2.渲染管线

本章介绍了实时图形的核心组件，即图形渲染管道，也简称为“管道”。管道的主要功能是在给定虚拟摄像机的情况下生成或渲染二维图像，三维物体，光源等。因此，渲染管道是实时渲染的基础工具。使用管道的过程如图2.1所示。图像中对象的位置和形状由其几何形状，环境特征以及相机在该环境中的位置决定。对象的外观受材质属性，光源，纹理（应用于表面的图像）和着色方程的影响。

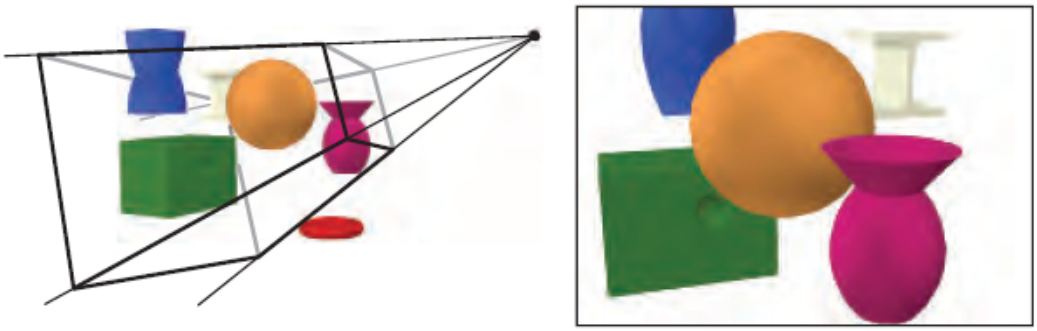

图2.1。在左图中，虚拟摄像机位于金字塔的顶端（四条线会聚的点）。仅渲染视图体积内的基元。对于以透视方式呈现的图像（如这里的情况），视图体积是平截头体，即一个头部被截取的棱椎。右图显示了相机“看到的内容”。请注意，左图中的红色圆环形状不在右侧渲染中，因为它位于视锥体外。此外，左图像中的扭曲蓝色棱镜被夹在平截头体的顶平面上。

我们将解释渲染管道的不同阶段，重点是功能而不是实现。应用这些阶段的相关细节将在后面的章节中介绍。

2.1结构

在物理世界中，管道概念以许多不同的形式表现出来，从工厂装配线到快餐厨房。它也适用于图形渲染。管道由若干阶段组成，每个阶段执行庞大任务的一部分。

流水线的每个阶段是并行执行的，每个阶段取决于前一阶段的结果。理想情况下，一个非流水线系统被划分成n个流水线阶段，可以带来n倍的速度提升。性能的提高是使用流水线操作的主要原因。例如，一堆数量庞大的三明治可以被一些分工好的工人快速准备好 —— 一个准备面包，另一个添加肉，另一个添加配料。每个人都将自己的结果传递给下一个人，并立即开始下一个三明治的工作。如果每个人花费20秒来执行他们的任务，则每20秒，每分钟三次，最大速率可以生产一个三明治。尽管流水线是并行执行的，但它们会被拖延直到最慢的阶段完成其任务。例如，假设肉类添加阶段变得更加复杂，需要30秒。现在最好的速度是一分钟两个三明治。在这个特殊的流水线，肉类阶段是瓶颈，因为它决定了整个生产的速度。于是配料阶段在等待肉类阶段的完成时处于饥饿状态（对于顾客也是）。

这种管道结构也可以在实时计算机图形的背景下找到。将实时渲染管线粗略划分为四个主要阶段 - **应用，几何处理，光栅化和像素处理** - 如图2.2所示。这种结构是渲染管线的核心在实时计算机图形应用程序中，因此是后续章节中讨论的重要基础。这些阶段通常又是一个管线，这意味着它由几个子阶段组成。我们区分这里显示的功能阶段和它们的实现结构。功能阶段有明确的任务要执行，但没有指定任务在管线中执行的方式。给定的实现可以将两个功能阶段组合成一个单元或者执行使用可编程核心，同时将另一个更耗时的功能阶段划分为多个硬件单元。

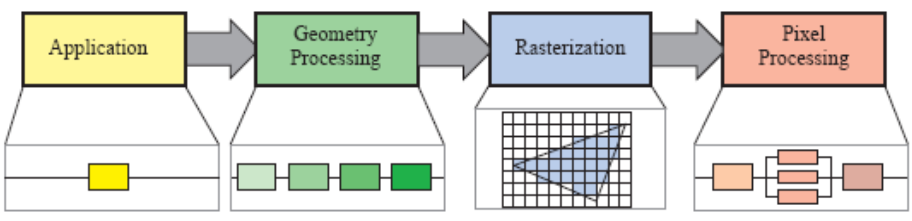

图2.2。渲染管道的基本构造，包括四个阶段：应用程序，几何处理，光栅化和像素处理。这些阶段中的每一个本身可以是管线，如几何处理阶段下面所示，或者阶段可以（部分）并行化，如像素处理阶段所示。在此图示中，应用程序阶段是单个进程，但此阶段也可以是流水线或并行化的。注意，光栅化找到图元内的像素，例如三角形。

渲染速度可以以每秒帧数（FPS）表示，即每秒渲染的图像的数量。它也可以用赫兹（Hz）表示，它只是1 /秒的符号，即更新的频率。通常以毫秒（ms）表示渲染图像所需的时间。生成图像的时间通常会有所不同，具体取决于每帧期间执行的计算的复杂程度。每秒帧数用于表示特定帧的速率，或表示某个使用持续时间内的平均性能。赫兹用于硬件，例如显示器，其能被设置为固定速率。

顾名思义，应用程序阶段由应用程序驱动，因此通常由在通用CPU上运行的软件实现。这些CPU通常包括多个内核，这些内核能够并行处理多个执行线程。这使CPU可以高效地运行应用程序阶段负责的各种任务。传统上在CPU上执行的某些任务包括碰撞检测，全局加速算法，动画，物理仿真以及许多其他任务，具体取决于应用程序的类型。下一个主要阶段是几何处理，它处理变换，投影和所有其他类型的几何处理。此阶段计算要绘制的内容，应如何绘制以及应在何处绘制。几何阶段通常在包含许多可编程内核以及固定操作硬件的图形处理单元（GPU）上执行。光栅化阶段通常将三个顶点作为输入，形成一个三角形，然后找到该三角形内所有要考虑的像素，然后将其转发到下一个阶段。最后，像素处理阶段为每个像素执行一个程序以确定其颜色，并可以执行深度测试以查看其是否可见。它还可以执行每个像素的操作，例如将新计算的颜色与以前的颜色混合。光栅化和像素处理阶段也完全在GPU上进行处理。所有这些阶段及其内部管道将在接下来的四个部分中讨论。有关GPU如何处理这些阶段的更多详细信息，请参阅第3章。

2.2应用程序阶段

由于通常在CPU上执行，因此开发人员可以完全控制应用程序阶段发生的事情。因此，开发人员可以完全确定实现，以后可以对其进行修改以提高性能。此处的更改也会影响后续阶段的性能。例如，应用程序阶段算法或设置可以减少要渲染的三角形的数量。

综上所述，某些应用程序工作可以由GPU使用称为计算着色器的单独模式来执行。此模式将GPU视为高度并行的通用处理器，而忽略了专门用于渲染图形的特殊功能。在应用程序阶段结束时，要渲染的几何图形被传送到几何阶段。这些是渲染图元（rendering primitives），即点，线和三角形，它们最终可能最终出现在屏幕上（或使用的任何输出设备）。这是应用程序阶段最重要的任务。

此阶段基于软件的实现的结果是，它不像几何处理，光栅化和像素处理阶段那样分为子阶段。但是为了提高性能，该阶段通常在多个处理器上并行执行核心。在CPU设计中，这被称为超标量构造，因为它能够在同一阶段同时执行多个进程[1]。第18.5节介绍了使用多个处理器内核的各种方法。

1.于CPU本身的流水线规模要小得多，因此可以说应用程序阶段可进一步细分为几个流水线阶段，但这与此处无关。

在此阶段通常实现的是碰撞检测。在两个物体之间检测到碰撞之后，可以生成响应并将其发送回碰撞的物体以及力反馈设备。在应用程序阶段，还要处理来自其他来源的输入，例如键盘，鼠标或头戴式显示器。根据此输入，可以发生几种不同类型的动作。加速算法，例如特定的剔除算法（第19章），也在这里实现，以及渲染管线其余部分无法处理的其他任何事情。

2.3几何阶段

GPU上的几何处理阶段负责大部分对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（图2.3）。

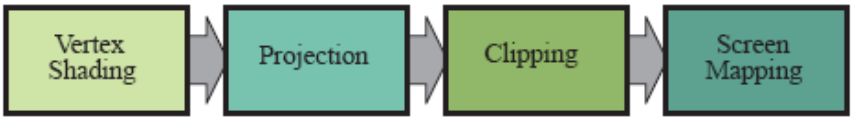

图2.3几何阶段分为由几个功能阶段组成的管线。

2.3.1顶点着色

顶点着色有两个主要任务，即计算一个顶点的位置和评估任何程序员可能希望作为顶点输出数据的东西，例如法线和纹理坐标。在传统意义上，一个对象的大部分光影表现是通过将光线应用于每个顶点的位置和法线和存储在顶点种的颜色来计算的。然后将这些颜色插值到整个三角形上。因此，该可编程顶点处理单元被称为顶点着色器[1049]。随着现代GPU的出现，以及每个像素发生部分或全部着色，此顶点着色阶段变得更加通用，并且可能根本不评估任何着色方程式，具体取决于程序员的意图。现在，顶点着色器是一个更通用的单元，专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节中的方法对对象进行动画处理。

我们首先描述如何计算顶点位置，这始终需要的一组坐标。在它进入屏幕的过程中，模型被转换为几个不同的空间或坐标系。最初，模型在其自己的模型空间中，这仅表示该模型根本没有进行转换。每个模型都可以与一个模型变换关联，以便可以定位和定向。可能有多个模型变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(instances)）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图形。

模型变换变换的是模型的顶点和法线。对象(object)（这里指模型）自己坐标称为模型坐标，并且在将模型变换应用于这些坐标之后，可以说模型位于世界坐标或世界空间中。世界空间是唯一的，在对模型进行了各自的模型变换后，所有模型都存在于同一空间中。

如前所述，仅被摄像机（或观察者）所看到的模型被渲染。摄像机在世界空间中具有特定方向来校准摄像机视角和在特定的位置来放置摄像机。为了便于投影和裁剪操作，使用视图变换对摄像机和所有模型进行变换。视图变换的目的是将摄影机放置在原点并将其对准目标，使其沿负z轴方向看，y轴指向上方，x轴指向右侧。我们使用-z轴约定；有些文字更喜欢看向+ z轴。区别主要是语义上的，因为彼此之间的转换很简单。应用视图变换后的实际位置和方向取决于基础应用程序编程接口（API）。如此划定的空间称为相机空间(camera space)，或更普遍地称为视野空间(view space)或眼睛空间。视图转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使用4×4矩阵实现，这是第4章的主题。但是，重要的是要认识到顶点的位置和法线都可以被程序员喜欢的任何方式来计算。

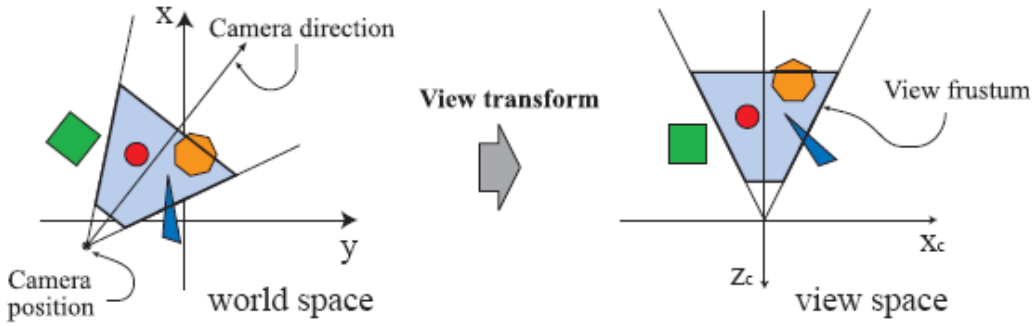

图2.4。在左图中，自上而下的视图显示了在+ z轴朝上的世界中，按用户期望的方式定位和定向的相机。视图变换可重新定向世界，以使相机位于其原点，沿其负z轴看，而相机的+ y轴朝上，如右图所示。这样做是为了使剪切和投影操作更简单，更快捷。浅蓝色区域是视图体积。在此，假定透视图，因为视图体积为平截头体。类似的技术适用于任何类型的投影。

接下来，我们描述顶点着色的第二种输出类型。要产生逼真的场景，仅渲染对象的形状和位置是不够的，但是还必须对它们的外观进行建模。此说明包括每个对象的材质，以及任何照在该对象上的光源的效果。材质和灯光可以采用多种方式建模，从简单的颜色表示到详尽的物理描述。

确定光线对材质的影响的这种操作称为着色(shading)。它涉及计算模型对象上各个点的着色方程式。通常，其中一些对模型顶点的计算是在的几何图形处理期间执行的，而其他一些计算可能是在逐像素处理期间执行的。每个顶点可以存储各种材质数据，例如点的位置，法线，颜色或任何评估着色方程式所需的其他数字信息。然后将顶点着色的结果（可以是颜色，矢量，纹理坐标以及任何其他种类的着色数据）发送到光栅化和像素处理阶段以进行插值，并用于计算表面的着色。

在本书中，尤其将更深入地讨论GPU顶点着色器的顶点着色形式，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操作，这会将视图体积转换为单位立方体，其顶点位于（-1，-1，-1）和（1、1、1）。并且可以使用定义相同体积但不同范围单位立方体，例如0≤z≤1。单位立方称为规范视图体积。进行投影操作，通常在GPU上由顶点着色器完成。有两种常用的投影方法，即正交投影（也称为平行投影）和透视投影。参见图2.5。实际上，正交投影只是平行投影的一种类型。特别是在建筑领域，还有其他一些用途，例如斜投影和轴测投影。旧的街机游戏Zaxxon从后者命名。

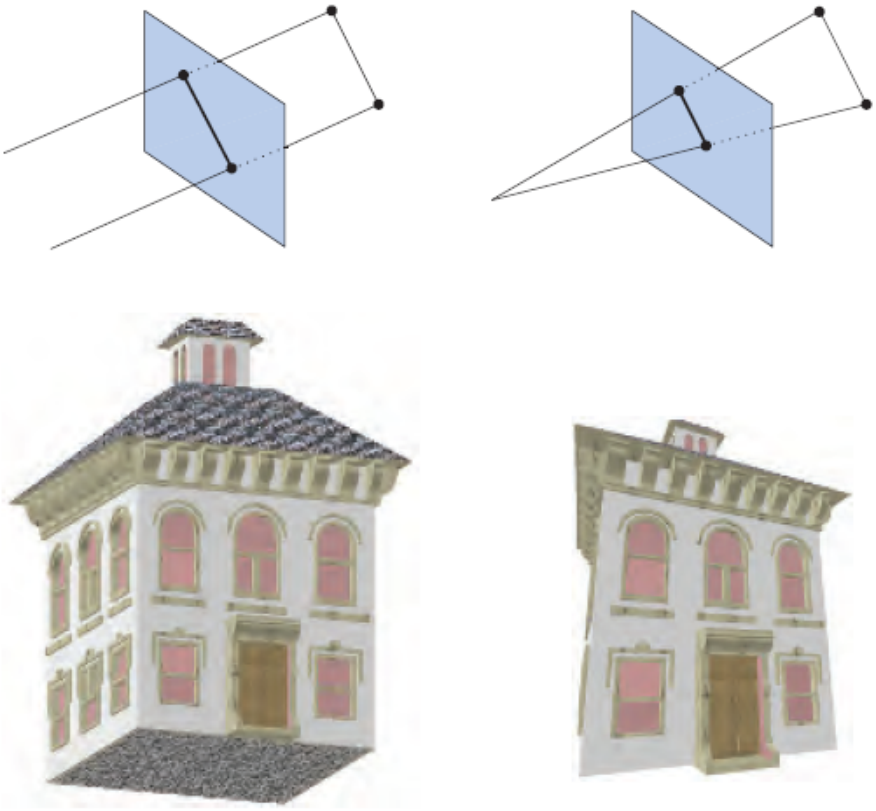

图2.5 左侧是正交投影或平行投影；右边是透视图。

请注意，投影操作表示为矩阵（第4.7节），因此有时可以将其与其余的几何变换连接在一起。

正交视图的视图体积通常是一个矩形框，而正交投影会将此视图体积转换为单位立方体。正交投影的主要特征是，平行线在变换后保持平行，此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，投影后出现的越小。此外，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尺寸的方式。从几何学上讲，视场称为视锥，是具有矩形底面的截顶金字塔。视锥也将转换为单位立方体。正交变换和透视变换都可以使用4×4矩阵构造（第4章），并且在进行任何变换之后，都将模型称为裁剪坐标。这些实际上是齐次坐标，将在第4章中进行讨论，因此，这发生在被w除之前。 GPU的顶点着色器必须始终输出此类型的坐标，以使下一个功能阶段（裁剪）正常工作。

尽管这些矩阵将一个体积转换为另一个体积，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2.5节所述。这样，模型可以从三个维度投影到两个维度。

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，GPU上将按照以下顺序进行几个可选阶段：曲面细分(tessellation)，几何着色(geometry shading)和流输出(stream output)。它们的使用取决于硬件的功能（并非所有GPU都具有）以及程序员的需求。它们彼此独立，并且通常不常用。在第3章中将对每个进行更多说明。

第一个可选阶段是细分。假设您有一个弹跳的球物体。如果用一组三角形表示它，则可能会遇到质量或性能问题。您的球从5米远处看起来可能不错，但近距离可以看到各个三角形，尤其是沿着轮廓的三角形。如果用更多的三角形制作球以提高质量，则当球太远且仅覆盖屏幕上的几个像素时，可能会浪费大量的处理时间和内存。通过细分，可以生成具有适当数量的三角形的曲面。

我们已经讨论了一些三角形，但是到目前为止，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。细分阶段本身包含一系列阶段（外壳着色器，细分和域着色器），这些阶段将这些面片顶点集转换为（通常）更大的顶点集，然后用于创建新的三角形集。场景的摄像头可用于确定生成了多少个三角形：靠近时会生成许多三角形，而远离时会生成很少的三角形。

下一个可选阶段是几何着色器。该着色器早出现于曲面细分着色器，因此在GPU上更常见。就像曲面细分着色器一样，它可以吸收各种图元并可以产生新的顶点。这是一个非常简单的阶段，因为此创建的范围受到限制，输出基元的类型受到更多的限制。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。每个火球都可以由一个点，单个顶点表示。几何着色器可以获取每个点并将其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），从而为我们提供了更具说服力的图元进行着色。

最后一个可选阶段称为流输出。在此阶段，我们可以将GPU用作几何引擎。此时，我们可以选择将其输出到数组以进行进一步处理，而不是将处理后的顶点向下发送到要渲染到屏幕的剩余管线中。这些数据可以在以后的过程中由CPU或GPU本身使用。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（细分，几何体着色和流输出），并且每个阶段都是可选的。不管使用哪个（如果有）选项，如果我们继续沿管线移动，我们都有一组具有齐次坐标的顶点，不论如何都可以在相机视图中查看它们。

2.3.3裁剪

只需要将全部或部分视锥体内的图元传递到光栅化阶段（以及随后的像素处理阶段），然后将其绘制在屏幕上。完全位于视锥体内的图元将原样传递到下一个阶段。完全不在视图体积之外的基元不会进一步传递，因为它们不会被渲染。只是部分在视锥体内的图元需要被裁剪。例如，有一条直线有一个顶点在视锥体内另一个顶点在视锥体外那么这条直线需要被视锥体裁剪，将外部顶点替换为位于该线和视图体积的交点处的新顶点。投影矩阵的使用意味着将变换后的图元剪裁在单位立方体上。在裁剪之前执行视图转换和投影的优点是使裁剪问题一致。图元总是被裁剪在单位立方体上。

裁剪过程如图2.6所示。除了单位立方体的六个剪切平面之外，用户还可以定义其他剪切平面以可视方式剪切对象。在第818页的图19.1中显示了显示这种类型的可视化效果的图像，称为切片

裁剪步骤使用投影产生的4值齐次坐标执行裁剪。在透视空间中，值通常不会跨三角形线性内插。需要第四个坐标，以便在使用透视投影时正确地插入和剪切数据。最后，执行透视划分，将所得三角形的位置放入三维归一化的设备坐标中。如前所述，此视图的体积范围是（-1，-1，-1）到（1,1,1）。几何阶段的最后一步是从该空间转换为窗口坐标

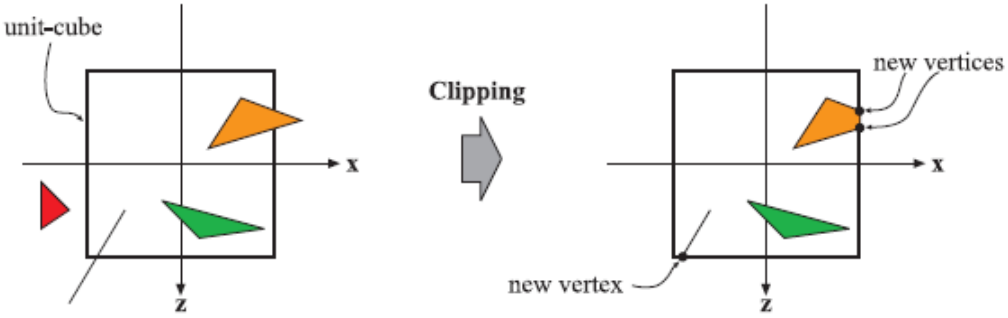

图2.6。投影变换后，仅将单位立方体内的图元（对应于视锥中的图元）进行后续处理。因此，完全位于单位立方体外的图元将被丢弃，并完全保留完全位于单位立方体内的图元。与单位立方体相交的图元被裁剪到单位立方体上，从而生成新的顶点，而旧的顶点被丢弃。

2.3.4屏幕映射

只有视图体积内的（剪切后）图元传递到屏幕映射阶段，进入该阶段时坐标仍然是三维的。每个图元的x坐标和y坐标都将转换为屏幕坐标。屏幕坐标和z坐标也称为窗口坐标。假定场景应渲染到窗口中，最小角在（x1，y1）处，最大角在（x2，y2），其中x1 <x2并且y1 <y2。然后，屏幕映射是平移，随后是缩放操作。新的x和y坐标称为屏幕坐标。 z坐标（对于OpenGL为[−1，+1]，对于DirectX为[0，1]）也映射到[z1，z2]，其中z1 = 0和z2 = 1为默认值。但是，可以使用API进行更改。窗口坐标以及此重新映射的z值将传递到光栅化器阶段。屏幕映射过程如图2.7所示。

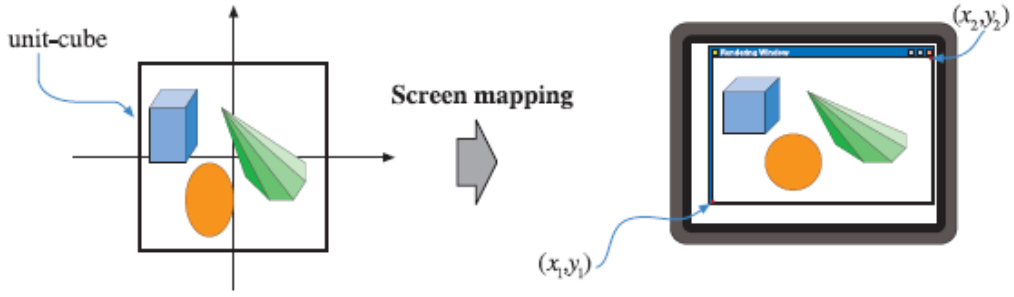

图2.7。投影变换后，图元位于单位立方体中，并且屏幕映射过程会在屏幕上查找坐标。

接下来，我们描述整数和浮点值与像素（和纹理坐标）的关系。给定水平像素数组并使用笛卡尔坐标，最左像素的左边界在浮点坐标中为0.0。 OpenGL一直使用此方案，DirectX 10及其后续版本使用它。该像素的中心为0.5。因此，像素[0，9]的范围覆盖了[0.0，10.0）的范围。转换很简单

$$d = floot(c) \qquad (2.1)$$
$$c = d + 0.5 \qquad (2.2)$$

其中d是像素的离散（整数）索引，c是像素内的连续（浮点）值。

尽管所有API的像素位置值都从左到右增加，但在某些情况下OpenGL和DirectX之间的上下边界零位置不一致。[2]OpenGL始终偏爱笛卡尔系统，将左下角视为值最低的元素，而DirectX有时会根据上下文将左上角定义为此元素。每种都有逻辑，在不同之处没有正确答案。例如，（0，0）位于OpenGL中图像的左下角，而在DirectX中位于左上角。从一个API迁移到另一个API时，必须考虑到这一差异。

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。除了在指定特定版本时编写“ DirectX”和在讨论该特定API时编写“ Direct3D”之间，我们没有区别，而是通篇编写“ DirectX”来遵循常用用法。

2.4光栅化

给定经过变换操作和投影操作的顶点及其关联的着色数据（全部来自几何阶段），下一阶段的目标是查找要渲染的图元（例如三角形）内的所有像素（图片元素的简称）。我们称这种过程为光栅化，它分为两个功能子阶段：三角形设置（也称为图元装配）和三角形遍历。这些显示在图2.8的左侧。注意，它们也可以处理点和线，但是由于三角形是最常见的，因此子阶段的名称中带有“三角形”。光栅化，也称为扫描转换，是从屏幕空间中的二维顶点（每个顶点具有z值（深度值）以及与每个顶点相关的各种阴影信息）到屏幕上的像素的转换。光栅化也可以被视为几何处理阶段和像素处理阶段之间的同步点，因为在这里，三角形是由三个顶点形成的，并最终传递到像素处理阶段。

三角形是否被视为与像素重叠取决于您如何设置GPU管线。例如，您可以使用点采样来确定“内部”。最简单的情况是在每个像素的中心使用一个点采样，因此，如果该中心点在三角形内部，则相应的像素也被认为在三角形内部。您还可以使用超采样或多采样抗锯齿技术对每个像素使用一个以上的采样（第5.4.2节）。还有一种方法是使用保守光栅化，其定义是，如果像素的至少一部分与三角形重叠，则该像素“位于三角形内”（第23.1.2节）。

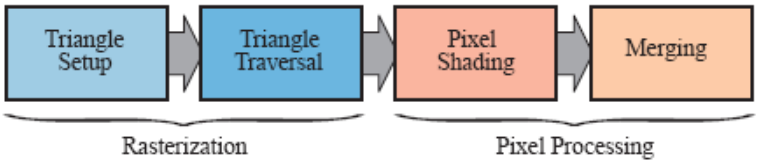

图2.8。左：光栅化分为两个功能阶段，称为三角形建立和三角形遍历。右图：像素处理分为两个功能阶段，即像素处理和合并。

2.4.1三角形建立

在这一阶段，计算三角形的微分，边方程和其他数据。这些数据可用于三角形遍历（第2.4.2节），以及用于插值几何阶段产生的各种着色数据。这部分的任务通常由硬件完成。

2.4.2三角形遍历

在这里检查每个中心（或样本）被三角形覆盖的像素，并为与三角形重叠的像素部分生成一个片段。更多详细的采样方法可以在第5.4节中找到。查找哪些像素在三角形内通常被称为三角形遍历。每个三角形片段的属性都是使用在三个三角形顶点之间插值的数据生成的（第5章）。这些属性包括片段的深度，以及来自几何阶段的任何着色数据。 McCormack. [1162]提供了更多关于三角形遍历的信息。在这里，也可以在三角形上执行透视校正插值[694]（第23.1.1节）。然后将图元内部的所有像素（或样本）发送到像素处理阶段，如下所述。

2.5像素处理

在这一点上，由于所有先前阶段组合的结果，已经找到了在三角形或其他图元内部考虑的所有像素。像素处理阶段分为像素着色和合并，如图2.8右侧所示。像素处理是对一个图元内部的像素或样本执行计算和操作的阶段。

2.5.1像素着色

使用经过插值的着色数据作为输入，此阶段可以执行任何每个像素的着色计算。最终结果是一种或多种颜色将传递到下一个阶段。与通常由硬件执行的三角形设置和遍历阶段不同，像素着色阶段由可编程GPU内核执行。为此，程序员为像素着色器（或片段着色器，在OpenGL中众所周知）提供了一段程序，该程序可以包含任何所需的计算。此阶段可以使用多种技术，其中最重要的一种是纹理化。在第6章中将更详细地讨论纹理化。简单地说，对一个对象进行纹理化意味着出于各种目的将一个或多个图像“粘合”到该对象上。此过程的一个简单示例如图2.9所示。图像可以是一维，二维或三维图像，其中二维图像是最常见的。最简单的说，最终产品是每个片段的颜色值，并将它们传递到下一个子阶段。

图2.9左上方显示了没有纹理的龙模型。图片中的碎片，将纹理“粘合”到龙上，结果显示在左下方。

2.5.2合并

每个像素的信息都存储在颜色缓冲区中，颜色缓冲区是颜色的矩形阵列（每种颜色的红色，绿色和蓝色分量）。合并阶段负责将像素着色阶段产生的片段颜色与当前存储在缓冲区中的颜色进行组合。此阶段也称为ROP，代表“光栅操作（管道）”或“渲染输出单元”。与像素着色阶段不同，执行此阶段的GPU子单元通常是不可完全可编程的。但是它是高度可配置的，可实现各种效果。

此阶段还负责解决可见性。这意味着在渲染整个场景后，颜色缓冲区应包含场景中从相机的角度可见的图元的颜色。对于大多数甚至所有图形硬件，这都是通过z缓冲区（也称为深度缓冲区）算法[238]完成的。 Z缓冲区的大小和结构与颜色缓冲区相同，并且对于每个像素，Z缓冲区将z值存储到当前最接近的图元。这意味着，当将图元渲染到某个具体的像素时，该图元在该像素处的z值将被计算并与同一像素处z缓冲区的内容进行比较。如果新的z值小于z缓冲区中的z值，则正在渲染的图元比之前在那个像素处最接近相机的图元更接近相机。因此，该像素的z值和颜色将使用所绘制图元的z值和颜色进行更新。如果计算的z值大于z缓冲区中的z值，则保持颜色缓冲区和z缓冲区不变。 z缓冲区算法很简单，具有O（n）收敛（其中n是要渲染的图元的数量），作用于任何z值能通过和其相关的像素被计算的图元。还要注意，该算法允许大多数图元以任何顺序呈现，这是其普及的另一个原因。但是，z缓冲区仅在屏幕上的每个点存储一个深度，因此不能用于部分透明的图元。透明图元必须在所有不透明基元之后渲染，以从后到前的顺序或使用其他顺序的算法（第5.5节）呈现这些内容。透明是z缓冲区的主要弱点之一。

我们已经提到了颜色缓冲区用于存储颜色，而z缓冲区用于存储每个像素的z值。但是，还有其他通道和缓冲区可用于筛选和捕获片段信息。 Alpha通道与颜色缓冲区关联，并为每个像素存储相关的不透明度值（第5.5节）。在较早的API中，alpha通道还用于通过alpha测试功能选择性地丢弃像素。如今，可以将丢弃操作插入到像素着色器程序中，并且可以使用任何类型的计算来触发丢弃。可以使用这种类型的测试来确保完全透明的片段不会影响z缓冲区（第6.6节）。

模板缓冲区是一个屏幕外缓冲区，用于记录渲染图元的位置。通常每个像素包含8位。可以使用各种方法将图元渲染到模板缓冲区中，然后可以使用模板缓冲区的内容来控制渲染到颜色缓冲区和z缓冲区。例如，假设已将填充圆绘制到模板缓冲区中。允许以填充圆作为父物体的后续图元和填充圆组合渲染到颜色缓冲区中。模板缓冲区可以是生成某些特殊效果的强大工具。流水线末端的所有这些功能都称为光栅操作（ROP）或混合操作。可以将当前颜色缓冲区中的颜色与三角形内要处理的像素的颜色混合。这可以实现诸如透明度或颜色采样累积的效果。如前所述，混合通常可以使用API进行配置，而不是完全可编程的。但是，某些API支持光栅顺序视图，也称为像素着色器顺序，允许可编程混合功能。

帧缓冲区通常由系统上的所有缓冲区组成。

当图元到达并通过光栅化阶段时，从相机的角度可见的图元将显示在屏幕上。屏幕显示颜色缓冲区的内容。为了避免让人类观看者在对其进行栅格化并将其发送到屏幕时看到它们，使用了双重缓冲。这意味着场景的渲染发生在屏幕后方的缓冲区中。将场景渲染到后缓冲区中后，后缓冲区的内容将与先前在屏幕上显示的前缓冲区的内容交换。交换通常发生在垂直回扫过程中，这是安全的时间。

有关不同缓冲区和缓冲方法的更多信息，请参见第5.4.2、23.6和23.7节

2.6纵观渲染管线

点，线和三角形是用于构建模型或对象的渲染图元。假设该应用程序是交互式计算机辅助设计（CAD）应用程序，并且用户正在检查华夫饼制作机的设计。在这里，我们将在整个图形渲染管道中使用此模型，包括四个主要阶段：应用程序，几何，光栅化和像素处理。通过透视投影将场景渲染到屏幕上的窗口中。在这个简单的示例中，华夫饼制作机模型同时包含线（以显示零件的边缘）和三角形（以显示表面）。华夫饼制作机的盖子可以打开。一些三角形是由带有制造商徽标的二维图像制成的。对于此示例，除了在光栅化阶段发生的纹理应用之外，表面着色是在几何阶段完全计算出来的。

应用程序阶段

CAD应用程序允许用户选择和移动模型的各个部分。例如，用户可以选择盖子，然后移动鼠标将其打开。应用阶段必须将鼠标移动转换为相应的旋转矩阵，然后确认呈现该矩阵时已将其正确应用于盖子。另一个示例：播放动画，使动画沿预定路径移动，以从不同的角度显示华夫饼制作机。然后必须由应用程序根据时间更新相机参数，例如位置和视图方向。对于要渲染的每个帧，应用程序阶段将相机的位置，照明和模型的图元传入管线中的下一个主要阶段-几何阶段。

几何阶段

对于透视图，我们在此假定应用程序已提供了投影矩阵。同样，对于每个对象，应用程序都已计算出一个矩阵，该矩阵描述了视图变换以及对象本身的位置和方向。在我们的示例中，华夫饼制造商的底座将具有一个矩阵，而盖子则具有另一个矩阵。在几何阶段，使用此矩阵转换对象的顶点和法线，从而将对象变化到视图空间。然后，可以使用材质和光源属性来计算顶点处的阴影或其他计算。然后使用单独的用户提供的投影矩阵执行投影，将对象转换为代表眼睛所见的单位立方体的空间。单位立方体外部的所有图元都将被丢弃。将与该单位立方体相交的所有图元都裁剪到该单位立方体上，以便获得一组完全位于单位立方体内的图元。然后将这些顶点映射到屏幕上的窗口中。在完成所有这些按三角形和按顶点进行的操作之后，将所得数据传递到光栅化阶段。

光栅化

然后光栅化所有在上一阶段幸存下来的图元，这意味着找到了图元内部的所有像素，并将它们进一步发送到管线以进行像素处理。

像素处理

此处的目标是计算每个可见图元的每个像素的颜色。那些与任何纹理（图像）相关联的三角形将根据需要应用这些图像进行渲染。可见性通过z缓冲区算法以及可选的丢弃和模板测试来解决。依次处理每个对象，然后将最终图像显示在屏幕上。

结语

渲染管线源于针对实时渲染应用程序的数十年的API和图形硬件演变。重要的是要注意，这不是唯一的渲染管线。离线渲染管道经历了不同的进化路径。电影制作的渲染通常是通过微多边形管线完成的[289，1734]，但是光线追踪和路径追踪近来已取而代之。 11.2.2节中介绍的这些技术，这些技术也可以用于建筑和设计的预可视化。

多年来，应用程序开发人员使用此处描述的过程的唯一方法是通过使用中的图形API定义的固定功能管道。固定功能管道之所以如此命名，是因为实现它的图形硬件包含无法灵活编程的元素。主要的固定功能机器的最后一个例子是2006年推出的Nintendo的Wii。另一方面，可编程GPU使得可以确切确定在整个生产流程的各个子阶段中应用了哪些操作。对于本书的第四版，我们假设所有开发都是使用可编

程GPU完成的。

进一步阅读和资源

布林的著作《A Trip Down the Graphics Pipeline 》 [165]是一本关于从头开始编写软件渲染器的旧书。它是学习实现渲染管线的一些精妙之处，解释关键算法（例如裁剪和透视插值）的好资源。古老的（至今仍经常更新）《 OpenGL编程指南》（又称“红皮书”）[885]提供了图形管道的完整描述以及与其使用相关的算法。本书的网站realtimerendering.com提供了指向各种管线图，渲染引擎实现等的链接。