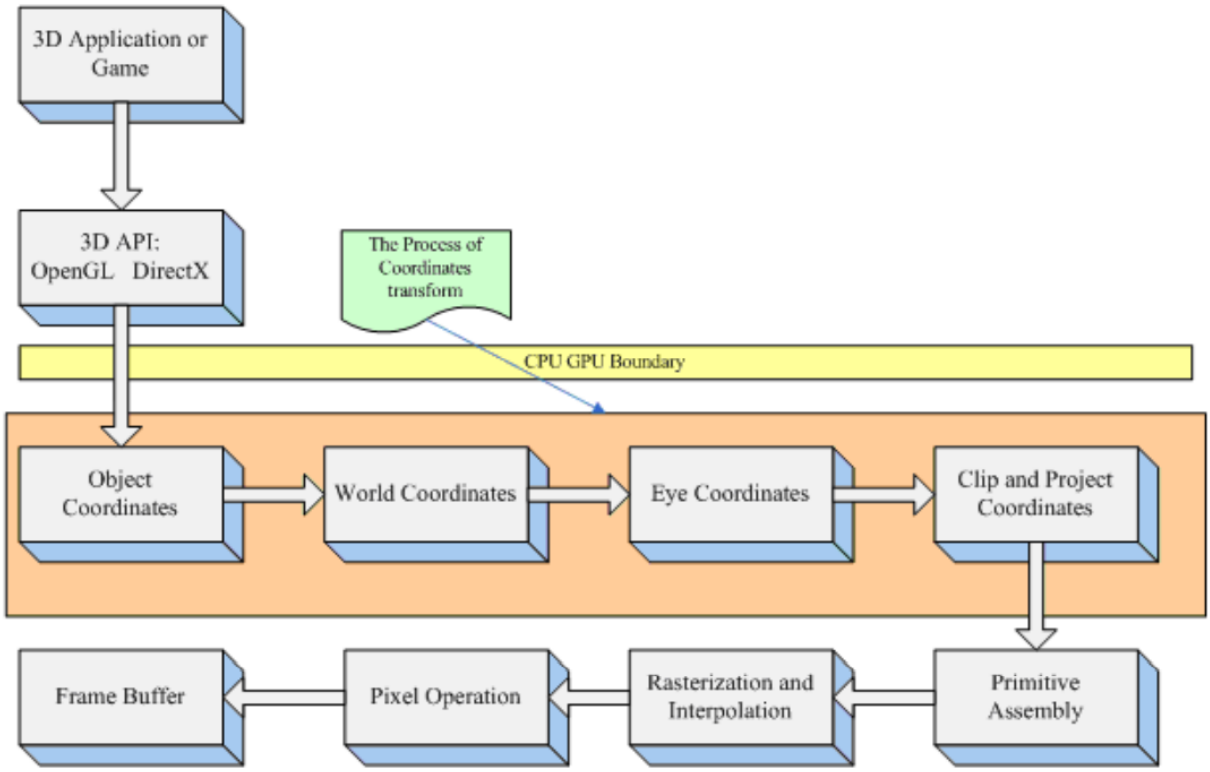
图形绘制管线描述 GPU 渲染流程，即“给定视点、三维物体、光源、照明模 式，和纹理等元素，如何绘制一幅二维图像”
图形绘制管线分为三个主要阶段：**应用程序阶段、几何阶段、光栅阶段。**

应用程序阶段：使用高级编程语言（C、C++、JAVA 等）进行开发，主要和 CPU、内存打交道；

几何阶段：主要负责顶点坐标变换、光照、裁剪、投影以及屏幕映射，该阶段基于 GPU 进行运算，在该阶段的末端得到了经过 变换和投影之后的顶点坐标、颜色、以及纹理坐标；

光栅阶段：基于几何阶段的输出数据，为像素（Pixel）正确配色，以便绘制完整图像，该阶段进行的都是单个像素的操作，每个像素的信息存储在颜色缓冲 器（color buffer 或者 frame buffer）中。



GPU 的整个处理流程

2.1 几何阶段

几何阶段的主要工作是“变换三维顶点坐标”和“光照计算”，显卡信息中通常会有一个标示为**“T&L”硬件**部分，所谓“T&L”即 Transform & Lighting。

根据顶点坐标变换的先后顺序，主要有如下几个坐标空间（坐标类型）：
Object space，模型坐标空间

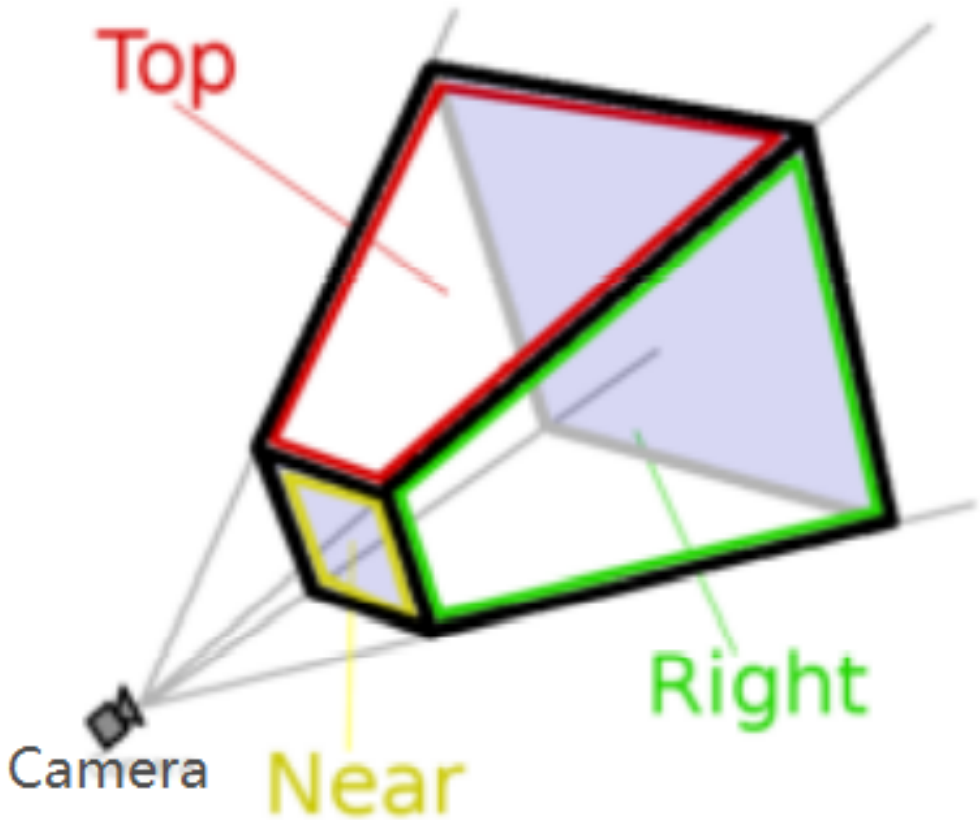
World space，世界坐标系空间
Eye space，观察坐标空间
Clip and Project space，屏幕坐标空间

2.1.1 从object space到world space

object space：
其一，object space coordinate 就是模型文件中的顶点值（在模型建模时得到的）
其二，object space coordinate 与其他物体没有任何参照关系
world space：无论在现实世界，还是在计算机的虚拟空间中，物体都必须和一个固定的坐标原点进行参照才能确定自己所在的位置，我们将一个模型导入计算机后，就应该给它一个相对于坐标原点的位置，这个位置就是 world space coordinate。
world matrix：从 object space coordinate 到 world space coordinate 的变换过程由一个四阶矩阵控制，通常称之为 world matrix。

有光照计算发生：world coordinate space（通常），eye coordinate space

2.1.2 从world space到eye space



eye space:以 camera（视点或相机）为原点，由视线方向、视角和 远近平面，共同组成一个梯形体的三维空间，称之为 viewing frustum（**视锥**），如上图所示。

2.1.3 从 eye space 到 project and clip space

视锥裁剪(Frustum Culling):在上图中，近平面，是梯形体较小的矩形面，作为投影平面；远平面是梯形体较大的矩形，在这个梯形体中的所有顶点数据是可见的；超出这个梯形体之外的场景数据，会被视点去除。这一步通常 称之为“clip（裁剪）”。

裁剪算法：识别指定区域内或区域外的图形部分的过程称之为裁剪算法。

规范立方体（Canonical view volume, CVV）：因为在不规则的体（viewing frustum）中进行裁剪并非易事，所以经过图形学前辈们的精心分析，裁剪被安排到一个单位立方体中进行，该立方体的对角顶点分别是 (-1,-1,-1)和(1,1,1)，通常称这个单位立方体为规范立方体。

CVV 的近平面（梯形体较小的矩形面）的 X、Y 坐标对应屏幕像素坐标（左下角是 0、0），Z 坐标则是代表画面像素深度。

从视点坐标空间到屏幕坐标空间 （screen coordinate space）事实上是由三步组成：

1. 用透视变换矩阵把顶点从视锥体中变换到裁剪空间的 CVV 中；--> 投影
   -->投影方法： 正交投影（也称平行投影）和透视投影

2. 在 CVV 进行图元裁剪；
   -->只有当图元完全或部分的存在于视锥内部时，才需要将其光栅化;
   -->当一 个图元完全位于视体（此时视体以及变换为 CVV）内部时，它可以直接进入下 一个阶段;
   -->完全在视体外部的图元，将被剔除；
3. 屏幕映射：将经过前述过程得到的坐标映射到屏幕坐标系上

视点去除：不但可以在 GPU 中进行，也可以使用高级语言（C\C++） 在 CPU 上实现。使用高级语言实现时，如果一个场景实体完全不在视锥中，则 该实体的网格数据不必传入 GPU，如果一个场景实体部分或完全在视锥中，则 该实体网格数据传入 GPU 中。所以如果在高级语言中已经进行了视点去除，那 么可以极大的减去 GPU 的负担。

图元：组成图形的最小单位（点、线、面）。

2.2 Primitive Assembly && Triangle setup （图元装配和处理三角形）

图元装配（Primitive Assembly ）：即将顶点根据 primitive（原始的连接关系），还原出网格结构。网格由顶点和索引组成，在之前的流水线中是对顶点的处理， 在这个阶段是根据索引将顶点链接在一起，组成线、面单元。之后就是对超出屏 幕外的三角形进行裁剪。

为 了减少需要绘制的顶点个数，而识别指定区域内或区域外的图形部分的算法都称之为裁减。裁减算法主要包括：视域剔除（View Frustum Culling）、背面剔除 （Back-Face Culling）、遮挡剔除（Occlusing Culling）和视口裁减等。

2.3 光栅化阶段

2.3.1 Rasterization

光栅化（Rasterization）：决定哪些像素被集合图元覆盖的过程（Rasterization is the process of determining the set of pixels covered by a geometric primitive）。经过上面诸多坐标转换之后，现在我们得到了每个点的屏幕坐标值（Screen coordinate），也知道我 们需要绘制的图元（点、线、面）。但此时还存在两个问题：

问题一：点的屏幕坐标值是浮点数，但像素都是由整数点来表示的，如果确 定屏幕坐标值所对应的像素?

“绘制的位置只能接近两指定端点间的实际线段位置， 例如，一条线段的位置是（10.48，20.51），转换为像素位置则是（10，21）。

问题二：在屏幕上需要绘制的有点、线、面，如何根据两个已经确定位置的 2 个像素点绘制一条线段，如果根据已经确定了位置的 3 个像素点绘制一个三角形面片?

对于问题二涉及到具体的画线算法，以及区域图元填充算法。通常的画线算 法有 DDA 算法、Bresenham 画线算法；区域图元填充算法有，扫描线多边形填 充算法、边界填充算法等。

这个过程结束之后，顶点(vertex)以及绘制图元（线、面）已经对应到像素 (pixel)。

2.3.2 Pixel Operation（Raster Operation）

在更新帧缓存之前，执行最后一系列针对每个片段的操作，其目 的是：计算出每个像素的颜色值。

1：消除遮挡面

2：Texture operation，纹理操作，也就是根据像素的纹理坐标，查询对应的纹理值

3：Blending（混色）

alpha 混合技术：根据目前已经画好的颜色，与正在计算的颜色的透明度（Alpha）， 混合为两种颜色，作为新的颜色输出。通常称之为 alpha 混合技术。

每个像素：一个RGB颜色值和一个 （Z Buffer）Z缓冲器深度值，一个alpha值

从绘制管线得到一个 RGBA，使用 over 操作符将该值与原像素颜色值进行 混合，公式如下：

$$c_d = a \bullet c_a + (1-a)c_s \qquad 【over\ 操作符】$$

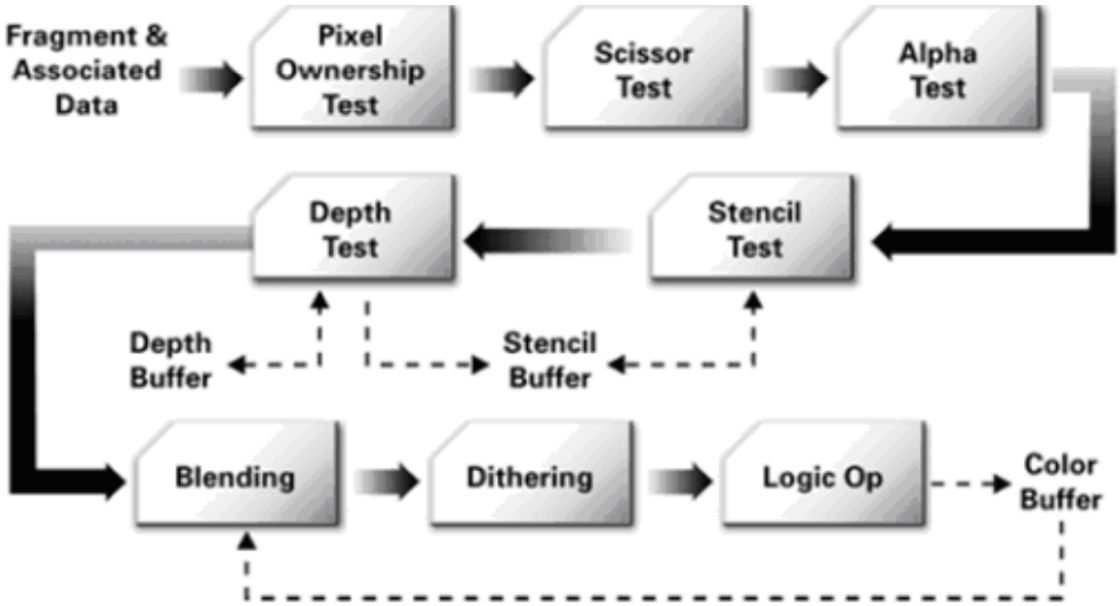
a 是透明度值（alpha）， ca 表示透明物体的颜色， cs 表示混合前像素的颜色值， cd 是最终计算得到的颜色值。Z缓冲器深度值：排序相关。

为了在场景中绘制透明物体，通常需要对物体进行排序。首先，绘制不透明的物体；然后，在不透明物体的上方，对透明物体按照由后到前的顺序进行混合处理。如果按照任意顺序进行混合，那么会产生严重的失真。

4：Filtering

将正在算的颜色经过某种 Filtering（滤波或者滤镜）后输出。 可以理解为：经过一种数学运算后变成新的颜色值。

该阶段之后，像素的颜色值被写入帧缓存中。



OpenGL 和 Direct3D 中的 Raster Operations（像素操作流程）

OGRE 中有一种技术 称为 compositor（合成器）。

OGRE是什么？ https://www.oschina.net/p/ogre?hmsr=aladdin1e1

2.4 图形硬件

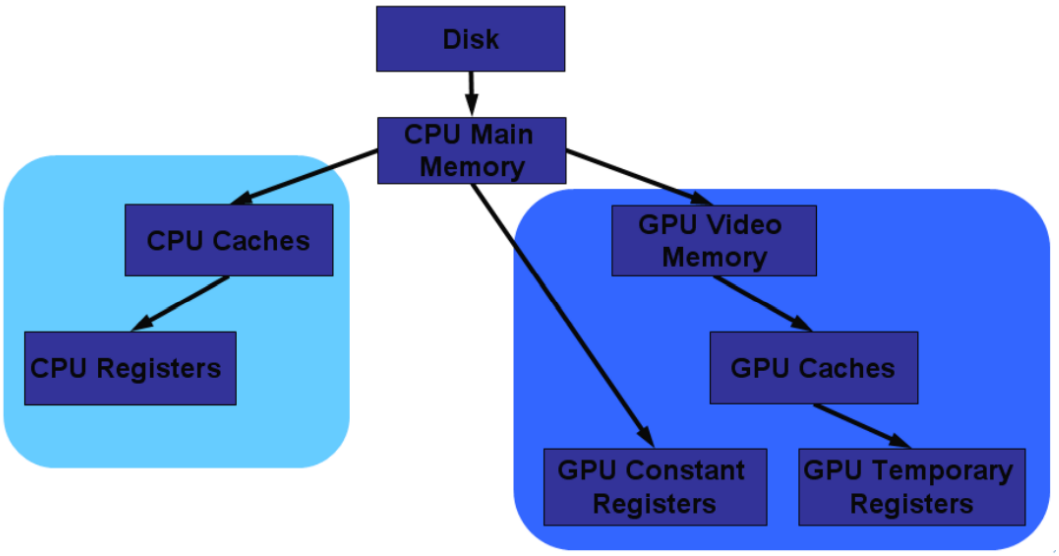
图形硬件的相关知识，主要包括 **GPU 中数据的存放硬件**， 以及**各类缓冲区的具体含义和用途**，如：z buffer（深度缓冲区）、stencil buffer （模板缓冲区）、frame buffer（帧缓冲区）和 color buffer（颜色缓冲区）

2.4.1 GPU 内存架构

寄存器和内存的区别：

物理结构：寄存器是 cpu 或 gpu 内部的存储单元，即寄存器是嵌入在 cpu 或者 gpu 中的，而内存则可以独立存在；

功能：寄存器是有限存储容量的高速存储部件，用来暂存指令、数据和位址。Shader 编程是基于计算机图形硬件的，这其中就包括 GPU 上的寄存器类型，glsl 和 hlsl 的着色虚拟机版本就是基于 GPU 的寄存器和指令集而区分的。



GPU 存储架构

2.4.2 Z Buffer 与 Z 值

Z buffer又称为 depth buffer（深度缓冲区），其中存放的是视点到每个像素所对应的空间点的距离衡量，称之为Z值或者深度值。

不同的图形硬件使用不同位数的Zbuffer： 16 位，24 位， 32 位。编程时要注意这个点。

可见物体的Z值范围位于【0，1】区间，默认情况下，最接近眼睛的顶点（近裁减面上）其Z值为0.0，离眼睛最远的顶点（远裁减面上）其Z值为1.0。

Z 值决定了物体之间的相互遮挡关系，如果没有 足够的精度，则两个相距很近的物体将会出现随机遮挡的现象，这种现象通常称 为“flimmering”或”Z-fighting”

2.4.3 Stencil Buffer

Stencil buffer（模板缓冲区）是一个额外的 buffer，通常附加到 z buffer 中。例如：15 位的 z buffer 加上 1 位的 stencil buffer(总共 2 个字节)； 或者 24 位的 z buffer 加上 8 位的 stencil buffer（总共 4 个字节）。是一个用来“做记号”的 buffer，例如：在一个像素的 stencil buffer 中存放 1，表示该像素 对应的空间点处于阴影体（shadow volume）中。

2.4.4 Frame Buffer

Frame buffer（帧缓冲器）用于存放显示输出的数据，这个 buffer 中的数据一般是像素颜色值。frame buffer 通常都在显卡上，当显卡会集成到主板上时 frame buffer 被放在内存区域（general main memory）