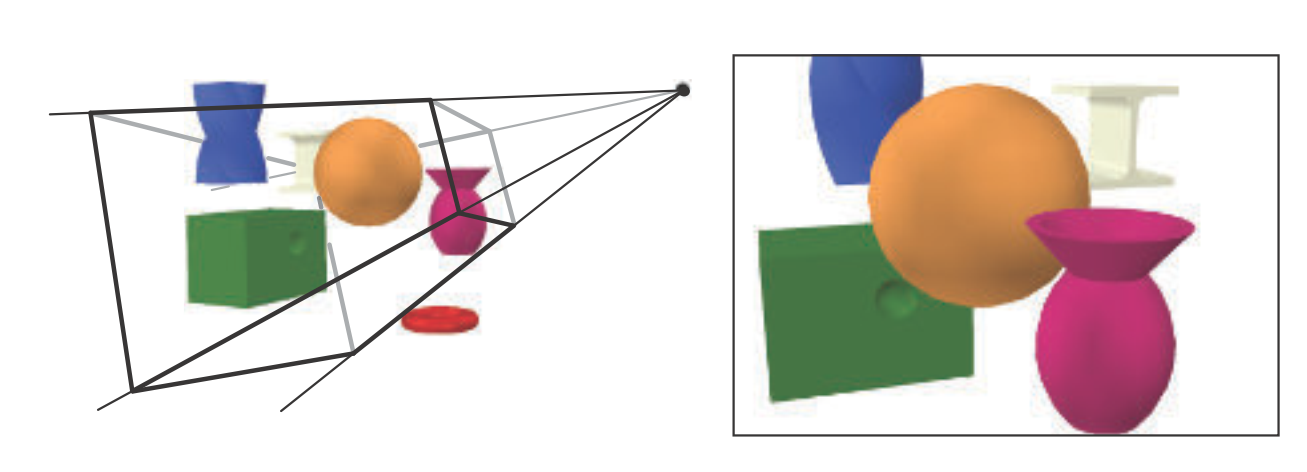
笔者最近在回顾一些图形学基础知识，遂整理在此，此文主要讲述图形渲染管线。

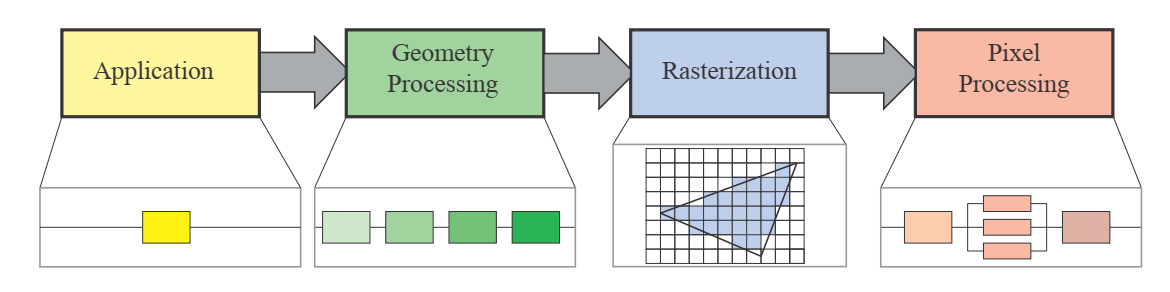
图形渲染管线(Graphics rendering pipeline)的主要功能是在给定一个虚拟相机、三维物体、光源和其他信息的情况下，生成或者说渲染出一副二维图像。在二维图像中，物体的位置取决于物体的三维位置和形状、环境的一些性质以及相机的位置，物体的表面颜色则取决于材质属性、光源、纹理和着色方程等。



此图呈现了一个透视投影的例子，虚拟相机位于视锥体的顶端，且只有视锥体以内的物体被渲染。那个红色的圆环被抛弃掉，以及那个蓝色的物体也被截取了一半

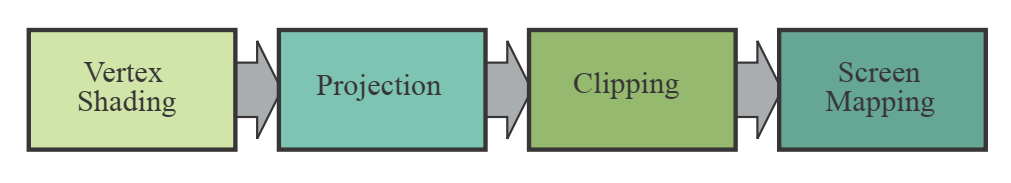
使用管线的一个主要原因是它是并行的。理论上一个非并行的系统被划分为n个阶段同时执行，可以获得n倍的加速，但实际上取决于最慢的瓶颈。

图形渲染管线可简单分为四个阶段：应用程序、几何处理阶段、光栅化和像素处理阶段，每个阶段也都有并行化的子管线



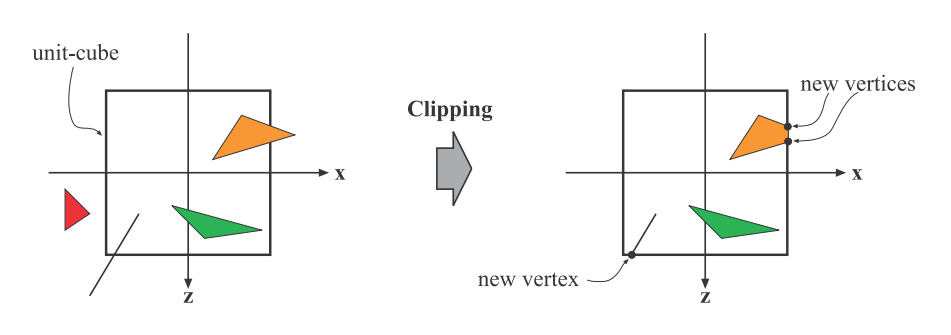
1、应用程序阶段。主要是运行在CPU上面，通过多线程运算来实现并行，此阶段完全被开发者控制。这个阶段最主要的功能就是向下一阶段传递几何图元(primitive)，如一些点、线、三角形。其他的功能比如有：碰撞检测、一些加速算法、动画、物理模拟、接收处理外设输入等。此外，还有一些应用程序的工作可以在GPU上面进行，这个单独的模式称之为计算着色器(compute shader)，此时就把GPU当作一个高度并行的处理器，而非用于渲染

2、几何处理阶段。运行在GPU上面，主要处理逐顶点或者逐三角形的操作，诸如变换、投影等，用于决定哪些物体要被画、怎样画出来以及在哪画，可以被更细分为四个阶段：顶点着色器、投影、裁剪、屏幕映射。



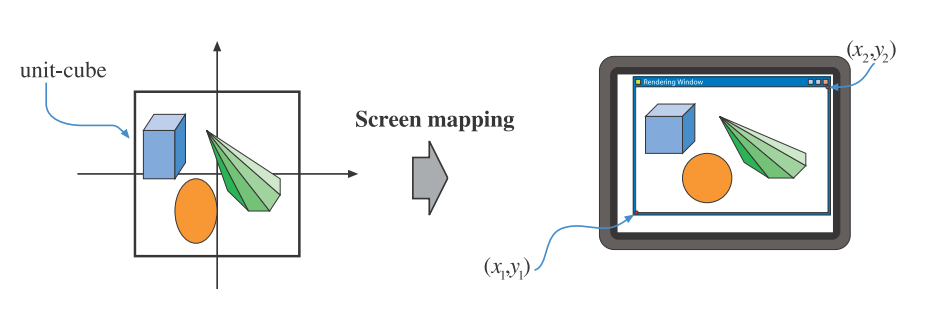
顶点着色器用于计算1)顶点位置和2)顶点的附属数据，前者是通过一系列变换矩阵，将物体坐标从物体坐标系转到世界坐标系，再转到相机坐标系；后者则是计算一些顶点的附属数据，如法线、颜色、纹理坐标等，这些数据会在传入光栅化阶段时被插值，而被每个像素所使用。顶点着色器不能创造也不能销毁顶点，一个顶点的数据也不能传给其他顶点，因为各个顶点的操作是高度独立的，所以也就可以在GPU里面并行处理

随后进行投影和裁剪，将相机空间的视域体（透视投影-视锥体；正交投影-长方体）转化为规范视域体(canonical view volume)，其范围为(-1,-1,-1)到(1,1,1)(不同的API可能不同)，也称之为裁剪空间，但仍然为四维齐次坐标，并把与视域体边界相交的线段进行裁剪，如图所示：



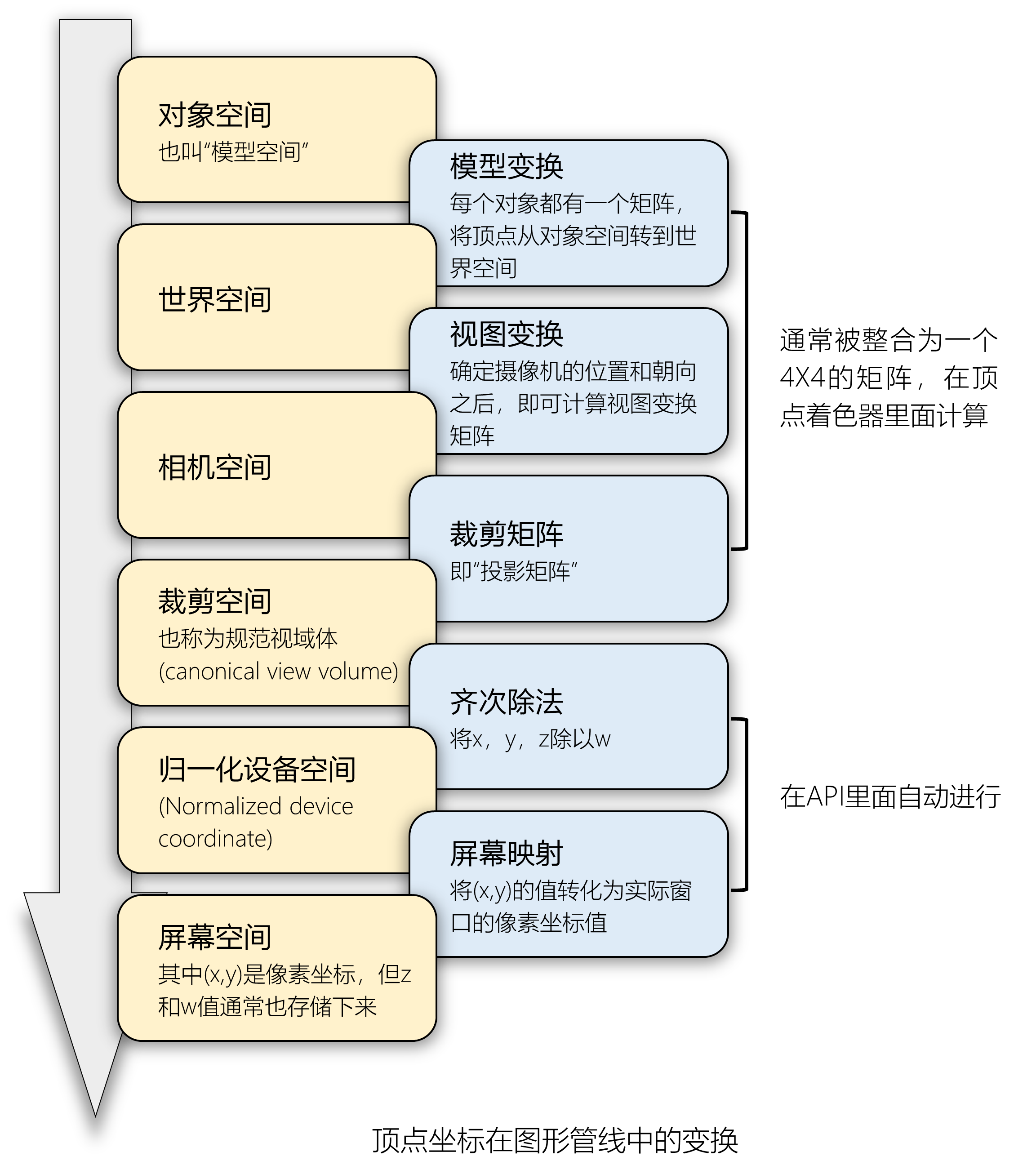
此图中，在裁剪时用的是四维齐次坐标进行插值，注意一定要对w值进行插值，特别是执行透视投影的时候

几何处理最后一个子阶段是屏幕坐标映射，经过前面的投影和剪裁后，得到四维齐次坐标，再除以w后，进入三维的归一化设备坐标(normalized device coordinates)，此时的z被存入z-buffer里面，(x,y)的值从(-1,1)映射到不同的窗口尺寸范围，如图所示：



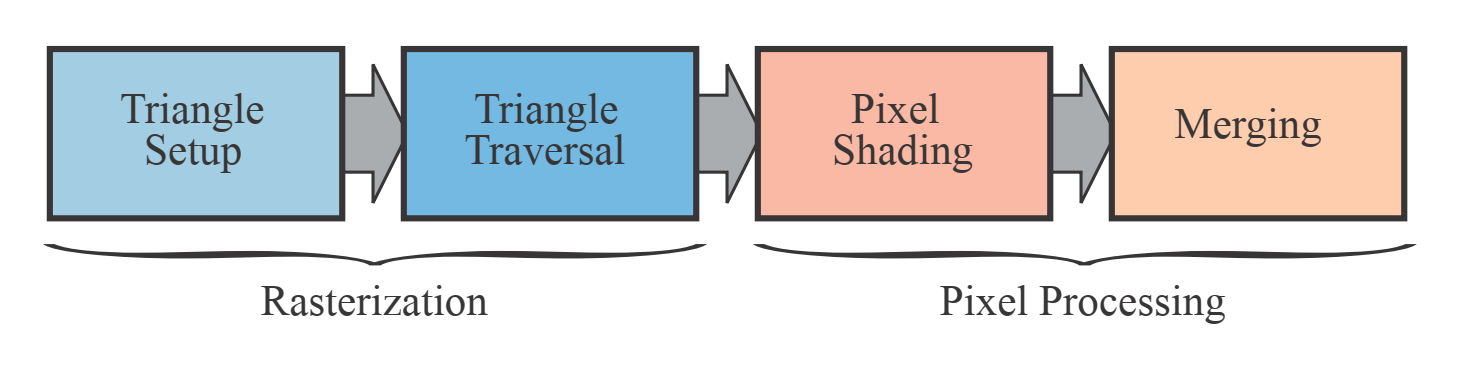
此图展示了屏幕映射的过程，注意不同的API对于坐标系定义不一致

上面所述的顶点操作可归结为一张图：



所有管线都有刚刚描述的这些几何处理的过程，但是一些API支持更强的GPU另外执行下面一些操作：曲面细分(tessellation)，几何着色器(geometry shading)，流输出(stream output)，并不常用。

3、光栅化阶段。将一系列带有深度信息的屏幕坐标点及其附属的其他信息转移到屏幕上的每个像素。首先是将顶点装配为三角形，当然也适用于点或者线的情况，称之为图元装配，然后遍历每个三角形，通过对顶点的信息进行插值从而生成每个片元(fragment，可称之为像素)，每个片元包含深度以及用于着色的信息。生成每个片元后，传递给像素处理阶段：



4、像素处理阶段。决定每个像素的颜色以及通过深度测试确定其可见性，可分为像素着色器(pixel shading，也称片元着色(fragment shading))和混合两个子阶段。

像素着色器是完全可编程的，常用于计算和输出片元的颜色。比如执行纹理贴图(texturing)，贴图就好比给一个光秃秃的模型粘上一层图像，让其栩栩如生。当然贴图也有许多其他的用途，如法线贴图、凹凸贴图等，之后的文章会讲到。



最初，像素着色器的输出只能是混合阶段，但是现在可以输出到多个buffer里面，称之为MRT(multiple render target). 这样就可以同时输出多种信息，一个典型例子是延迟渲染

像素处理的第二个子阶段，混合。它不是高度可编程而是高度可配置化的，混合的主要用途是将像素着色器里面的颜色与当前颜色缓冲的颜色进行混合，比如各种叠加方式。混合也被用于计算像素可见性，几乎现在所有的图形硬件都在使用z-buffer算法：z-buffer的尺寸和color-buffer一致，其主要思想是比较当前深度值和新的fragment的深度值，将较小者留下并更新当前buffer的颜色和深度值，此算法的优点是复杂度低(仅为O(n)，n为图元数量)且在混合时不需要考虑次序，所以极为流行，但其缺点是不能用于半透明图元的混合。

回想一下，如果在片元着色器里面对每一个三角形的每个片元都进行了计算，但是到混合阶段却因为可见性被抛弃了，那么之前的计算实际上都浪费了。所以，现代的GPU都实现了一种名为early-z的功能，即在片元着色器之前进行可见性测试。

除了color-buffer和z-buffer外，还有alpha-buffer，在旧的API中被用于进行透明度测试以确定是否抛弃这个像素，但在现代的API中，其通常是用在片元着色器里面参与计算或者决定是否抛弃像素。此外还有模板缓冲(stencil buffer)等，此处不赘述

整个图形管线将许多顶点数据进行处理，根据用户的需求和交互确定环境属性、变换矩阵等，最终渲染出二维图像显示在屏幕上。本文只概述了最常用实时图形管线的流程，随着现代GPU的发展，也出现了很多新的技术，后续有机会和大家逐一分享学习，请多指教！