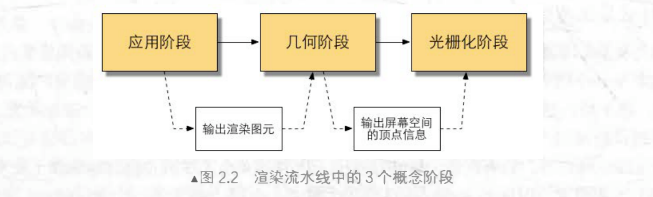
# Shader入门精要笔记

首要点：保持一定的**耐心**，与C++/C#等语言相比，成功编写了代码，但是为什么要这么写它们是如何执行的仍然并不了解。想要了解Shader就得了解渲染流水线的工作方式，所以保持耐心，打好基础这很重要。

### 2.1渲染流水线

目的：由一个三维场景出发，然后输出一个二维图像。换句话说，就是计算机通过一系列的顶点和纹理信息最终将这些信息转化为人眼可见的图像。该工作通常有CUP和GPU共同完成。

概念：一个渲染流程在**概念**上可以划分为三个阶段：应用阶段，几何阶段，光栅化阶段。每个阶段通常也是一个流水线系统。



·应用阶段

该阶段通常有CPU主导：

①场景信息，如摄像机位置以及视椎体，场景中包含的模型和光源。

②做一个粗粒度剔除，将不可见的物体剔除提高渲染性能。

③设置模型的渲染状态，包括但不限于使用的shader，纹理和材质。

④传递渲染需要的几何信息点，线，面等。

PS：渲染一个物体的信息被称为 渲染图元（Rendering Primitive）

·几何阶段

处理所有和我们要绘制的几何相关的事情，如在哪绘制，如何绘制。通常在GPU上进行。具有以下几个主要任务

①将渲染图元的顶点坐标转化到屏幕空间中

①每个顶点的深度值和着色相关的信息。

·光栅化

使用几何阶段传递的信息生成屏幕上的像素并显示。在GPU上运行。

①决定每个渲染图元中的哪些像素应该被绘制在屏幕中。

### 2.2 CPU和GPU之间的通信

渲染起点为CPU，即应用阶段，通信可大致分为3个阶段

①将数据加载到显存中。

渲染所需的数据从硬盘->RAM->显存 这是因为显卡对于显存中的数据访问速度更快，且大多数显卡对于RAM没有访问权限。对于一些数据CPU还需要对其进行处理，所以会存放在RAM中不会被立即释放。

②设置渲染状态

设置模型所需的几何着色器/片元着色器、光源及材质等。设置好后通知GPU渲染 即draw call

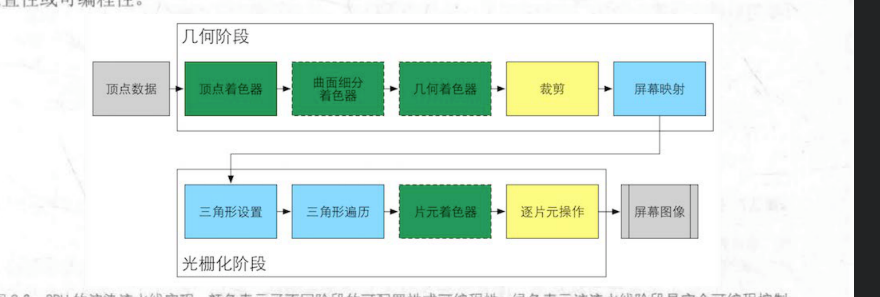
③调用DrawCall

DrawCall实质为一个命令，指向一个需要被渲染的图元（primitives）列表。由CPU发起GPU接受。GPU通过显存中数据进行渲染。

### GPU流水线

PS:GPU流水线为实际的流程，渲染流水线为概念上的

流程图如下：



绿色实线：表示必须由开发者编程实现。 绿色虚线：开发者可选择进行编程 黄色：可配置 蓝色：不可修改，为固定流程。

几何阶段：

顶点着色器：完全可编程。通常用于实现顶点的空间转换以及顶点颜色设置。

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，用于执行逐图元着色，或产生更多图元。

裁剪：可配置。将不存在相机视野内的顶点减掉。

屏幕映射：将每个图元坐标转换到屏幕坐标系中

光栅化阶段：

三角形设置：将几何阶段的顶点转换为一个个三角形

三角形遍历：为三角形内所包含的片元设置相关信息

片元着色器：可选着色器，实现逐片元的着色操作

逐片元操作：可配置操作。负责许多重要操作，如修改颜色，深度测试，模板测试，混合等。

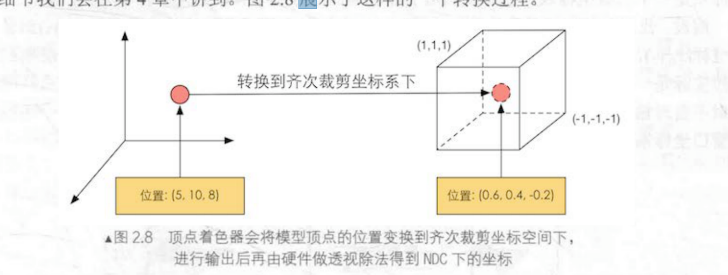
### 顶点着色器

主要任务：坐标转换和逐顶点光照设置以及输出后续阶段所需的数据（如颜色）。

·坐标转换：将顶点坐标从模型空间转换到齐次裁剪空间。

Ps：o.pos = mul(UNITY\_MVP,v.position) 这代码功能是将顶点坐标转换到齐次裁剪坐标系下。再由硬件做透视除法后最终得到归一化设备坐标（Normalized Device Coordinates, NDC）

Unity中的NDC



OpenGL同时也是Unity使用的NDC z分量取值为[-1,1] DirectX z值分量为[1,0]

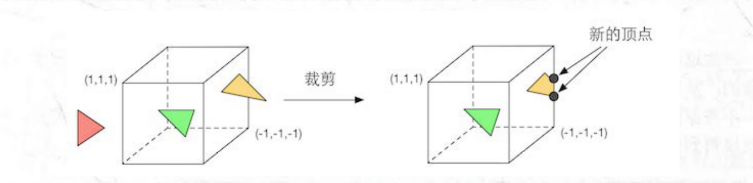
### 裁剪

主要任务：将那些不在摄像机视野内的物体剔除掉。

·完全在相机内：不处理

·完全不在相机内：直接剔除掉

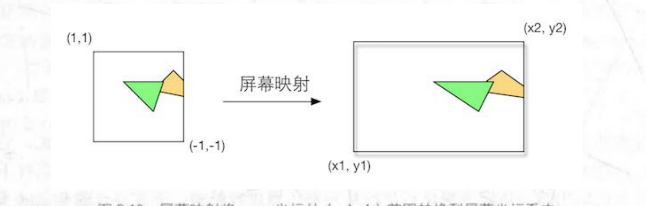
·不完全在相机内：沿着坐标边界进行裁剪。



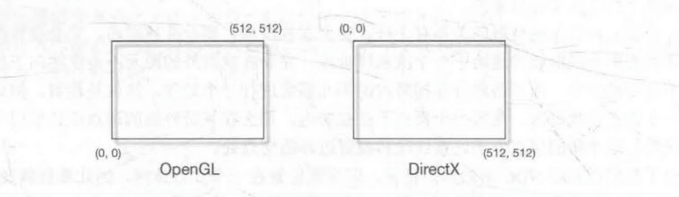
### 屏幕映射

主要任务：顶点着色器处理后，顶点仍然在三维坐标下。屏幕映射的目的就是将每个图元的x,y转换到屏幕坐标系下。

映射：屏幕坐标系的取值范围与屏幕分辨率相关。 齐次裁剪坐标系的取值范围[1,-1] 转换到屏幕坐标系就是一个缩放操作，齐次裁剪坐标系的z值则不进行操作，直接保留在屏幕坐标系输出给下一个阶段（可用于深度测试）。



PS：不同接口屏幕坐标系的起点不同 OpenGL为左下角 DirectX为左上角

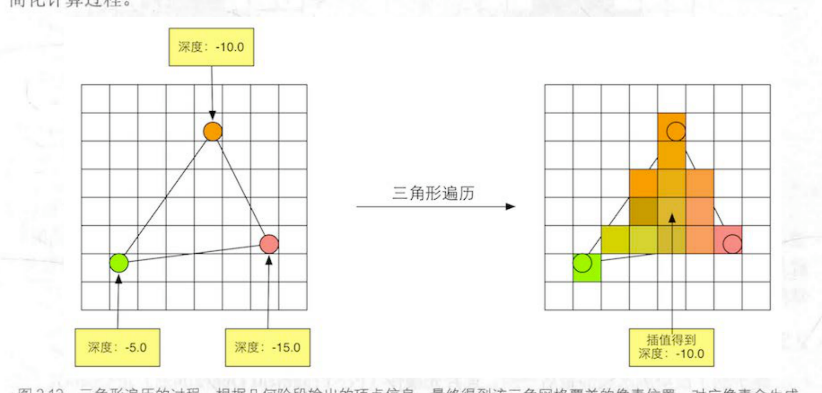


### 三角形设置

主要任务：该阶段进入光栅化，需要顶点位置以及它们相关的额外信息。如深度值，法线和视角方向。 三角形设置的目的就是将一条边两个顶点连接起来最终得到一个个三角形。

### 三角形遍历

主要任务：该阶段会遍历上一个阶段的计算结果，来判断一个三角形区域覆盖了哪些像素，并使用三角网格覆盖的区域进行像素插值。



被三角形覆盖的像素将会生成一个片元，该阶段输出的是一个片元序列。片元并不只是一个像素，而是包含了许多信息的集合。如屏幕坐标，深度信息等。

### 片元着色器（ DIrectX中被称为 像素着色器）

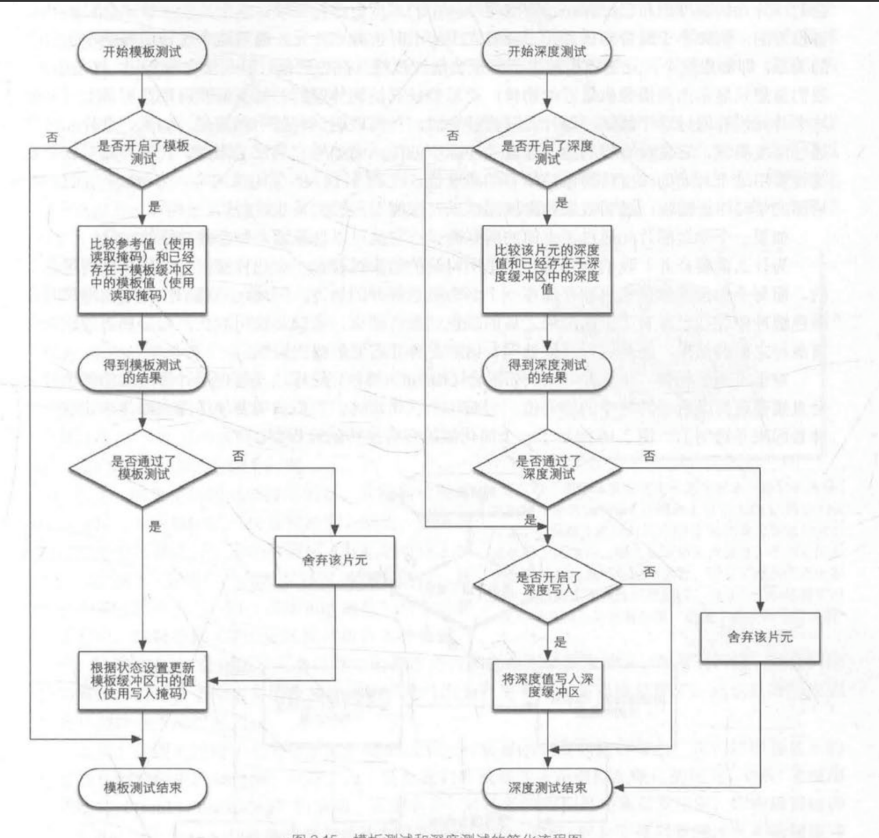
主要任务：利用上一个阶段得到的信息然后输出一个或者多个颜色值。该阶段是一个非常重要的可编程着色阶段。

### 逐片元操作（DirectX中被称为 输出合并阶段）

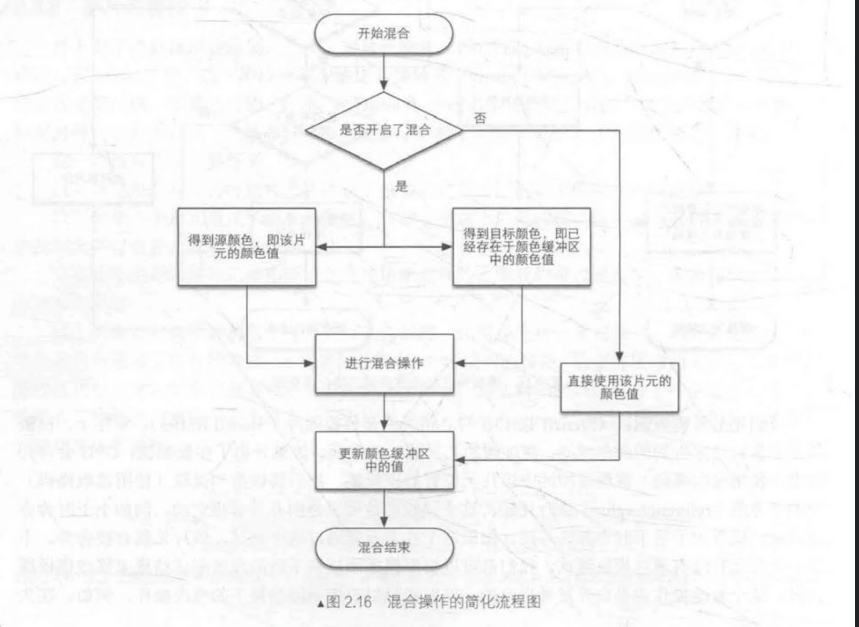
主要任务：①决定每一个片元的可见性。即各个测试工作：深度测试，模板测试

②如果片元通过了测试，将把改片元的颜色值和已经存储在颜色缓冲区中的颜色进行合并。

Ps：该阶段是高度可配置的



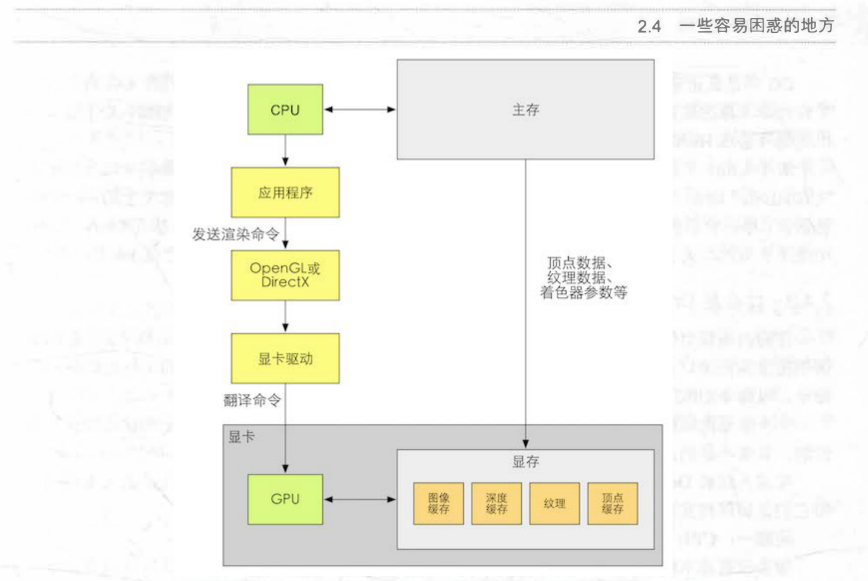
模板测试和深度测试简化过程



混合操作的简化流程

PS：对于大多数GPU来说，他们会尽可能的在执行片元着色器之前就进行这些测试，以降低性能开销。 现代GPU会判断片元着色器中的操作是否会和提前测试发生冲突，如果有则不会进行提前测试。 如：你在片元着色器里面手动丢掉了一些片元，那么就该漏出后面的片元，如果提前测试那么后面的片元已经被丢掉了。 GPU绘制使用了双缓存机制。以保证我们不会看见正在光栅化的画面。

### 扩展



OpenGL和DirectX是上层应用和GPU通信的桥梁

显卡驱动是用于将OpenGL或DirectX转化为GPU能听懂的指令，由显卡制作商提供。

**HLSL，GLSL，CG分别是什么**

都是着色语言

HLSL：High Level Shading Language。 由微软DirectX开发，由微软控制着着色器的编译，即便使用了不同的硬件结果是一致的。但也因此支持HLSL的设备有限，基本上是微软自家的产品。

GLSL：OpenGL Shading Language. 由OpenGL开发。 OpenGL没有提供着色器编辑器，而是有显卡驱动来完成着色器编译工作。只要显卡驱动支持对GLSL的编译就可以运行，这导致GLSL具有很强的可移植性，同时也会导致不同的硬件供应商会导致编译结果不一致的情况。

CG: C for Graphic。它会根据平台的不同，编译成不同的中间语言。同样具有很强的可移植性。 同时由于和微软有合作，所以语法上HLSL非常相似，可以无缝移植成HLSL但可能无法完全发挥出OpenGL的最新特性。

**为什么Draw Call多了会影响帧率**

这是因为每次CPU向GPU发送DrawCall指令时，都需要向GPU发送许多内容，如命令和数据等。在这阶段（即应用阶段）需要很多准备工作，之后将数据发送给GPU，由于GPU渲染能力很强，所以渲染速度高于提交数据。如果DrawCall数量过多，CPU就会把大量时间花在提交DrawCall上，造成CPU过载。

**如何减少Draw Call**

Draw Call过多导致CPU过载，一个明显的优化方法就是将一些细小的Draw Call合并为一个大的Draw Call。这也叫批处理（Batching）。

批处理：在CPU的内存中合并网格，然后提交到GPU中。

合并网格是需要时间的，所以批处理更适用于静态的物体，对于这些静态的物体只需要合并一次即可。对于动态的也是可以进行合并的，但是由于物体是在不断运动的，每一帧都需要合并，这对空间和时间都会造成一定的影响。

此外在游戏开发中还有另外两点需要注意

①尽量减少使用大量很小的网格，如果不可避免时，考虑是否可以进行合并

②避免使用过多的材质，尽量在不同的网格中使用相同的材质。