变换后其顶点顺序会变成顺时针方向的。这个顺序的变化可能会导致不正确的光照和背面剔除的发生。为了去检查一个提供的矩阵是否通过某种方式进行了反射，可以计算该矩阵左上角的三阶行列式的值。如果这个值是负的，那么这个矩阵就是有反射的。

例子：一个特定方向上的缩放。缩放矩阵S只会沿着x,y或z轴进行缩放。如果要进行其他方向的缩放，就需要进行混合变换。假设缩放应该沿着由标准正交且右朝向（本人猜测就是左手系的意思）的向量fx, f y和 f z为轴。首先，构造如下的矩阵F：

（等式略，见原文59页）

我们的想法是先让这三个轴组成的坐标系与标准坐标系相一致，然后使用标准缩放矩阵，最后在转换回去。第一步就是与F的转置矩阵，也就是逆矩阵，相乘。接着就是真正的缩放操作，然后转换回去。这个转换如等式4.11所示：

X = FS(s)FT

4.1.4 剪切

另一种经典的转换是设置剪切矩阵。这可以，举个例子，用来在游戏中扭曲整个场景来创造一种梦幻的效果或者通过抖动（见9.3.1节）来创造模糊反射。这里有六个基本的剪切矩阵，他们被记为Hxy(s), Hxz(s), Hyx(s), Hyz(s), Hzx(s), and Hzy(s)。第一个下标标志着哪一个坐标被剪切矩阵给改变了，第二个下标标志着哪一个坐标进行了裁剪。Hxz(s)作为一个剪切矩阵的例子展示在等式4.12。可见，利用下标我们可以找到下方矩阵中参数s的位置；x（数字索引为0）确认为第0行，同时z（数字索引为2）确认列数为2，因此s被定为在这里：

（等式略，见原文60页）

点p乘以这个矩阵的效果是得到一个点：(px+ spz py pz)T。为了更生动直观，这个转换被用于一个单位矩形展示在图4.3。Hij(s)（通过联系j号坐标的值来剪切i号坐标）的逆矩阵通过对相反的方向进行剪切得到，也就是H ij-1 (s) = Hij (−s).

有一些计算及图形文章使用削微不同的剪切矩阵：

（等式略，见原文61页）

然而，两个下标都用来表示通过第三个坐标进行剪切操作的坐标。这两种不同的描述通过这种方式联系起来H’ij (s, t) = Hik(s)Hjk(t)，此处k表示索引为3的坐标。矩阵的正确与否取决于你的个人口味和API是否支持。

最后，需要注意的是因为任意剪切矩阵的行列式|H| = 1，所以它是一个保体积变换。

4.1.5 转换串联

由于矩阵乘法运算的不可交换性，矩阵出现的顺序就变得异常重要。转换的串联也因此被称作是顺序相关的。

作为一个顺序相关的举例，让我们考虑下两个矩阵，S和R。S(2, 0.5, 1)使用因子2缩放x元件，使用因子0.5缩放y元件。Rz(π/6)绕z轴（由本书页指向外）逆时针旋转弧度π/6。这些个矩阵可以用两种方式相乘，并且获得完全不同的结果。这两种情况展示在图4.4中。

我们将矩阵串联成一个队列的原因显然有一个是为了获得更高的效率。举个例子，想想你有一个有几千个顶点的对象，并且这个对象必须要缩放，旋转并且最后还要平移。现在，相对于让所有顶点分别乘以这三个矩阵，这三个矩阵串联形成了一个矩阵。这个矩阵接着应用在顶点上。这个复合矩阵就是C = TRS。记住这里的顺序：缩放矩阵S，应该首先用于顶点，因此它出现在这个复合体的右侧。这个顺序实际上隐含地运行为TRSp = (T(R(Sp)))。

值得注意的是，虽然矩阵串联是顺序相关的，但这些矩阵却可以按需要进行分组。比如，相对于TRSp你想要一次性计算刚体运动变换TR。我们可以把这两个矩阵分为一组，(TR)(Sp)，并用一个中间结果代替。因此，矩阵串联是满足结合律的（associative）。

4.1.6 刚体变换

当一个人掏出了一根硬硬的东西，比如桌上的（哔——），并把它移动到其他的位置时，可能是她的衬衫口袋，只有这个对象的朝向和位置会变化，而它的形状通常是不会受到影响的。这种变换，包括只由平移和旋转组成的序列，被称为刚体变换（rigid-body transform）并且具有维持长度，角度和手系（个人理解为，是否可能镜像）的特点。

任意刚体矩阵，X，可以被写为一个平移矩阵T(t)和一个旋转矩阵R组成的序列。因此，X可以被写作等式4.14所示的矩阵：

（等式略，见原文62页）

X的逆矩阵计算如下X−1 = (T(t)R)−1 = R−1T(t)−1 = RT T(−t)。因此，要计算这个逆矩阵，只要把矩阵R左上的三阶矩阵转置，并且修改T中平移值的符号。这两个新矩阵用相反的顺序相乘后得到的就是逆矩阵了。

另外一种计算X的逆的方式是考虑将R（看做是三阶矩阵）和X看做下面的记法形式：

（等式略，见原文63页）

在这里，0是一个3x1的元素全为0的列向量。通过一些简单的运算就可以得到等式4.16所示的逆矩阵：

（等式略，见原文62页）

4.1.7 法线变换

一个矩阵可以用来一致地变换点，线，多边形以及其他几何体。同样的矩阵也可以用于变换沿着这些线或是多边形表面的切向量。然而，这个矩阵往往无法用于变换一个非常重要的几何属性，表面法线（包括顶点光照法线）。图4.5展示了如果使用同一个矩阵会发生什么。

相较于直接乘以这个矩阵自身，适当的方法应该是使用这个矩阵伴随矩阵的转置。伴随矩阵的计算在A.3.1节有叙述。伴随矩阵被证明总是存在的。法线在经过转换后不能保证一定是单位长度的，所以一般来说需要进行标准化。

转换法线的传统方法是计算其逆矩阵的转置。这种方法往往是有效的。然而，完全的逆运算是不必要的，而且有时可能无法构建。逆矩阵是伴随矩阵除以原始矩阵的行列式得到的。如果行列式是0，那么该矩阵就是奇异的，逆矩阵也就不存在了。

实际上只是计算一个完整的四阶矩阵的伴随矩阵也可能非常昂贵，并且往往是不必要的。因为法线是一个向量，所以平移不会影响它。进一步说，大部分模型变换是仿射（译者：由三阶的线性变换和一次平移变换组合而成）的。他们并不会修改传入的齐次坐标的w元素，也就是说，他们没有计算投影。在这些（常见的）情况下，法线变换就只需要计算左上三阶矩阵的伴随矩阵。

通常我们甚至连这个伴随矩阵的计算都不需要。假设我们知道变换矩阵完全是由一连串的平移，旋转以及一致性缩放运算组成的（没有拉伸或挤压）。平移不影响法线。均衡的缩放因子只是简单地改变法线的长度。剩下的只有一系列的旋转，它总会形成某种形式的净旋转，其他的什么也不剩。一个旋转矩阵的定义基于一个事实也就是它的转置就是它的逆。逆矩阵的转置可以被用于变换法线，两次转置（或两次取逆）会相互抵消。总的来说，结果就是在这些情况下，原始的转换矩阵本身也可以直接用于转换法线。

最后，对于所产生法线的完整重标准化也不总是必要的。如果只有平移和旋转被组合在一起，法线的长度在经过矩阵转换后就不会发生变化，也就不需要重新标准化了。如果也包含了均衡的缩放，整体的缩放因子（如果一直，否则提取——4.2.3节）可以用来直接标准化取得的法线。举个例子，如果我们知道经过一系列的缩放后对象变成了5.2倍大，那么这个矩阵变换后取得的法线就可以通过除以5.2来进行重标准化。作为替换，为了构造一个可以直接产生标准化结果的法线转换矩阵，我们可以直接让原始矩阵左上的三阶矩阵元素各除以这个缩放因子一次。

注意法线变换在系统中并非一个问题，在转换之后，表面法线是由三角面直接导出的（例如利用三角形两边的叉乘）。切向量与法线在本质上是不同的，并且往往直接用原始矩阵进行变换。

4.1.8 逆的计算

很多种情况下逆矩阵都是需要的，比如，当在坐标系之间来回转换时。基于转换的可用信息，可以用下列三种方法中的一个来计算一个矩阵的逆。

·如果这个矩阵是一个单一的变换或是简单变换由已给的参数组成的队列，那么我们就可以通过简单地计算“参数的逆”以及矩阵顺序得到。比如，如果M = T(t)R(φ)，那么M-1 = R(-φ)T(-t)。

·如果这个矩阵已知是正交的，那么M-1 = MT，也就是说，转置就是逆。旋转的任意队列也是一个旋转，所以是正交的。

·如果啥也不知道，那么伴随矩阵方法（902页等式A.38），克莱姆法则（译者：并不知道克莱姆法则如何用来求逆），LU分解或者高斯消元法可以用于计算逆（见A.3.1节）。克莱姆法则以及伴随矩阵方法一般来说更好，因为他们的分支运算更少；在现代的架构中避免使用“if”是明智的。见4.1.7节来了解如何使用伴随矩阵方法来求转换后法线的逆。

在优化时也可以考虑一下逆运算的目的是什么。举个例子，如果逆运算是用来转换向量的，那么通常来说只有左上的3x3的部分需要进行逆运算（见前面的章节）。

4.2 特殊的矩阵转换和运算

在本节中，许多对实时图型来说必要的矩阵转换和运算将会被引入和导入。首先，我们将介绍欧拉变换（及其参数提取），它是一种非常直观的描述方向的方法。接着我们会讲到从一个单一的矩阵中检索出一系列的基础变换。最后，将介绍一种使一个实体绕任意轴旋转的方法。

4.2.1 欧拉变换

这个转换是一种直观的方式构建一个矩阵来定向你自己（也就是相机）或其他任意在某个确定方向上的实体。它取名自瑞士大数学家莱昂哈德·欧拉（1707-1783）。

首先，必须建立某种默认的视野方向。大多数情况下，它沿着z轴负方向同时头顶指向y轴，就像图4.6所示。欧拉变换是三个矩阵的乘积，也就是图中所示的旋转。更正式的写法，这个转换，记为E，正如等式4.17所给出的：

E(h, p, r) = Rz(r)Rx(p)Ry(h).

由于E是旋转的复合体，所以它显然是正交的。因此它的逆可以表示为E−1 = ET = (RzRxRy)T = RT y RT x RT z，当然，直接使用E的转置会更简单。

欧拉角h,p以及r表示按什么顺序以及头，俯仰和翻转各自绕他们的轴旋转多少。这种转换很直观并且因此可以很简单的用外行（译者：直白？）的语言来描述。举个例子，改变头部角度使得观察者摇头说“不”，改变俯仰角使得他点头，改变翻转角使他向侧面倾斜他的头。相对于讨论绕x,y,z轴旋转，我们讨论变更头部（面向），俯仰和翻转。注意这种转换不仅可以定向摄像机，同样可以定向任意对象或实体。这些转换可以被执行在世界空间下的全局轴或是相对于一个局部参考系。

当你使用欧拉变换的时候，有时会出现一种叫做“万向节锁”（gimbal lock）的东西。它发生在当旋转时一个自由度会丧失。举个例子，假如变换的顺序是x/y/z。考虑一下第二次旋转的时候绕y轴旋转π/2。这样会旋转本地的z轴并使其对齐初始的x轴，这也就是说最后绕z轴的旋转是冗余得。

另一种观察一个自由度丧失的方法是设置p = π/2然后检查Euler矩阵E(h, p, r)发生了什么：

（等式略，见原文67页）

由于矩阵只依赖一个角(r + h)，我们可以说一个自由度已经丧失了。

关于绕每个本地轴的旋转，虽然欧拉角通常在模型系统中使用x/y/z的顺序计算，但其他的顺序也是可行的。比如，z/x/y被用于动画，z/x/z被同时用于动画和物理。所有这些都是指定三个独立旋转的有效方法。最后那个顺序，z/x/z对于某些应用来说更适用，因为只有当绕x轴旋转弧度π（半圈）的时候万向节锁会发生。这也就是说，通过“毛球定理”我们知道万向节锁是不可避免的，这里没有一个完美的队列可以避免它。

对然对于小的角度变化或是试图定向来说欧拉角是有用的，但它也有一些其他的限制。把两组欧拉角组合起来是困难的。举个例子，从一组到另一组欧拉角的插值并不是简单的对每个角进行插值。实际上，两组不同的欧拉角也可能给出相同的朝向，所以任何插值都不应当旋转对象（译者：此处插值的问题解释不太清晰，读者可搜索“欧拉角插值”等关键字进行辅助理解）。这也是为什么使用替代的方向表示也是值得研究的原因了比如四元数，这在这一章之后的部分会有描述。

4.2.2 从欧拉转换中提取参数

走某些情况下，尝试从一个正交矩阵中去提取欧拉参数，h,p和r是有用的。这个程序展示在等式4.19：

（等式略，见原文68页）

将等式4.19中的三个旋转矩阵串联组合后得到

（等式略，见原文68页）

从这个等式我们看出俯仰参数可以由sin p = f21得到。同理，使用f01除以f11，类似的使用f20除以f22，就会得到下列的两个关于头部和翻转参数的额外等式：

（等式略，见原文68页）

因此，欧拉参数h（头），p（俯仰）以及r（翻转）可以由矩阵F使用函数atan2(y, x)（见第1章第7页）得到，如等式4.22：

（等式略，见原文68页）

然而，这里我们需要掌握一种特殊的情况。当cos p = 0时，这种情况就会出现，因为f01 = f11 = 0，因此atan2函数将无法使用。由cos p = 0 我们可以得到sin p = ±1，因此F可以简化为

（等式略，见原文68页）

剩下的参数任取h = 0，然后sin r / cos r = tan r = f10 / f00，也就得到r = atan2(f10, f00)。

注意由arcsin（见B.1节）的定义我们知道，-π/2≤p≤π/2，这也就意味着如果F在构建时的p参数超过了这个区间，那么最初的参数就无法被提取。h,p和r并非唯一的，意味着不只一组欧拉参数可以用于获得同一个转换。关于欧拉角转换的更多相关信息可以参考Shoemake1994年的研究。上面列出的简单方法可能会导致数值不稳定的问题，这一点通过牺牲一定的速度是可以得到避免的。

例子：转换约束。想象你拿着一把扳手来拧螺丝，为了把螺丝拧对位置你必须让这个扳手绕x轴旋转。现在假设你的输入设备（鼠标，VR手套，空间球等等）给了你一个正交转换来移动这个扳手。现在你遇到的问题是，你不想对扳手应用这个变换，因为它应该只能绕x轴旋转才对。因此想要重新限制输入变换P为绕x轴的旋转，只需要使用本节描述的方法，简单地提取出欧拉角，h,p和r，然后重新构建一个新的矩阵Rx(p)。这是一个很吃香的转换，它会让扳手绕x轴旋转（如果P含有这样的运动的话）。

4.2.3矩阵分解

到目前为止，我们一直假设在了解我们使用的转换矩阵的源头和历史的基础上工作。这并不是一般情况：举个例子，与转换后的对象相关联的只有一个串联组合得到的矩阵。从一个复合矩阵中检索各种转换的任务被称作“矩阵分解”（matrix decomposition）。

有许多原因让我们去检索一组转换。用途包括：

·只想提取一个对象的缩放因子。

·查找特定系统所需的转换。比如VRML使用一个转换节点（见4.1.5节）但不允许使用任何四阶矩阵。

·确定一个模型是否只经历了刚体变换。

·在只有关于对象的矩阵是有效的动画的关键帧之间进行插值。

·从一个旋转矩阵中去除剪切操作。

我们已经介绍了两种分解，从一个刚体变换中导出平移和旋转矩阵（见4.1.6节）以及从一个正交矩阵中导出欧拉角（4.2.2节）。

正如我们所看到的，检索平移矩阵是很简单的，因为我们只是简单地需要四阶矩阵的最后一列元素。我们还可以通过检查矩阵的行列式是否为负来确定是否发生了反射。要分离出旋转，缩放和剪切运算就需要更多的确定性计算了。

幸运的是，有许许多多关于这个主题的研究，和网上就可以获取的代码。Thomas和Goldman各自提出了许多针对各种类型转换的不同方法。Shoemake改进了他们应对仿射矩阵的技术，他的算法不依赖于参考系并试图分离矩阵以获得刚体变换。

4.2.4 任意轴旋转

有时能够使一个实体绕任意轴旋转一个角度是非常方便的。假设旋转轴r是标准化的并且转换被构建为绕r旋转弧度α。

为了实现这个，首先我们要找到任意的两个轴，它们具有单位长度，且与r三者两两正交，也就是说，是标准正交的。然后它们就构成了一组基（basis，not gay）。我们的想法是将基准（bases，见A.3.2节）从标准基转移到这组新的基，然后绕，假设是，x轴（也就是之后应该对应r的那个）旋转弧度α，并且最后转换回标准基。这个步骤被展示在图4.7中。

第一步是计算这组标准正交基的轴。第一个轴是r，也就是我们想要绕着旋转的那个。我们现在集中精力找第二个轴s，因为我们知道第三个轴t可以通过前两个轴叉乘得到t = r x s。一种数值稳定（译者：可能由于某些原因导致数值误差较大，比如当两个极其相近的数相减时就会导致有效数字较大丧失，造成误差，因为数字精度是有限的。）的方法是找到r中（绝对值）最小的分量，然后将其设置为0。交换剩下的两个分量，并让这两个分量的第一个乘以-1。数学化得展示如下：

（等式略，见原文71页）

这样就可以保证s(上面有横杠，我不会打)是正交（垂直）于r的，并且(r, s, t)是一组标准正交基。我们使用这三个向量作为行组成下面的矩阵：

（等式略，见原文71页）

这个矩阵转化向量r到x轴(ex)，s到y轴，t到z轴。所以绕标准化向量r旋转弧度α的最终转换为

X = MT Rx(α)M.

换句话说，这意味着我们第一个转换后r就变成了x轴（使用M），接着我们绕x轴旋转弧度α（使用Rx(α)），然后我们使用M的逆转换回来，这种情况下MT就是M的逆，因为M是正交的。

另外一种绕任意的标准化的轴r旋转弧度φ的方法是由Goldman（译：黄金侠？？？）提出得。在这里，我们简单介绍一下他的转换：

（等式略，见原文71页）

在4.3.2节我们将会提出另一种解决这个问题的办法，使用四元数。在那一节中也有更多的针对旋转问题的有效率的算法，比如从一个向量转到另一个向量。

4.3 四元数

虽然四元数是1843年Sir William Rowan Hanilton发明的，作为复数的扩展，但直到1985年才被Shoemake引入到计算及图形的领域。四元数是一个具有引人注目特点的构建转换的强力工具，并且在某些情况下比欧拉角和矩阵更好用，特别是在旋转和定向方面。给定一个轴和角度表示，转化为或是从四元数转化的过程是直观的，但是欧拉角的双向转换却十分具有挑战性。四元数可以用来实现稳定平衡的方向插值，这在欧拉角看来是难以做到很漂亮的。

一个复数具有一个实部和一个虚部。每个复数都由两个实数组成，第二个实数会被乘以√−1。类似的，四元数具有四个部分。前三个值与旋转轴密切相关，而旋转角度则同时影响所有的四个部分（详见4.3.2节）。每一个四元数由四个实数表示，每一个关联不同的部分。因为四元数具有四个分量，我们选择将它们表示为向量，但为了区别于向量，我们在它们上面加一顶帽子：qˆ（译：这个尖角在正上方）。我们先从四元数的数学背景开始，这些也将接下来用于构建有趣且有用的转换。

4.3.1 数学背景

我们先从四元数的定义开始。

定义。一个四元数q^可以被定义为以下几种等价形式。

qˆ = (qv, qw) = iqx + jqy + kqz + qw = qv + qw,

qv = iqx + jqy + kqz = (qx, qy, qz),

i2 = j2 = k2 = −1, jk = −kj = i, ki = −ik = j, ij = −ji = k.

变量qw被称为四元数q^的实部。虚部是qv，同时i，j，k被称为虚单位。

对虚部qv我们可以使用所有一般的向量运算，比如加法，缩放，点乘，叉乘或其他。使用这个四元数的定义，两个四元数q^和r^的乘法运算展示在下方。注意虚单位的乘法是不支持交换律的。

乘法：（等式略，见原文73页）

从这个等式中我们可以看出，我们同时使用了叉乘和点乘来计算两个四元数的积。按这个四元数的定义，加法，共轭（conjugate），范数（norm）以及单位数（identity）的定义也需要确定：

（等式略，见原文73页）

当n(qˆ) = √qˆqˆ∗经过化简之后（结果如上），虚部被抵消而只剩下一个实部（译：就是虚数单位都莫得了）。范数有时表示为||qˆ|| = n(qˆ)。以上等式的一个结果就是乘法逆元，记为qˆ−1，可以被推导出来了。等式qˆ−1qˆ = qˆqˆ−1 = 1对于逆本身来说也必须是成立的（就像通常的乘法逆元一样）。我们从范数的定义中推导出一个公式：

（等式略，见原文73页）

我们因此可以得到下方的乘法逆元：

（等式略，见原文73页）

这个求逆的公式使用了缩放乘法，这种缩放乘法运算也可以从等式4.29的乘法等式中推导而来：sqˆ = (0, s)(qv, qw) = (sqv, sqw)同时qˆs = (qv, qw)(0, s)=(sqv, sqw)，这意味着缩放乘法满足交换律：sqˆ = qˆs = (sqv, sqw)。

下面的几个性质很容易就可以由定义推导而来：

（等式略，见原文74页）

一个单位四元数qˆ = (qv, qw)是满足这个的n(qˆ) = 1。因此它qˆ也可以被写成这样

qˆ = (sin φuq, cos φ) = sin φuq + cos φ,

其中uq是一个三维向量，且||uq|| = 1，因为

（等式略，见原文74页）

当且仅当uq · uq =1= ||uq||2。就像在下一节将要看到的，单位四元数非常适合通过一个非常高效的方式来构建旋转和方向。但在此之前，我们首先要介绍一些单位四元数的额外运算。

对于复数来说，二维的单位向量可以被写作cos φ + isin φ = eiφ。这个等式在四元数中写为

qˆ = sin φuq + cos φ = eφuq .

单位四元数的对数和幂函数如下所示：

（等式略，见原文74页）

4.3.2 四元数变换

我们现在要学习四元数的一个子类，这些单位长度的四元数被称为“单位四元数”（unit quaternions）。关于单位四元数最重要的一点就是它们可以表示任意的三维旋转，并且这种表示极其简洁明了。

现在我们将讲述一下是什么让单位四元数对于旋转和方向如此有用。首先，将一个点或向量p = (px py pz pw)T的四个坐标放入一个四元数p^的四个分量中，并且假设我们有一个单位四元数q^ = (sin φuq, cos φ)。接着

qˆpˆqˆ−1

让p^（也就是p）绕轴uq旋转角度2φ。注意因为q^是一个单位四元数，所以qˆ−1 = qˆ\*。这个旋转显然可以用于绕任意轴旋转，如图4.8。

任意非零的实数倍的q^都代表相同的变换，也就是说q^和-q^代表相同的旋转。也就是说，翻转轴uq以及实部qw，构建了一个于原始四元数的旋转完全相同的四元数。这也意味着从一个矩阵中提取一个四元数可能返回q^或-q^。

给出两个单位四元数q^和r^，先执行q^再执行r^的两者的串联四元数p^（也可以被解释为一个点p），如等式4.41所示：

r(qˆpˆqˆ∗)ˆr∗ = (ˆrqˆ)pˆ(ˆrqˆ) ∗ = cˆpˆˆc∗.

在这里，c^ = r^q^是代表了单位四元数q^和r^串联结果的单位四元数。

矩阵变换

因为有些系统中硬件使用的是矩阵乘法并且实际上矩阵乘法要比等式4.40效率更高，所以我们需要转变的方法来从一个四元数转换到一个矩阵，反之亦然。一个四元数q^可以被转换为一个矩阵Mq，如等式4.42所示：

（等式略，见原文76页）

在这里，标量s = 2 / n(q^)。相对于单位四元数，就可以化简为

（等式略，见原文76页）

一旦这个四元数被构成了，就不再需要计算任何的三角函数了，所以在实际的转化过程中会更具效率。

相反的计算，从一个正交矩阵Mq转化为一个单位四元数q^就稍微有点复杂了。这个过程的关键就是从等式4.43中的矩阵得到下面的差值：

（等式略，见原文76页）

这些个等式的含义就是如果qw我们是知道的，那么向量vq的值就可以被计算出来，继而q^也就可以得到了。Mq的迹（见898页）计算如下

（等式略，见原文76页）（译：此等式中似乎把n(q^)与qx2 + qy2 + qz2 + qw2划等号，这与之前n(q^)的定义不符，不太明白，猜测可能是因为单位四元数的缘故，所以结果看起来没有问题）

这个结果相对于一个单位四元数可以得到以下转化：

（等式略，见原文76页）

为了获得一个数值稳定的程序，用特别小的数字进行除法是需要避免的。因此，首先设t = q2 w − q2 x − q2 y − q2 z，并由此推出

（等式略，见原文77页）

这也就意味着，m00,m11,m22以及u中的最大值实际上就决定了qx,qy,qz和qw中的最大值。如果qw是最大的，等式4.46就可以用来推导出四元数。否则，我们注意到下列等式：

（等式略，见原文77页）

使用上述等式中适合的一个来计算qx,qy以及qz中的最大值，然后再使用等式4.44来计算q^中剩余的分量。幸运的是，这里有这个算法的代码——见本章最后的扩展阅读和资源部分。

球面线性插值

球面线性插值是一个计算，给出两个单位四元数q^和r^以及一个参数t ∈ [0, 1]，计算一个插值后的四元数。举例来说，这个计算对于动画对象非常有用。它对于插值相机的方向来说比较差劲，因为相机的“上”向量可能会在插值过程中变得倾斜，这往往是一个让人烦恼的效果。

这个运算使用复合四元数s^表示为代数形式如下所示：

s(qˆ, rˆ, t)=( rˆqˆ−1) t qˆ.

然而，为了在软件中执行，下面的slerp格式替代球面线性插值是更合适的：

（等式略，见原文77页）

为了计算等式中所需的φ，可以用到下面的定义：cos φ = qxrx + qyry + qzrz + qwrw。对于t ∈ [0, 1]，slerp函数计算（唯一的，当且仅当q^和r^不互为相反数）由q^（t = 0）到r^（t = 1）在一个四维单位球面上共同组成的最短弧的插值四元数。这个弧位于由q^,r^和原点组成的平面与四维单位球面的相交处（译：也就是交线）。如图4.9所示。计算得到的旋转四元数绕固定轴匀速旋转。像这样的，匀速的加速度为0的曲线，被称为测地线（geodesic curve）。

Slerp函数完美契合在两个方向间的插值并且它的表现良好（固定轴，匀速）。当时用多个欧拉角进行插值时，情况就不一样了。实际上，直接计算一个slerp是一个非常昂贵的计算因为它疯狂地调用了三角函数。李（Li）提供了一种更加快速的且不会丧失任何精度的增值方法用于计算slerp。

当超过两个角，假设q0^,q1^,…,qn-1^，是允许的，并且我们想计算从q0^到q1^再到q2^直到qn-1^的插值，slerp可以经过非常简单的改变而直接使用于此。现在，当我们在接近，假设是qi^，我们就会用qi-1^和qi^作为参数来slerp。当穿过了qi^之后，我们接着就会使用qi^和qi+1^作为参数来slerp。这可能会导致在方向插值中出现突然的抖动，正如图4.9中的那样。这与点的线性插值中发生的现象很类似；见578页图13.2的右上部分。有些读者在阅读过13章有关样条的内容后可能会希望再次阅读下面的段落。

使用一丢丢样条曲线可能是更好的插值方式。我们在qi^和qi+1^之间引入四元数ai^和ai+1^。球形立方插值可以通过四元数qi^,ai^,qi+1^和qi+1^的设置而定义。令人震惊的是，这些额外的四元数通过下面这种方法被计算出来：

（等式略，见原文79页）

qi^和ai^会使用一个如等式4.52所示的光滑的三次样条曲线来进行球面插值。

（等式略，见原文79页）

正如上面可见的，squad函数实际上由重复地使用slerp进行球面插值构成（见13.1.1节了解更多关于点的重复线性插值的信息）。这个插值可能会超过原始的方向qi^，i ∈ [0,...,n − 1]，但不会超过ai^——这被用于表示原始方向上的切线方向。

从一个向量旋转为另一个

一个常用的运算是通过可能的最短路径从一个方向s转换到另一个方向t。四元数的数学运算大大简化了这一过程，并且还展示了四元数与这种表达法的密切关系。首先，将s和t标准化。然后用u = (s × t)/||s × t||计算单位旋转轴u。接着，e = s · t = cos(2φ) and ||s × t|| = sin(2φ)，此处2φ是s与t的夹角。接着我们就得到了表示s到t旋转的四元数q^ = (sin φu, cos φ)。实际上使用半角公式（见919页）和三角恒等式（等式B.9）可以化简（等式略）为

（等式略，见原文79页）

直接使用这种方式生成四元数（而不是通过叉乘s和t再进行标准化）避免了s和t在方向上非常相近时产生的数值不稳定的问题。当s个t指向相反方向时，这两种方法都会出现稳定性的问题，因为会出现除以0的情况。当发现这种特殊情况时，任意垂直于s的旋转轴都可以被用于旋转到t。

有时我们需要用矩阵来代表一个s到t的旋转。在经过对等式4.43进行了一系列代数和三角简化之后，旋转矩阵变成了

（等式略，见原文80页）

在这个等式中，我们使用了以下中间计算：

（等式略，见原文80页）

我们可以看到，这个简化式中所有的平方根和三角函数都消失了，因此，这是一种构建矩阵的高效率方式。

注意，当s和t平行或接近平行的时候必须小心翼翼。因为 ||s×t|| ≈ 0。如果φ ≈ 0，我们可以返回单位矩阵。然而，如果2φ ≈ π，那么我们可以绕任意轴旋转弧度π。这个轴可以通过让s和任意不平行于s的向量叉乘得到（见4.2.4节）。M¨oller and Hughes使用户主矩阵（Householder matrices）来用一种不同的方法处理这种特殊的矩阵。

例子：一个相机的定位及定向。假设一个虚拟相机（或视点）的默认位置为(0 0 0)T同时默认的视角方向v为沿着z轴负方向，也就是v = (0 0 − 1)T。现在，我们的目标是构建一个转换将这个相机移动到一个新的位置p，并看向一个新的方向w。首先为摄像机定向，这可以通过从默认视角方向旋转到目标视角方向完成。R(v, w)负责这个。定位就简单的通过平移到p完成，从而得到最终转换X = T(p)R(v, w)。实际上，在完成第一次旋转之后，往往希望可以通过另一次向量到向量的旋转来使视野的“上”可以指向某个希望的方向。

4.4 顶点混合

假设现在一个数字角色的手臂动画由两部分组成，一个前臂和一个上臂，如图4.10左侧所示。这个模型可以使用刚体变换（4.1.6节）来完成动画。然而，这两部分的交接处不会像是一个真正的肘部。这是因为使用了两个独立的对象，因此，关节是由这两个单独对象的重叠部分组成的。然而，静态的模型部分并没有解决使关节活动的问题。

顶点混合（vertex blending）是这个问题的一个可能的解决方案。这项技术还有许多其他的名字，比如蒙皮（skinning），包皮（enveloping）以及子空间骨骼变形（skeleton-subspace deformation）。虽然此处给出的算法确切起源不太清楚，但定义骨骼以及皮肤对变化的反应是计算机动画中的一个古老命题。在其最简单的形式中，前臂和上臂还是像之前一样各自动画，但在关节处，两个部分通过一个具有弹性的“皮肤”连接在一起。因此，这个弹性区域会有一部分顶点使用前臂的矩阵变换而另一部分使用上臂的矩阵变换。这导致有些三角形中的顶点需要使用不同的矩阵来变换，而不是每个三角形只使用一个矩阵。见图4.10。这项基础技术有时被称为缝合（stitching）。

把这一步迈更远一些，可以允许一个独立的顶点被多个不同的矩阵变换，将结果坐标加权混合。这通过使用动画对象上绑定的一具骨骸，通过骨骼的变换可能会通过用户定义的权重影响到每一个顶点。因为整个胳膊可能都是“有弹性的”，也就是说，所有的顶点可能都受到不止一个矩阵的影响，这整个网格一般被称为皮肤（skin， 覆盖于骨骼上）。见图4.11。许多商业建模系统具有同样的谷歌建模功能。尽管他们的名字是骨骼，但实际上它们不一定必须是刚性的。举个例子，Mohr和Gleicher提出了加入额外的关节来实现诸如肌肉膨胀效果的想法。James和Twigg讨论可以使用可以挤压和拉伸的骨骼来实现动画蒙皮。

数学化来说，这个方法如等式4.56所示，这里p是原始顶点，u(t)就是位置与时间t相关的转换后顶点。这里有n个骨骼影响着p的位置，它们的位置由世界坐标表示。矩阵Mi从最初的谷歌坐标系转换到世界坐标系。一般来说，骨骼在其坐标系的原点位置有控制关节。举个例子，一个前臂骨骼会将肘关节作为原点，因为动画的旋转矩阵使手臂的这一部分绕这个关节运动。Bi(t)矩阵是随着时间对象动画的第i个骨骼在世界空间下的变换，它一般是一系列矩阵的组合，比如层次结构中上层骨骼（译：父骨骼）的变换以及本地的动画矩阵。Woodland深度讨论了一种维护和更新Bi(t)矩阵动画函数的方法。最后，wi是骨骼i对于顶点p的权重。这个顶点混合等式就是

（等式略，见原文83页）

每根骨骼根据它自己的参考系将一个顶点转换到一个坐标，并且最终的坐标是一系列计算得到的点插值得到的。在一些关于蒙皮的讨论中并没有明确提及矩阵Mi，而是被作为Bi(t)的一部分来看待。我们在这里将它显示出来是因为它非常有用，它几乎总是矩阵串联过程中的一部分。

事实上，矩阵Bi(t)和Mi-1在每一帧的每根骨骼都要进行串联，并且每个结果矩阵都要用于转换顶点。顶点p通过不同骨骼的复合矩阵变换，然后使用权重wi进行混合——因此被称为顶点混合（vertex blending）。这些个权重非负且和为1，所以发生的事情就是顶点被转换到几个位置然后在他们之间进行插值。因此，变换后的点u会位于点集Bi(t)Mi-1p组成的凸包中，其中I = 0…n – 1（t不变）。法线一般也可以使用等式4.56进行转换。根据使用的变换（比如，如果一个骨骼被挤压或拉伸了一个需要考虑的数值），可能会需要使用Bi(t)Mi−1逆的转置来代替，如4.1.7节所述。

顶点混合非常适合用于GPU。网格中的这一大堆顶点可以一次性被传递进GPU置于静态缓存并重复使用。每一帧中，只有骨骼矩阵变化，同时一个顶点着色器计算它们对存储的网格的影响。通过这种方法，从CPU上传输和处理的数据量得到最小化，允许GPU极具效率地渲染网格。最简单的情况是模型所有的骨骼矩阵可以一起使用；否则就必须将模型分割并复制一部分骨骼。

当我们使用顶点着色器时，我们可以使用超过范围[0, 1]的或是和不为1的权重值。然而这只会用于当时用一些其他的混合算法时，比如目标变形（morph targets，见4.5节）。

基础的顶点混合有一个缺点就是引入了需求外的折叠，扭曲和自交可能会发生。见图4.12。一种最好的解决方案是使用对偶四元数（dual quaternions），由Kavan等人提出。这种技术使得蒙皮辅助维持了原始变换的刚性，所以避免了肢体“糖果包装纸”式的扭曲。计算量小于线性皮肤混合消耗的1.5倍并且结果非常杰出，这也使得这项技术迅速被采用。感兴趣的读者可以参考下这篇文章，其中也对以往线性混合改进的纵览。

4.5 变形

在进行动画时从一个三维模型进行变形可能会非常有用。想象一下现在有一个模型播放到时刻t0同时我们希望可以在时刻t1将它变为另一个模型。在从t0到t1的整个过程中，我们需要通过某种插值来获得一个连续的“混合”模型。图4.13展示了一个变形的例子。

变形有两个非常关键的问题，分别叫做“顶点对应”（vertex correspondence）问题以及“插值”（interpolation）问题。提供两个任意模型，它们具有不同的拓扑结构，不同的顶点数目，以及不同的网格连接，往往从设置这些顶点的对应关系开始。这是一个艰难的问题，并且在这个领域有许多研究。我们推荐感兴趣的读者去阅读Alexa的研究。

然而，如果这两个模型具有一对一的顶点对应关系，那么插值既可以建立在每个顶点的基础上。这也就是说，第一个模型中的每个顶点，也一定有且仅有一个顶点存在于第二个模型中，反之亦然。这使得插值变成一个非常简单的任务。举个例子，线性插值可以被直接用于这些顶点（见13.1节使用其他方式进行插值）。为了计算一个位于时刻t ∈ [t0, t1]变形后的顶点，我们首先计算s = (t − t0)/(t1 − t0)，然后线性混合顶点，

m = (1 − s)p0 + sp1,

此处p0和p1分别对应位于不同时刻t0和t1的相同顶点。

一种使用者具有更直观控制的有趣的变形，通常被称作“对象变换”（morph targets）或“混合图形”（blend shapes）。它的基本概念如图4.14所示。

我们从一个中性的模型开始，在当前情况下就是一张脸。让我们使用N来表示这个模型。另外，我们还有一组不同的面部表情。在这个示例中只有一个表情，也就是微笑脸。一般情况下，我们可以允许k≥1个不同的表情，记为Pi（I ∈ [1,...,k]）。作为一个预处理，“表情差异”计算如下：Di = Pi – N，也就是使每个模型减去中性模型。

此时，我们有了一个中性模型N以及一组表情差异Di。通过如下公式可以获得一个推演模型M：

（等式略，见原文87页）

我们使用权重wi在中性模型的基础上增加需要的不同模型的特征。如图4.14，设置w1 = 1是我们可以准确地获得一张微笑脸如中间的插图。使用w1 = 0.5可以获得一张皮笑肉不笑的脸，等等。同时我们可以使用负权重或是大于1的权重。

我们还可以为这张简单的脸部模型增加另外一个具有“悲伤”眉毛的脸。对眉毛使用负权重就可以创建一个“开心”的眉毛。由于位移是相加的，这个眉毛的表情可以与微笑的嘴一起使用。对象变形是一个非常强力的技术，提供了许多动画的控制，因为一个模型的特点可以独立于其他模型运作。Lewis等人引入了构成空间变形（pose-space deformation），它合并了顶点混合与对象变形。支持DirectX 10的硬件可以使用涌出（stream-out）以及其他高级功能来允许单个模型使用更多的对象同时这个效果只在GPU进行计算。

一个同时使用蒙皮和变形的真实示例如图4.15所示。

4.6 投影

在实际渲染一个场景之前，场景中所有相关的对象都必须被投影到某些平面或某些简单的体块中。在这之后，裁剪和渲染才会运行（见2.3节）。

在本章到目前为止的转换中还从未影响过第四个分量，即w分量。这是因为，点和向量在转换后保持了它们的类型。同时，4阶矩阵的底部一行总是（0 0 0 1）。透视投影矩阵（perspective projection matrices）则是例外：底部一行包含了向量和点的操作数，同时往往需要一些齐次化过程（也就是说，w往往不是1，所以需要除以w来获得非齐次点）。本节第一次处理的正交投影（orthographic projection）是一个经常被使用的一种简单的投影类别。它没有影响w分量。

本节中，我们假设视角正方向为z轴负方向，同时y轴正方向为上x周正方向为右。这是一个右手系。一些文本和软件，比如DirectX，使用左手系，也就是目光指向z轴正方向。这两种坐标系都是同样有效的，并在最后得到相同的效果。

4.6.1 正交投影

正交投影的一个特点就是在投影后平行的直线依然保持平行。下方展示的矩阵Po是一个简单的正交投影矩阵，它使得一个点的x,y分量不变化，同时使其z分量为0，也就是说，它把物体正交地投影到平面z = 0：

（等式略，见原文89页）

这个投影的效果如图4.16所示。显然，Po是不可逆的，因为它的行列式|Po| = 0。换句话说，这个变幻从三维降维到二维，并且没有办法恢复降掉的那一维。使用这种正交投影来观察的一个问题是它同时把z轴正向和负向的点都投影到了投影平面。我们往往把z值（也包括x,y值）限制在一个固定范围内，从被称为n（近平面）到f（远平面）。这也是下一个变换的目的。一个更常用的正交投影矩阵使用六元组表示（l, r, b, t, n, f）分别代表左，右，底，顶，近和远平面。这个矩阵的实质是缩放和平移AABB，它是由这些平面构成的一个以原点为中心的轴向立方体。AABB的最小顶点为（l, b, n）最大顶点为（r, t, f）。重要的是我们要注意到n > f，因为我们在这个体积空间中是沿z轴负方向看得。我们的常识告诉我们近的数值应该要比远的小。同样沿z轴负方向的OpenGL将他们在正交矩阵生成过程glOrtho中的输入近值小于远值，然后在内部对这两个值取反。另一种理解的方法是OpenGL的近远值是沿着视角方向（z轴负向）的（正数）距离，而不是z视坐标值。在OpenGL中，这个轴向立方体的最小顶点为（-1，-1，-1）最大顶点为（1，1，1）；而在DirectX中则为（-1，-1，0）和（1，1，1）。这个立方体被称为“规则观察空间”（canonical view volume）同时这个空间内的坐标被称为“规范化设备坐标”（normalized device coordinates）。这个变换过程如图4.17所示。至于变换到规则观察空间内的原因是在这里的裁剪要效率得多。

在变换到规则观察空间后，将要渲染的多边形的顶点将会被裁剪到这个立方体上。立方体内的多边形最后会通过映射保留的单位立方体到屏幕上来渲染。这个正交变换如下所示：

（等式略，见原文91页）

通过这个等式，Po可以被写作一个平移T(t)接着一个缩放矩阵S(s)的组合，这里s = (2/(r − l), 2/(t − b), 2/(f − n))，t = (−(r + l)/2, −(t + b)/2, −(f + n)/2)。这个矩阵是可逆的，也就是说，P−1 o = T(−t)S((r − l)/2,(t − b)/2,(f − n)/2)。

在计算机图形学中，投影后往往使用左手坐标系——也就是说，对于视窗，x轴指向右，y轴指向上，z轴指入视窗。因为我们定义AABB的方式中远值小于近值，正交变换总是会包含一次镜像变换。关于这点，假设原始的AABB与目标规则观察空间大小相同。也就是说AABB的坐标是（l, b, n）为（-1, -1, 1）同时（r, t, f）为（1, 1, -1）。等式4.60如下

（等式略，见原文92页）

是一个镜像矩阵。这个镜像转换使得右手视觉坐标系（沿着z轴负向看）翻转到左手规范化设备坐标系。

DirectX将z深度映射到区间[0, 1]而不是OpenGL的[-1, 1]。这可以通过在正交矩阵之后使用一个简单的缩放和平移矩阵来实现，也就是，

（等式略，见原文92页）

因此DirectX中使用的正交矩阵为

（等式略，见原文92页）

它通常以转置的形式出现，因为DirectX使用先行后列（row-major）的形式表达矩阵。

4.6.2 透视投影

一种比正交投影有趣的多的变换是透视投影，它用于绝大多数计算机图形学的应用。此时，本来平行的直线在投影后不再平行；相反，他们可能在极限处相交于一个点。透视更符合我们对这个世界的认知，也就是近大远小。

首先，我们会展示一个投影到平面z = -d, d > 0的透视投影矩阵的有益推导。我们从世界空间开始推导以简单理解世界到视觉空间的转换是如何生效的。在推导之后是，举个例子，OpenGL中更传统更常用的矩阵。

假设摄像机（视点）位于原点，我们想要投影一个点p到平面z = -d, d > 0, 从而获得一个新的点q = (qx, qy, -d)。图4.18描述了这个小剧本。利用图中所示的相似三角形，可以得到关于q的x分量的如下推导：

（等式略，见原文93页）

关于q的其他分量的表达式为qy = −dpy/pz（取得方式类似qx），qz = −d。与上方等式结合，我们可以得到如下所示的透视投影矩阵Pp：

经过等式4.66的简单验证，证明这个矩阵可以得到正确的透视投影：

（等式略，见原文94页）

证明的最后一步是整个向量除以w分量（此情况下为−pz/d），以使最后一个位置为1。结果中z的值总是-d，因为我们是投影到此平面上的。

直观上，很容易理解为什么齐次坐标允许投影。齐次化的一个几何解释是它将点（px, py, pz）投影到平面w = 1上。与正交转换类似，这里也有一个透视转换，它并不是真的投影到一个平面（这是不可逆转的），而是把视锥转换到前面所说的规则观察空间。这里我们假设视锥开始于z = n结束于z = f，0 > n > f。z = n处的矩形最小顶点为(l, b, n)最大顶点为(r, t, n)。如图4.19所示。

参数(l, r, b, t, n, f)定义了相机的视锥。视角的水平区间取决于锥体左右两侧平面（由l和r定义）的夹角。同理，视角的垂直区间取决于顶部和底部的两个平面（由t和b定义）的夹角。视角的范围越大，相机可以“看到”的就越多。不对称的锥体可以通过r ≠ −l 或 t ≠ −b来建立。举个例子，不对称锥可以用于立体视觉（见18.1.4节）以及CAVEs。

视野的范围对于提供一个场景的感觉是非常重要的一个方面。眼睛本身的视野相对于计算机屏幕具有一个物理区间。这个关系是

φ = 2 arctan(w/(2d)),

此处φ是视野的区间，w是垂直于视线的物体的宽度，d是物体的距离。举个例子，一个21英寸的显示器大约有16英寸宽，同时25英寸是一个最小的推荐观看距离，这也就产生了35度的物理视野。在12英寸的距离，这个视野是67度；18英寸时，是48度；30英寸时，是30度。同样的公式也可以用于从相机尺寸计算视野，例如，一个标准50mm镜头的35mm相机（具有36mm帧宽）得到φ = 2 arctan(36/(2 ∗ 50)) = 39.6度。