# 渲染流水线

#### 应用阶段：

1 准备好场景数据

2 粗粒度的剔除工作，剔除不可见的物体

3 设置好每个模型的渲染状态。包括使用的材质，使用的纹理，使用的shader等等。输出渲染所需的几何信息，即渲染图元。

#### 几何阶段：

1 该阶段最重要的任务是把顶点坐标变换到屏幕空间，再交给光栅器

2 该阶段输出屏幕空间的二维顶点坐标、每个顶点对应的深度值，着色等信息。

#### 光栅化阶段：

1 该阶段用传入的数据产生屏幕上的像素，并渲染出最终的图像。

2 需要对上个阶段得到的逐顶点数据进行插值，然后进行像素处理。

# 应用阶段:

1 把数据加载到显存中。

A 数据充HDD中加载到系统内存RAM，再将网格，纹理等数据加载到显存中VRAM。

2 设置渲染状态, 这些状态定义了场景中的网格是怎样被渲染。

3 调用DrawCall。告诉GPU，可以按照渲染状态来渲染了。

这个命令会指向一个需要被渲染的图元列表。

# GPU流水线

# 几何阶段：

顶点着色器：完全可编程。用于实现顶点的空间变换，顶点着色等功能。

曲面细分着色器：可选着色器，用于细分图元

几何着色器：可选着色器，用于执行逐个图元的着色操作，或者被用于产生更多的图元

裁剪：可配置阶段。不在摄像机内的顶点裁减掉，并剔除某些三角图元的面片。可以用自定义的裁剪平面来配置裁剪区域，可以指定裁剪三角图元的正面还是背面。

屏幕映射：不可配置，不可编程。把图元的坐标转换到屏幕坐标系中。

光栅化阶段：

三角形设置和三角形遍历 固定函数。

片元着色器，完全可编程，用于逐个片元着色。

逐片元操作：不可编程，可配置。执行 修改颜色，深度缓冲，进行混合等重要操作。

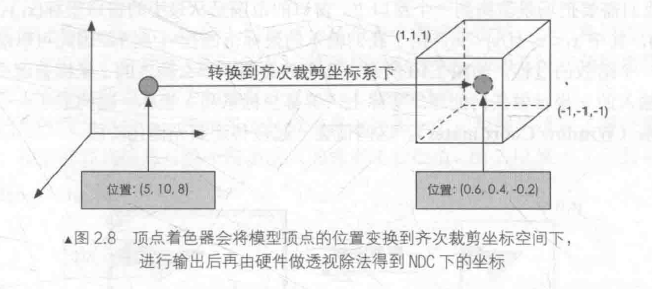
#### 顶点着色器

1 处理单位是 顶点。每个顶点都会调用一次顶点着色器。处理的顶点之间相互独立。GPU可以并行化处理他们。

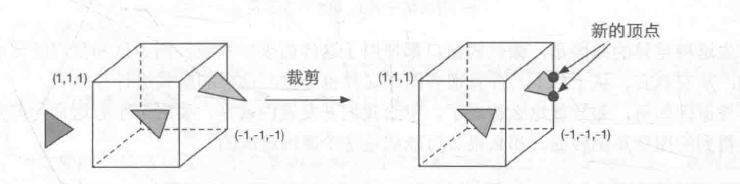
2 主要工作是：坐标变换和逐个顶点光照。还可以输出后续阶段所需的数据。坐标变换的重要任务是 把顶点坐标从模型空间转换到齐次裁剪空间。如以下代码：

o.pos = mul(UNITY\_MVP, v.position)

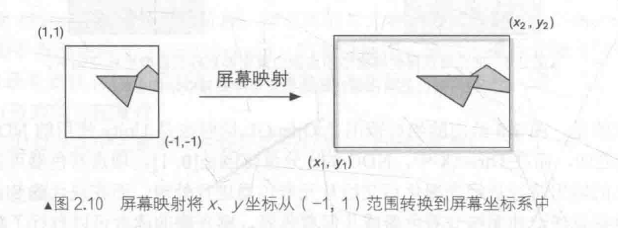
将顶点坐标 转换为 齐次裁剪坐标系下，接着由硬件做透视除法后，得到归一化的设备坐标(NDC)。OpenGL和Unity的NDC的z范围是 -1到1，Dx是0到1。最常见的输出路径是 经光栅化后 交给 片元着色器。



#### 裁剪

需要裁剪的片元是部分在视野中，部分不在视野中的片元。例如，一个线，A端在视野内，B端不在，则在裁剪的位置生成C端。

#### 屏幕映射

输入的数据是 处于单位立方体内的三维坐标。屏幕映射的作用是把每个图元的xy坐标转换到屏幕坐标系下。从(-1, 1) 映射到(x1,x2)。Z坐标不处理, 与窗口构成了窗口坐标系。



# 光栅化阶段

从上个阶段输出的信息是屏幕坐标系下的顶点位置和其他相关信息，如深度值，法线方向，视角方向。

光栅化的两个重要目标：1 计算每个图元覆盖了哪些像素；2 为这些像素计算他们的颜色

#### 三角形设置

目前为止输入的数据还都是三角形顶点。这个阶段会计算光栅化一个三角网格所需的信息，获取每条边上的像素坐标。

#### 三角形遍历

该阶段检查每个像素是否被一个三角网格所覆盖，如果覆盖则会生成一个片元。该片元包括了屏幕坐标，深度信息，顶点信息等等。

#### 片元着色器

输入是上一个阶段对顶点信息插值得到的结果，具体来说是从 顶点着色器 输出的数据插值得到的。

输出是一个或者多个颜色值。

进行了纹理采样，顶点着色器 输出每个顶点对应的纹理坐标，然后经过光栅化进行线性插值获得每个像素的纹理坐标。

局限在于 片元单独着色，不会告诉邻居信息。特例是片元着色器可以访问到导数信息。

#### 逐片元操作

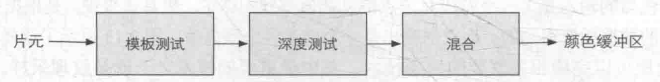
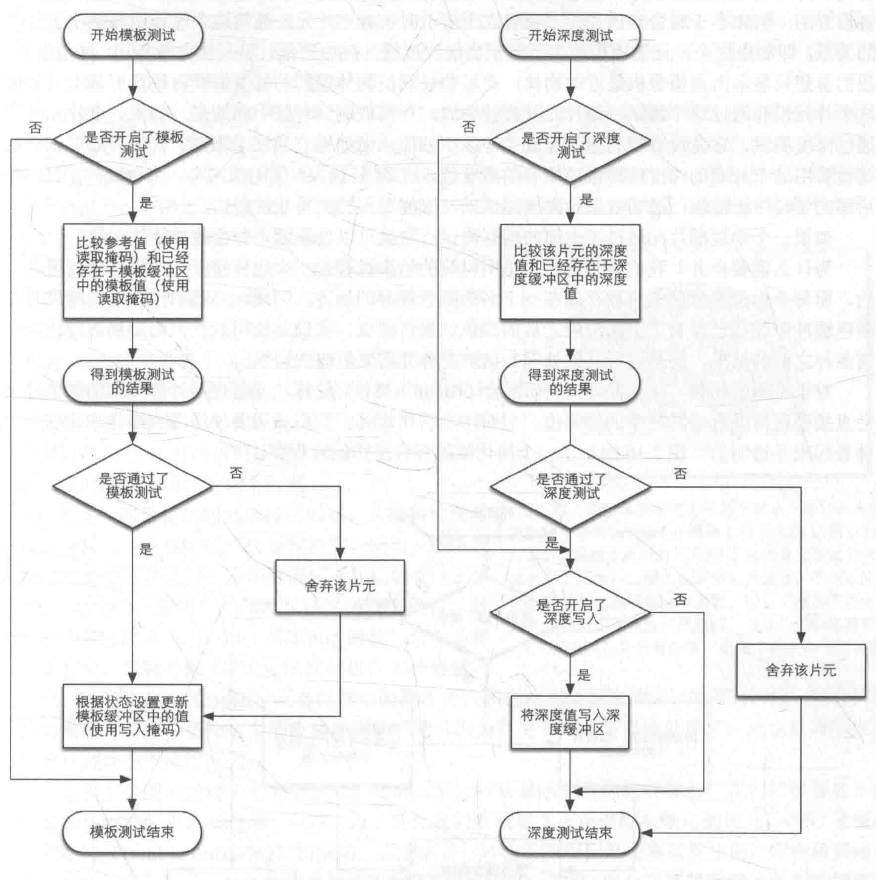
Dx叫做 输出合并阶段。

主要任务：

1 决定每个片元的可见性。涉及到了测试工作：深度测试、模板测试等。

2 如果一个片元通过了所有测试，就把这个片元的颜色值和已存储在颜色缓冲区中的颜色进行混合。

高度可配置。



#### 模板测试

1用到的是 模板缓冲，他与 颜色缓冲，深度缓冲 一样。

2过程：开启模板测试后，GPU会首先读取 模板缓冲区 中该片元的位置，然后将该值与读取到的 参考值 进行比较，比较函数可以由开发者指定的。

3不管一个片元有没有通过模板测试，我们都可以根据 模板测试的结果来修改模板缓冲区(同深度缓冲)。例如，失败时 模板缓冲不变，通过时 模板缓冲区的值+1等。

4作用： 模板测试通常用于限制渲染的区域，高级用法是 渲染阴影，轮廓渲染等。

#### 深度测试

与 模板测试 不同的地方是第3条：

如果一个片元没有通过深度测试，它就没有权利更改深度缓冲区中的值。

如果通过了深度测试，开发者可以指定是否要用这个片元的深度值覆盖掉原来的深度值，这是通过 开启/关闭深度写入来做到的。

作用：透明效果和深度测试，深度写入 关系密切。

#### 合并

问题 颜色缓冲区中的颜色从哪里来的？

解答 渲染过程 是 一个物体接着一个物体画到屏幕上的。例如先画A，再画B，这样在画B的时候，颜色缓冲区中的某些位置就有A的数据。

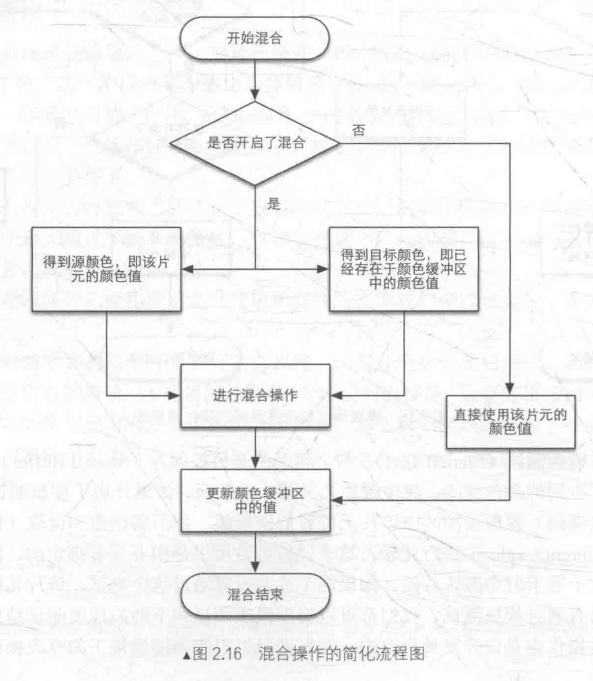
对于不透明的物体，可以关闭混合，这样片元着色器计算得到的颜色值 就会直接覆盖 颜色缓冲区中的值。半透明的物体，我们就需要使用混合操作，这个操作可以是 相加，相减，相乘等。

##虽然逻辑上这些测试是在 片元着色器 之后进行的， 但是大多是GPU会在 片元着色器 之前就进行这些测试。Early-Z技术：将深度测试提前执行的技术。

提前测试的问题：如果将测试提前，则会与片元着色器产生冲突。例如：A是玻璃，B是实物。如果提前检测，A把B完全遮挡。所以只能在片元着色器中进行融合，之后再进行测试。因此，透明度此时会导致性能下架

#### 双重缓冲

经过以上步骤，颜色缓冲中的数据即是显示的颜色，GPU使用双重缓冲的策略切换 后置缓冲和前置缓冲中的数据。



# GLSL，HLSL，CG

#### GLSL

优点在于跨平台性，可以在Windows，Linux，Mac甚至移动平台上使用。跨平台性是由于OpenGL没有提供着色器编译器，而是由显卡驱动来完成着色器的编译工作。即 只要显卡驱动支持对GLSL的编译 即可运行。对GLSL的编译结果 取决于硬件供应商，例如NVIDIA，ATI等。所以同一份代码编译结果可能会不一致。

#### HLSL

由微软控制着着色器的编译，无论硬件如何，同一个着色器的编译的结果是一样的。支持HLSL的平台有限，例如Windows，Xbox360, PS3等。

#### CG

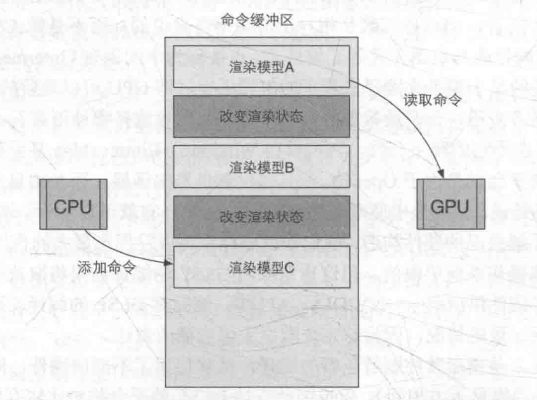
会根据平台的不同，编译成相应的中间语言。CG语言是与微软合作的，因此CG可以无缝的移植成HLSL代码，但可能无法完全发挥出OpenGL的最新特性。

# DrawCall

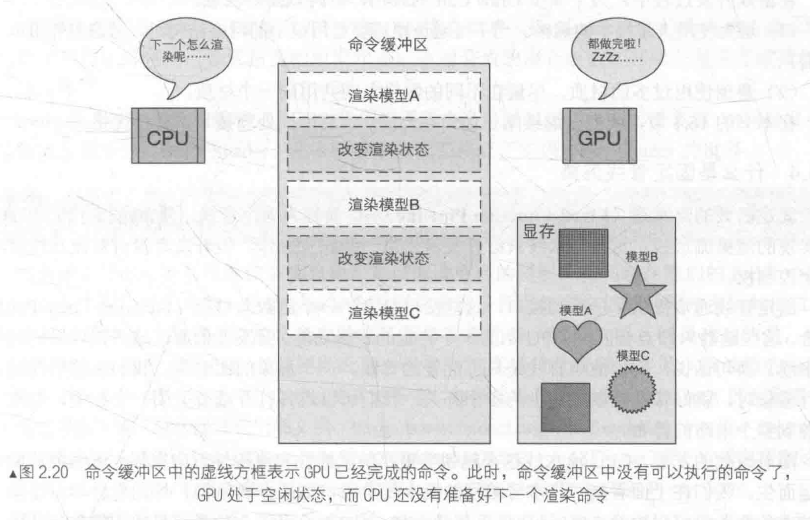
1 DrawCall 含义是 CPU调用图像编程接口， 以命令GPU进行渲染的操作。

2 DC 造成性能问题的元凶是CPU，而不是GPU。

3 CPU和GPU是如何进行流水线任务的？

 利用 命令缓冲区，CPU向其中添加命令，GPU从中读取命令，添加和读取的过程是独立的。命令缓冲区中的命令，DC是其中一种，还有 改变渲染状态的命令(例如 改变使用的着色器，使用不同纹理等)

4 DC多了为什么影响帧率？

 每次调用DC前，CPU需要向GPU发送很多内容，包括数据，状态和命令。在提交DC前，CPU还有很多工作，例如检查渲染状态等。GPU渲染200个或2000个三角形没有什么区别。DC多了，CPU会把大量时间花费在提交DC上，造成CPU的过载。

5 如何减少DC？

进行批处理，Batching。

注意：CPU在内存中合并网格，而合并的过程也是需要消耗时间的。因此，批处理更适合对静态的物体，只需要合并一次即可。当然，可以对动态的物体批处理，但需要每帧都重新进行合并然后发送给GPU，耗时耗内存。

所以：1 避免使用大量很小的网格；2 避免使用过多的材质。

# 固定管线渲染

固定函数的流水线，简称 固定管线。这种流水线只给开发者提供了一些配置操作。固定管线提供了一系列接口，这些接口包含一个函数入口点，匹配GPU上的一个特定逻辑功能。

例如，在电路上有许多开关，我们可以选择打开或者关闭其中的开关。