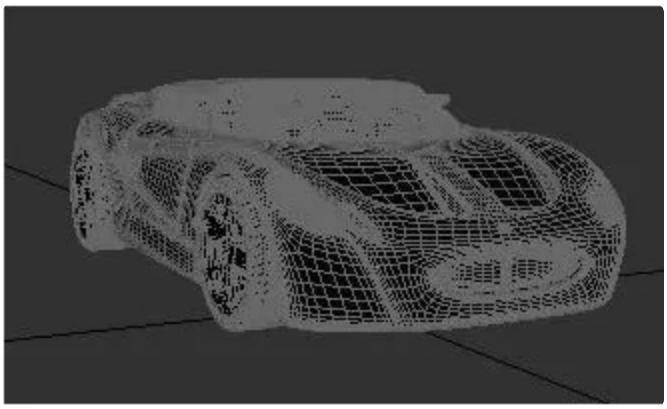
渲染管线概述

CPU && GPU

1. CPU处理并准备好需要显示内容交付给GPU;
2. GPU通过计算,转换将显存中的图形数据显示到屏幕像素中;
3. GPU需要处理屏幕上的每一个像素点，并保证这些像素点的更新是流畅的，这就对GPU的并行计算能力要求非常高。
4. 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。因此计算能力不如CPU;

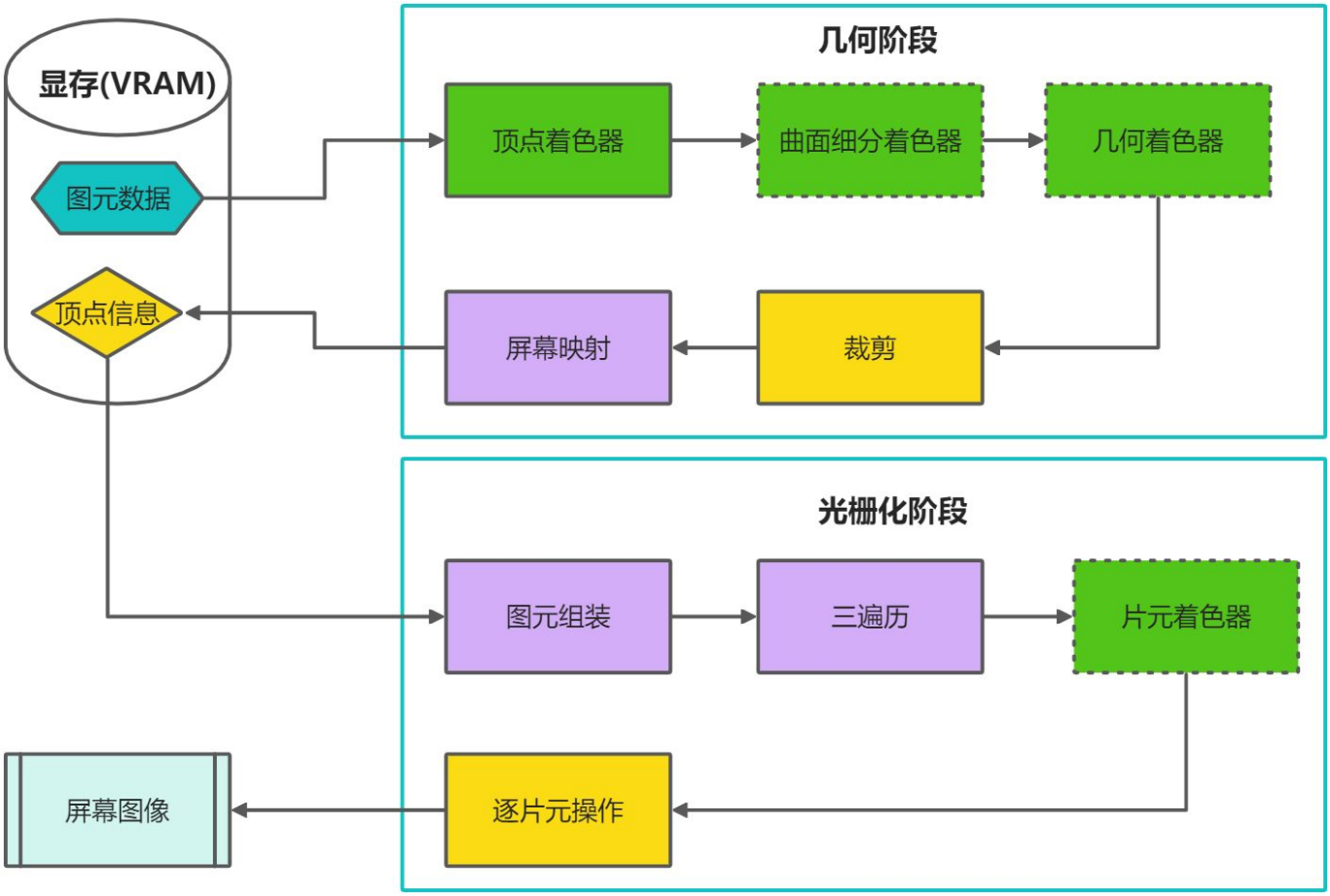
GPU面对3D游戏中成千上万的三角面，如果仅仅是逐一单个处理计算，效率低的惊人。



3D模型网格

渲染管线（Render Pipeline）

渲染管线又称渲染流水线; 在渲染流程中，CPU与GPU通力合作渲染图像。在运算过程中，CPU不断地将要处理的数据丢给GPU，GPU调动一个个计算单元对这些数据进行处理，最后组装出产品——图像。根据这个流程将渲染管线划分为以下三个阶段;



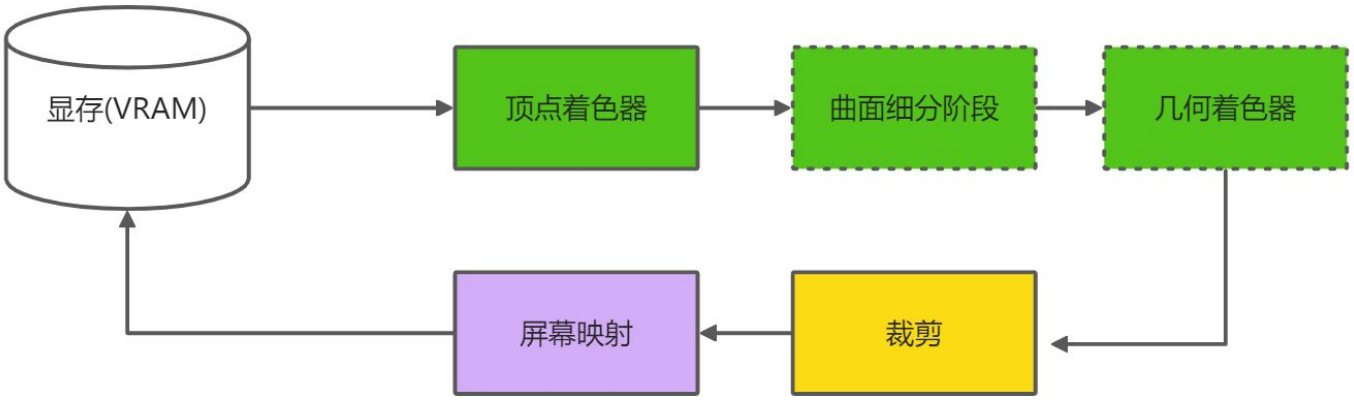
应用阶段（Application Stage）

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，

几何阶段（Geometry Stage）

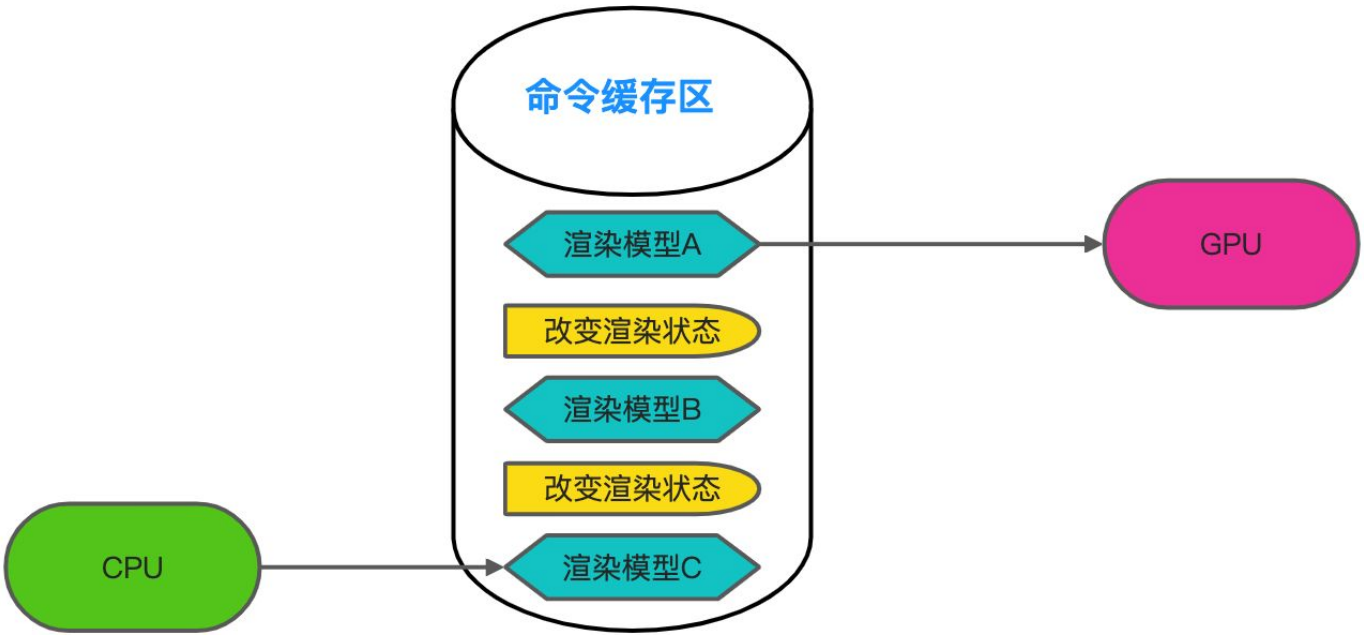
这一个由GPU主导的阶段，也就是说，从这个阶段开始，我们进入了上文所说的“流水线”。几何阶段将把CPU在应用阶段发来的数据进行进一步处理，而这个阶段又可以进一步细分为若干个流水线阶段，可以类比理解为工厂流水线上进行的一道道工序。



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），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

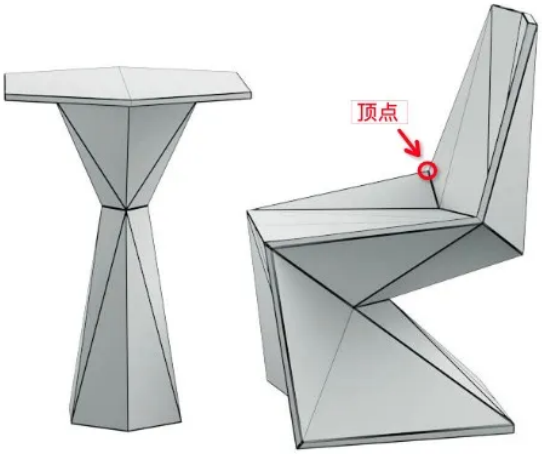
在应用阶段，CPU从硬盘中把需要的数据拿出来放进了内存中，经过之前所述的一系列操作后，再打包发给GPU进行进一步处理。虽然从渲染的角度来看，当CPU把数据传递到显存中后，这些数据在内存中的使命就结束了，可以移除了，但对于一些特殊数据仍然可以“幸存”；譬如游戏中有一面墙，它不仅需要被渲染出来，还需要进行计算物体碰撞，那么CPU在将它的网格丢给GPU后并不会马上把它从内存中移除，因为CPU还需要用这个网格计算碰撞。
在应用阶段，尽管在CPU已经把数据准备得十分充分，但在完成传送任务后，CPU也不能一走了之，它还需要向GPU下达一个渲染指令，这个指令就是**Draw Call**，由于之前我们已经把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！”；



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）

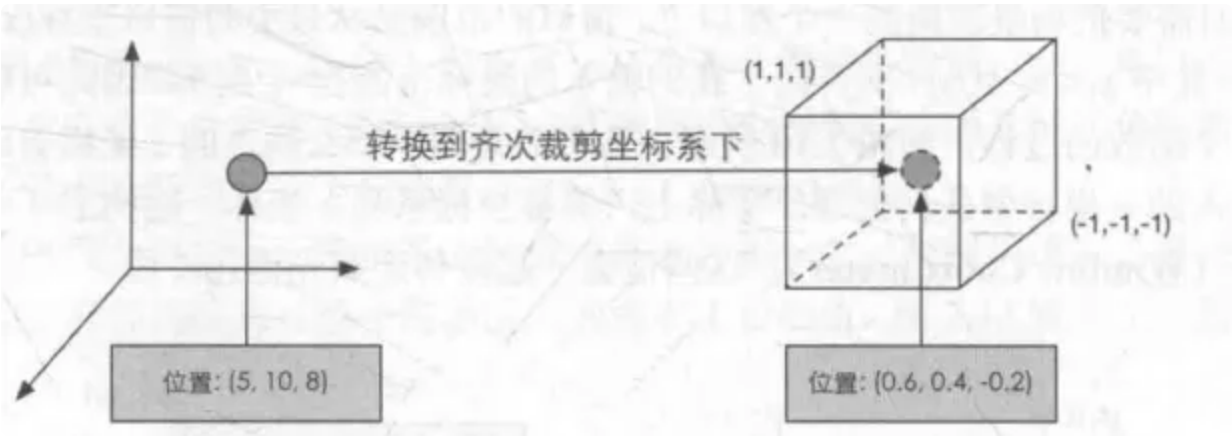
顶点着色器是GPU流水线的第一个阶段也是必需的阶段，这一块可以由开发者完全控制。顶点着色器的处理单位是顶点，也就是说，输入进来的每个顶点都会调用一次顶点着色器。顶点着色器本身不可以创建或者销毁任何顶点，而且无法得到顶点与顶点之间的关系。例如，我们无法得知两个顶点是否属于同一个三角网格。但正是因为这样的相互独立性，GPU可以利用本身的特性并行化处理每一个顶点，这意味着这一阶段的处理速度会很快。



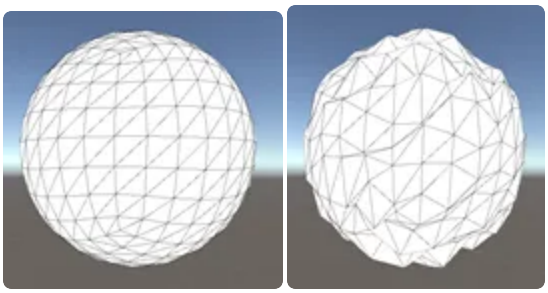
顶点着色器需要完成的工作主要有:坐标变换和逐顶点光照。

坐标变换。

顾名思义，就是对顶点的坐标进行某种变换。顶点着色器可以在这一步中改变顶点的位置，这在顶点动画中是非常有用的。例如，我们可以通过改变顶点位置来模拟水面、布料等。但需要注意的是，无论我们在顶点着色器中怎样改变顶点的位置,一个最基本的顶点着色器必须完成的一个工作是，把顶点坐标从**模型空间**转换到**齐次裁剪空间**。

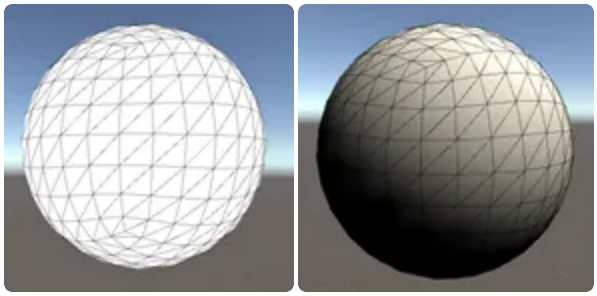


坐标变换



处理前　　处理后

逐顶点色彩信息处理



处理前　　处理后

图中小球通过各顶点法向与光源坐标进行了简单的漫反射计算

3D图形学定义了以下几种坐标系空间：

1. 模型空间：Model Space，也叫Local Space

直观的讲，这个坐标系一般以模型中心为原点，所有的模型在建模的时候给定的模型顶点坐标都以这个坐标系为基准，用户最开始所指定的顶点坐标也是位于该空间。

给出模型空间的好处在于方便建模，以及单个模型的重重利用。因为一个物体可被放置到场景的多个地方，这时每个物体的顶点坐标显然是不一样的，但可以共享同一个模型。

2. 世界空间：World Space

这个坐标系即3D场景给各个物体指定坐标的基准。场景有不同的物体具有不同的世界坐标。

3. 视角空间：View Space

这个坐标是以照相机为基准的，以照相机位置为原点，照相机朝向z轴正方向，右边为x轴正方向，上边为y轴正方向。之所以设置这个坐标系，主要是为了方便接下来的投影及裁剪操作。如果直接在世界空间下进行，由于照相机位置、朝向灵活多变，计算将会十分复杂。有了视角空间，会大大方便计算。

4. 投影、裁剪空间：Projection Clip Space

设立这个空间的目标是能够方便地对渲染图元进行裁剪：完全位于这块空间内部的图元将会被保留，完全位于这块空间外部的图元将会被剔除，而与这块空间边界相交的图元就会被裁剪。

I. 模型、世界空间变换

从模型空间到世界的变换主要包括：缩放、旋转和平移。缩放和旋转操作通过3X3矩阵及可实现，为了实现平移操作，则需要4X4型矩阵，因此所有的空间变换统一采用4X4矩阵，且顶点坐标也采用相应的[x,y,z,w]型。大多数情况下，w=1，[x,y,z]与顶点本身坐标保持一致。此外，多出的w在投影变换中发挥了至关重要的作用。

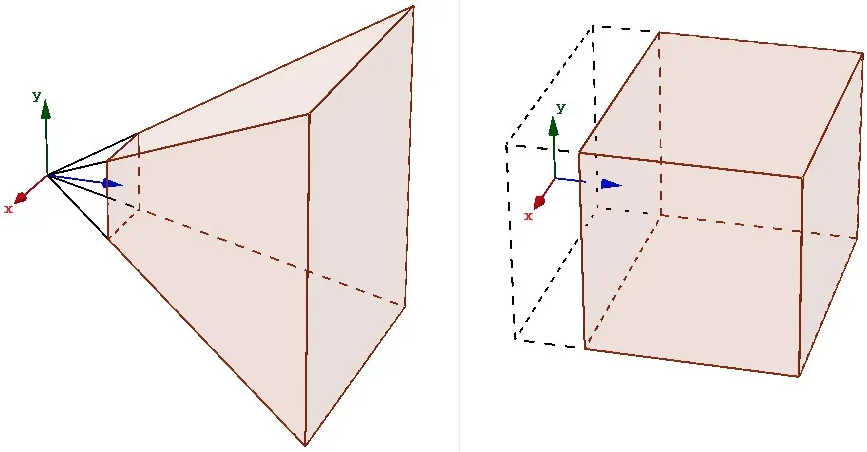
II. 世界、视角空间变换

从世界空间到视角空间通过相应的“视角矩阵”实现

III. 视角、齐次裁剪空间

从视角空间到**齐次裁剪空间**空间依靠“投影矩阵”来实现。投影有两种：正交投影和透明投影，大多数情况下，比如游戏中，用到的投影为透视投影，因为这种投影方式与人观察物体的方式是一样的。

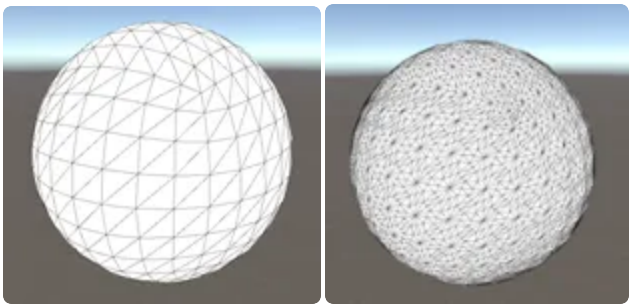
要计算投影矩阵，首先要确定照相机的几项基本参数：近、远平面（n,f），投影平面的宽、高比(r)，以及上、下视野角度大小(a)。近、远平面规定照相机能看到的最近和最远的距离。有了这些参数，所有能投影到屏幕上的点组成了如下所示的多面体：



投影变换的效果即把这个多面体转换成长方体，长、宽分别位于[–1,1]之间，z位于[0,1]之间。真正的裁剪操作就是在这个长方体中进行的，因此将大大简化裁剪的计算。

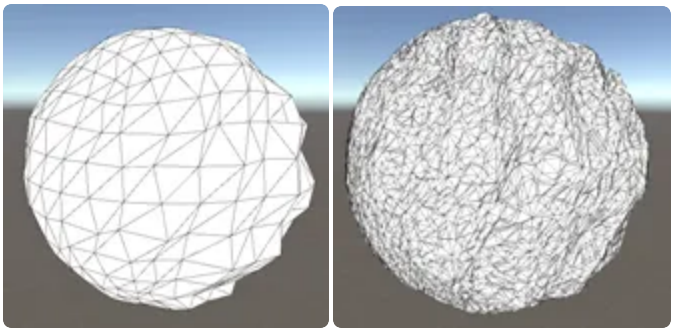
关于投影变换，要注意一点的是，很多人误以为投影即把三维顶点投影到二维平面 上，投影变换后顶点的z坐标即被抛弃，只剩下x,y坐标用于后面的屏幕变换。实际上，投影变换后z坐标并没有消失，位于[0,1]之间。屏幕坐标的变换不再使用z坐标，但z坐标在后面的Output Merger阶段用于深度比较时发挥的关键作用。

曲面细分阶段（Tessellation Stage）



处理前　　处理后

在这一阶段，开发者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。



处理前　　处理后

贴图置换

曲面细分阶段分为以下三个流程

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

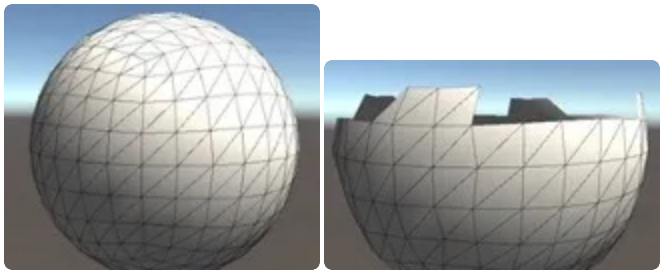
真正的细分阶段；尽管开发者无法在这个阶段进行编程，但GPU将会根据Hull-Shader Stage中的标记进行细　　分；就像是流水线上的工人木讷地照着传过来的钢筋上的标记安装上旋钮。

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。

曲面细分流程

几何着色器（Geometry Shader）



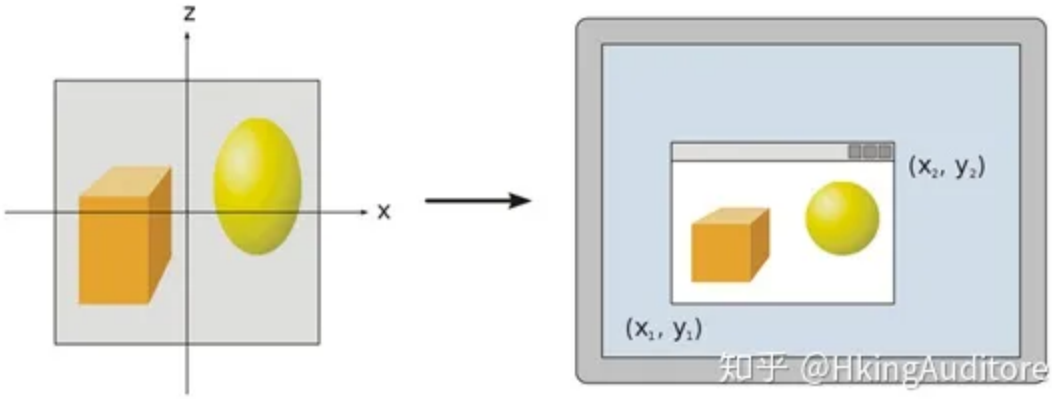
处理前　　处理后

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、

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。

屏幕映射（Screen Mapping）



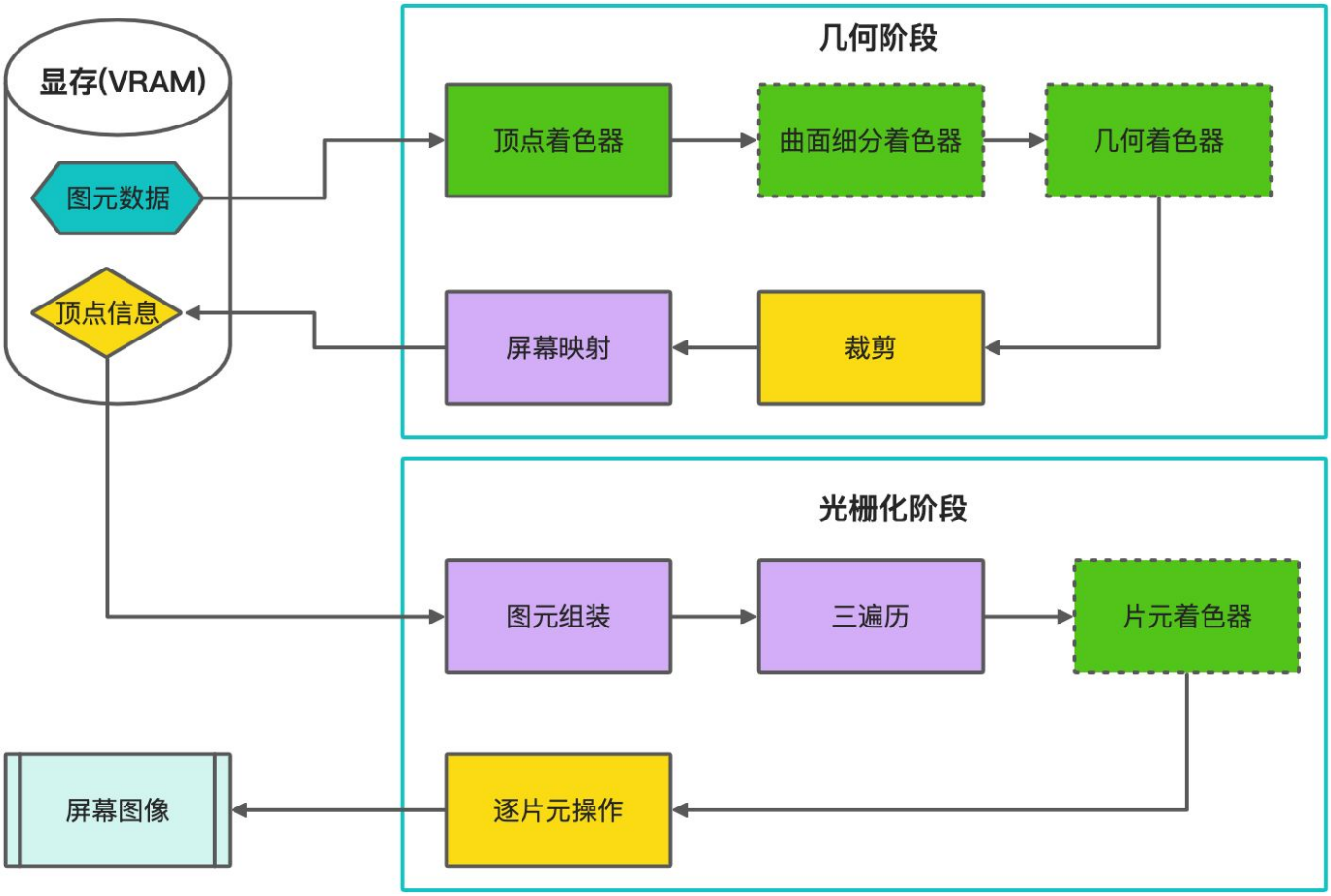
在把不需要的顶点裁剪掉后，GPU需要把顶点映射到屏幕空间，这是一个从三维空间转换到二维空间的操作

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。

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）

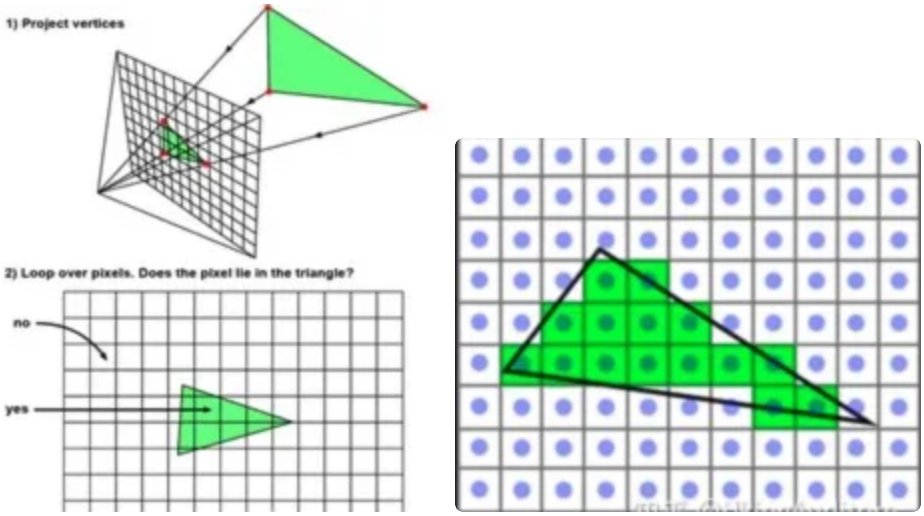
到此，GPU也只是完成了渲染的一半工作，因为现在我们只是得到了一些顶点，他们还不是能被显示在屏幕上的像素。



图元组装（Primitive Assembly）

又称为**三角形设置（Triangle Setup**。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：

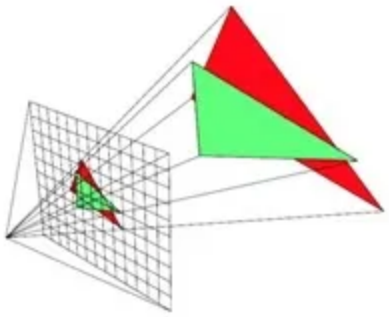
1: **Standard Rasterization**（中心点被覆盖即被划入片元）

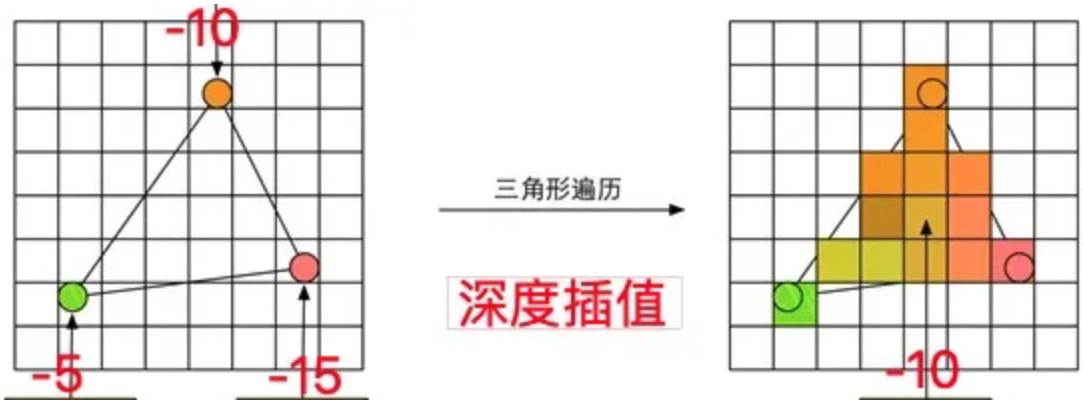
2: Outer-conservative Rasterization（只要被覆盖了，哪怕只有一点也被划入片元）

3: **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。

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）

又称为**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，开发者可以决定这些片元该着上什么样的颜色。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：

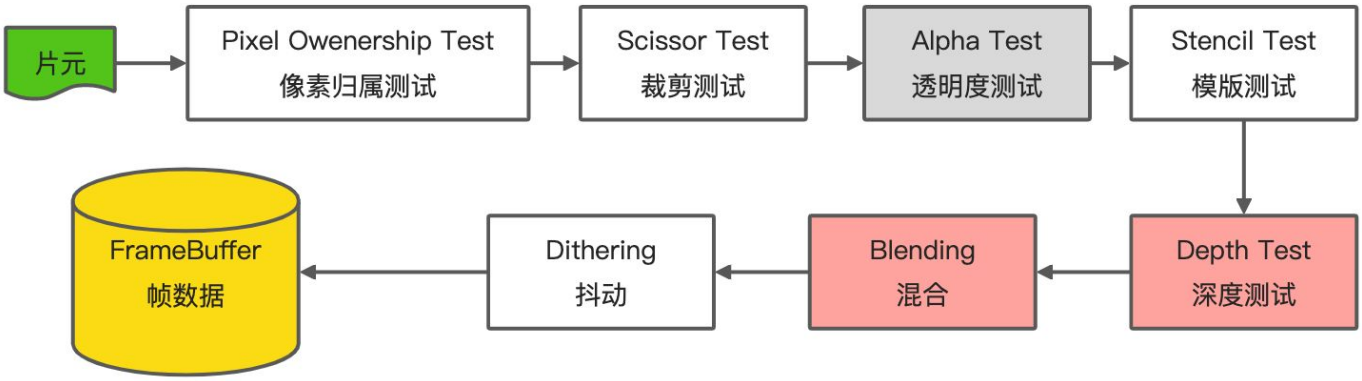
纹理采样着色效果

此外，开发者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）

从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。



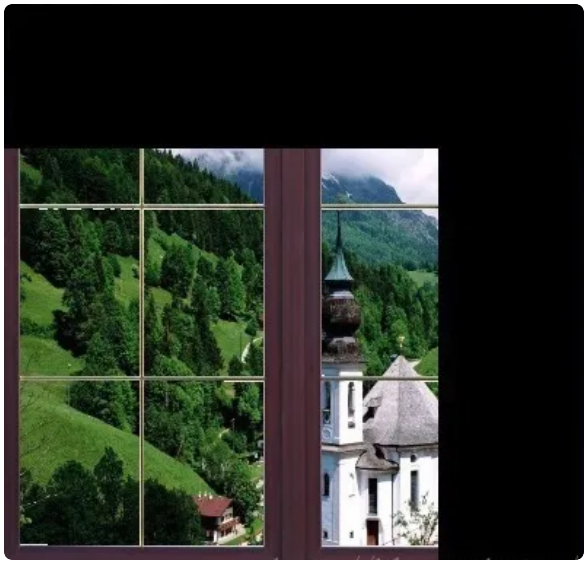
Pixel ownership test（像素归属测试）

判断像素在 Framebuffer 中的位置是不是为当前 OpenGL ES Context 所有，即测试某个像素是否属于当前　　的 Context 或是否被展示（是否被用户可见）；

Scissor Test（裁剪测试）:

在裁剪测试中，允许开发者开设一个裁剪框，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。

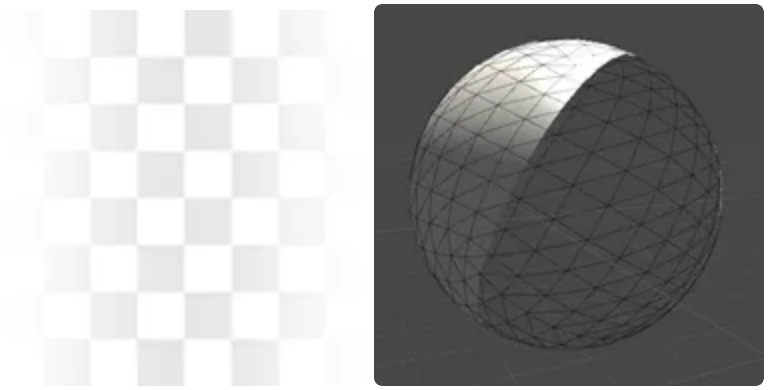
裁剪前



裁剪后

Alpha Test (透明度测试):

在透明度测试中，允许开发人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。否则丢弃掉该片元; 不过在OpenGL3.1后这个API被删除了，幸运的是你可以在片元着色器中实现类似的效果。



使用的纹理　　透明度测试效果

Stencil Test（模版测试）:

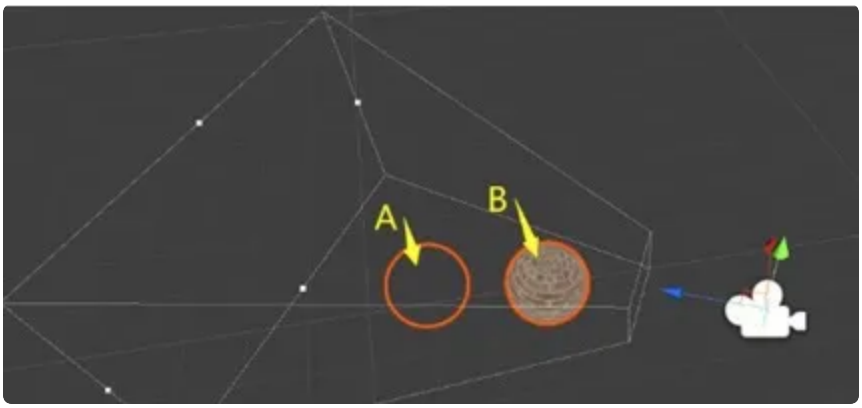
模板测试是一个相对复杂的测试。在模板测试中，GPU将读取片元的模板值与模板缓冲区的模板值进行比较，如何比较可以由开发者决定，如果比较不通过，这个片元将被舍弃。

box与plane的位置关系　　模板测试结果

譬如图中panel是在box上方,按照正常情况下是应该盖住了box; 我们让box模版值==0通过测试,通过测试后后把模版缓冲区的值+1; 也让plane的box的模版值==0就可以通过测试; 我们先让box渲染(设置box的渲染队列比plane前), 当box渲染完成后会把模版缓冲区的值+1,这就会导致box区域的plane无法测试通过;片元会被丢弃;最终实现了box的穿透效果

Depth Test（深度测试）:

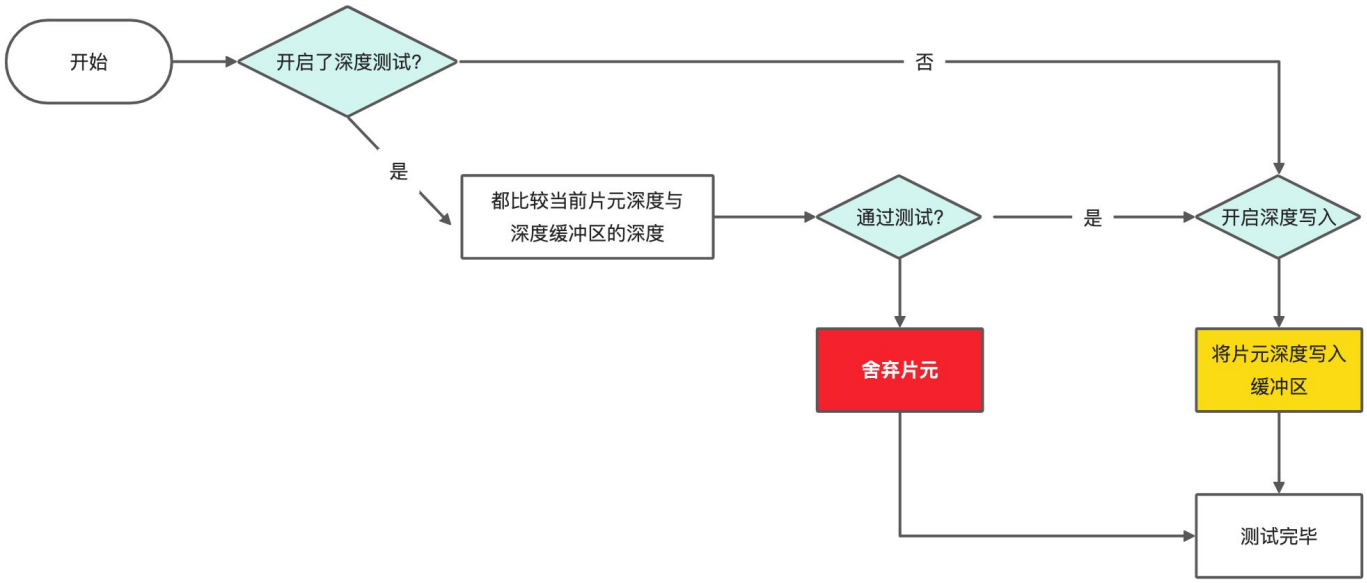
深度测试是一个十分重要的测试。在深度测试中，GPU将读取片元的深度值（就是我们前面留下来的坐标z分量）与缓冲区的深度值进行比较，比较方式同样是可以配置的。用通俗的说法解释，深度测试允许开发者设置如何渲染物体之间的遮挡关系。



A、B两个小球与摄像机的关系

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。

深度测试流程

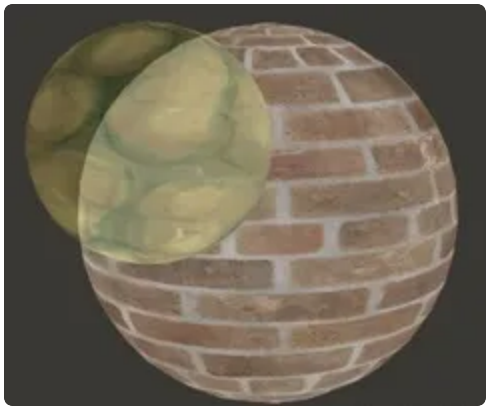


混合（Blend）

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，

1: 直接进行颜色的替换; 2:根据不透明度进行**混合（Blend）**

混合操作同样是可配置的，开发者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。