The Graphics Rendering Pipeline

渲染管线，这章主要讲光栅化渲染管线。

> *毕业前实习时，也实现过一个简单的软光栅化渲染管线，再复习一下。*

> *在计算机图形学领域，shading指基于表面相对灯光的角度、距灯光的距离、相对于相机的角度和材质的属性等来修改物体/表面/多边形的颜色，进而创造一个具有真实感效果的过程。*
> *In computer graphics, shading refers to the process of altering the color of an object/surface/polygon in the 3D scene, based on things like (but not limited to) the surface's angle to lights, its distance from lights, its angle to the camera and material properties (e.g. bidirectional reflectance distribution function) to create a photorealistic effect. Shading is performed during the rendering process by a program called a shader.*

graphics rendering pipeline, 也被称为"the pipeline"，即图形渲染管线。

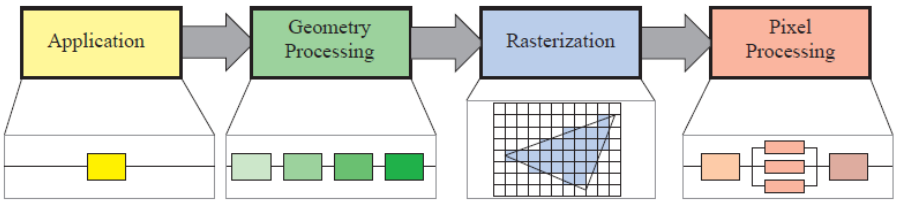
图形渲染管线的主要功能是根据给定的虚拟相机、三维物体和光源等，生成(或渲染)一个二维图像。

2.1 架构

一条渲染管线由几个阶段(stages)组成，每个阶段完成一个大的任务。

pipeline stages并行执行，同时每个阶段要依赖上一个阶段的处理结果。理想情况，一个nonpipelined 系统被分成 n个pipelined stages，可以得到n倍的加速。

Real-time rendering pipeline可以粗略地分为四个主要的stages: *application*, *geometry processing*, *rasterization*和*pixel processing*，如下图。



从上图可以看出一个stage可能本身也是一个pipeline，如上图*Geometry Processing*stage。一个stage也可能是(partly)并行化的，如:*Pixel Processing* stage。

Application stage用application驱动，由运行在CPU上实现的软件。一些通常在CPU上执行的任务：collision detection, global acceleration algorithms, animation, physics simulation等。

*Geometry processing*处理变换、映射等geometry handling类型的任务。这个stage计算出: what is do be drawn, how it should be drawn, and where it should be drawn. geometry stage通常运行在GPU上。

Rasterization stage通常以三角形的三个顶点作为输入，计算出属于三角形内部的像素，然后把这些像素传给下一个stage。

Pixel processing stage对于每个像素执行一个程序(program，应该是指shader)，决定该像素的颜色，也可能执行深度测试(depth testing)，来判断其是否可见。这个stage也可能执行per-pixel(逐像素)操作，例如：blending the newly computed color with a previous color。rasterization和pixel processing这两个stages也完全在GPU上处理。

2.2 The Application Stage

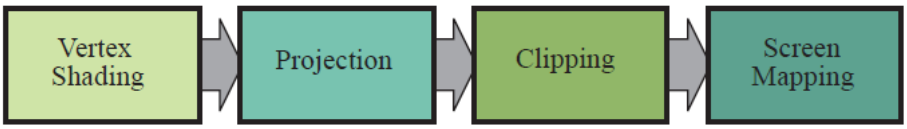
开发者对*application stage*发生的事有绝对的控制权，因为这个阶段是在CPU上执行的。例如，一个*application stage*的算法或设置可以减少被渲染的三角形的数量。

但是一些application work也可以通过GPU完成，这要用到计算着色器(*compute shader*)。*compute shader*把GPU当成一个高度并行的通用处理器(highly parallel general processor)，而忽略了它渲染图形的专门功能。

在*application stage*的末尾，要被渲染的geometry被提供给*geometry processing stage*。这些是*rendering primitives*，例如，points, lines,和triangles，这些最终可能会出现在屏幕上(或者别的任何输出设备)。这是application stage最重要的任务。

2.3 Geometry Processing

Geometry processing stage (GPU上)负责大部分per-triangle(逐三角形的)、per-vertex(逐顶点的)操作。每个阶段又可以进一步分成以下几个功能性阶段(functional stages): vertex shading, projection, clipping, and screen mapping。如下图:



2.3.1 Vertex Shading

Vertex Shading有两个主要的任务，即：

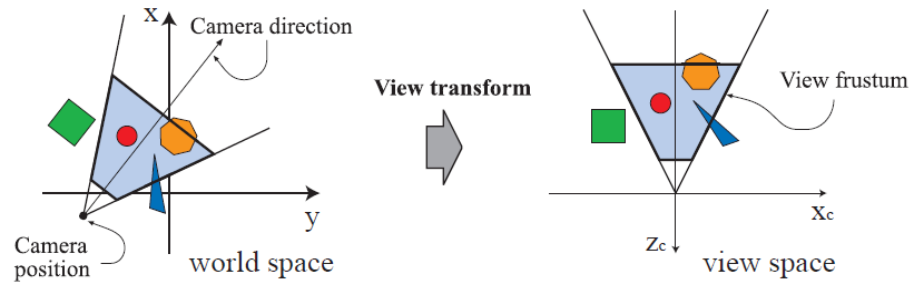
1. 计算顶点(vertex)的位置(postition)
2. 计算任何程序员可能想要的顶点输出数据(vertex output data)，例如法线、纹理坐标。

传统上，大部分物体着色(shade of an object)是通过对每个顶点位置和法线应用光照并把产生的颜色存储在顶点(vertex)中来计算的。这些颜色将会在每一个三角形内部插值。因为这个原因，这个可编程的顶点处理单元被命名为vertex shader。随着现代GPU的出现，随着部分或全部的逐像素着色(per-pixel shading)的出现，vertex shading stage功能变得更加泛化，可能根本不进行任何着色(或者说不计算任何着色方程，shading equation)，这取决于程序员的意图。现在vertex shader是一个更通用的单元，用于设置和每个顶点相关联的数据。

在被显示到屏幕的过程中，一个模型(model)被变换到好几个不同的空间(*spaces*)或者坐标系统(*coordinate systems*)。最初一个模型存在于它自己的模型空间(*model space*)，即根本没有发生变换。每一个模型可以与一个模型变换(*model transform*)相关联，这样它就可以被定位或定向。也可能发生单个模型(single model)与多个model transforms相关联。也允许在同一场景中，相同模型(same model)的多个拷贝(叫作实例，*instances*)有不同的位置、朝向或尺寸，不且不需要复制基本几何体(basic geometry)。

被model transform变换的是模型(model)的顶点(vertices)和法线(normals)。一个物体的坐标叫作模型坐标(model coordinates)，模型施加完model transform后，就说模型处于世界坐标(*world coordinates*)或世界空间(*world space*)。世界空间是唯一的，所有模型经过各自的model transforms后，都处于同一个空间/坐标系了。

只有被camera(或observer)看到的模型(models)才会被渲染。Camera在世界空间有一个位置(location)和方向(direction)，用于放置和瞄准相机。为了方便投影(projection)和裁剪(clipping)，相机和所有模型(models)会经历*view transform*。View transform的目的是把camera放置在原点(origin)上，并使得camera看向负z轴(negative z-axis)，y 轴朝上(pointing upward)，x 轴朝右。本书使用$-z$轴的约定(右手坐标系)，一些教材可能喜欢让camera看向$+z$轴。模型经view transform变换后就处于*camera space*了，或者叫*view space*，*eye space*。下图是一个view transform影响camera和models的例子。



为了产生一个具有真实感的场景，仅仅渲染物体的形状和位置是不够的，也要渲染他们的“样子”(appearance)。这个 描述包含每个物体的材质和照射到物体的光源的效果。材质和光源多种方式建模，从简单的颜色到复杂的物理描述。

决定灯光在材质上的效果的操作被称为着色(*shading*)。它涉及到在物体各个点上计算着色方程*shading equation*。通常，这些计算的一部分是在模型顶点(vertices)的几何处理阶段完成的，而另一些是在per-pixel处理时完成的。各种各样的材质数据可以被存储在每个顶点(vertex)上，如：点的位置，法线 ，颜色或者任何别的用于计算着色方程的数值信息。Vertex shading的结果(可能是colors, vectors, texture coordinates，以及别的数据)被送到rasterization和pixel processing阶段，被插值，被用于计算表面着色(shading of the surface)。

作为vertex shading的一部分，渲染系统完成*projection*(投影)和随后的*clipping*(裁剪)，这会把视景体(view volume)转换成一个单位立方体(unit cube)，它的顶点坐标范围为(-1, -1, -1)和(1, 1, 1)。这个单位立方体被称为标准视景体(*canonical view volume*)。首先完成投影(Projection)，在GPU上是通过vertex shader完成的。有两种常见的投影方法，即正交投影(*orthographic projection*)，也称作平行投影(*parallel projection*)，和透视投影(*perspective projection*)。另外几种投影方式，如：oblique和axonometric投影。

正交投影的视景体(view volume)通常是一个的长方体，而正交投影会把这个长文体变换成一个单位立方体(unit cube)。正交投影的主要特点是变换后平行线仍然保持平行。这个变换是平移和缩放的组合。

透视投影更复杂点。在透视投影中，物体离像机越远，物体在投影后看起来越小。另外，平行线可能会相交。投视变换模仿我们感知物体尺寸的方式。在几何学上，透视投影的视景体(view volume)被称为视锥体(*frustum*)，即一个截头锥体。视锥体也会被变换成一个单位立方体(unit cube)。正交变换和透视变换都可以用4×4矩阵来构造，经过两者中任何一种变换后，都说模型处于裁剪空间或裁剪坐标(*clip coordinates*)。这些实际上是齐次坐标(homogeneous coordinates)，出现在被w除之前。为了使下一个功能阶段，clipping(裁剪)，正确工作，GPU的vertex shader必须总是输出这种类型的坐标。

尽管这些矩阵把一个volume(视景体)变换成另一个(unit cube)，他们被称投影(projections)，因为显示后(after display)，z坐标不再存储在生成的图像中，而是存储在z-buffer中。以这种方式，模型被从三维空间投影到了二维空间。

2.3.2 Optional Vertex Processing

上面讲的vertex processing是每个渲染管线都有的。一旦这个过程完成后，还有一些在GPU上发生的可选的阶段(stages)，按此顺序：tessellation，geometry shading和流输出(stream output)。是否使用由硬件的能力(并不是所有的GPUs都支持)和程序员的意愿决定。它们之间相互独立，通常用不到它们。

第一个可选的阶段是曲面细分/表面细分(*tessellation*)。设想你有一个弹球对象。如果你用一个单独的三角形集合(a single set of triangles)来表示它，你可能会遇到质量或性能问题。你的球可能在5米远处看起来挺好的，但是一旦靠近，就会看到一个个三角形。如果你使用更多的三角形来提高球的渲染质量，当球离相机远只覆盖屏幕上一点点像素的时候，你可能会浪费相当大的处理时间和内存。通过曲面细分(tessellation)可以使用适量个数的三角形来生成一个弯曲表面(curved surface)。

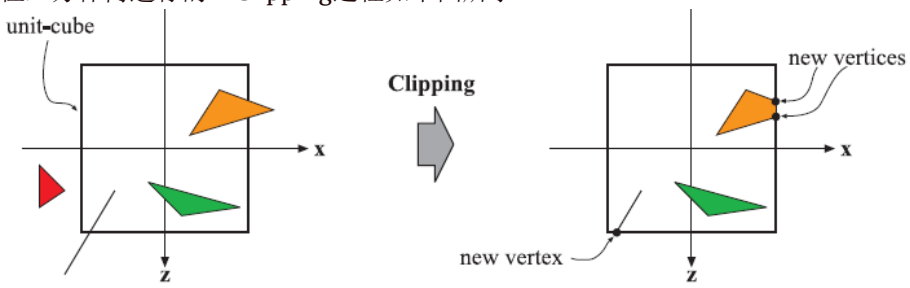
下一个可选阶段(stage)是*geometry shader*。Geometry shader在历史上出现的时间(不是指在渲染管线的顺序)上早于表面细分(tessellation shader)，而且相比之下，在GPUs上更常见。它和tessellation shader相似，因为它接收各种类型的基本体(primitives，比如点、线、面)，然后可以产生新的顶点(vertices)。它是一个更简单的stage，因为这种creation在范围和output primitives的类型上有受到了限制。想像一下模拟烟花爆炸。每个火球可以用一个点(a single vertex)来表示。Geometry shader可以接收每一个点，把它转变成一个面向观察者、覆盖一些像素的正方形(由两个三角形组成)，提供了一个能令人更加信服的用于着色的基本体。

最后一个可选的stage被叫做*stream output*。这个stage允许把GPU当成一个几何引擎(geometry engine)。这个stage不是把处理过的顶点数据沿着渲染管线继续向下传送渲染到屏幕上，而是可以选择(optionally)把这些数据输出到一个数组中用于进一步处理。这些数据可以被CPU或GPU使用。这个stage通常被用于粒子模拟，如烟花的例子。

这三个stages按这样的顺序执行: tessellation, geometry shading, stream output，而且每一个都是可选的。每哪一个被使用了，如果我们沿着渲染管线走，我们得到的齐次坐标下的顶点集，这将会被用于检测相机是否可以看到它们。

2.3.3 Clipping

Primitives(点、线、三角形，这里裁剪是对这些基本体进行的)只有在完全或部分出现在视景体(view volume)内部时，才能被传递到光栅化阶段(rasterization stage)以及随后的像素处理阶段(pixel processing stage)，才会被绘制到屏幕上。一个完全落在视景体内的primitive会按它原有的样子传递到下一个阶段。完全处于视景体外的Primitives不会被进一步传递到下一个阶段，因为它们不会被渲染。只有那些部分落在视景体内的物体才会被裁剪，这样会使得视景体外的顶点被新的primitive与视景体相交的点替代。由于使用了投影矩阵，这意味着这些经投影变换后的primitives是相对于一个单位立方体进行裁剪的。在裁剪(clipping)之前，先进行view transformation (从世界空间变换到view space)和投影(projection)的优势是使得裁剪问题具有一致性，所有primitives总是在一个单位立方体内进行的。Clipping过程如下图所示。

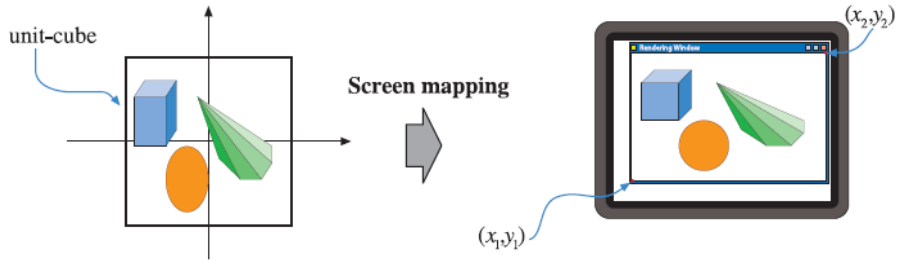


除了视景体的六个裁剪面，用户也可以定义附加的裁剪面(clipping planes)来可视化地修剪物体。(还没读到具体怎么操作)

裁剪这一步(clipping step)使用由投影产生的有四个分量(4-value)的齐次坐标来完成。透视空间中(perspective space)中，坐标不值不是在三角形中进行普通的线性插值。为了在使用透视投影时，数据能够被合理地插值和裁剪，才需要用到第四个坐标值。最后一步是透视除法(*perspective division*)，这会把三角形放到一个三维的标准化设备坐标(*normalized device coordinates*)中。像之前提到的那样，这个视景体的坐标值变化范围是$(-1,-1,-1)$到$(1,1,1)$。在几何阶段(geometry stage)的最后一步是把标准化设备坐标(空间)标转化到窗口坐标(window coordinates)。

2.3.4 Screen Mapping

只有经裁剪后仍然在视景体内部的primitives才会被传递到屏幕映射阶段(screen mapping stage)，且刚进入这一段时的坐标仍然是三维的。每个primitive的x,y坐标通过变换成为屏幕坐标(*screen coordinates*)。屏幕坐标和z坐标一起被称为窗口坐标(*window coordinates*)。假设场景被渲染进一个窗口，其中两个为(x_1,y_1)和$(x_2,y_2),x_1<x_2,y_1<y_2$。屏幕映射是一个伴随着缩放操作的变换。新的$x,y$坐标被称为屏幕坐标。$z$坐标(OpenGL:$[-1,1]$, DirectX: [0, 1])被映射到$[z_1,z_2]$，默认情况下$z_1=0,z_2=1$ (可以调用 API修改值)。窗口坐标(带有重新映射过的z值)被传递到光栅化阶段(rasterizer stage)。屏幕映射过程如下图:

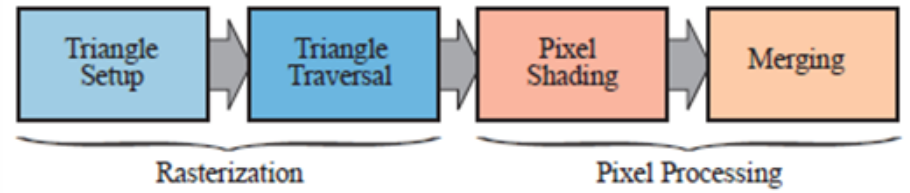


接下来，我们描述与像素(和纹理坐标)关联的整数值和浮点值之间的转换关系。给定一组水平排列的像素(pixels)，使用笛卡尔坐(Cartesian coordinates)，则最左侧边上的像素坐标为0.0(用浮点数表示)。OpenGL总是使用这种策略，DirectX 10及以上也使用它。这个最左边的像素的中心为0.5。因此像素$[0,9]$(可以看作一组像素的索引)跨越坐标$[0.0,10.0]$。转换(conversions)可简单地表述为:
$d = floor(c)$,
$c = d + 0.5$,
此处d是像素离散的(整型)索引，而c是在像素内的连续的(浮点)值。

在所有图形APIs中像素的位置的横坐标都是从左向右逐渐变大的，但纵坐标0点的位置在OpenGL和DirectX中是不一致的。OpenGL把左下角作为值最小元素(原点)，而DirectX有时根据上下会把左上角定义为原点。

Rasterization 光栅化

给定变换和投影后的带有着色数据(shading data)的顶点(vertices)(都来自几何处理阶段)，下一个阶段的目标是找到所有在基本体(primitive，如:点、线、三角形)内部的像素(pixels，即picture elements的简称)。我们称这个过程为光栅化(*rasterization*)，它被分为两个功能子阶段：三角形设置(triangle setup，也称为图元装配，primitive assembly)和三角形遍历(triangle traversal)，如下图所示。这些过程即适用于点(points)、线(lines)，也适用于三角形(trangles)，但三角形是最常见的，所以本阶段用“三角形”来称这些基本体。



光栅化，也叫作扫描转换(*scan conversion*)，是从屏幕空间的具有z值(即深度值，depth value)和其它着色信息的二维顶点(vertices)到屏幕像素的转换。也可以认为光栅化是几何处理和像素处理的一个同步点(synchronization point)，因为由三个顶点(vertices，除了位置坐标，还包含别的信息，纹理坐标，法线)构成的三角形是在这里被发送到像素处理阶段的。

三角形是否覆盖某个像素，由你怎么设置GPU的渲染管线决定。例如，你可能使用点采样来决定是否在三角形内部("insideness")。最简单的情形是使用在每个像素中心的单点样本，这样如果这个中心点在三角形内，则认为对应的像素也在三角形内。对于一个像素，你可能使用多于一个采样点(使用supersampline或multisampling反锯齿技术)。另外一种方法是使用保守的光栅化，只要一个像素至少一部分与三角形重叠，就认为此像素在三角形内("inside")。

2.4.1 Triangle Setup 三角形设置

在这个阶段differentials, edge equations和此三角形的别的数据被计算了。这些数据可能用于接下来的三角形遍历(triangle traversal)，和几何阶段产生的各种着色数据(shading data)的插值。使用固定功能的硬件来完成这个任务。

2.4.2 Triangle Traversal 三角形遍历

这里是检测每个像素的中心点(或采样点)是否被三角形覆盖了的地方，并且为被三角形覆盖的部分产生一个*fragment*。之后，在第五章会介绍更多的精细复杂的采样方法。查找哪些采样点或像素在三角形内部的过程通常称为三角形遍历(*triangle traversal*)。三角形的每一个fragment的属性是通过在三角形三个顶点

之间进行数据插值产生的。这些属性包括fragment的深度和所有从几何阶段得到的着色数据。这里也是透视校正插值(perspective-correct interpolation)完成的地主。所有在一个primitive(点、线、三角形)内部的像素或采样点被送到像素处理阶段(pixel processing stage，见下方)。

2.5 Pixel Processing

此时，所有被认为在三角形或别的primitive内部的点已经被发现了(作为之前所有阶段组合的结果)。像素处理阶段被分为两部分: 像素着色(*pixel shading*)和合并(*merging*)，如上图右图。像素处理阶段是在primitive内部的像素(pixels)或样本(samples)上完成逐像素(per-pixel)或者(per-sample)操作或计算的地方。

2.5.1 Pixel Shading (像素着色)

所有逐像素的着色计算都是在这里完成的，使用插值了的着色数据作为输入。最终结果是一个或多个将被传入到下一个阶段的颜色。Pixel Shading用可编程的GPU cores完成。为此，程序员需要为pixel shader(OpenGL中称为fragment shader)提供一个程序(里面可以包含任意想要的计算)。在这儿，各种各样的技术可以被使用，其中最重要的一个是*texturing*。简单地讲，texturing一个物体意味着把一个或多个图像(images)“粘/贴到”("gluing")一物体上。用到的图像可以是一维的、二维的，甚至三维的，其中二维的最常见。最终的产出是每个fragment的像素值，这些数据会被传递到下一个子阶段。

上图左上角的图片是没有纹理的龙，右边是使用的纹理图片，左下角是使用纹理后的龙。

2.5.2 Merging 合并/融合(混合？)

每一个像素的信息被存储在颜色缓存(*color buffer*)中，color buffer是colors(每个颜色具有红、绿、蓝分量)的矩形数组(二维数组)。Merging stage的任务是结合(combine)pixel shading阶段产生的fragment color和当前已经在缓存(buffer)中存储着的颜色。这个阶段也被称为ROP，表示“raster operations (pipeline)” (光栅操作？)或者“渲染输出单元”(render output unity)。和shding stage不同，执行Merging阶段的GPU 子单元(subunit)通常是不可编程的。但是，这是可高度配置的，能够实现各种效果(透明度混合？)。

这个阶段也负责解决**可见性问题(visibility)**。这意味着当整个场景被渲染后，color buffer中存储的应该是在相机视点可见的场景中的primitive的颜色。对于大部分甚至所有图形硬件(graphics hardware)，这是通过z-buffer(也称为*depth buffer*)算法实现的。z-buffer和颜色缓存(color buffer)的大小和形状一样。对于每一个像素(pixel)，z-buffer存储当前离相机最近的primitive的z-value，z值。这意味着当一个primitive被渲染到某个像素时，这个primitive在这个像素的z值会被计算并与z-buffer中相同像素点的的内容进行比较。如果新的z-value比z-buffer中的z-value小，则正在被渲染的primitive成为当前在那个像素点离相机最近的primitive。因此，那个像素点的z-value和color会被用正在渲染的primitive的z-value和color更新。如果计算得到z-value比z-buffer中的大，则color buffer和z-buffer会保持不变。z-buffer算法是简单的，具有$O(n)$的收敛性(n是要被渲染的primitives的数量)，对于绘制任何primitive都支持，只要每个相关像素的z-value可以被计算。而且要注意这个算法允许大部分prmitives可以被以任何顺序渲染。然而，z-buffer在屏幕的每一点的仅存储一个单独的深度(depth)，它不可

以用于局部透明(partially transparent)的primitives。这些局部透明的primitives必须在所有不透明(opaque)物体渲染完之后再渲染，且以从后向前的顺序(back-to-front)，或者使用一个单独的顺序无关的(order-inedpendent)算法。透明度(transparency)是基本z-buffer的一个主要弱点(major weakness)。

我们已经提到过color buffer为每一个像素存储颜色，z-buffer存储z-values。然而，可以有别的通道(channels)和缓存(buffers)用于过滤(filter)和捕捉(capture)fragment信息。*Alpha channel*和color buffer相关联，并为每个像素存储一个不透明度值。在老点的APIs中，alpha通道用于选择性地丢弃(discard)像素(经过alpha test feature)。现如今，一个discard操作可以被插入到pixel shader程序，任何计算类型(type of computation)可以用于触发一个discard。这个测试类型(type of test)可以用于确保完全透明的fragments不影响z-buffer。

*Stencil buffer*是一个离屏缓存(offscreen buffer)，用于记录已被渲染的primitive的位置(locations)。通常对于每个像素，*stencil buffer*包含8个位(bits)。可以使用各种函数把primitives渲染进stencil buffer，且这个buffer的内容可以被用于控制向color buffer和z-buffer的渲染。作为一个例子，假设有一个填充的圆(filled circle)被绘制进stencil buffer。这可以和一个操作结合，使得随后primitives的绘制仅允许发生在这个圆存在的地方。Stencil buffer可以是一个强大的工具，用于生成某些特效(special effects)。在渲染管线的最后部分，所有函数被称为光栅操作(*raster operations*, ROP)或*blend operations*。混合当前在color buffer中的颜色和正在被处理的像素(inside a triangle)的颜色是有可能的。这可以启用效果，如：透明(transparency)或者颜色样本的累积。正如提到的那样，blending通常是可使用API配置的，并不能完全可编程。然而，某些APIs支持光栅化顺序视图(raster order views)，也称为pixel shader ordering，这启用了可编程的blending能力(capabilities)。

通常*framebuffer*由系统中所有的buffers组成。

当primitives中已经达到并被传递到光栅化阶段(rasterizer stage)，从相机视点可见的primitives将会被展示在屏幕上。屏幕展示color buffer的内容。为了避免人看到primitives被光栅化和被送到屏幕上的过程，双缓存(*double buffer*)被启用了。这意味着场景是离屏渲染的，在一个后台缓存(*back buffer*)中。一旦场景被在back buffer中渲染完，back buffer的内容会和前台缓冲(*front buffer*，即之前屏幕上显示的东西)进行交换。交换经常发生在垂直回描(*vertical retrace*)时，一个这样做安全的时间。

总结

这里讲的渲染管线是数十年来面向实时渲染应用程序的API和图形硬件发展演变的结果。需要注意的是它不是唯一的渲染管线。离线渲染(offline rendering)管线有不同的发展路径。电影产品的渲染经常使用*micropolygon pipelines*，但量最近光线追踪逐渐有取代的地位。本书讲的内容需要适合用编程的GPUs(programmable GPUs)，而不是只支持固定渲染管线(*fixed-function pipeline*)的图形API (graphics API)。

关于渲染管线有两个重要的书:

1. *A Trip Down the Graphics Pipeline* (比较老一点)
2. *OpenGL Programming Guide*，也称为“红书”(Red Book)