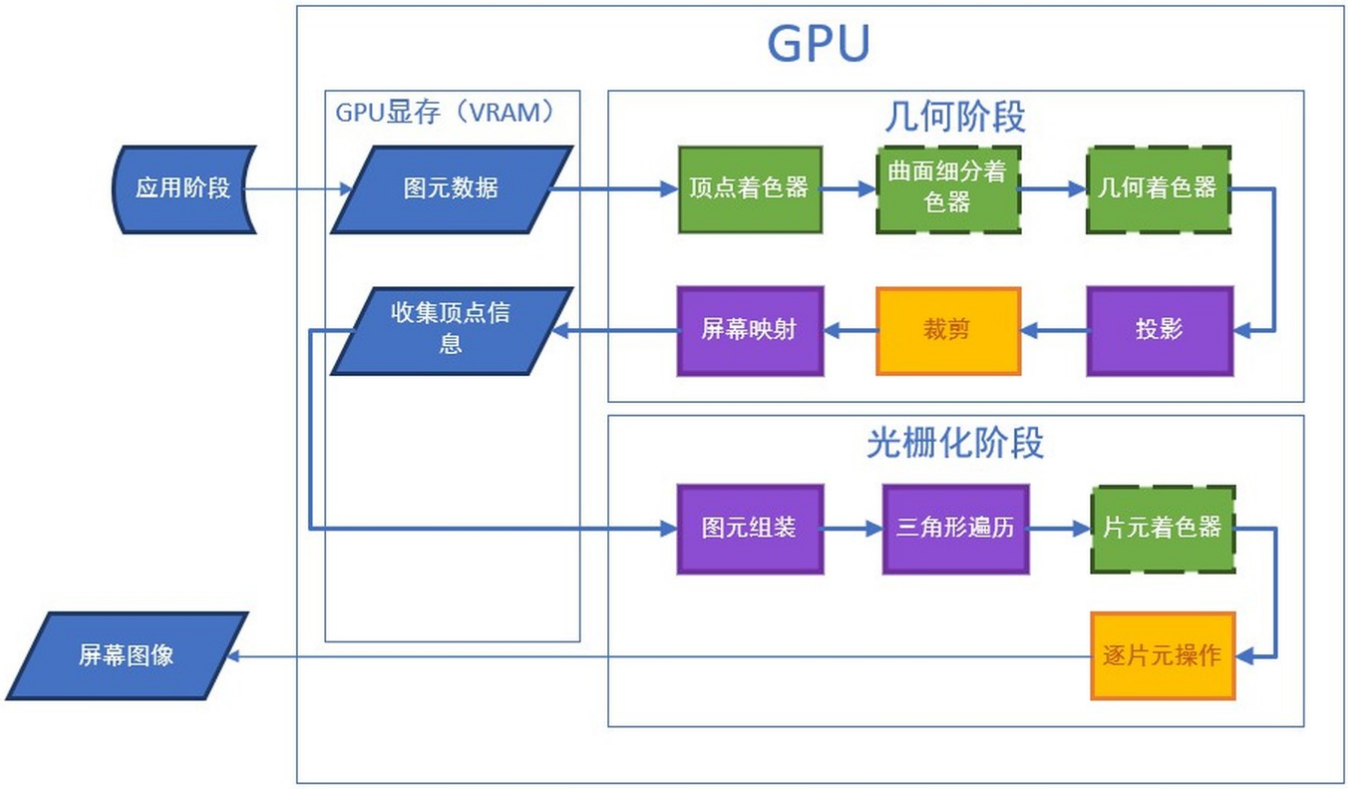


猴子也能看懂的渲染管线（Render Pipeline）

HkingA...
腾讯 技术美术

284 人赞同了该文章

CPU与GPU的区别

GPU的架构与CPU有极大的不同，这主要归因于两者不同的使用场合。试想一下，GPU面对3D游戏中成千上万的三角面，如果仅仅是逐一单个处理计算，损失的效率是极其惊人的。

3D游戏中包含着大量的三角面

这可以类比汽车工业的发展，在1913年前福特开发出汽车流水线前，汽车组装只能让一位位工人逐工序完成，年产不过12台，效率极低；而引入了流水线概念后，每位工人只需要做不停地做同一道工序，所有工序并行进行，极大地提高了工厂的生产效率，生产效率提高了8倍。

GPU对图像处理的高效率体现了同样的思路，GPU采用了数量众多的计算单元和超长的流水线，但每一个部分只有非常简单的控制逻辑（如同《摩登时代》中一个流水线工人只负责拧一个螺丝）。尽管计算能力不如CPU，但耐不住人多力量大；这就好比拿出一百道十以内加减法运算题给一百个小学生和一个资深大学教授来做，尽管小学生能力并不如何，但这么多个小学生同时做这些题消耗的总时长，总比一个学识渊博大学教授做要来得快。

渲染管线（Render Pipeline）

在渲染流程中，CPU与GPU正如上文一样通力合作渲染图像。在运算过程中，CPU如同进货的卡车不断地将要处理的数据丢给GPU，GPU工厂调动一个个如工人一般的计算单元对这些数据进行简单的处理，最后组装出产品——图像。

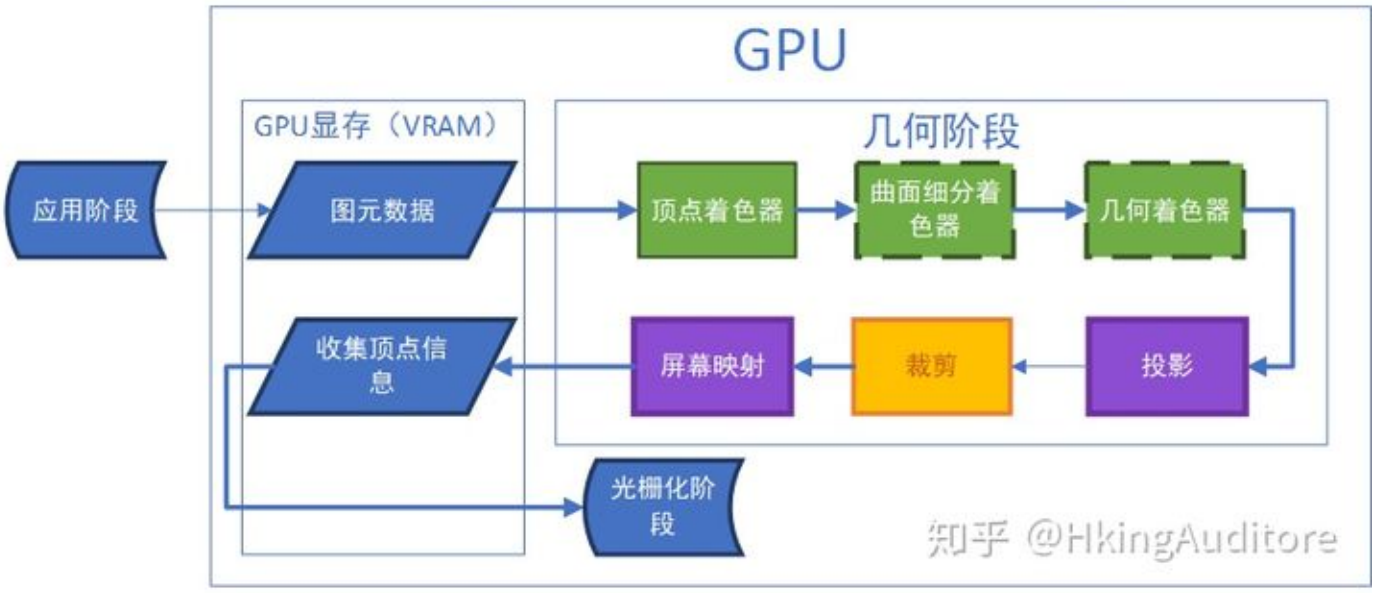
应用阶段（Application Stage）

这是一个由CPU主要负责的阶段，且完全由开发人员掌控。在这个阶段，CPU将决定递给GPU什么样的数据（譬如渲染目标场景中的灯光、场景的模型、摄像机的位置），有时候还会对这些数据进行处理（譬如只递给GPU可以被摄像机看见的元素，其他不可见的元素被**剔除（culling）**出去），并且告诉GPU这些数据的渲染状态（譬如纹理、材质、着色器等）。

我们同样用工业流水线进行类比，这一块相当于工厂的产品进口部门，采购员（CPU）联系发货单位（RAM）订购想要的原材料（数据），并经过一番精挑细选拿出自己满意的材料（数据处理，如剔除），把这些材料连同他们的加工方式（如应当使用的着色器）丢给工厂。值得注意的是，由于这一块采购员是与发货单位的商人而不是工厂里的工人交流， 所以他可以使用更复杂的语言（如高级程序语言）与商人讨价还价，而不是像在工厂中向工人发号施令时使用的指令（着色器语言）。

几何阶段（Geometry Stage）

这一个由GPU主导的阶段，也就是说，从这个阶段开始，我们进入了上文所说的“流水线”。几何阶段将把CPU在应用阶段发来的数据进行进一步处理，而这个阶段又可以进一步细分为若干个流水线阶段，可以类比理解为工厂流水线上进行的一道道工序。



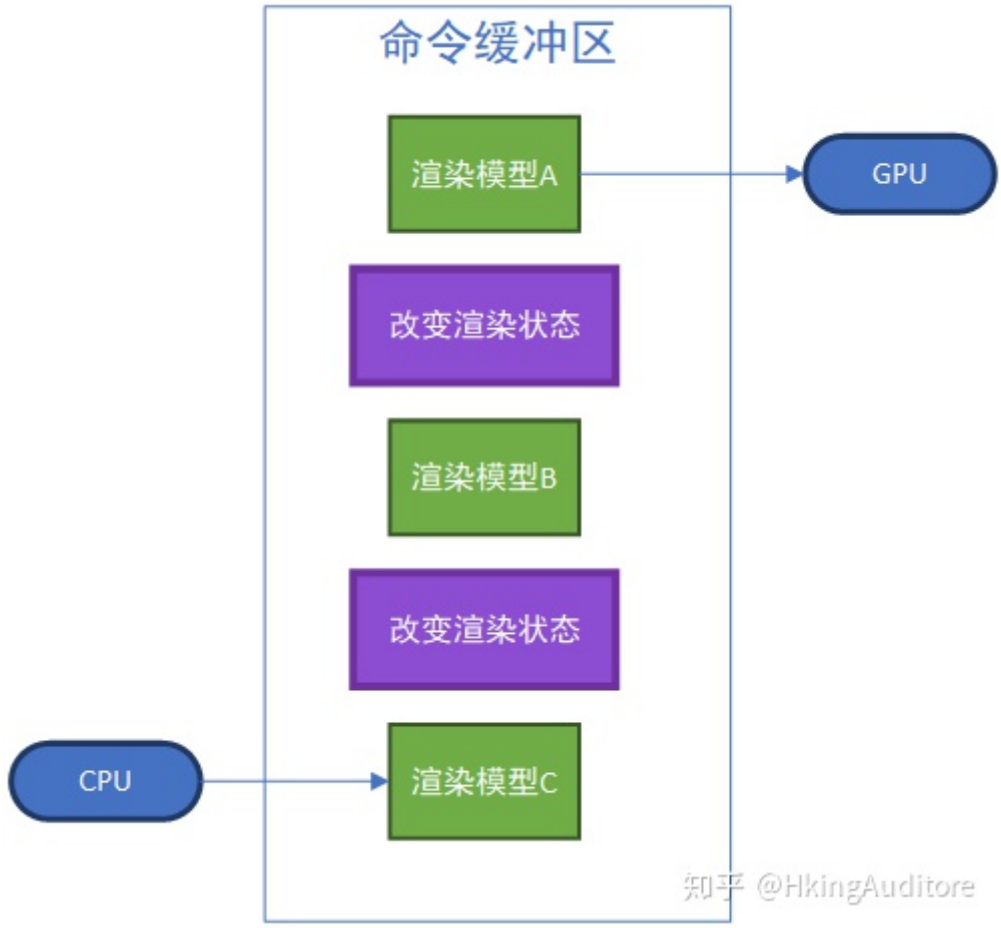
流程图中展现了几何阶段中几个常见的渲染步骤（不同的图像应用接口存在着些许不同，这里以OpenGL为例），其中，绿色表示开发者可以完全编程控制的部分，虚线外框表示此阶段不是必需的，黄色表示开发者无法完全控制的部分（但可以进行一些配置），紫色表示开发者无法控制的阶段（已经由GPU固定实现）。

下面详细解释这几个细分阶段所做的工作。

放入显存与Draw Call

在应用阶段，CPU从硬盘中把需要的数据拿出来放进了内存中，经过之前所述的一系列操作后，再打包发给GPU进行进一步处理。虽然从渲染的角度来看，当CPU把数据传递到显存中后，这些数据在内存中的使命就结束了，可以移除了，但对于一些特殊数据仍然可以“幸存”；譬如游戏中有一面墙，它不仅需要被渲染出来，还需要进行计算物体碰撞，那么CPU在将它的网格丢给GPU后并不会马上把它从内存中移除，因为CPU还需要用这个网格计算碰撞。

在应用阶段，尽管在CPU已经把数据准备得十分充分，但在完成传送任务后，CPU也不能一走了之，它还需要向GPU下达一个渲染指令，这个指令就是**Draw Call**，由于之前我们已经把这个数据准备得十分完善了，所以Draw Call仅仅是一个指向需要被渲染的图元列表，没有其他材质信息。这整个过程就好比进货人员把一捆写好了原料信息、加工方法的原材料丢到工厂的仓库后对工人下达命令：“来！加工他！”，而没有必要再啰嗦一次“用XXXXX方法，加上XXXXX工序，再加上XXXXX，来给我加工这个XXXXX。”

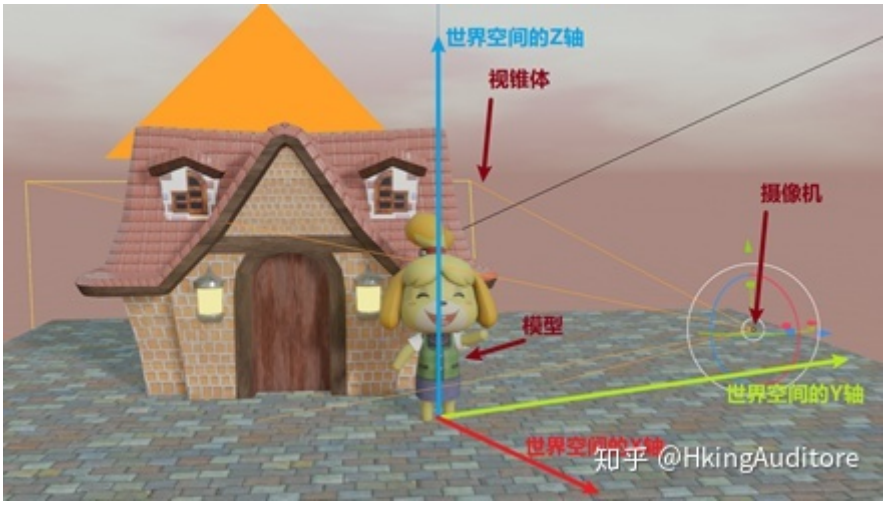


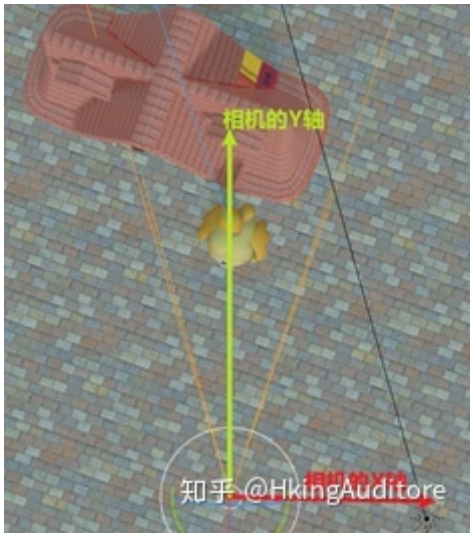
CPU向GPU发送的指令也是像流水线一样的——CPU往命令缓冲区中一个个放入命令，GPU则依次取出执行。在实际的渲染中，GPU的渲染速度往往超过了CPU提交命令的速度，这导致渲染中大部分时间都消耗在了CPU提交Draw Call上。有一种解决这种问题的方法是使用**批处理（Batching）**，即把要渲染的模型合并在一起提交给GPU。打个比方，工厂想要把100根钢筋中间截断，如果发货方采用的方法是一根一根钢筋送给工厂，那速度肯定是相当慢的；大部分情况都是发货方把这100根钢筋打包送给工厂，这样明显加快了效率。

顶点着色器（Vertex Shaders）

顶点着色器是GPU流水线的第一个阶段也是必需的阶段，这一块可以由开发者完全控制。值得一提的是，这顶点着色器中，我们无法创建或销毁任何一个顶点，也无法得到当前处理的这个顶点与其他顶点的关系。因为每次处理顶点都是独立的，不需要额外考虑其他，所以进行这一步速度会相当快。

在这里，GPU还需要进行**模型转化与相机转换（Model- & Camera transformation）**，在3D渲染中，我们必须要设置一个摄像机来接收图像，这个摄像机的视野决定了GPU最终会让我们看到什么样的画面，为了方便之后的运算，这里还需要根据视锥体（可以理解为摄像机能够看到的范围）将坐标空间由世界空间映射到摄像机的观察空间。



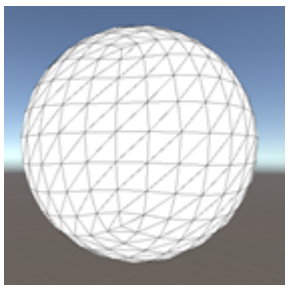


在世界空间下进行计算时，计算是相当不方便的；举个简单的例子，假如你看到了一个橙色的球，想告诉你不在场的朋友，于是发消息和朋友说：“诶！我在我的面前五米方向发现一个橙色的球！”，但可惜的是，你的朋友不知道你在哪，当然更不知道你的“面前五米”到底是什么，这个时候想向朋友介绍这个球的位置的最确切办法只有“东经XX度，北纬XX度”——听起来就相当复杂。但如果这个时候你能把你这个朋友拉到你现在站到你现在的位置，脸朝着你现在的朝向，那么你朋友就能理解你的“面前五米”了。而且显然，“面前五米”比“东经XX度，北纬XX度”要好算得多。

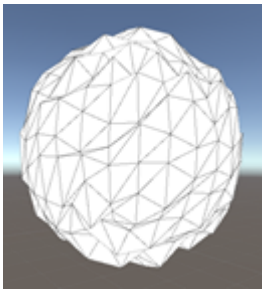
这个“把朋友拉到自己位置”的操作就是空间的转换。前面我们提到过，GPU的高效源于“流水线”里千千万万个头脑并不发达的“小学生”，为了加快“小学生”们的运算速度，GPU当然巴不得运算越简单越好；因此相比起在复杂的世界空间进行坐标[9]计算，把计算的空间换到摄像机的观察空间会大大地方便之后流水线的处理速度。尽管这个运算将牵扯到比较复杂的矩阵空间变换计算，但好在这是一劳永逸的。

此外，我们还可以进行的操作有：坐标变换（如动画）、逐顶点色彩处理（如光照、纹理采样）。

l 坐标变换



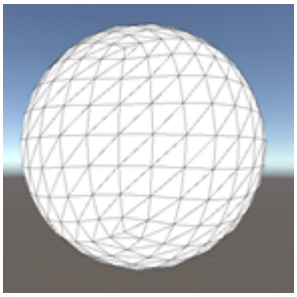
处理前



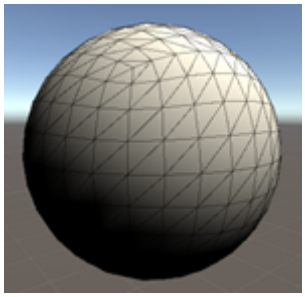
处理后

开发者可以编写程序在这个阶段修改顶点的坐标，诸如流动、摇曳等与顶点位置相关的动画操作都可以实现。上图通过坐标变换改变了原有球的形状。

| 逐顶点色彩信息处理



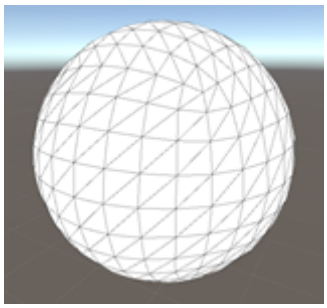
处理前



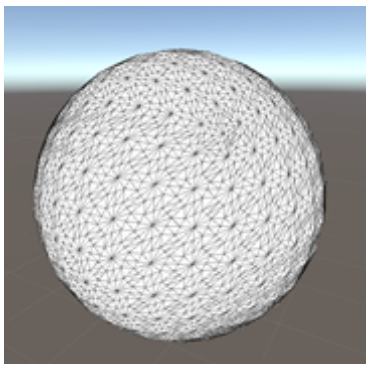
顶点着色器处理后再经过一系列处理显示出的颜色

开发者可以在这个阶段计算每个顶点的光照信息，计算光照、阴影等。图中小球即通过各顶点法向与光源坐标进行了简单的漫反射计算。当然除了计算光照，其他与顶点颜色相关的操作都可以在这个阶段里进行。值得一提的是，这里仅仅是“信息处理”，还不是真正的着色，可以理解为“为接下来的着色计算提供一些信息”。

曲面细分着色器（Tessellation Stage）

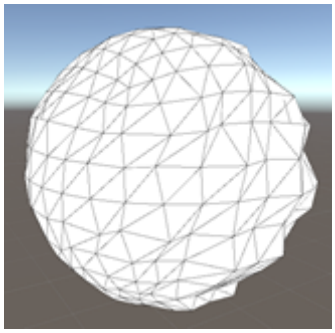


处理前

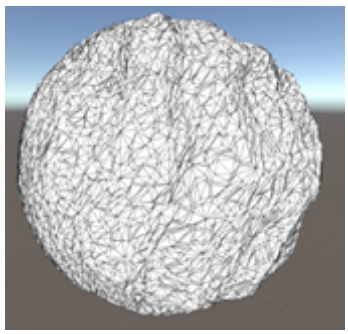


处理后

在这一阶段，程序员可以进行曲面细分操作，看起来就像在原有的图元内加入了更多的顶点。对于一些有大量曲面的模型，进行曲面细分可以让曲面更加圆润；如果为这些细分的顶点再准备一些位置信息，那么这些细分的顶点将有助于我们展现一个细节更加丰富的模型。这也是**贴图置换（Displacement Mapping）**的基本思路。

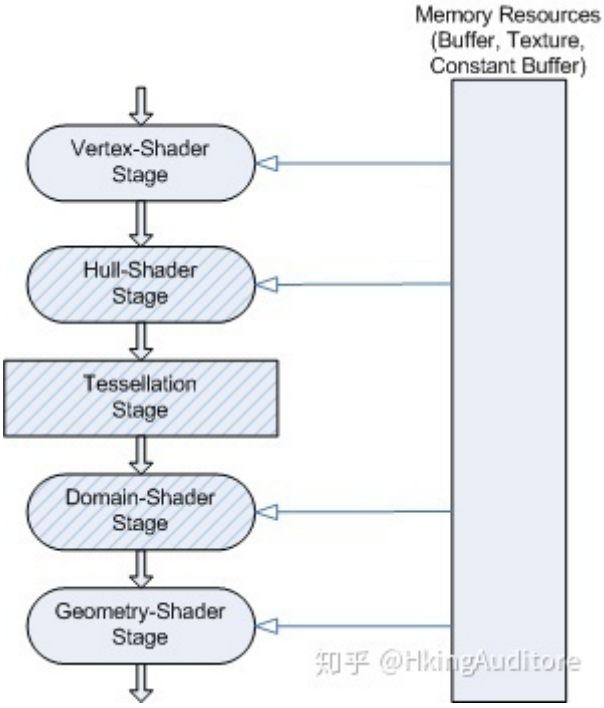


处理前



处理后

贴图置换



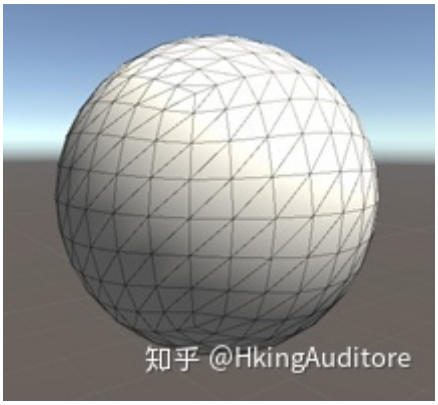
细分流程

在进入Tessellation Stage之前，流水线还将经过**Hull-Shader Stage**，这是一个可编程的阶段，开发者可以指挥GPU如何对顶点进行细分操作，但还不会真正进行细分，就像是指挥流水线上的工人说：“来，帮我把这给这根钢筋中间打上三个标记，好让后面的工人在上面安上旋钮。”

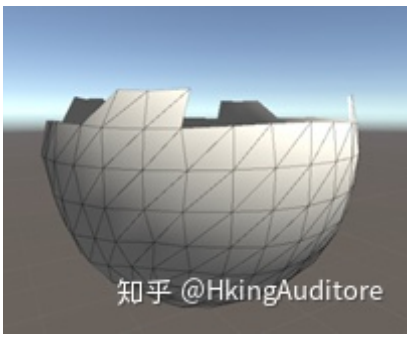
Tessellation Stage是真正的细分阶段；尽管开发者无法在这个阶段进行编程，但GPU将会根据Hull-Shader Stage中的标记进行细分；就像是流水线上的工人木讷地照着传过来的钢筋上的标记安装上旋钮。

在离开Tessellation Stage之后，流水线将进入**Domain-Shader Stage**，这是一个可编程的阶段，开发者可以指挥GPU对这些细分的顶点进行坐标计算；就像是指挥流水线上的工人如何调整上一个流程里工人安装上的旋钮，把钢筋摆成想要的形状。

几何着色器（Geometry Shader）



处理前

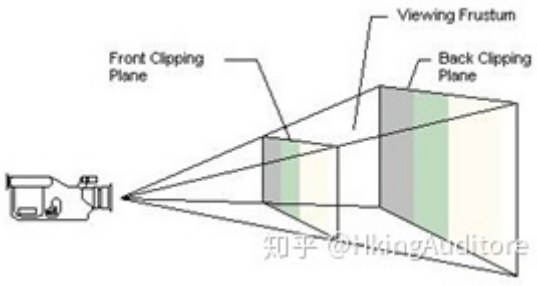


处理后

在这个阶段，开发者可以控制GPU对顶点进行增删改操作。几何着色器与顶点着色器都可以对顶点的坐标进行修改，但几何体着色器并行调用硬件困难，并行程度低，效率和顶点着色器有很大的差距；如果不是要做顶点增、删这些仅仅能用几何着色器实现的效果，那么还是用顶点着色器来完成吧。

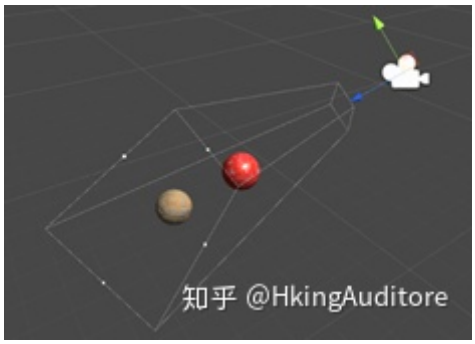
投影（Projection）

尽管至此GPU已经在三维空间中做了很多工作，但我们最终是要在一个二维的屏幕上查看我们渲染出来的图像——这就需要在GPU把三维空间映射到二维平面上了。不过值得注意的是，尽管这个过程叫做“投影”，但与数学上的投影还是有很大区别的，这个阶段中，GPU将顶点从摄像机观察空间转换到裁剪空间（又被称为齐次裁剪空间），为之后的剔除过程以及投射到二维平面做准备。

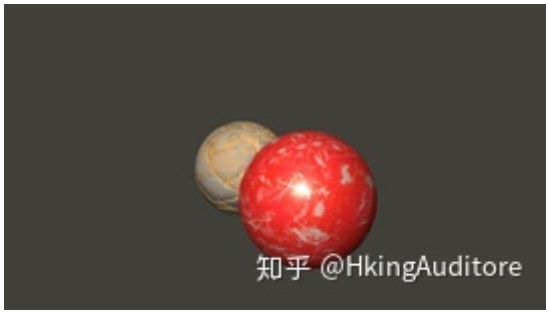


常见的投影方式有：透视投影与正交投影。他们在计算时计算时需要考虑**远裁剪平面（Far Clipping Plane）**和**近裁剪平面(Near Clipping Plane)**；透视投影需要额外考虑**视野(Field of**

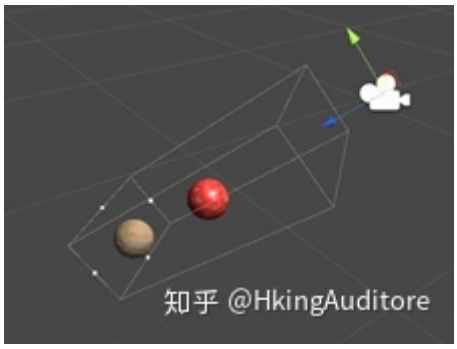
View)，即视锥体张开的角度，正交投影需要额外考虑**尺寸（Size）**，这个值用于衡量视锥体底的大小。



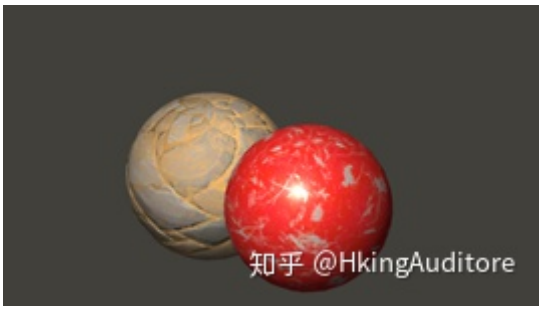
透视投影的视锥体



透视投影的效果



正交投影的视锥体



正交投影的效果

一遇到空间变换，我们就自然而然地想到矩阵变换[9]，感谢数学家给出的投影矩阵：

$$\mathbf{M}_{projection} = \begin{bmatrix} \frac{\cot\frac{FOV}{2}}{aspect} & 0 & 0 & 0 \\ 0 & \cot\frac{FOV}{2} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

知乎 @HongAudiotore

透视投影矩阵

$$\mathbf{M}_{projection} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

知乎 @HongAudiotore

正交投影矩阵

具体的推导过程在此不详细叙述。大多数人可能认为把顶点从观察空间转换到裁剪空间是一个三维到三维的矩阵变化，但观察公式我们发现这其实是一个四维到四维的矩阵变换；实际上，这里引入了齐次坐标系的概念，在三维中原有的三个分量x、y、z上又额外增加了w分量，使得可以通过矩阵乘的方式为三维坐标实现平移的效果。其次坐标的扩充的方法很简单，对于一个顶点（x,y,z），我们只需要给它增加一个w=1的分量即可，即（x,y,z,1）。

一个有趣的现象是，对于任意一个顶点，如果它乘上的是透视投影矩阵，它的w分量将不再是1，而如果它乘上的是正交投影矩阵的话，w分量仍然是1。发生这种现象的根本原因是透视空间并不是一个欧几里得空间——在透视空间中，平行线会在无穷远处相交，因此数学家们引入了一个额外的w分量来描述这种空间下的坐标，具体的过程此处不赘述。我们需要知道的是，在经过透视投影矩阵变化后，顶点中的w分量变成了一个衡量顶点到摄像机之间距离的参数。而正交投影矩阵的变化应该是最让人舒服的，它直接把空间变化为了一个x、y、z三个坐标都在[-1,1]区间内，w=1的立方体。

裁剪（Clipping）

在经过投影过程把顶点坐标转换到裁剪空间后，GPU就可以进行裁剪操作了。裁剪操作的目的就是把摄像机看不到的顶点剔除出去，使他们不被渲染到。判断顶点是否可以免受裁剪也十分简单，只需要满足

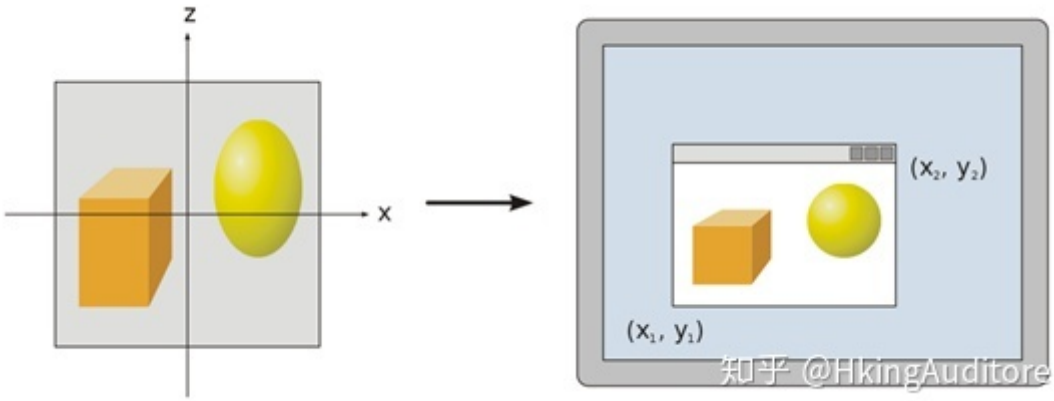
$$x, y, z \in [-w, w]$$

在把不需要的顶点裁剪掉后，GPU需要把顶点映射到屏幕空间，这是一个从三维空间转换到二维空间的操作，更符合大家对“投影”的理解。对透视裁剪空间来说，GPU需要对裁剪空间中的顶点执行齐次除法（其实就是将齐次坐标系中的w分量除x、y、z分量），得到顶点的**归一化的设备坐标（Normalized Device Coordinates, NDC）**，经过齐次除法后，透视裁剪空间会变成一个x、

y、z三个坐标都在[-1,1]区间内的立方体。对于正交裁剪空间就要简单得多，只需要把w分量去掉即可。

此时顶点的x、y坐标就已经很接近于它们在屏幕上所处的位置了，不过还有一个多出来的z分量，不过它也不会被白白丢弃，而是被写入了**深度缓冲（z-buffer）**中,可以做一些有关于顶点到摄像机距离的计算。

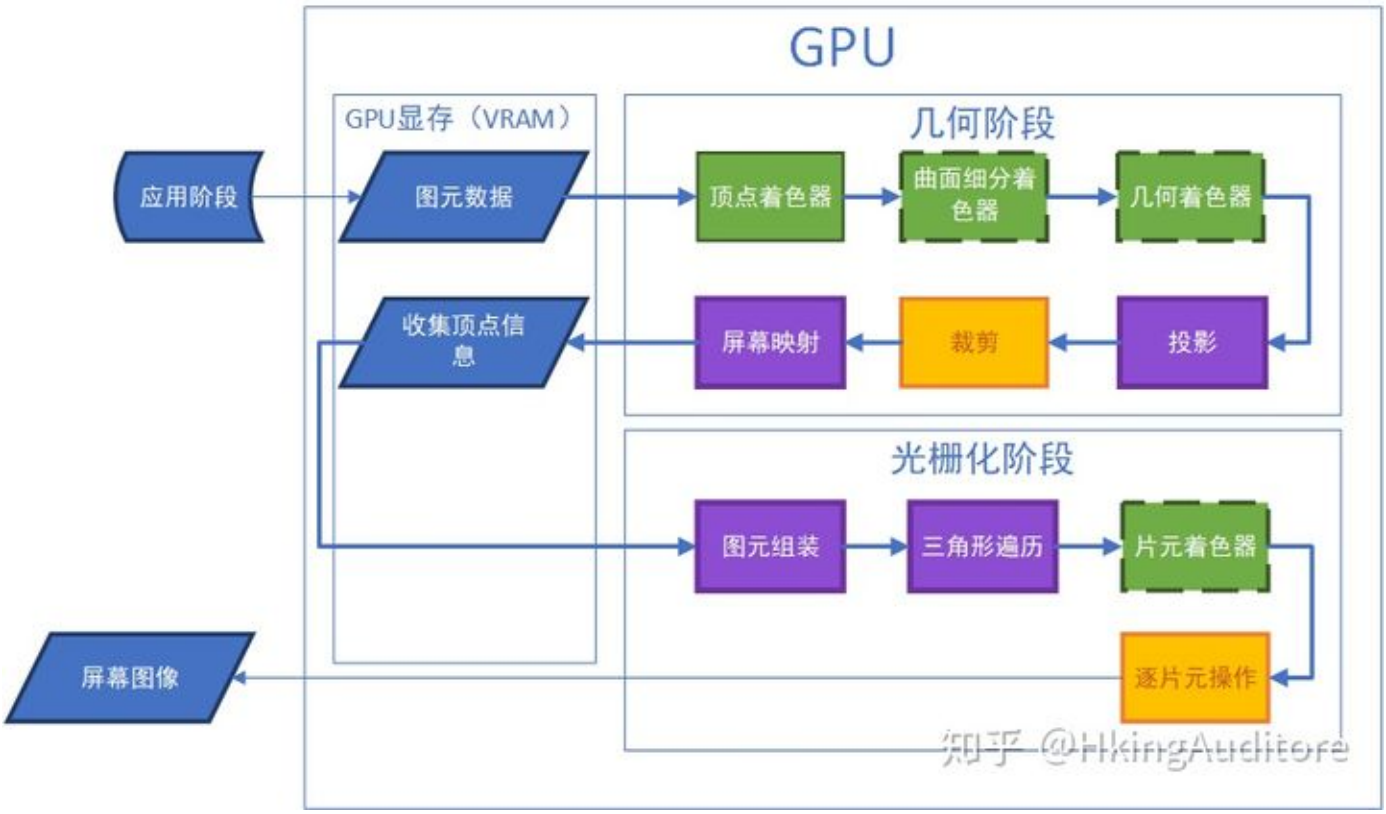
屏幕映射（Screen Mapping）



尽管GPU已经得到了顶点的x、y坐标，但他们处于[-1,1]区间中的，GPU还需要进行一定的计算才能把他们映射到我们的1920*1080甚至2560*1440的屏幕。得到的新坐标系称为窗口坐标系，虽然只需要两个坐标把顶点投射到屏幕上，但它仍然是三维的，这个多出来的z值就是在上面算出来的深度。

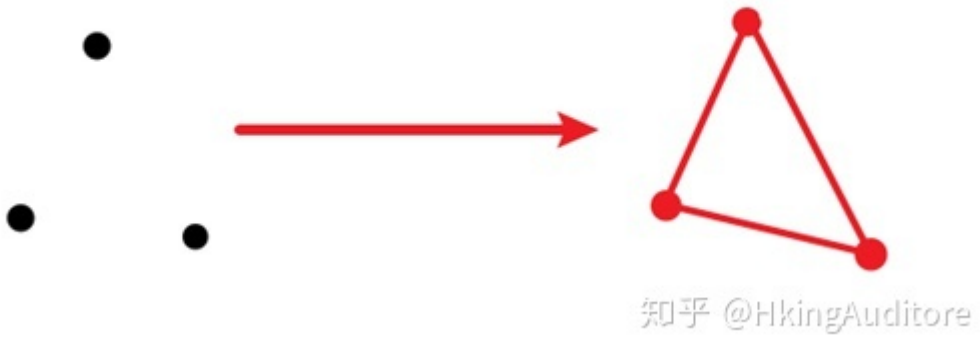
光栅化阶段（Rasterization Stage）

到此，GPU也只是完成了渲染的一半工作，因为现在我们只是得到了一些顶点，他们还不是能被显示在屏幕上的像素。

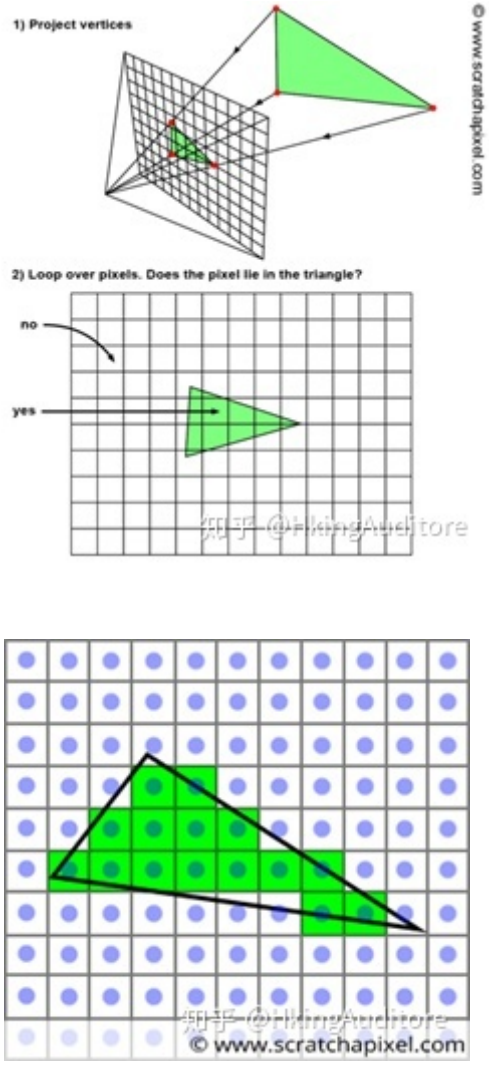


图元组装（Primitive Assembly）

有些资料把这个过程称为**三角形设置（Triangle Setup）**，不过个人认为叫做Primitive Assembly更为贴切。这个过程做的工作就是把顶点数据收集并组装为简单的基本体（线、点或三角形），通俗的说就是把相关的两个顶点“连连看”，有些能构成面，有些只是线，有些甚至没有与之配对的顶点只能当一个“单身狗”。

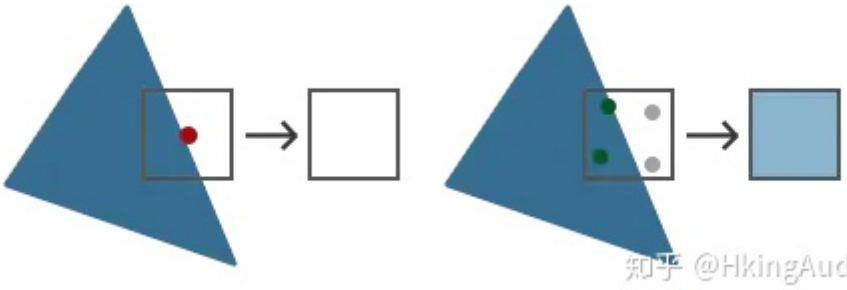


三角形遍历（Triangle Traversal）

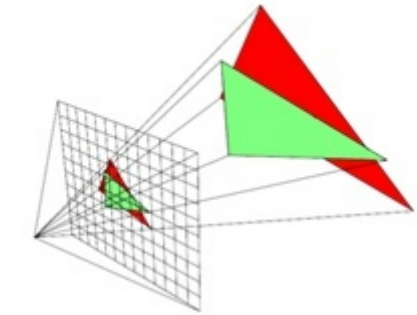


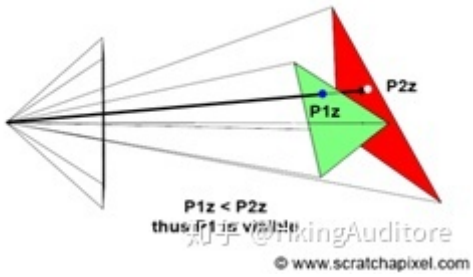
这个过程将检验屏幕上的某个像素是否被一个三角形网格所覆盖，被覆盖的区域将生成一个**片元（Fragment）**。当然，并不是所有的像素都会被一个三角形完整地覆盖，有相当多的情况都是一个像素块内只有一部分被三角形覆盖，对于这种情况，有三种解决方案：常用的有**Standard Rasterization**（中心点被覆盖即被划入片元）、**Outer-conservative Rasterization**（只要被覆盖了，哪怕只有一点也被划入片元）、**Inner-conservative Rasterization**（完全被覆盖才会被划入片元）。值得注意的是，片元不是真正意义上的像素，而是包含了很多种状态的集合（譬如屏幕坐标、深度、法线、纹理等），这些状态用于最终计算出每个像素的颜色。

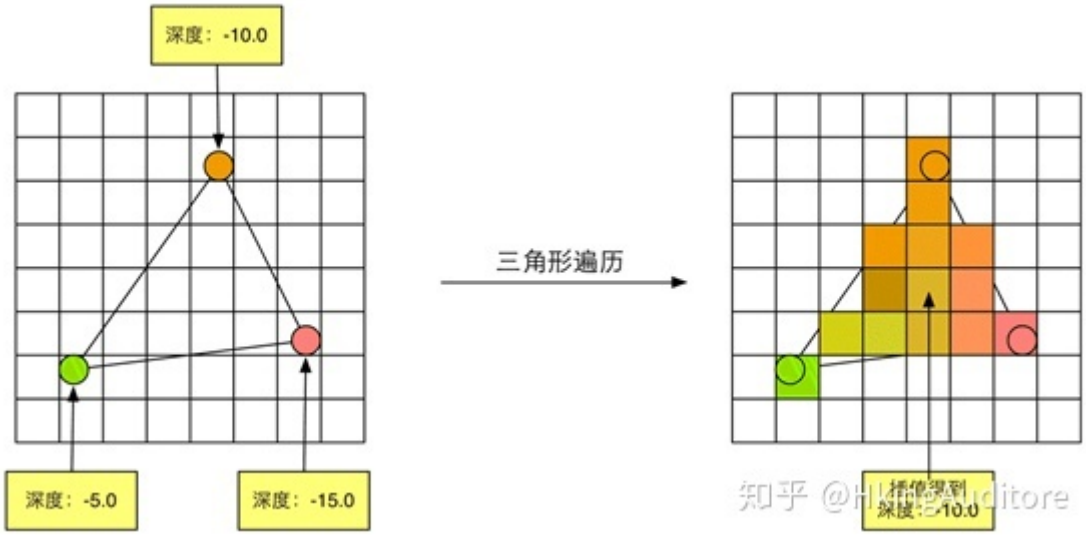
这一阶段牵扯到了**抗锯齿（Anti-aliasing）**操作。因为不管用什么划分片元的方法，三角形边缘部分总会显得很锐利。当然，程序员们也想出了各种各样的抗锯齿方法来解决这个问题，譬如**多重采样抗锯齿（MultiSampling Anti-Aliasing，MSAA）**，这种抗锯齿方法对中心点不在三角形内的边缘处采用不同程度的浓度进行计算。



除此以外，GPU还将对覆盖区域的每个像素的深度进行插值计算。因为对于屏幕上的一个像素来说，它可能有着多个三角形的重叠，所以这一步对于后面计算遮挡、半透明等效果有着重要的作用。





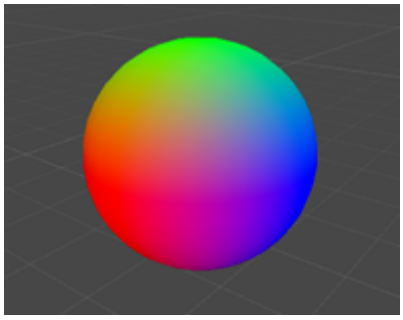


简单地说，这一步将告诉接下来的步骤，一个个三角形是怎么样覆盖每个像素的。

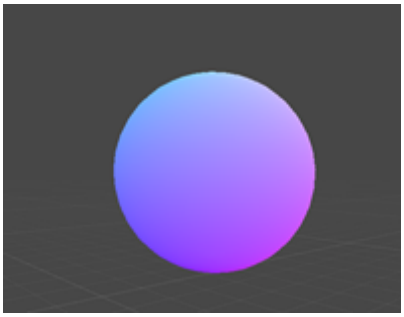
片元着色器（Fragment Shader）

这一阶段被一些资料称为**像素着色器（Pixel Shader）**，不过进行到这一步时片元还不是真正意义上的像素。这是十分重要的一步，它将为每个片元计算颜色，这意味着它们很快就能被我们在屏幕上看见了。

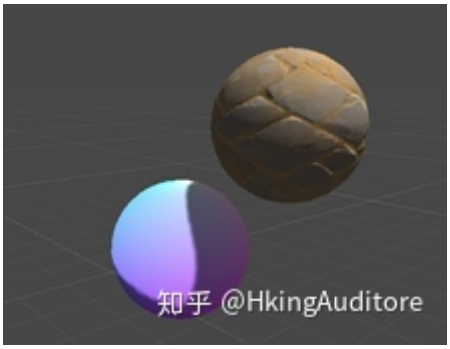
这个阶段是完全可编程的；在收到GPU为这个阶段输入了大量的数据后，程序员可以决定这些片元该着上什么样的颜色。



根据顶点法线计算颜色

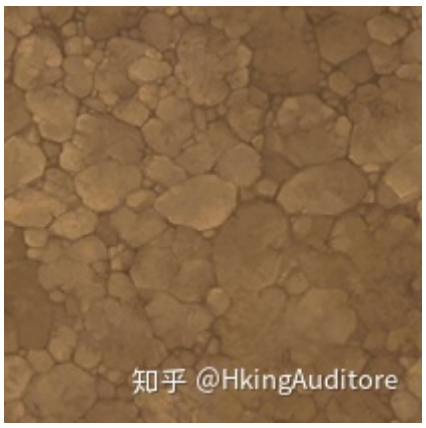


根据uv计算颜色

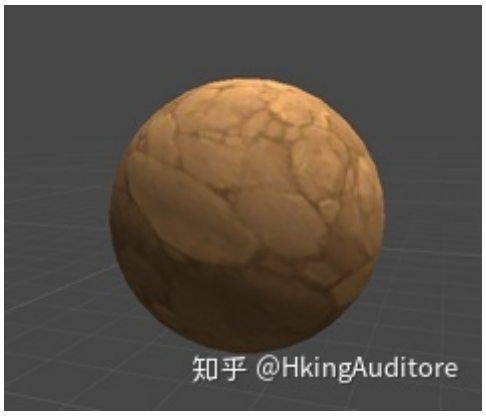


接受阴影

当然，除了自己计算色彩，使用纹理采样也是一种常见的做法：



使用的纹理



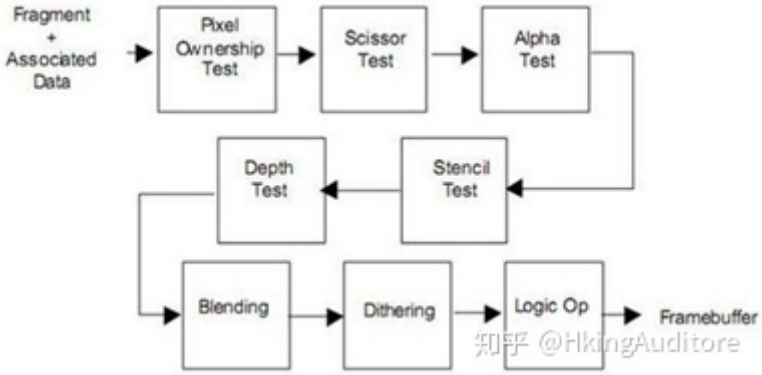
着色效果

此外，程序员还可以引入更多的信息计算颜色，包括法线贴图、高度图、糙度图等等。虽然片元着色器可以完成很多重要效果，但它仅可以影响单个片元。也就是说，当执行片元着色器时，它不可以将自己的任何结果直接发送给它附近的片元的。

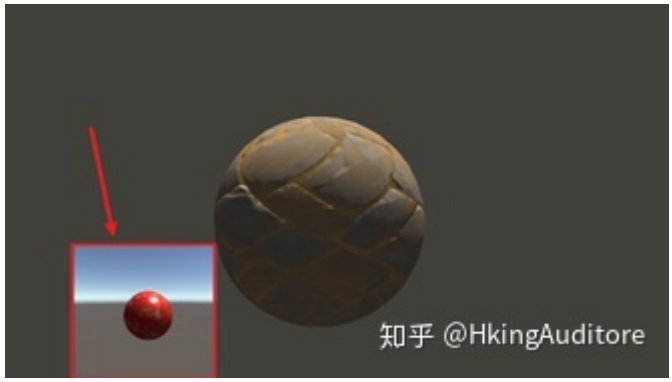
逐片元操作（Per-Fragment Operations）

在DirectX中，这一步又称作输出**合并阶段（Output-Merger）**。从两个名字中我们大致可以推测出GPU在这个阶段要做的事情：对每个片元进行操作，将它们的颜色以某种形式合并，得到最终在屏幕上像素显示的颜色。主要的工作有两个：对片元进行**测试（Test）**并进行**合并（Merge）**。

测试步骤决定了片元最终会不会被显示出来。在OpenGL中，主要的测试有：**裁剪测试（Scissor Test）**、**透明度测试（Alpha Test）**、**模板测试（Stencil Test）**以及**深度测试（Depth Test）**。这个阶段是高度可配置的。



裁剪测试（Scissor Test）

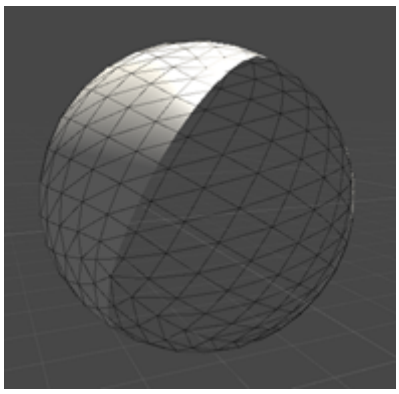


裁剪测试效果

在裁剪测试中，允许程序员开设一个裁剪框，只有在裁剪框内的片元才会被显示出来，在裁剪框外的片元皆被剔除。

透明度测试（Alpha Test）

使用的纹理

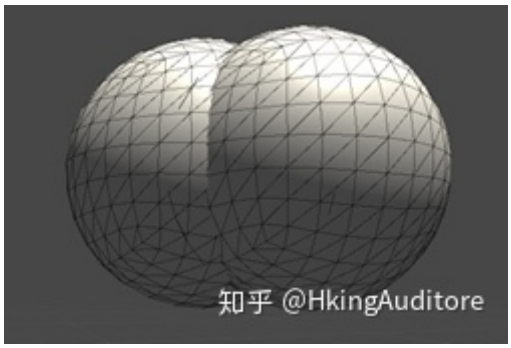


透明度测试效果

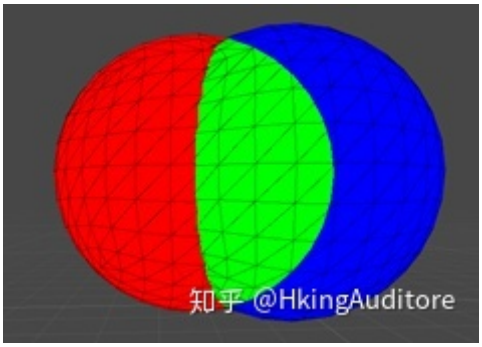
在透明度测试中，允许程序员对片元的透明度值进行检测，仅仅允许透明度值达到设置的阈值后才可以会绘制。在OpenGL3.1后这个API被删除了，但你可以在片元着色器中实现类似的效果。

模板测试（Stencil Test）

模板测试是一个相对复杂的测试。在模板测试中，GPU将读取片元的模板值与模板缓冲区[9]的模板值进行比较，如何比较可以由程序员决定，如果比较不通过，这个片元将被舍弃。



两小球的位置关系

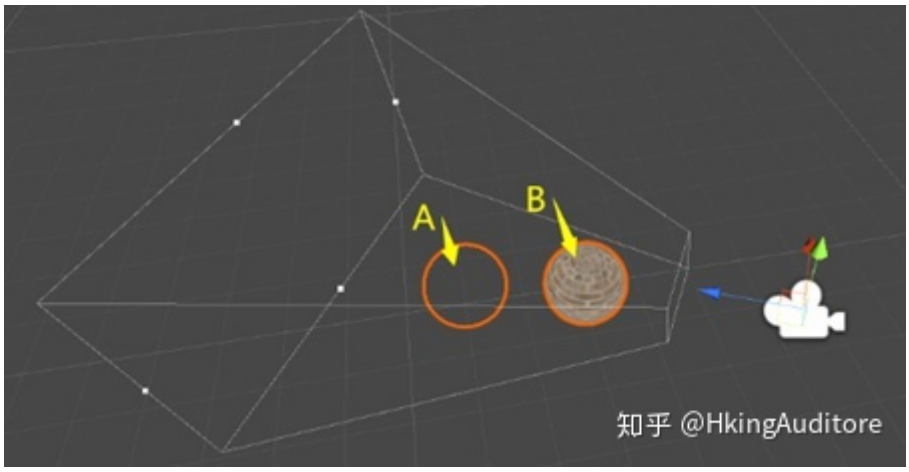


模板测试结果

譬如图中两个有重叠区域的小球；令左侧小球模板值为3，右侧小球在非重合处模板值为2，重合处模板值为3。现进行模板测试，令左侧小球始终被绘制为红色，令右侧小球满足：模板值与缓冲区相等的绘制为绿色，否则绘制为蓝色。于是我们发现，右侧小球重合处的片元通过了模板测试，被成功绘制为绿色。

深度测试（Depth Test）

深度测试是一个十分重要的测试。在深度测试中，GPU将读取片元的深度值（就是我们前面留下来的坐标z分量）与缓冲区的深度值进行比较，比较方式同样是可以配置的。用通俗的说法解释，深度测试允许程序员设置如何渲染物体之间的遮挡关系。



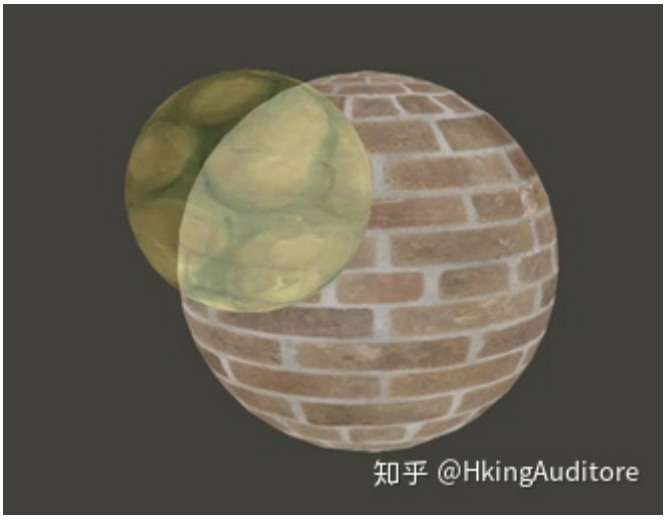
A、B两个小球与摄像机的关系

深度测试效果

如图，对于摄像机，尽管A小球在B小球的后方，但通过修改深度测试，我们让GPU把A没有被遮挡的部分隐藏了，反而让A被遮挡的部分显示出来的。

大量的被遮挡片元直到深度测试阶段才会被剔除，而在此之前它们同样地被计算，这占用了GPU大量的资源。因此有种优化技术是将**深度测试提前（Early-Z）**。但这带来了与透明度测试的冲突，例如某个片元甲虽然遮挡了另一个片元乙，但甲却是透明的，GPU应当渲染的是片元乙，这就产生了矛盾，这就是透明度测试会导致性能下降的原因。

如果一个片元通过了上面所有的测试，那它终于可以来到合并环节了。合并有两种主要的方式，一种是直接进行颜色的替换，另一种是根据不透明度进行**混合（Blend）**，而混合操作同样是可配置的，程序员可以设定是把这两种颜色进行相加、相减还是相乘等等，有点像在PS里的操作。



在A小球与B小球的遮挡部分进行柔和相加操作

在经过上面的层层测试后，片元颜色就会被送到颜色缓冲区。GPU会使用**双重缓冲（Double Buffering）**的策略，即屏幕上显示**前置缓冲（Front Buffer）**，而渲染好的颜色先被送入**后置缓冲（Back Buffer）**，再替换前置缓冲，以此避免在屏幕上显示正在光栅化的图元。

编辑于 2020-07-01 22:41

渲染器　GPU 通用计算　图形处理器（GPU）