2.3 几何处理

GPU上的几何处理阶段负责大多数每个三角形和每个顶点的操作。该阶段进一步分为以下功能阶段：顶点着色、投影、裁剪和屏幕映射（如图2.3）。

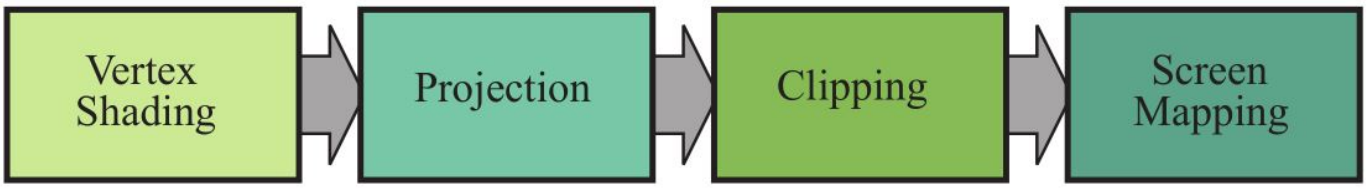

图2.3. 几何处理阶段分为一系列功能阶段。

2.3.1 顶点着色

顶点着色有两个主要任务，即计算顶点的位置和评估程序员可能喜欢的顶点输出数据，例如法线坐标和纹理坐标。传统上，对象的大部分阴影是通过将灯光应用到每个顶点的位置和法线并仅将结果颜色存储在顶点来计算的。然后将这些颜色插入整个三角形。出于这个原因，这个可编程的顶点处理单元被命名为顶点着色器[1049]。随着现代GPU的出现，以及每个像素发生的部分或全部着色，这个顶点着色阶段更加通用，并且可能根本不会求取任何着色方程的值，其工作主要取决于程序员的意图。顶点着色器现在是一个更通用的单元，专门用于设置与每个顶点关联的数据。例如，顶点着色器可以使用第4.4节和第4.5节中的方法为对象设置动画。

我们首先描述如何计算顶点位置，一组始终需要的坐标。在被屏幕显示的过程中，模型被转换成几个不同的空间或坐标系。最初，模型驻留在自己的模型空间中，这仅意味着它根本没有被转换。每个模型都可以与模型变换相关联，以便可以对其进行定位和定向。可以将多个模型转换与单个模型相关联。这允许同一模型的多个副本（称为实例）在同一场景中具有不同的位置、方向和大小，而无需复制基本几何体。

模型变换所变换的是模型的顶点和法线。对象的坐标称为模型坐标，在对这些坐标应用模型变换后，模型被称为位于世界坐标或世界空间中。世界空间是唯一的，模型经过各自的模型变换后，所有的模型都存在于同一个空间中。

如前所述，模型只有被相机（或观察者）看到才能渲染。相机在世界空间中有一个位置和一个方向，用于放置和瞄准相机。为了便于投影和剪辑，相机和所有模型都使用视图变换进行了变换。视图变换的目的是将相机放置在原点并瞄准它，使其看向负z轴的方向，y轴指向上方，x轴指向右侧。我们使用-z轴约定；一些文章也会使用向下看+z轴的约定。区别主要是语义上的，因为一个和另一个之间的转换很简单。应用视图变换后的实际位置和方向取决于底层应用程序编程接口 (API)。如此划定的空间称为相机空间，或更常见的是，视图空间或眼睛空间。视图变换影响相机和模型的方式示例如图 2.4所示。模型变换和视图变换都可以用4×4矩阵来实现，这是第4章的主题。但是，重要的是要意识到可以以程序员喜欢的任何方式计算顶点的位置和法线。

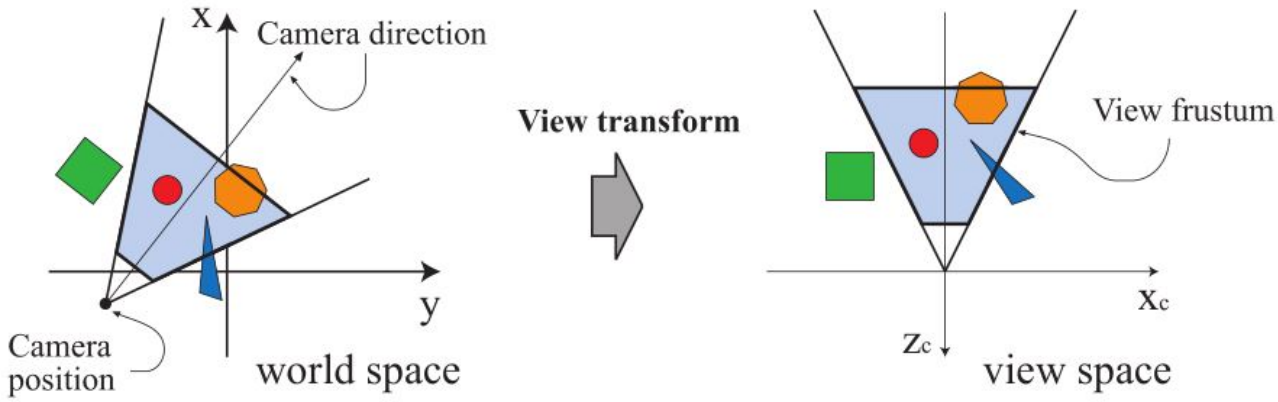

图2.4. 在左图中，自上而下的视图显示了在+z轴向上的坐标系中，按照用户希望的方式定位和定向的相机。视图变换重新定向了坐标系，使相机位于原点，沿其负z轴看，相机的+y轴向上，如右图所示。这样做是为了使裁剪和投影操作更简单、更快捷。浅蓝色区域是视锥体。在这里，假设透视图，因为视锥体是一个平截头体。类似的技术适用于任何类型的投影。

接下来，我们将描述顶点着色的第二种类型的输出。要生成逼真的场景，仅渲染对象的形状和位置是不够的，还必须对它们的外观进行建模。该描述包括每个物体的材质，以及任何光源照射在物体上的效果。材料和灯光可以通过多种方式建模，从简单的颜色到物理描述的精细表示。

这种确定光对材料效果的操作称为着色。它涉及计算对象上不同点的着色方程。通常，其中一些计算在模型顶点的几何处理期间执行，而其他计算可能在逐像素处理期间执行。各种材质数据可以存储在每个顶点，例如点的位置、法线、颜色或求取着色方程值所需的任何其他数字信息。然后，顶点着色结果（可以是颜色、矢量、纹理坐标以及任何其他类型的着色数据）被发送到光栅化和像素处理阶段进行插值并用于计算表面的着色。

GPU顶点着色器形式的顶点着色在本书中进行了更深入的讨论，尤其是在第3章和第5章中。

作为顶点着色的一部分，渲染系统先进行投影，然后进行裁剪，将视图体换为单位立方体，其极值点位于$(-1,-1,-1)$和$(1,1,1)$之间。可以使用不同的范围来定义相同的体积，例如，$(0 \leq z \leq 1)$。单位立方体称为正规化视图体。投影是在GPU上由顶点着色器首先完成的。常用的投影方法有两种，即正射投影（也称平行投影）和透视投影，如图2.5。事实上，正射投影只是一种平行投影。也可以使用其

他几种投影方式（特别是在建筑领域），例如斜投影和轴测投影。老式街机游戏Zaxxon就是以后者命名的。

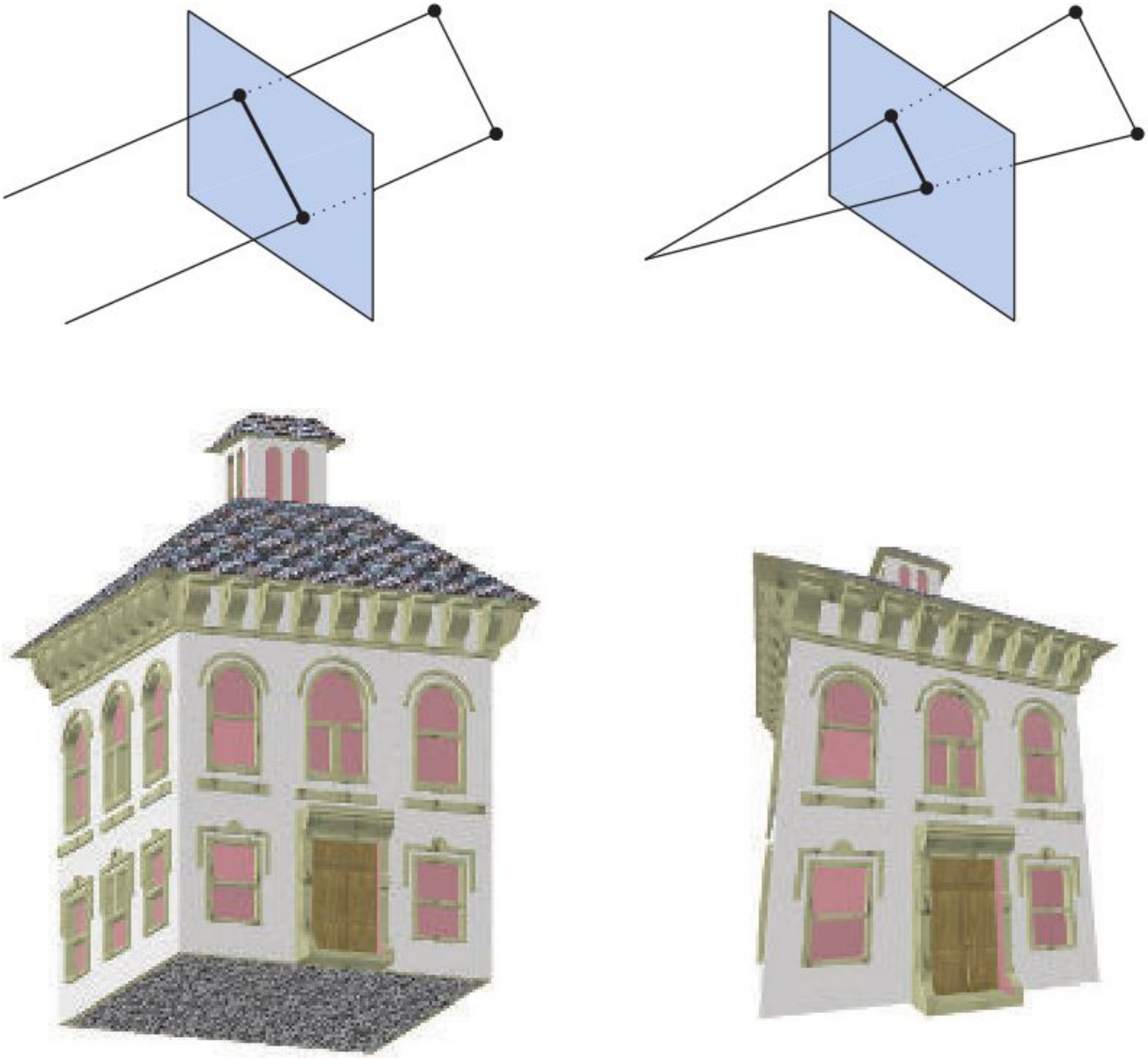

图2.5. 左侧是正射投影或平行投影；右边是透视投影。

请注意，投影表示为矩阵（第4.7节），因此它有时可能与几何变换的其余部分连接。

正交观察的视图体通常是一个矩形框，正交投影将这个视图体变换为单位立方体。正交投影的主要特点是平行线在变换后保持平行。这种转换是平移和缩放的组合。

透视投影有点复杂。在这种类型的投影中，物体离相机越远，投影后看起来越小。另外，平行线可能会聚在地平线上。因此，透视变换模仿了我们感知物体大小的方式。在几何上，称为截头锥体的视图体是一个具有矩形底面的截棱锥。截头锥体也转化为单位立方体。正交变换和透视变换都可以用4×4矩阵构造（第4章），并且在任一变换之后，模型都被称为在裁剪坐标中。这些实际上是齐次坐标，在第4章中

讨论过，因此这发生在除以w之前。GPU的顶点着色器必须始终输出这种类型的坐标，以便下一个功能阶段（裁剪）正常工作。

尽管这些矩阵将一个几何体转换为另一个几何体，但它们被称为投影，因为在显示之后，z坐标不存储在生成的图像中，而是存储在z缓冲区中，如第2.5节所述。通过这种方式，模型从三维投影到两维。

2.3.2 可选的顶点处理

每个管道都有刚刚描述的顶点处理。完成此处理后，可以在GPU上进行几个可选阶段，按顺序是：曲面细分、几何着色和流输出。它们的使用取决于硬件的能力——并非所有 GPU 都有它们——以及程序员的愿望。它们相互独立，一般不常用。 将在第3章中详细介绍每一个。

第一个可选阶段是曲面细分。假设你有一个弹跳球对象。如果用一组三角形表示它，则可能会遇到质量或性能问题。您的球在5米外可能看起来不错，但近距离观察单个三角形，三角形的轮廓，就会变得清晰可见。如果你用更多的三角形制作球来提高质量，当球很远并且只覆盖屏幕上的几个像素时，你可能会浪费大量的处理时间和内存。通过曲面细分，可以使用适当数量的三角形生成曲面。

我们已经讨论了一些三角形，但在管线中的这一点上，我们只处理了顶点。这些可用于表示点、线、三角形或其他对象。顶点可用于描述曲面，例如球。这样的表面可以由一组面片指定，每个面片由一组顶点组成。曲面细分阶段由一系列阶段本身组成——外包着色器(hull shader)、曲面细分器(tessellator)和域着色器(domain shader)——将这些面片顶点集转换为（通常）更大的顶点集，然后用于制作新的三角形集。场景的相机可用于确定生成了多少个三角形：面片很近时很多，远处时很少。

下一个可选阶段是几何着色器。该着色器早于曲面细分着色器，因此在GPU上更常见。它类似于曲面细分着色器，因为它接受各种类型的图元并可以生成新的顶点。这是一个更简单的阶段，因为此创建的范围有限，输出图元的类型也更有限。几何着色器有多种用途，其中最流行的一种是粒子生成。想象一下模拟烟花爆炸。每个火球都可以用一个点来表示，一个顶点。几何着色器可以将每个点变成面向观察者并覆盖多个像素的正方形（由两个三角形组成），从而为我们提供更令人信服的图元进行着色。

最后一个可选阶段称为流输出。这个阶段让我们使用GPU作为几何引擎。与将我们处理过的顶点沿着管道的其余部分发送到屏幕上不同，此时我们可以选择将这些顶点输出到数组中以供进一步处理。这些数据可以由CPU或GPU本身在以后的过程中使用。此阶段通常用于粒子模拟，例如我们的烟花示例。

这三个阶段按此顺序执行——曲面细分、几何着色和流输出——每个阶段都是可选的。无论使用哪个（如果有）选项，如果我们继续沿着管道向下走，我们就会得到一组具有齐次坐标的顶点，这些顶点将被检查相机是否能看到它们。

2.3.3 裁剪

只有全部或部分在视图体内部的图元需要传递到光栅化阶段（以及随后的像素处理阶段），然后在屏幕上绘制它们。完全位于视图体内部的图元将按原样传递到下一个阶段。完全在视图体积之外的基元不会

被进一步传递，因为它们没有被渲染。部分位于视图体内部的图元需要裁剪。例如，一条直线，在视图体外部有一个顶点，在视图体积内部有一个顶点，此时应该根据视图体对其进行裁剪；以便外部的顶点被位于该线和视图体之间的交点处的新顶点替换。投影矩阵的使用意味着变换后的图元被裁剪到单位立方体上。在裁剪之前进行视图变换和投影的好处是可以使裁剪问题保持一致；图元总是针对单位立方体进行裁剪。

裁剪过程如图2.6所示。除了视图体积的六个剪裁平面之外，用户还可以定义额外的剪裁平面来明显地剪裁对象。第818页的图19.1中显示了显示这种可视化类型的图像，称为剖视(sectioning)。

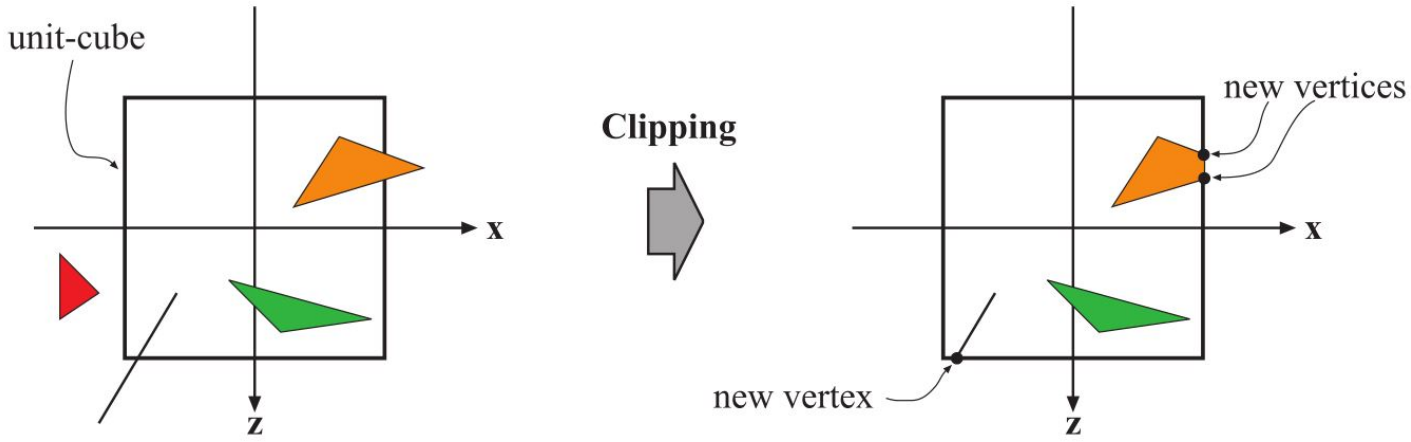

图2.6. 只需要单位立方体内部的图元（对应视锥体内部的图元）继续处理。因此，单位立方体外面的图元被丢弃，而完全在里面的图元被保留。与单位立方体相交的图元被裁剪在单位立方体上，从而产生新的顶点并丢弃旧的顶点。

裁剪步骤使用投影产生的4值齐次坐标进行裁剪。值通常不会跨透视空间中的三角形进行线性插值。需要第四个坐标，以便在使用透视投影时正确插入和裁剪数据。最后，执行透视除法，将生成的三角形的位置放入三维标准化设备坐标中。如前所述，此视图体积范围从$(-1,-1,-1)$到$(1,1,1)$。几何阶段的最后一步是从这个空间转换到窗口坐标。

2.3.4 屏幕映射

只有视图体内部的（裁剪的）图元被传递到屏幕映射阶段，进入这个阶段时坐标仍然是三维的。每个图元的x和y坐标被转换为屏幕坐标。屏幕坐标与z坐标一起也称为窗口坐标。假设场景应该被渲染到一个最小位置在(x_1, y_1)，最大位置在(x_2, y_2)处的窗口（其中$x_1 < x_2$和$y_1 < y_2$）。屏幕映射先是平移，然后是缩放操作。新的x和y坐标称为屏幕坐标。z坐标（OpenGL的$[-1,+1]$和DirectX的$[0,1]$）也被映射到$[z_1, z_2]$，其中$z_1 = 0$和$z_2 = 1$作为默认值。但是，这些可以通过API进行更改。窗口坐标连同这个重新映射的z值被传递到光栅化阶段。屏幕映射过程如图2.7所示。

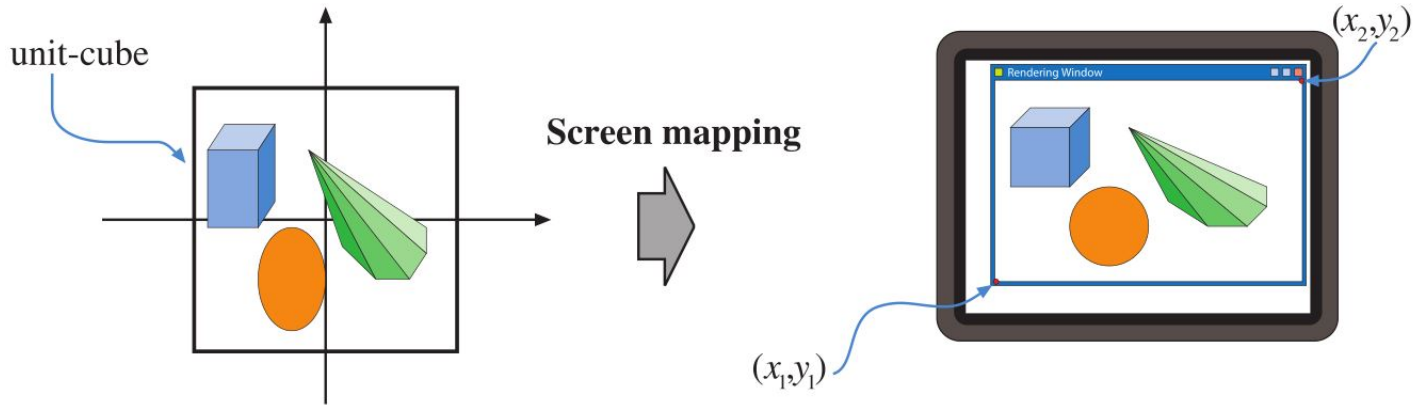

图2.7. 投影变换后的图元位于单位立方体中，屏幕映射程序负责在屏幕上找到坐标。

接下来，我们描述整数和浮点值如何与像素（和纹理坐标）相关。给定像素的水平数组并使用笛卡尔坐标，最左边像素的左边缘在浮点坐标中为0.0。OpenGL一直使用这种方案，DirectX10及其后续版本也使用它。该像素的中心为0.5。因此，一系列像素 [0,9] 覆盖了 [0.0,10.0) 的跨度。转换很简单：

$$d = floor(c) \tag{2.1}$$

$$c = d + 0.5 \tag{2.2}$$

其中d是像素的离散（整数）索引，c是像素内的连续（浮点）值。

虽然所有API的像素位置值都从左到右增加，但在OpenGL和DirectX[2]之间的某些情况下，顶部和底部边缘的零位置不一致。OpenGL始终偏爱笛卡尔系统，将左下角视为最低值元素，而DirectX有时根据上下文将左上角定义为该元素。每个人都有一个逻辑，在他们不同的地方不存在正确的答案。例如，$(0, 0)$在 OpenGL中位于图像的左下角，而在DirectX中位于左上角。在从一个API迁移到另一个API时，必须考虑到这种差异。

备注：

2. “Direct3D”是DirectX的三维图形API组件。DirectX包括其他API元素，例如输入和音频控件。我们不区分在指定特定版本时编写“DirectX”和在讨论此特定API时编写“Direct3D”，而是通过始终编写“DirectX”来遵循常见用法。