

图1：pipeline过程

Pipeline 是指图形系统对图像信息进行一系列处理的过程，这些图像信息包括顶点信息(坐标、法向量…),像素信息(图像、纹理…)等,最终这些处理好的数据将被放到帧缓冲区(FrameBuffer)中供应用获取。

a.输入装配器(Input assembler) : 从Vertex Buffer 或 Index Buffer 中搜集原始顶点数据。

b.顶点着色器(Vertex Shader): 每一个顶点都会运行一次Vertex Sahder,它将顶点的原始坐标(model space)转换成屏幕坐标(screen space)，并为pipeline的下一个阶段提供输入。

c.细分着色器(Teseellation) : 将几何图形细分成更多、更小的三角形使得渲染表面和边缘更加平滑。如下图:

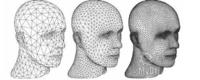


图2：细分着色器

d.几何着色器(Geometry Shader) :作用于每一个图原(primitive)(点，线，三角形),在操作的过程中可能会丢失部分图原，也能会增加图原，功能和Teseellation差不多，但更为复杂，Geometry Shader现在用的并不多，除了Intel的集成显卡外，它在其他类型显卡上的运行性能让人抓狂。

e.光栅化(Rasterization) : 可能叫做像素化更好理解一些。它将图原(primitive)离散成片原(fragments)，每一个片原代表FrameBuffer中的一个像素信息，对应于屏幕上的一点的颜色和不透明度等信息，屏幕外的片原将被丢弃，Vertex Shader 的输出结果会和片原做进一步的插值计算。由于深度关系，那些在其他片原后面的片原同样会被丢弃。如下两幅图:

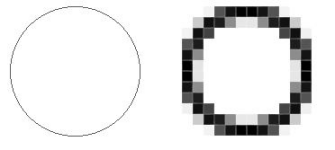
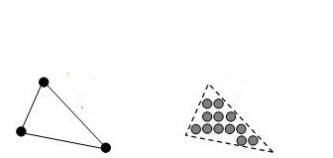
 

图3：边缘像素化 图4：区域像素化

f.片原着色器(Fragment Shader) : 对于每一个片原，Fragment Shader 都会调用一次，并决定片原使用何种颜色和何种深度值来填充哪一个FrameBuffer(s)。你可以使用来自Vertex Shader的被插值后的数据，这些数据可以包含用于光照的的纹理坐标和法线。

g.颜色混合器(Color blending) : 用于混合在同一FrameBuffer 中不同片原映射的同一像素的情况，你可以选择覆盖、相加或混合它们。

图1-Pipeline 过程图中，*绿色的模块*表示固定的功能，它们所做的工作已经被预定义好了，不过你可以通过一些参数来进行一些设置。而*橙色的模块*属于可编程部分，这意味着你可以编写自己的代码使显卡按照你的目的去执行。

在Vulkan中Pipeline 是不可改变的，这意味着如果你想改变着色器、绑定不同的FrameBuffer、改变混合功能(blend function)，etc，你必须从头开始重建Pipeline。喜忧参半…。

由于你使用Pipeline的具体目的不同，一些可编程模块就成了可选项，比如，如果你只想画一个几何图形，那么Teseellation 和 Geometry Shader 就可以不需要了。

**着色器模块 Shader modules**

Vulkan 和之前的图形API有所不同，为了避免不同厂商移植代码的复杂性，Vulkan的着色器代码(shader code)采用字节码格式(bytecode) SPIR-V，而非人类可以阅读的文本格式,如GLSL(opengl 的一套体系)等。但这并不意味着我们要亲自手写字节码， 幸运的是LunarG SDK 已经提供了glslangValidator.exe ，这个程序已经在你的安装目录中了，所以我们在编写着色器程序的时候任然使用我们所熟悉的GLSL格式，glslangValidator.exe 将为我们把GLSL转换成SPIR-V。

**Vertex shader**

它以vertex position 、mormal(法线)、texture Coordination 、color等作为输入，输出color、position、和texture Coordination等。最终作为输入传递给Fragment Sahder 的position 是以clip Coordination表示的。

clip Coordination 是齐次坐标( homogeneous coordinates)，他与FrameBuffer的坐标一一对应,如图:

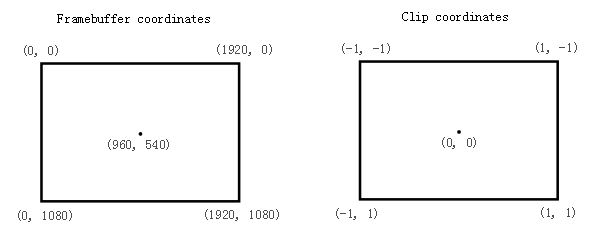


图5：坐标转换

如同Direct3D ，齐次坐标的z轴取值为0到1。   
对于我们将要画的三角形来讲，我们直接定义它的三个坐标为齐次坐标，这样可以更加简单，避免了坐标的转换。如图下图：

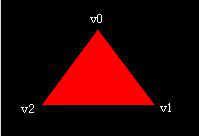


图6：运行效果图

通常坐标值都是被放在Vertex Buffer中的，但是Vertex Buffer 也是个比较大的概念，我们之后再讨论。这里我们将采用一种看起来不太规范的做法:直接把坐标硬编码到着色器代码里。

#version 450

#extension GL\_ARB\_separate\_shader\_objects : enable

out gl\_PerVertex {

vec4 gl\_Position;

};

vec2 positions[3] = vec2[](

vec2(0.0, -0.5),

vec2(0.5, 0.5),

vec2(-0.5, 0.5)

);

void main() {

gl\_Position = vec4(positions[gl\_VertexIndex], 0.0, 1.0);

}

着色器需要 GL\_ARB\_separate\_shader\_objects 扩展才能工作，gl\_VertexIndex是内置变量，代表Vertex Buffer的索引，这里表示顶点数组的索引,gl\_position内置变量表示输出坐标。四元组坐标(x,y,z,w)中,z和w 采用齐次坐标。

**Creating shader modules**

要把着色器程序传递到Pipeline之前，我们需要把它们包裹成:VkShaderModule。让我们创建一个函数来完成包裹的过程:

void createShaderModule(const std::vector<char>& code, VDeleter<VkShaderModule>& shaderModule) {}

然后还是创建信息描述的结构体对象vkShaderModuleCreateInfo

**Fixed functions**（不可编程部分）