

## Chapter 2 The Graphics Rendering Pipeline

### 图形渲染管线

Anonymous——“A chain is no stronger than its weakest link.”

佚名——“链条的坚固程度取决于它最薄弱的环节。”（一着不慎，满盘皆输。）

本章节介绍了实时图形学中的核心组件，它被称为“图形渲染管线（graphics rendering pipeline）”，也被简称为“管线”。渲染管线的核心功能就是利用给定的虚拟相机、三维物体、光源等信息，来生成或者渲染（render）一张二维图像。因此，渲染管线是实时渲染中的底层工具，其功能如图 2.1 所示。最终图像中物体的位置和形状，由其几何结构，环境特征以及相机位置所决定。而物体的外观则会受到材质属性、光源、纹理（应用在物体表面上的图像）以及着色方程的影响。

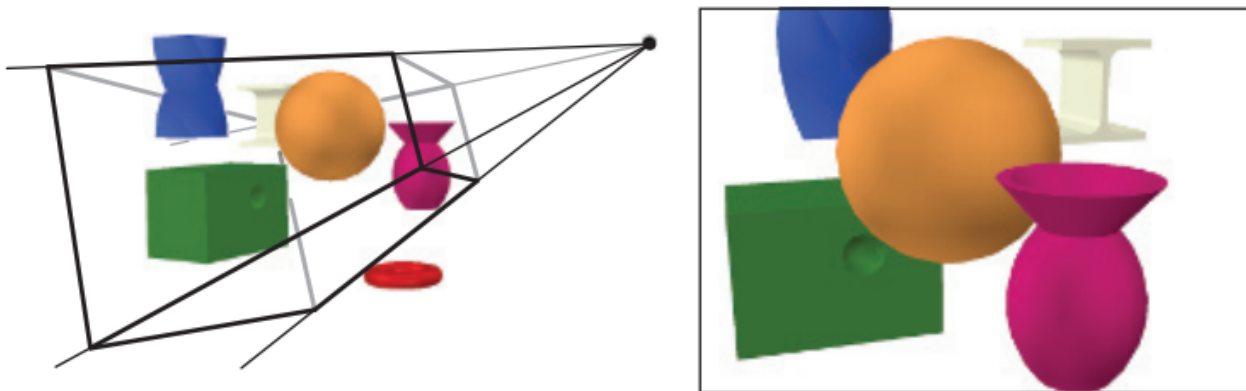


图 2.1: 在左图中，虚拟相机位于“金字塔”的顶端（四条直线汇聚的地方），只有位于可视空间内部的物体才会被渲染。对于一个透视渲染的图像而言（如这里的例子），可视空间是一个截锥体（frustum，复数形式为 frusta），通常被叫做视锥体，即一个带有矩形底座的、被截断的金字塔形状。右图显示了虚拟相机所看到的画面，请注意左图中的红色甜甜圈并没有出现在右图中，这是因为它位于视锥体之外；同时左图中扭曲的蓝色棱柱体被视锥体的上平面裁剪了。

我们将会介绍渲染管线中的各个阶段，在本章节中，我们将会重点关注各个阶段的功能而不是实现方式，有关如何应用这些阶段的内容，将在后续章节中进行详细介绍。

## 2.1 渲染管线的架构

在现实世界中，流水线（pipeline）的概念有很多种表现形式，从工厂的装配线到快餐厨房等，它也同样适用于图形渲染领域。一个流水线中会包含若干个阶段，每个阶段负责完成总任务中的一部分任务[715]。

流水线阶段可以并行执行，其中每个阶段都依赖于前一个阶段的结果。在理想状态下，一个非流水线化的系统可以被划分为  $n$  个流水线阶段，从而提升  $n$  倍的速度，这种性能上的提升是使用流水线的主要原因。举个例子，一组员工就可以快速制作一大批的三明治：一个人负责准备面包，一个人负责添加肉片，另一个人负责添加其他配料。每个人都会将结果传递给流水线上的下一个人，同时立即开始下一个三明治的制作。如果每个人都需要 20 秒的时间来完成各自的任務，那么每 20 秒就可以生产一个三明治，一分钟就可以生产 3 个。虽然流水线阶段可以并行进行，但是整个流水线的效率会被执行速度最慢的那个阶段所影响。比如说：如果给三明治加肉片的阶段变得比之前更复杂了，现在需要 30 秒时间才能完成，那么现在流水线的最快速度就是一分钟生产两个三明治了。对于这个三明治流水线而言，加肉阶段就是整个流水线的瓶颈，因为它决定了整个生产过程的最终速度。在等待加肉阶段完成之前，加料阶段是“饥饿（starved）”的（用户也是）。

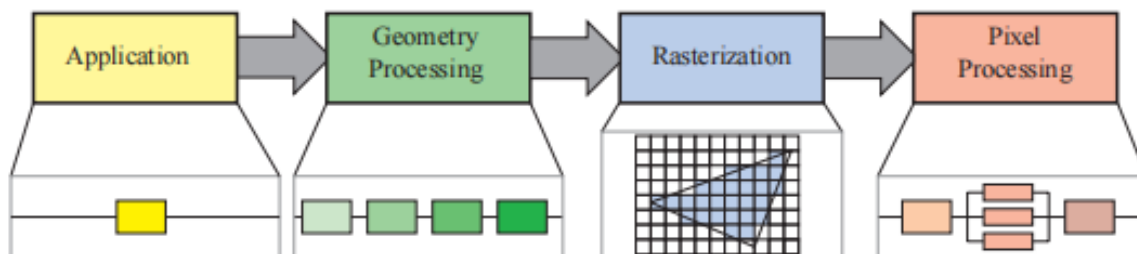


图 2.2：渲染管线的基本结构，包含以下四个阶段：应用阶段、几何处理阶段、光栅化阶段和像素处理阶段。每个阶段本身也可以是一个流水线，例如几何处理阶段下方的图示（包含 4 个子阶段）；有些阶段也可以（部分）并行化执行，例如像素处理阶段下方的图示（三个子阶段并行执行）。在上图中，应用阶段是一个单独的处理过程，但是这个阶段同样也可以流水线化或者并行化。另外请注意，光栅化会查找一个图元内部的像素，例如一个三角形内部的像素。

在实时渲染领域中也都可以找到这样的流水线结构，一种粗略的划分方法是将渲染管线分为四个阶段——应用阶段（application）、几何处理阶段（geometry processing）、光栅化阶段（rasterization）和像素处理阶段（pixel processing），如图 2.2 所示。这个结构（渲染管线的引擎）是实时计算机图形程序的核心，也是后续章节的基础概念。每个阶段本身通常也是一个流水线，这意味着每个阶段也是由几个子阶段构成的。在这里我们将功能性阶段（functional stage）

（如上图所示）和其实现结构的概念（the structure of implementation）区分开来。一个功能性阶段有一个特定的任务需要执行，但是并没有指定它在管线中的实现方式。一个给定的实现方式也可能会将两个功能性阶段合并成一个功能单元，或者使用可编程核心来执行；同时它也可以将一个很耗时的功能性阶段，划分为几个硬件单元来完成。

渲染速度可以用每秒帧数（FPS）来进行表示，即每秒显示的帧数；也可以用赫兹（Hz）来进行表示，这个单位代表了  $1/\text{seconds}$ ，即更新的频率。通常我们也会直接使用渲染一张图像所花费的时间（毫秒）来表示渲染速度，生成每帧图像所花费的时间往往并不相同，这取决于每帧中所执行计算的复杂度大小。FPS 可以用来表示某一帧的速率，也可以表示一段时间内的平均性能。Hz 一般被设定成一个固定的值，多用于硬件设备中，例如显示器等。（FPS：帧数，Hz：硬件刷新率）。

顾名思义，应用阶段（application）是由应用程序进行驱动的，它在软件中进行实现，运行在通用 CPU 上。这些 CPU 一般都具有多个核心，可以并行处理多个线程（thread）的任务，这使得 CPU 可以高效执行由应用阶段所负责的各种任务，一般 CPU 会负责碰撞检测，全局加速算法，动画，物理模拟等任务，具体会执行哪些任务取决于应用程序的类型。下一个主要阶段是几何处理阶段（geometry processing），它负责处理变换（transform），投影（projection）以及其他所有和几何处理相关的任务。这个阶段需要计算哪些物体会被绘制，应该如何进行绘制，以及应当在哪里绘制等问题。几何阶段通常运行在硬件处理单元（GPU）上，它包含一系列的可编程单元和固定操作硬件。光栅化阶段（rasterization）通常会将构成一个三角形的三个顶点作为输入，找到所有位于三角形内部的像素，并将其转发到下一个阶段中。最后一个阶段是像素处理阶段（pixel processing），对于每个像素而言，都会执行一个程序来决定它的颜色；并执行深度测试，来判断这个像素是否可见；这里还可以执行一些逐像素的操作，例如将新计算的颜色和之前的颜色进行混合。光栅化阶段和像素处理阶段同样完全运行在 GPU 上。所有这些阶段及其内部的子流水线阶段，将会在接下来的四个小节中进行讨论。有关 GPU 如何处理这些阶段的细节，详见第 3 章。

## 2.2 应用阶段

由于应用阶段通常都运行在 CPU 上，因此开发者可以完全控制在应用阶段发生的事情，开发者可以决定应用程序的具体实现方式，也可以在之后对其进行修改优化，从而提高程序的性能表现。对应用阶段的修改也会影响后续阶段的性能表现。例如：一个应用阶段中的算法或者设置，可以减少后续需要进行渲染的三角形数量。

尽管如此，有一些应用阶段中的任务也可以让 GPU 来进行执行，即通过使用一个叫做计算着色器（compute shader）的独立模式，该模式会将 GPU 视为一个高度并行的通用处理器，而忽略其专门用于图形渲染的特殊功能。

在应用阶段的最后，需要进行渲染的几何物体会被输入到几何处理阶段中，这些几何物体被称作为渲染图元（rendering primitive），即点、线和三角形。这些图元最终可能会出现在屏幕上（或者是任何正在使用的显示设备上），这也是应用阶段中最重要的任务。

由于这一阶段基于软件实现（software-based），其中一个结果就是，它并没有像之后的几何处理、光栅化和像素处理等阶段一样，被进一步划分出子阶段。

由于 CPU 自身就是一个规模很小的流水线，因此我们也可以说，应用阶段也被进一步划分为了几个子阶段，但是这与本章节的主题关系不大。

但是为了提升效率，应用阶段通常也会利用多个处理器核心来并行化执行，在 CPU 设计中，这被称为多核（multi-core）构造，因为它可以在同一阶段执行多个进程。在[章节 18.5](#) 中我们介绍了一些多核心调度的方法。

碰撞检测（collision detection）通常会在这个阶段中实现。当检测到两个物体之间的碰撞之后，会产生相应的响应，并返回给碰撞物体，同时也返回给力反馈设备（如果有的话）。应用阶段同样也是处理其他来源输入的地方，例如键盘、鼠标或者头戴式显示器等，会根据不同的输入，从而采取不同的操作。此外，一些加速算法例如特殊的剔除算法（[第 19 章](#)）等，以及渲染管线剩余部分无法处理的一切问题，都会应用阶段中完成。

## 2.3 几何处理阶段

运行在 GPU 上的几何处理阶段会负责大部分的逐三角形（per-triangle）和逐顶点（per-vertex）操作。将几何处理阶段再细分下去，可以划分为以下几个功能性阶段：顶点着色（vertex shading）、投影（projection）、裁剪（clipping）和屏幕映射（screen mapping），如[图 2.3](#) 所示。



图 2.3：将几何处理阶段进一步划分成一个包含若干功能性阶段的流水线。

## 2.3.1 顶点着色

顶点着色 (vertex shading) 的任务主要有两个，一个是计算顶点的位置，另一个是计算那些开发人员想要作为顶点数据进行输出的任何参数，例如法线 (normal) 和纹理坐标 (texture coordinate) 等。在早些时候，物体的光照是逐顶点计算的，通过将光源应用于每个顶点的位置和法线，从而计算并存储最终的顶点颜色；然后再通过对顶点颜色进行插值，来获取三角形内部像素的颜色，因此这个可编程的顶点处理单元被命名为顶点着色器 (vertex shader) [1049]。随着现代 GPU 的出现，以及几乎全部的着色计算都在逐像素的阶段进行，因此顶点着色阶段变得越来越通用，甚至可能并不会在该阶段中进行任何的着色计算，当然这也取决于开发人员的意图，我们仍然可以在顶点着色器中进行着色计算。顶点着色器如今是一个更加通用的单元，它负责计算并设置与每个顶点都相关的数据。例如在[章节 4.4](#)和[章节 4.5](#)中，顶点着色器可以用来计算物体的动画。

首先我们描述一下顶点位置是如何被计算出来的，它需要一组顶点坐标来作为输入。在物体最终进入屏幕的过程中，它需要在不同的空间 (space) 或者坐标系

(coordinate system) 下，进行若干次变换。最开始时，模型位于自身的模型空间 (model space) 中，也可以简单地认为它没有进行任何变换。每个模型都可以与一个模型变换 (model transform) 相关联，以便调整自身的位置和朝向。我们可以将若干个模型变换和同一个模型相关联，这样我们就能够在不复制这个模型的前提下，在一个场景中放置同一个模型的多个副本 (也叫做实例，instance)，每个实例都拥有各自不同的位置和朝向 (即模型变换)。

模型变换会对模型的顶点和法线进行变换，模型本身所处的坐标系叫做模型坐标系 (model coordinates)，它的坐标也被称为模型坐标。当对这些坐标进行模型变换之后，这个模型便位于世界坐标系 (world coordinate) 或者叫做世界空间 (world space) 中。世界空间是唯一的，当各个模型经过各自的模型变换之后，所有的模型便都位于一个相同的空间中。

如前文所述，只有被相机 (或者观察者) 看到的模型才会被渲染，相机在世界空间中有一个位置参数和方向参数，用于放置相机和调整相机的朝向。为了便于后续的投影操作和裁剪操作，相机和世界空间中的所有模型，都会应用观察变换 (view transform)，观察变换的主要目的是将相机放置在原点上，并调整相机的朝向，使其看向  $z$  轴负半轴的方向，同时  $y$  轴指向上方， $x$  轴指向右方。本文中我们将会使用指向负  $z$  轴的约定，有一些书籍则更喜欢让相机看向正  $z$  轴的方向，二者之间的区别主要是语义上的，因为二者之间的转换十分简单。在应用观察变换之后，模型的具体位置和具体方向取决于底层图形 API 的实现方式。这样形成的空间被称作相机空间

(camera space)，或者是更加常见的观察空间 (view space) 和眼睛空间 (eye space)。图 2.4 是观察变换的一个例子，它展示了观察变换是如何对相机和模型产生影响的。模型变换和观察变换都是使用  $4 \times 4$  矩阵实现的，我们将在第 4 章讨论这个话题。这里我们需要认识到，顶点的坐标和法线，都可以按照程序员所喜欢的任何方式来进行计算。

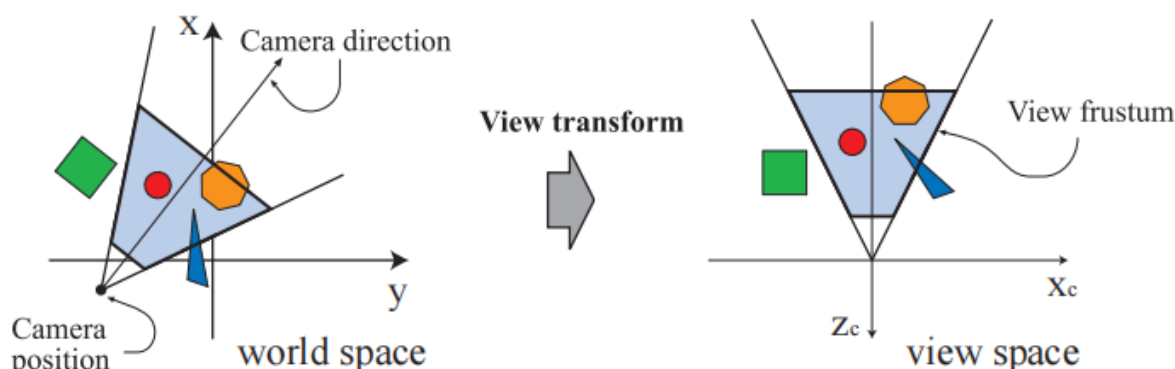


图 2.4: 左图是一个俯视图，虚拟相机按照用户的想法进行放置，在这个空间中，正  $z$  轴指向上方。观察变换重新定位了这个空间，使得相机位置位于原点，看向负  $z$  轴的方向，同时正  $y$  轴指向上方，正  $x$  轴指向右方。这样做可以使得投影操作和裁剪操作变得更加简单。图中的蓝色区域是可视空间，这里假设相机进行的是透视观察，因为此时视野的形状是一个视锥体。类似的变换操作可以应用于任何类型的投影。

接下来，我们将描述顶点着色的第二类输出。为了创建一个真实的场景，仅仅是渲染物体的位置和形状是不够的，我们还需要对物体的外观信息进行建模，包括物体的材质 (material) 信息以及光源照射在物体表面上的效果。从最简单的颜色描述到基于物理的详细描述，材质和光源可以通过很多方式进行建模。

确定光照作用于材质上所产生的效果，这个操作被称为着色，它涉及到在模型的不同位置上计算着色方程。通常来说，其中一些计算是在模型顶点的几何处理阶段中执行的，其他计算可能会在逐像素处理中完成。顶点上可以存储各种各样的数据，例如顶点位置、法线、颜色或者着色方程所需要的其他数值信息。顶点着色的结果（可能是颜色、向量、纹理坐标或者其他类型的着色数据）会被发送到光栅化阶段中进行插值，并在像素处理阶段中用于计算表面的着色。

顶点着色在 GPU 中会以顶点着色器的形式体现，我们将在本书中进行更加深入的讨论，尤其是第 3 章和第 5 章。

作为顶点着色的一部分，渲染系统还会进行投影操作和裁剪操作，这两个操作会将整个可视空间变换为一个标准立方体 (standard cube)，其端点位于  $(-1, -1, -1)$  和  $(1, 1, 1)$  处，立方体的边长为 2。可以使用不同的范围来定义相同的空间，例如

$0 \leq z \leq 1$ ，这个标准立方体被称为规范可视空间（canonical view volume）。首先会进行投影操作，这是在 GPU 上的顶点着色器中完成；有两种常见的投影方法，一种是正交投影（orthographic），也可以叫做平行投影（parallel）；另一种是透视投影（perspective），如图 2.5 所示。事实上，正交投影是平行投影中的一种，其他类型的平行投影也有一些应用，尤其是在建筑领域，例如：斜平面投影（oblique）和轴测投影（axonometric），有一款老式街机游戏《Zaxxon》就使用了轴测投影方式。

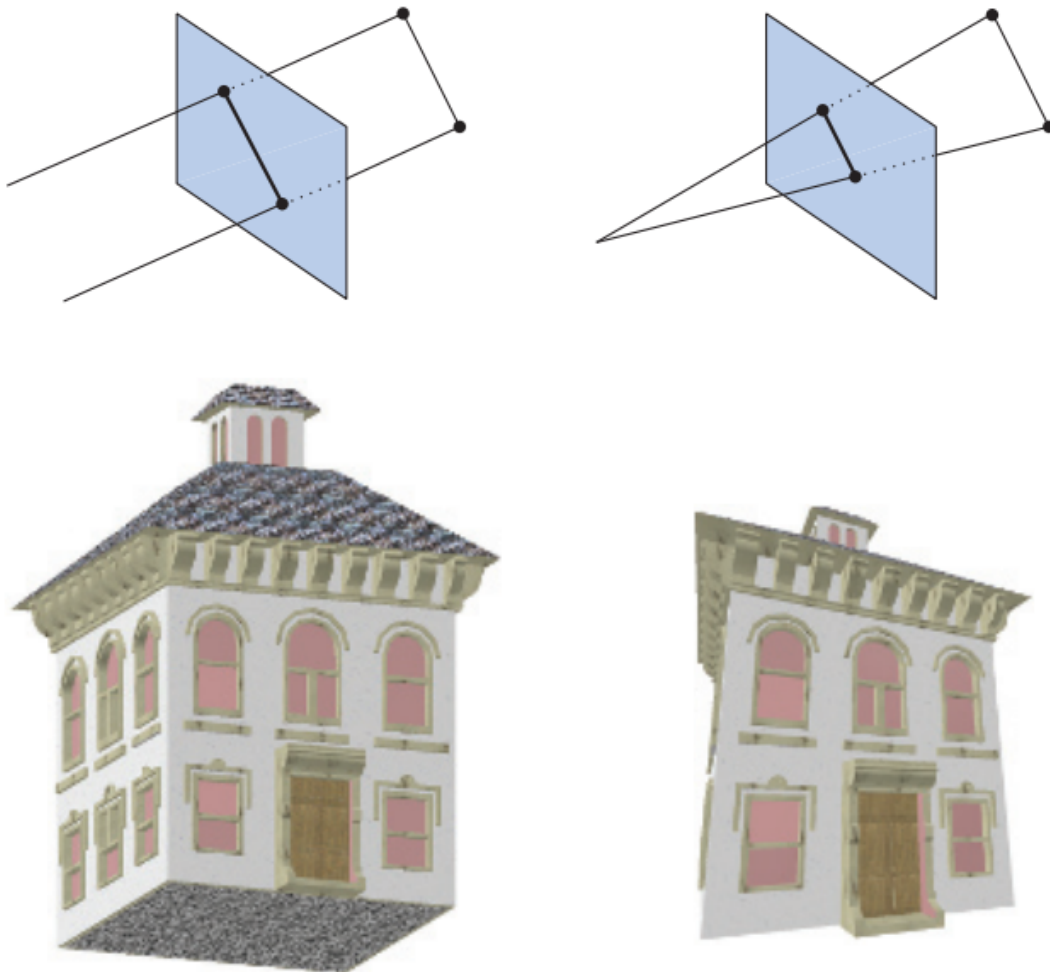


图 2.5：左侧是正交投影，或者叫做平行投影；右侧是透视投影。

请注意，投影操作是通过投影矩阵（[章节 4.7](#)）完成的，因此有时候它会和其余几何变换连接起来。

正交视图的可视空间通常是一个长方体，正交投影会将这个可视空间变换为一个标准立方体。正交投影最主要的特征就是，投影变换之前的平行线，在正交投影之后仍然是平行的。这样的投影变换由一个位移变换和一个缩放变换组成。

透视投影要更加复杂一点。在透视投影中，距离相机越远的物体，在投影变换之后就越小；此外，平行线在透视投影之后也会在视界处汇聚，也就是说，透视投影模拟了

我们感知物体大小的方式。从几何学上看，透视投影的可视空间（也叫做视锥体）是一个具有矩形底面的截断金字塔，这个视锥体也会被投影变换为一个标准立方体。正交投影和透视投影都可以使用一个  $4 \times 4$  矩阵来进行描述（第 4 章），在投影变换之后，模型所处的坐标系被称为裁剪坐标系（clip coordinates），在坐标除以  $w$  分量之前，事实上它们都是齐次坐标（homogeneous coordinate），我们将会在第 4 章中进行详细讨论。GPU 的顶点着色器必须始终输出这种类型的坐标，以便于下一个功能性阶段（裁剪）可以正确执行。

尽管这些投影变换矩阵会将模型从一个空间变换到另一个空间，但是它们仍然被叫做投影，这是因为在显示之后，坐标的  $z$  分量并不会被存储在生成的图像中，而是存储在一个叫做  $z$ -buffer 的地方，详见章节 2.5。通过这种方式，模型便从三维空间投影到了二维空间中。

### 2.3.2 可选的顶点处理

每个渲染管线中，都会有刚才所描述的顶点处理阶段，当完成顶点处理之后，还有几个可以在 GPU 上执行的可选操作，它们的执行顺序如下：曲面细分

（tessellation）、几何着色（geometry shading）和流式输出（stream out）。是否使用这些可选操作，一方面取决于硬件的功能（并不是所有 GPU 都支持这些功能），另一方面取决于程序员的意愿。这些功能相互独立，而且一般并不是很常用，详细内容见第 3 章。

第一个可选阶段是曲面细分（tessellation），想象现在有一个使用三角形进行表示的弹性小球，我们可能会遇到质量和性能的取舍问题。在 5 米外观察这个小球可能看起来会很不错，但是如果离近了看，我们会发现部分三角形，尤其是小球轮廓边缘处的三角形会非常明显。如果我们给这个小球添加更多的三角形来提高表现质量，那么当这个小球距离相机很远，仅仅占据屏幕上几个像素的时候，我们会浪费大量的计算时间和内存。这个时候，使用曲面细分可以为一个曲面生成数量合适的三角形，同时兼顾质量和效率。

在前文中我们已经讨论了一些有关三角形的话题，但是到目前为止，我们在管线中只对顶点进行了处理。这些顶点可以用来表示点、线、三角形或者其他物体，顶点也可以用来描述一个曲面（例如一个球体），这样的曲面可以通过几个部分组合而成，每个部分也都由一组顶点构成。曲面细分阶段本身也包含了一系列子阶段——壳着色器（hull shader）、曲面细分器（tessellator）和域着色器（domain shader），它们可以将当前的顶点集合（通常）转换为更大的顶点集合，从而创建出更多的三角形。

场景中的相机位置可以用来决定需要生成多少个三角形：当距离相机很近时，则生成较多数量的三角形；当距离相机很远时，则生成较少数量的三角形。

下一个可选阶段是几何着色器（geometry shader），这个着色器出现的比曲面细分着色器更早，因此在 GPU 上也更加常见。它和曲面细分着色器的相似点在于，它也将各种类型的图元作为输入，然后生成新的顶点。这是一个较为简单的阶段，因为它能够创建的范围是有限的，能够输出的图元则更加有限。几何着色器有好几种用途，其中最流行的一种就是用来生成粒子。想象我们正在模拟一个烟花爆炸的过程，每颗火花都可以表示为一个点，即一个简单的顶点。几何着色器可以将每个顶点都转换成一个正方形（由两个三角形组成），这个正方形会始终面朝观察者，并且会占据若干个像素，这为我们提供了一个更加令人信服的图元来进行后续的着色。

最后一个可选阶段叫做流式输出（stream out）。这个阶段可以让我们把 GPU 作为一个几何引擎，我们可以选择将这些处理好的数据输入到一个缓冲区中，而不是将其直接输入到渲染管线的后续部分并直接输出到屏幕上，这些缓冲区中的数据可以被 CPU 读回使用，也可以被 GPU 本身的后续步骤使用。这个阶段通常会用于粒子模拟，例如我们刚才所举的烟花案例。

以上三个阶段会按照曲面细分、几何着色和流式输出的顺序进行执行，每一个阶段都是可选的。无论我们选择使用其中的哪些阶段，最终我们都会拥有一组使用齐次坐标进行表示的顶点，并输入到管线的下一阶段中，并检查这些顶点是否会被相机看见。

### 2.3.3 裁剪

只有完全位于可视空间内部，或者部分位于可视空间内部的图元，才需要被发送给光栅化阶段（以及后续的像素处理阶段），然后再将其绘制到屏幕上。完全位于可视空间内部的图元，将会按照原样传递给下一阶段；完全位于可视空间之外的图元，将不会传递给下一阶段，因为它们是不可见的，也不会被渲染；而对于那些一部分位于可视空间内部，一部分位于可视空间外部的图元，则需要进行额外的裁剪操作。例如：一个顶点在外，一个顶点在内的线段会被可视空间裁剪（clip），裁剪之后会生成一个新的顶点，用来替代可视空间之外的那个顶点，这个新顶点位于线段和可视空间的交点处。我们使用投影矩阵将可视空间变换为一个标准立方体，这意味着所有的图元都需要被这个标准立方体所裁剪。使用观察变换和投影变换保证了裁剪操作的一致性，即图元始终只需要针对这个标准立方体进行裁剪即可。

裁剪过程如图 2.6 所示，除了可视空间的六个裁剪平面之外，用户还可以定义额外的裁剪平面来对物体进行裁剪，这种操作被称为切片（sectioning），图 19.1 展示了这一过程。

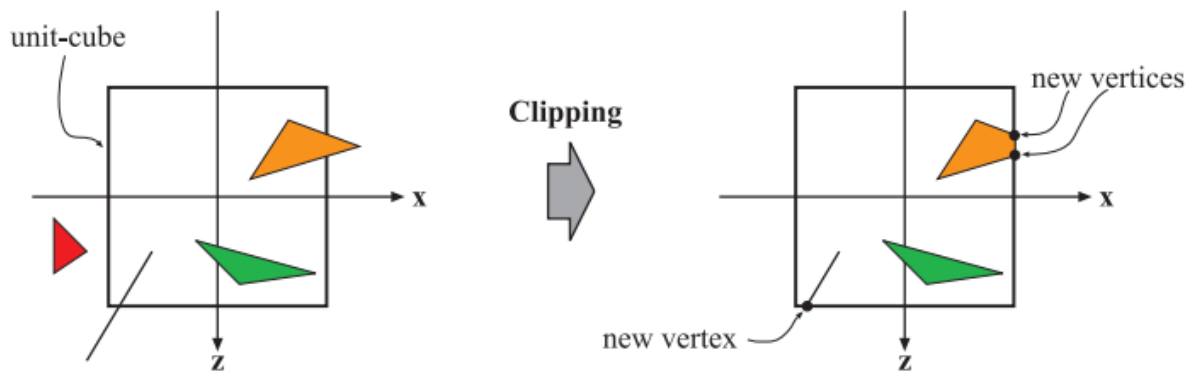


图 2.6: 在投影变换之后, 只有位于标准立方体内部的图元 (对应视锥体内部的模型), 才需要进行后续处理。因此完全位于标准立方体外部的图元将会被直接丢弃, 完全位于标准立方体内部的图元将会被保留。而与标准立方体相交的图元, 将会被标准立方体裁剪, 即生成一些新的顶点, 而位于立方体之外的旧顶点将会被直接丢弃。

这里我们会使用投影变换生成的四维齐次坐标, 来完成这个剪切操作, 齐次坐标在透视空间中的三角形上进行的插值, 通常并不是线性的, 同时我们需要使用齐次坐标的第四个值, 以便在透视投影之后进行正确的插值和裁剪。最后会进行透视除法 (perspective division), 将得到的三角形位置转换到三维标准化设备坐标系 (normalized device coordinates, NDC) 中。前文中我们提到, 这个标准立方体的范围是  $(-1, -1, -1)$  到  $(1, 1, 1)$ , 几何处理阶段的最后一步就是将这个空间转换为窗口坐标系。

### 2.3.4 屏幕映射

只有位于可视空间内部的图元才会被传递到屏幕映射 (screen mapping) 阶段。当这些图元进入这一阶段时, 其坐标还是三维的, 其中  $x$  坐标和  $y$  坐标会被转换为屏幕坐标 (screen coordinate), 屏幕坐标和  $z$  坐标在一起, 被称作窗口坐标

(window coordinate)。假设这个场景会被渲染到一个窗口中, 窗口左下角的坐标为  $(x_1, y_1)$ , 右上角的坐标为  $(x_2, y_2)$ , 其中  $x_1 < x_2, y_1 < y_2$ 。屏幕映射包含了一个缩放操作, 映射后的新  $x, y$  坐标会被称为屏幕坐标。 $z$  坐标同样也会被映射到  $[z_1, z_2]$  的范围中, 默认是  $z_1 = 0, z_2 = 1$  (OpenGL 中的范围是  $[-1, +1]$ , DirectX 中的范围是  $[0, 1]$ ), 但是这些范围也可以使用相应的 API 来进行修改。窗口坐标和这个被重映射的  $z$  值一起, 都会被传递到光栅化阶段中。屏幕映射的过程如图 2.7 所示。

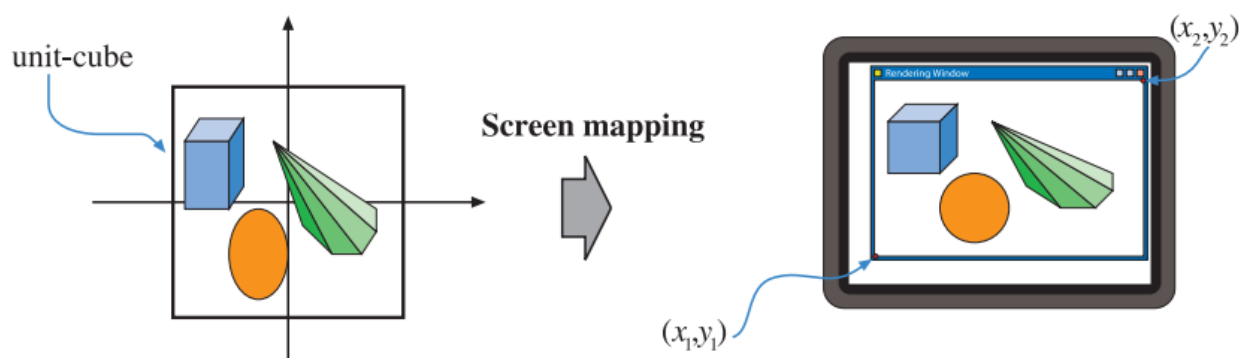


图 2.7：投影变换之后的图元都位于标准立方体内部，屏幕映射负责找到图元在屏幕上的坐标位置。

接下来我们将描述整型数值和浮点型数值与像素（以及纹理坐标）之间的对应关系，给定一个使用笛卡尔坐标进行描述的水平像素数组，那么最左侧像素的中心点坐标为 0.5，该像素左边界的坐标为 0.0。OpenGL 以及 DirectX 10 之后的 API 均采用了这一规定。因此， $[0, 9]$  范围的像素占据了  $[0.0, 10.0)$  的范围，也就是说，转换规则为：

$$d = \text{floor}(c), \quad (2.1)$$

$$c = d + 0.5, \quad (2.2)$$

其中  $d$  代表了像素的位置索引（整型）， $c$  代表了像素内的连续值（浮点类型）[\[692\]](#)。

虽然所有 API 都定义像素值从左向右会不断增大，但是在某些情况下，OpenGL 和 DirectX 在竖直方向上的起始位置是不一样的。

“Direct3D”是 DirectX 的三维图形 API，DirectX 中还包含了许多其他 API，例如输入控制和音频控制等。在讨论这个 API 的时候，我们一般不去区分“Direct3D”和 DirectX 之间的区别，这里我们统一使用“DirectX”。

OpenGL 倾向于全部使用笛卡尔坐标系，始终将最左下角的像素视为最小像素（即起始位置的像素）；而在 DirectX 中，根据上下文的不同，有时候会将左上角的位置作为最小像素。在这个问题上并没有一个标准答案，每个规定都有着自己的逻辑。例如：在 OpenGL 中， $(0, 0)$  点位于屏幕的左下角；而在 DirectX 中，这个点位于屏幕的左上角。当我们想要迁移 API 的时候，这个差异是十分重要的。

## 2.4 光栅化阶段

现在我们已经有了被正确变换和正确投影的顶点数据，以及它们相应的着色数据（在几何处理阶段中获取的），下一阶段的目标是找到位于待渲染图元（例如三角形）中的所有像素值（pixel, picture element 的缩写）。我们将这个过程称为光栅化（rasterization），我们可以将其划分成两个子阶段：三角形设置（triangle set up, 也叫做图元装配, primitive assembly）和三角形遍历（triangle traversal），其过程如图 2.8 所示。这里需要注意的是，这两个子阶段也同样适用于点和线，只是因为三角形是更加常见的图元，因此这两个子阶段的名字中才带有了三角形。光栅化也被称为扫描变换（scan conversion），这是一个将屏幕空间中二维顶点，转换到屏幕上像素的过程，其中每个顶点都对应一个  $z$  值（深度缓冲）和各种各样的着色信息。光栅化也可以被认为是一个几何处理阶段和像素处理阶段之间的同步点，因为光栅化阶段的三角形，是由几何处理阶段输出的顶点组成的，并且最终会输出到像素处理阶段中。

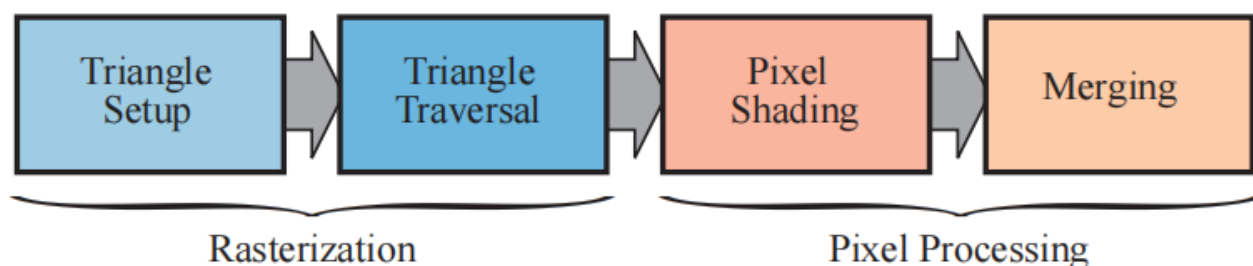


图 2.8：左侧：光栅化阶段被划分为了两个功能性阶段，分别被称为三角形设置和三角形遍历。右侧：像素处理阶段也被划分为了两个功能性阶段，分别被称为像素着色和合并。

判定某个三角形和屏幕上的哪些像素重合，取决于我们如何实现 GPU 管线。例如：你可以使用点采样（point sample）来判定某个点是否位于三角形内部。最简单的方式就是直接将每个像素的中心点来作为该像素的样本，如果该像素的中心点位于三角形内部的话，那么我们就认为该像素也位于三角形的内部。我们还可以通过超采样（supersampling）或者多重采样抗锯齿技术（multisampling antialiasing），来对每个像素进行多次采样，详见[章节 5.4.2](#)。另一种方法是使用保守光栅化（conservative rasterization），即当某个像素只要有一部分与三角形重叠时，我们就认为该像素位于三角形内部，详见[章节 23.1.2](#)。

### 2.4.1 三角形设置

三角形的微分（differential）、边界方程（edge equation）和其他数据，都会在这个阶段进行计算，这些数据可以用于三角形遍历（[章节 2.4.2](#)），以及对几何处理阶

段产生的各种着色数据进行插值。这个功能一般会使用固定功能的硬件实现。

## 2.4.2 三角形遍历

在这一阶段，会对每个被三角形覆盖的像素（中心点或者样本点在三角形内部的像素）进行逐个检查，并生成一个对应的片元（fragment）。我们可以在[章节 5.4](#) 中找到更加详细的采样方法。找到那些位于三角形内部的点或者样本，这个过程通常被称为三角形遍历，并且会对三角形三个顶点上的属性进行插值，来获得每个三角形片元的属性（[第 5 章](#)），这些属性包括片元的深度，以及几何阶段输出的相关着色数据等。McCormack 等人[\[1162\]](#)提供了有关三角形遍历的更多信息。在光栅化阶段也会对三角形进行透视正确的插值[\[694\]](#)（[章节 23.1.1](#)）。片元内部的像素或者样本会被输入到像素处理阶段中，下面我们将会对其进行介绍。

## 2.5 像素处理阶段

经过之前若干阶段的处理，这里我们已经找到了所有位于三角形（或者其他图元）内部的像素。像素处理阶段也可以被划分为像素着色（pixel shading）和合并（merging）两个阶段，如[图 2.8](#) 所示。在像素处理阶段，会对图元内部的像素（或者样本）进行逐像素（或者逐样本）的计算和操作。

### 2.5.1 像素着色

这里会使用插值过的着色数据作为输入，来进行逐像素的着色计算，其结果是生成一个颜色值或者多个颜色值，这些颜色值会被输入到下一阶段中。三角形设置和三角形遍历使用了专门的硬件单元进行执行，而像素着色阶段则是由可编程的 GPU 核心来执行的。为此，程序员需要为像素着色器（在 OpenGL 中叫做片元着色器）提供一个实现程序，这个程序中包含了任何我们想要的着色计算操作。这里可以使用各种各样的技术，其中最重要的一个技术就是纹理化（texturing），详见[第 6 章](#)。简单来说，纹理化就是将一个图像或者多个图像“粘合（gluing）”在物体表面，从而实现各种各样的效果和目的。[图 2.9](#) 展示了一个纹理化的简单例子，一般这些纹理都是二维图像，但是有时候也可以是一维图像或者三维图像。简单来说，像素着色阶段最终会输出每个片元的颜色值，这些颜色值会被输入到下一个子阶段中。



图 2.9：左上角是一个没有纹理的龙模型，右边的纹理会被“粘合”在模型的表面，其结果如左下角图像所示。

## 2.5.2 合并

颜色缓冲（color buffer）是一个矩形阵列，它存储了每个像素中的颜色信息（即颜色的红绿蓝分量）。在之前的像素着色阶段中，我们计算了每个片元的颜色，并将其存储在颜色缓冲中，而合并阶段的任务就是将这些片元的颜色组合起来。这个阶段也被叫做 ROP，意思是“光栅操作管线（raster operations pipeline）”或者“渲染输出单元（render output unit）”，这取决于你问的是谁。与像素着色阶段不同，执行这一阶段的 GPU 子单元，并不是完全可编程的；但它仍然是高度可配置的，可以实现各种效果。

合并阶段还负责解决可见性问题，即当整个场景被渲染的时候，颜色缓冲应当只包含那些相机可见的图元颜色。对于大部分或者几乎所有的图形硬件而言，这个操作是通过 z-buffer（深度缓冲）实现的[238]。z-buffer 具有与颜色缓冲相同的尺寸，对于其中的每个像素，它存储了目前距离最近的图元  $z$  值。这意味着，当一个图元要被渲染到某个像素上时，会计算这个图元的  $z$  值，并将其与 z-buffer 中的对应像素深度进行比较。如果这个新的  $z$  值比当前 z-buffer 中的像素深度更小，说明这个新图元距离相机更近，会挡住原来的图元，因此需要使用新图元的  $z$  值和颜色值来对 z-buffer 和颜色缓冲进行更新；如果新图元的  $z$  值大于对应像素在 z-buffer 中的  $z$  值，说明这个新图元距离相机更远，则 z-buffer 和颜色缓冲将会保持不变。z-buffer 算法十分简单，其时间复杂度为  $O(n)$ （ $n$  为需要被渲染的图元数量）；且这个算法适用于任何能够计算出  $z$  值的图元。同时请注意，z-buffer 允许图元以任意顺序进行渲染，这也是它流行的另一个原因。但是 z-buffer 在每个屏幕像素上，只存

储了一个深度值，因此它不适用于透明物体的渲染。透明物体必须要等到所有的不透明物体都渲染完成之后，才能进行渲染，而且需要严格按照从后往前的顺序进行渲染，或者使用一个顺序无关的透明算法（[章节 5.5](#)）。透明物体的渲染是 z-buffer 算法的主要弱点之一。

我们刚才提到使用颜色缓冲来存储每个像素的颜色，使用 z-buffer 来存储每个像素的 z 值。但是还有一些其他的通道和缓冲可以用来过滤和捕获片元的信息，例如与颜色缓冲相关联的透明通道（alpha channel），它存储了每个像素的不透明度（opacity，[章节 5.5](#)）。在一些较老的 API 中，透明通道也可以被用来进行透明测试（alpha test），来选择性的丢弃一些像素。如今这样的片元丢弃操作可以在像素着色器中完成，而且任何我们想要的计算都可以用来触发这个丢弃操作。这种类型的测试可以用来确保那些完全透明的片元，不会对 z-buffer 产生影响（[章节 6.6](#)）。

模板缓冲（stencil buffer）是一个离屏缓冲区（offscreen buffer），它可以用来记录被渲染图元的位置信息，通常它的每个像素包含 8 bit。图元可以通过各种各样函数来被渲染到模板缓冲中，同时模板缓冲可以用来控制渲染到颜色缓冲和 z-buffer 中的内容。举个例子：假设现在有一个实心圆被写入到了模板缓冲中，现在我们通过一个操作，可以只允许后续图元被渲染到这个实心圆所在位置的颜色缓冲中。模板缓冲十分强大，可以用于生成一些特殊效果。所有这些在管线末尾的功能都被叫做光栅操作（raster operation, ROP）或者混合操作（blend operation）。我们也可以将当前颜色缓冲中的颜色，与三角形中正在处理的颜色相混合，从而实现一些透明效果或者颜色样本累积的效果。上文中我们提到，混合操作通常并不是完全可编程的，一般只能通过使用 API 来进行配置。但是某些 API 支持光栅顺序视图（raster order view），也可以被称作像素着色器排序，它支持可编程的混合操作。

系统中的所有缓冲区在一起，被统称为帧缓冲（frame buffer）。

当图元到达并通过光栅化阶段时，这些从相机角度可见的图元将会被显示在屏幕上，屏幕上所显示的内容就是颜色缓冲中的内容。由于渲染需要花费一定时间，为了避免观察者看到图元渲染并显示在屏幕上的过程，一般都会使用双缓冲机制（double buffering），这意味着场景的渲染都会在屏幕外的后置缓冲区中进行。当场景被渲染到后置缓冲区之后，后置缓冲区会与显示在屏幕上的前置缓冲区（front buffer）交换内容。这个交换的过程通常发生在垂直回扫（vertical retrace）的过程中，因此这样做是可行的。

有关不同缓冲和缓冲方法的更多内容，详见[章节 5.4.2](#)，[章节 23.6](#) 和 [章节 23.7](#)。

## 2.6 回顾整个管线

点，线和三角形是用来构建模型和物体的渲染图元。假设现在有一个交互式的计算机辅助设计（computer aided design, CAD）程序，用户正在查看一个华夫饼机的设计模型。现在我们将跟随这个模型，完整通过整个图形渲染管线，它包含四个主要阶段：应用阶段、几何处理阶段、光栅化阶段和像素处理阶段。场景将以透视视角渲染到一个屏幕窗口中，在这个简单的例子中，华夫饼机的模型由线段（展示各个部分的边界）和三角形（展示模型的表面）组成，华夫饼机还有一个可以打开的盖子，上面有一些三角形具有制造商标志的二维纹理。在这个例子中，除了纹理贴图的使用发生在像素处理阶段之外，其余所有的表面着色计算都在几何处理阶段完成。

### 应用阶段

CAD 程序允许用户选择并移动部分模型，例如用户可能会选中华夫饼机的盖子，并通过拖动鼠标来打开它。应用程序阶段需要将用户的鼠标移动，转换为相应的旋转矩阵，然后确保这个旋转矩阵在渲染的时候会被正确应用。另一个例子是播放相机动画，让相机沿着预定的路线进行移动，从不同的视角来展示这个华夫饼机。程序需要按照动画的设定，来对相机的相关参数进行更新，例如位置、视角方向等。在渲染每一帧之前，应用程序需要将相机位置，光照和模型的图元信息，都发送给管线的下一阶段——几何处理阶段。

### 几何处理阶段

对于透视视角而言，我们这里假设应用阶段已经提供了一个投影矩阵（ $\mathbf{P}$ ）；同样的，对于每个物体而言，应用程序都会计算出一个矩阵（ $\mathbf{MV}$ ），这个矩阵描述了观察变换以及物体本身的位置和朝向。在我们的例子中，华夫饼机的底座会对应一个矩阵，而盖子则会对应另外一个矩阵。在几何阶段，物体的顶点和法线会通过这个矩阵被变换到观察空间中，然后使用材质和光照信息，来进行顶点着色以及其他一些计算。然后会使用一个由用户提供的投影矩阵，来将相机的可视空间变换为一个标准立方体，所有位于标准立方体外部的图元都会被直接丢弃，所有与标准立方体相交的图元都会被裁剪，从而获得完全位于标准立方体内部的图元。最后顶点会被映射到屏幕上的窗口中。当所有这些逐三角形和逐顶点的操作完成之后，生成的结果数据会被输入到光栅化阶段。

### 光栅化阶段

所有在前一阶段中被保留下来的图元，在这个阶段中都会进行光栅化，即找到所有位于图元内部的像素，然后将其发送到管线的像素处理阶段。

## 像素处理阶段

这一步的目标是计算出每个可见图元所覆盖像素的颜色值。与纹理（图像）相关联的三角形会使用这些纹理进行渲染。图元的可见性通过使用 z-buffer 算法来进行解决，也可以使用可选的图元丢弃操作和模板测试。每个物体都会被轮流处理，最终生成一副图像，然后显示在屏幕上。

## 总结（Conclusion）

本章节所描述的渲染管线，是几十年来，面向实时渲染程序的 API 以及图形硬件发展而来的结果。很重要的一点是，这并不是唯一可能的渲染管线，离线渲染中也有一套渲染管线，但是经历了与实时渲染完全不同的演化路径。用于电影制作的渲染以往都会采用微多边形（micropolygon）渲染管线[289, 1734]，但是最近几年来，光线追踪和路径追踪已经渐渐占据了上风。这些技术在建筑可视化领域和设计可视化领域中也很多应用，它们会在[章节 11.2.2](#) 中提及。

多年以来，开发人员只能使用图形 API 所定义的固定渲染管线（fixed-function pipeline）来完成这个过程。固定渲染管线之所以名字中带有“固定”，是因为其中的功能是基于硬件进行实现的，而这些硬件无法进行灵活的编程调整。固定渲染管线的最后一个重要机器，是于 2006 年推出的 Nintendo Wii。另一方面，可编程 GPU 可以准确的确定，渲染管线中的每个阶段应用的都是哪些操作。在本书中，我们假设所有的开发都是使用可编程 GPU 来完成的。

## 补充阅读和资源

Blinn 的《A Trip Down the Graphics Pipeline》[165]是一本从头开始编写软件渲染器的老书，这是一个很好资源，你可以从中了解如何实现一个渲染管线的细节，它介绍了一些诸如裁剪和透视插值的关键算法。《OpenGL: Programming Guide》（简称“Red Book”，即红宝书）[885]是一本高龄的书（但是会经常更新），它提供了有关图形渲染管线的详细描述，以及所使用算法的详细描述。在我们的配套网站 [realtimerendering.com](http://realtimerendering.com) 上，提供了各种各样的管线程序、渲染引擎的实现以及其他信息。

