RGB模式是以色光三原色为基础建立的色彩模式。RGB图像只使用三种颜色，当不等量的三种色光进行叠加混合时，即会在屏幕上重现自然界各种各样的颜色。

红、绿、蓝每一种颜色各有256级亮度，用数字表示为从0、1、2…..直到255，256级的RGB色彩总共能组合出约1678万种色彩。

CMY是3种印刷油墨名称的首字母：青色Cyan、洋红色Magenta、黄色Yellow。K是黑色。

OpenGL： 一个定义了函数布局和输出的图形API的正式规范。

GLFW

GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建OpenGL上下文，定义窗口参数以及处理用户输入

Cmake

CMake 可以编译[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81" \t "https://baike.baidu.com/item/_blank)、制作程序库、产生适配器（wrapper）、还可以用任意的顺序建构[执行档](https://baike.baidu.com/item/%E6%89%A7%E8%A1%8C%E6%A1%A3" \t "https://baike.baidu.com/item/_blank)。CMake 支持 in-place 建构（二进档和源代码在同一个目录树中）和 out-of-place 建构（二进档在别的目录里），因此可以很容易从同一个源代码目录树中建构出多个二进档。CMake 也支持静态与动态程式库的建构。

CMake是一个工程文件生成工具。用户可以使用预定义好的CMake脚本，根据自己的选择（像是Visual Studio, Code::Blocks, Eclipse）生成不同IDE的工程文件

GLAD

GLAD是继GL3W，GLEW之后，当前最新的用来访问[OpenGL](https://www.baidu.com/s?wd=OpenGL&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/zjz520yy/article/details/_blank)规范接口的第三方库，一个拓展加载库，用来为我们加载并设定所有OpenGL函数指针，从而让我们能够使用所有（现代）OpenGL函数。

GLAD使用步骤：GLAD可以使OpenGL基础渲染变得十分简单，只需要简单四个步骤就可以完成基础渲染。流程如下：

1.初始化GLAD库，加载所有OpenGL函数指针。

2.创建着色器并附加到着色器程序。

3.构建缓冲对象并附加到顶点属性。

4.使用着色器程序，利用渲染接口将缓冲对象按照指定图元类型渲染出来。

视口(Viewport) 需要渲染的窗口。可设定渲染窗口的尺寸大小

当用户改变窗口的大小的时候，视口也应该被调整。对窗口注册一个回调函数(Callback Function)，它会在每次窗口大小被调整的时候被调用。

渲染循环(Render Loop)

在我们主动关闭它之前不断绘制图像并能够接受用户输入。因此，我们需要在程序中添加一个while循环，在我们让GLFW退出前一直保持运行。

glClear函数来清空屏幕的颜色缓冲，因为新的渲染前希望清屏。

glClearColor来设置清空屏幕所用的颜色

双缓冲（Double Buffer)

应用程序使用单缓冲绘图时可能会存在图像闪烁的问题。 这是因为生成的图像不是一下子被绘制出来的，而是按照从左到右，由上而下逐像素地绘制而成的。最终图像不是在瞬间显示给用户，而是通过一步一步生成的，这会导致渲染的结果很不真实。为了规避这些问题，我们应用双缓冲渲染窗口应用程序。前缓冲保存着最终输出的图像，它会在屏幕上显示；而所有的的渲染指令都会在后缓冲上绘制。当所有的渲染指令执行完毕后，我们交换(Swap)前缓冲和后缓冲，这样图像就立即呈显出来，之前提到的不真实感就消除了。

顶点数组对象：Vertex Array Object，VAO

顶点缓冲对象：Vertex Buffer Object，VBO

索引缓冲对象：Element Buffer Object，EBO或Index Buffer Object，IBO

图形渲染管线（Graphics Pipeline，管线）

一个顶点在呈现为像素之前经过的全部过程。

3D坐标转为2D坐标的处理过程是由OpenGL的图形渲染管线（实际上指的是一堆原始图形数据途经一个输送管道，期间经过各种变化处理最终出现在屏幕的过程）管理的。图形渲染管线可以被划分为两个主要部分：第一部分把你的3D坐标转换为2D坐标，第二部分是把2D坐标转变为实际的有颜色的像素。2D坐标和像素也是不同的，2D坐标精确表示一个点在2D空间中的位置，而2D像素是这个点的近似值，2D像素受到你的屏幕/窗口分辨率的限制

着色器

一个运行在显卡上的小型程序。很多阶段的图形管道都可以使用自定义的着色器来代替原有的功能。

着色器是使用一种叫GLSL的类C语言写成的。GLSL是为图形计算量身定制的，它包含一些针对向量和矩阵操作的有用特性。

着色器的开头总是要声明版本，接着是输入和输出变量、uniform和main函数。每个着色器的入口点都是main函数，在这个函数中我们处理所有的输入变量，并将结果输出到输出变量中。

图形渲染管线接受一组3D坐标，然后把它们转变为你屏幕上的有色2D像素输出。图形渲染管线可以被划分为几个阶段，每个阶段将会把前一个阶段的输出作为输入。所有这些阶段都是高度专门化的（它们都有一个特定的函数），并且很容易并行执行。正是由于它们具有并行执行的特性，当今大多数显卡都有成千上万的小处理核心，它们在GPU上为每一个（渲染管线）阶段运行各自的小程序，从而在图形渲染管线中快速处理你的数据。这些小程序叫做着色器(Shader)。

有些着色器允许开发者自己配置，这就允许我们用自己写的着色器来替换默认的。这样我们就可以更细致地控制图形渲染管线中的特定部分了，而且因为它们运行在GPU上，所以它们可以给我们节约宝贵的CPU时间。OpenGL着色器是用OpenGL着色器语言(OpenGL Shading Language, GLSL)写成的

图元

为了让OpenGL知道我们的坐标和颜色值构成的到底是什么，OpenGL需要你去指定这些数据所表示的渲染类型。我们是希望把这些数据渲染成一系列的点？一系列的三角形？还是仅仅是一个长长的线？做出的这些提示叫做图元(Primitive)，任何一个绘制指令的调用都将把图元传递给OpenGL。这是其中的几个：GL\_POINTS、GL\_TRIANGLES、GL\_LINE\_STRIP。

顶点vertex

以数组的形式传递3个3D坐标作为图形渲染管线的输入，用来表示一个三角形，这个数组叫做顶点数据(Vertex Data)；顶点数据是一系列顶点的集合。一个顶点(Vertex)是一个3D坐标的数据的集合。而顶点数据是用顶点属性(Vertex Attribute)表示的，

顶点着色器、几何着色器、片段着色器、片段着色器

当我们特别谈论到顶点着色器的时候，每个输入变量也叫顶点属性(Vertex Attribute)。

图形渲染管线的第一个部分是顶点着色器(Vertex Shader)，它把一个单独的顶点作为输入。顶点着色器主要的目的是把3D坐标转为标准化设备坐标，同时顶点着色器允许我们对顶点属性进行一些基本处理。

图元装配(Primitive Assembly)阶段将顶点着色器输出的所有顶点作为输入（如果是GL\_POINTS，那么就是一个顶点），并所有的点装配成指定图元的形状。

图元装配阶段的输出会传递给几何着色器(Geometry Shader)。几何着色器把图元形式的一系列顶点的集合作为输入，它可以通过产生新顶点构造出新的（或是其它的）图元来生成其他形状。

几何着色器的输出会被传入光栅化阶段(Rasterization Stage)，这里它会把图元映射为最终屏幕上相应的像素，生成供片段着色器(Fragment Shader)使用的片段(Fragment)。在片段着色器运行之前会执行裁切(Clipping)。裁切会丢弃超出你的视图以外的所有像素，用来提升执行效率。

OpenGL中的一个片段是OpenGL渲染一个像素所需的所有数据。

片段着色器的主要目的是计算一个像素的最终颜色，这也是所有OpenGL高级效果产生的地方。通常，片段着色器包含3D场景的数据（比如光照、阴影、光的颜色等等），这些数据可以被用来计算最终像素的颜色。

在所有对应颜色值确定以后，最终的对象将会被传到最后一个阶段，我们叫做Alpha测试和混合(Blending)阶段。这个阶段检测片段的对应的深度（和模板(Stencil)）值，用它们来判断这个像素是其它物体的前面还是后面，决定是否应该丢弃。这个阶段也会检查alpha值（alpha值定义了一个物体的透明度）并对物体进行混合(Blend)。所以，即使在片段着色器中计算出来了一个像素输出的颜色，在渲染多个三角形的时候最后的像素颜色也可能完全不同。

图形渲染管线非常复杂，它包含很多可配置的部分。然而，对于大多数场合，我们只需要配置顶点和片段着色器就行了。几何着色器是可选的，通常使用它默认的着色器就行了。

在现代OpenGL中，我们必须定义至少一个顶点着色器和一个片段着色器（因为GPU中没有默认的顶点/片段着色器）。

标准化设备坐标(Normalized Device Coordinates)

顶点在通过在剪裁坐标系中剪裁与透视除法后最终呈现在的坐标系。所有位置在NDC下-1.0到1.0的顶点将不会被丢弃并且可见。

一旦你的顶点坐标已经在顶点着色器中处理过，它们就应该是标准化设备坐标了。标准化设备坐标是一个x、y和z值在-1.0到1.0的一小段空间。任何落在范围外的坐标都会被丢弃/裁剪，不会显示在你的屏幕上。

顶点缓冲对象(Vertex Buffer Objects, VBO）

一个调用显存并存储所有顶点数据供显卡使用的缓冲对象。

定义这样的顶点数据以后，我们会把它作为输入发送给图形渲染管线的第一个处理阶段：顶点着色器。它会在GPU上创建内存用于储存我们的顶点数据，还要配置OpenGL如何解释这些内存，并且指定其如何发送给显卡。顶点着色器接着会处理我们在内存中指定数量的顶点。顶点缓冲对象(Vertex Buffer Objects, VBO)管理这个内存，在GPU内存（通常被称为显存）中储存大量顶点。

OpenGL有很多缓冲对象类型，顶点缓冲对象的缓冲类型是GL\_ARRAY\_BUFFER。OpenGL允许我们同时绑定多个缓冲，只要它们是不同的缓冲类型。我们可以使用glBindBuffer函数把新创建的缓冲绑定到GL\_ARRAY\_BUFFER目标上。

调用glBufferData函数，它会把之前定义的顶点数据复制到缓冲的内存中

第一件事是用着色器语言GLSL(OpenGL Shading Language)编写顶点着色器，然后编译这个着色器。

向量

在图形编程中我们经常会使用向量这个数学概念，因为它简明地表达了任意空间中的位置和方向，并且它有非常有用的数学属性。在GLSL中一个向量有最多4个分量，每个分量值都代表空间中的一个坐标，它们可以通过vec.x、vec.y、vec.z和vec.w来获取。注意vec.w分量不是用作表达空间中的位置的（我们处理的是3D不是4D），而是用在所谓透视除法(Perspective Division)上。

着色器程序对象(Shader Program Object)

着色器程序对象(Shader Program Object)是多个着色器合并之后并最终链接完成的版本。如果要使用刚才编译的着色器我们必须把它们链接(Link)为一个着色器程序对象，然后在渲染对象的时候激活这个着色器程序。已激活着色器程序的着色器将在我们发送渲染调用的时候被使用。

顶点缓冲数据会被解析

位置数据被储存为32位（4字节）浮点值。

每个位置包含3个这样的值。

在这3个值之间没有空隙（或其他值）。这几个值在数组中紧密排列(Tightly Packed)。

数据中第一个值在缓冲开始的位置。

索引缓冲对象(Element Buffer Object)：

一个存储索引供索引化绘制使用的缓冲对象。

glVertexAttribPointer函数

第五个参数 步长 第六个参数 （最后一个）的类型是void\*，所以需要我们进行这个奇怪的强制类型转换。它表示位置数据在缓冲中起始位置的偏移量(Offset)。

每个顶点属性从一个VBO管理的内存中获得它的数据，而具体是从哪个VBO（程序中可以有多个VBO）获取则是通过在调用glVetexAttribPointer时绑定到GL\_ARRAY\_BUFFER的VBO决定的。由于在调用glVetexAttribPointer之前绑定的是先前定义的VBO对象，顶点属性0现在会链接到它的顶点数据。

顶点数组对象(Vertex Array Object, VAO)

存储缓冲区和顶点属性状态。

可以像顶点缓冲对象那样被绑定，任何随后的顶点属性调用都会储存在这个VAO中。这样的好处就是，当配置顶点属性指针时，你只需要将那些调用执行一次，之后再绘制物体的时候只需要绑定相应的VAO就行了。这使在不同顶点数据和属性配置之间切换变得非常简单，只需要绑定不同的VAO就行了。glEnableVertexAttribArray和glDisableVertexAttribArray的调用。

通过glVertexAttribPointer设置的顶点属性配置。

通过glVertexAttribPointer调用与顶点属性关联的顶点缓冲对象。

和顶点缓冲对象一样，EBO也是一个缓冲，它专门储存索引，OpenGL调用这些顶点的索引来决定该绘制哪个顶点。所谓的索引绘制(Indexed Drawing)正是顶点重复问题的解决方案。

线框模式(Wireframe Mode)

要想用线框模式绘制你的三角形，你可以通过glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE)函数配置OpenGL如何绘制图元。第一个参数表示我们打算将其应用到所有的三角形的正面和背面，第二个参数告诉我们用线来绘制。之后的绘制调用会一直以线框模式绘制三角形，直到我们用glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL)将其设置回默认模式。

uniform

Uniform是一种从CPU中的应用向GPU中的着色器发送数据的方式，一个特殊类型的GLSL变量。它是全局的（在一个着色器程序中每一个着色器都能够访问uniform变量），并且只需要被设定一次。但uniform和顶点属性有些不同。首先，uniform是全局的(Global)。全局意味着uniform变量必须在每个着色器程序对象中都是独一无二的，而且它可以被着色器程序的任意着色器在任意阶段访问。第二，无论你把uniform值设置成什么，uniform会一直保存它们的数据，直到它们被重置或更新。

我们可以在一个着色器中添加uniform关键字至类型和变量名前来声明一个GLSL的uniform。从此处开始我们就可以在着色器中使用新声明的uniform了。

片段插值

这个图片可能不是你所期望的那种，因为我们只提供了3个颜色，而不是我们现在看到的大调色板。这是在片段着色器中进行的所谓片段插值(Fragment Interpolation)的结果。当渲染一个三角形时，光栅化(Rasterization)阶段通常会造成比原指定顶点更多的片段。光栅会根据每个片段在三角形形状上所处相对位置决定这些片段的位置。  
基于这些位置，它会插值(Interpolate)所有片段着色器的输入变量。比如说，我们有一个线段，上面的端点是绿色的，下面的端点是蓝色的。如果一个片段着色器在线段的70%的位置运行，它的颜色输入属性就会是一个绿色和蓝色的线性结合；更精确地说就是30%蓝 + 70%绿。

着色器类

把着色器类全部放在在头文件里，主要是为了学习用途，当然也方便移植。着色器类储存了着色器程序的ID。它的构造器需要顶点和片段着色器源代码的文件路径，这样我们就可以把源码的文本文件储存在硬盘上了。除此之外，为了让我们的生活更轻松一点，还加入了一些工具函数：use用来激活着色器程序，所有的set…函数能够查询一个unform的位置值并设置它的值。

纹理

一种包裹着物体的特殊类型图像，给物体精细的视觉效果。为了能够把纹理映射(Map)到三角形上，我们需要指定三角形的每个顶点各自对应纹理的哪个部分。这样每个顶点就会关联着一个纹理坐标(Texture Coordinate)，用来标明该从纹理图像的哪个部分采样（译注：采集片段颜色）。之后在图形的其它片段上进行片段插值(Fragment Interpolation)。

纹理坐标在x和y轴上，范围为0到1之间（注意我们使用的是2D纹理图像）。使用纹理坐标获取纹理颜色叫做采样(Sampling)。纹理坐标起始于(0, 0)，也就是纹理图片的左下角，终始于(1, 1)，即纹理图片的右上角。

纹理环绕方式：纹理缠绕(Texture Wrapping)： 定义了一种当纹理顶点超出范围(0, 1)时指定OpenGL如何采样纹理的模式。

| 环绕方式 | 描述 |
| --- | --- |
| GL\_REPEAT | 对纹理的默认行为。重复纹理图像。 |
| GL\_MIRRORED\_REPEAT | 和GL\_REPEAT一样，但每次重复图片是镜像放置的。 |
| GL\_CLAMP\_TO\_EDGE | 纹理坐标会被约束在0到1之间，超出的部分会重复纹理坐标的边缘，产生一种边缘被拉伸的效果。 |
| GL\_CLAMP\_TO\_BORDER | 超出的坐标为用户指定的边缘颜色。 |

纹理过滤

GL\_NEAREST（也叫邻近过滤，Nearest Neighbor Filtering）是OpenGL默认的纹理过滤方式。当设置为GL\_NEAREST的时候，OpenGL会选择中心点最接近纹理坐标的那个像素。纹理过滤(Texture Filtering)： 定义了一种当有多种纹素选择时指定OpenGL如何采样纹理的模式。这通常在纹理被放大情况下发生。

GL\_LINEAR（也叫线性过滤，(Bi)linear Filtering）它会基于纹理坐标附近的纹理像素，计算出一个插值，近似出这些纹理像素之间的颜色。一个纹理像素的中心距离纹理坐标越近，那么这个纹理像素的颜色对最终的样本颜色的贡献越大。

当进行放大(Magnify)和缩小(Minify)操作的时候可以设置纹理过滤的选项，比如你可以在纹理被缩小的时候使用邻近过滤，被放大时使用线性过滤。

多级渐远纹理(Mipmap)

GL\_NEAREST\_MIPMAP\_NEAREST 使用最邻近的多级渐远纹理来匹配像素大小，并使用邻近插值进行纹理采样

GL\_LINEAR\_MIPMAP\_NEAREST 使用最邻近的多级渐远纹理级别，并使用线性插值进行采样

GL\_NEAREST\_MIPMAP\_LINEAR 在两个最匹配像素大小的多级渐远纹理之间进行线性插值，使用邻近插值进行采样

GL\_LINEAR\_MIPMAP\_LINEAR 在两个邻近的多级渐远纹理之间使用线性插值，并使用线性插值进行采样

stb\_image.h是Sean Barrett的一个非常流行的单头文件图像加载库，它能够加载大部分流行的文件格式，并且能够很简单得整合到你的工程之中。

多级渐远纹理(Mipmaps)： 被存储的材质的一些缩小版本，根据距观察者的距离会使用材质的合适大小。

stb\_image.h： 图像加载库。

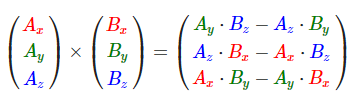
片段着色器也应该能访问纹理对象，但是我们怎样能把纹理对象传给片段着色器呢？GLSL有一个供纹理对象使用的内建数据类型，叫做采样器(Sampler)，它以纹理类型作为后缀，比如sampler1D、sampler3D，或在我们的例子中的sampler2D。我们可以简单声明一个uniform sampler2D把一个纹理添加到片段着色器中，稍后我们会把纹理赋值给这个uniform。

一个纹理的位置值通常称为一个纹理单元(Texture Unit)。一个纹理的默认纹理单元是0，它是默认的激活纹理单元。

纹理单元的主要目的是让我们在着色器中可以使用多于一个的纹理。通过把纹理单元赋值给采样器，我们可以一次绑定多个纹理，只要我们首先激活对应的纹理单元。

叉乘

叉乘只在3D空间中有定义，它需要两个不平行向量作为输入，生成一个正交于两个输入向量的第三个向量。



GLM是OpenGL Mathematics的缩写，它是一个只有头文件的库，也就是说我们只需包含对应的头文件就行了，不用链接和编译。

坐标系统：

局部空间(Local Space，或者称为物体空间(Object Space))

局部空间是指物体所在的坐标空间，即对象最开始所在的地方。

世界空间(World Space)

指顶点相对于（游戏）世界的坐标。

观察空间(View Space，或者称为视觉空间(Eye Space))

将世界空间坐标转化为用户视野前方的坐标

裁剪空间(Clip Space)

在一个顶点着色器运行的最后，OpenGL期望所有的坐标都能落在一个特定的范围内，且任何在这个范围之外的点都应该被裁剪掉(Clipped)。被裁剪掉的坐标就会被忽略，所以剩下的坐标就将变为屏幕上可见的片段。

屏幕空间(Screen Space)

为了将坐标从一个坐标系变换到另一个坐标系，我们需要用到几个变换矩阵，最重要的几个分别是模型(Model)、观察(View)、投影(Projection)三个矩阵。我们的顶点坐标起始于局部空间(Local Space)，在这里它称为局部坐标(Local Coordinate)，它在之后会变为世界坐标(World Coordinate)，观察坐标(View Coordinate)，裁剪坐标(Clip Coordinate)，并最后以屏幕坐标(Screen Coordinate)的形式结束。

为了将顶点坐标从观察变换到裁剪空间，我们需要定义一个投影矩阵(Projection Matrix)，它指定了一个范围的坐标，比如在每个维度上的-1000到1000。投影矩阵接着会将在这个指定的范围内的坐标变换为标准化设备坐标的范围(-1.0, 1.0)。所有在范围外的坐标不会被映射到在-1.0到1.0的范围之间，所以会被裁剪掉。在上面这个投影矩阵所指定的范围内，坐标(1250, 500, 750)将是不可见的，这是由于它的x坐标超出了范围，它被转化为一个大于1.0的标准化设备坐标，所以被裁剪掉了。

如果只是图元(Primitive)，例如三角形，的一部分超出了裁剪体积(Clipping Volume)，则OpenGL会重新构建这个三角形为一个或多个三角形让其能够适合这个裁剪范围。

纹理单元(Texture Units)： 通过绑定纹理到不同纹理单元从而允许多个纹理在同一对象上渲染。

向量(Vector)： 一个定义了在空间中方向和/或位置的数学实体。

矩阵(Matrix)： 一个矩形阵列的数学表达式。

GLM： 一个为OpenGL打造的数学库。

局部空间(Local Space)： 一个物体的初始空间。所有的坐标都是相对于物体的原点的。

世界空间(World Space)： 所有的坐标都相对于全局原点。

观察空间(View Space)： 所有的坐标都是从摄像机的视角观察的。

裁剪空间(Clip Space)： 所有的坐标都是从摄像机视角观察的，但是该空间应用了投影。这个空间应该是一个顶点坐标最终的空间，作为顶点着色器的输出。OpenGL负责处理剩下的事情（裁剪/透视除法）。

屏幕空间(Screen Space)： 所有的坐标都由屏幕视角来观察。坐标的范围是从0到屏幕的宽/高。

LookAt矩阵： 一种特殊类型的观察矩阵，它创建了一个坐标系，其中所有坐标都根据从一个位置正在观察目标的用户旋转或者平移。

欧拉角(Euler Angles)： 被定义为偏航角(Yaw)，俯仰角(Pitch)，和滚转角(Roll)从而允许我们通过这三个值构造任何3D方向。

**1.VBO/VAO**（顶点缓冲区对象或顶点数组对象）:

VBO/VAO（到底是啥，下回讲解）是cpu提供给GPU的顶点信息，包括了顶点的位置、颜色（只是顶点的颜色，和纹理的颜色无关）、纹理坐标（用于纹理贴图）等顶点信息。

**2.VertexShader**（顶点着色器）：

顶点着色器是处理VBO/VAO提供的顶点信息的程序。VBO/VAO提供的每个顶点都执行一遍顶点着色器。Uniforms（一种变量类型）在每个顶点保持一致，Attribute每个顶点都不同（可以理解为输入顶点属性）

**3.PrimitiveAssembly**（图元装配）：

顶点着色器下一个阶段是图元装配，图元（prmitive）是三角形、直线或者点精灵等几何对象。这个阶段，把顶点着色器输出的顶点组合成图元。

**4.rasterization（光栅化）：**

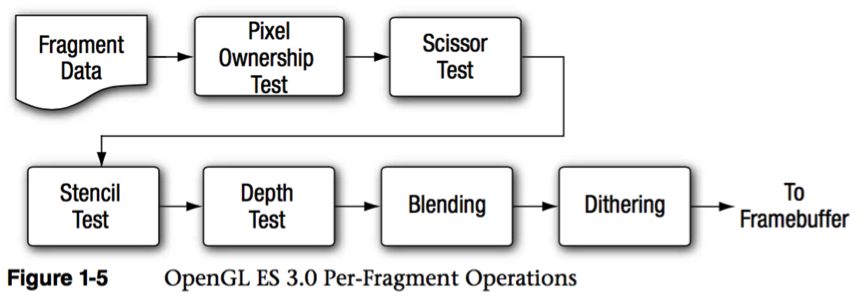
光栅化是将图元转化为一组二维片段的过程，然后，这些片段由片段着色器处理（片段着色器的输入）。这些二维片段代表着可在屏幕上绘制的像素。用于从分配给每个图元顶点的顶点着色器输出生成每个片段值的机制称作插值（Interpolation）。这句不是人话的话解释了一个问题，就是从cpu提供的分散的顶点信息是如何变成屏幕上密集的像素的，图元装配后顶点可以理解成变为图形，光栅化时可以根据图形的形状，插值出那个图形区域的像素（纹理坐标v\_texCoord、颜色等信息）。注意，此时的像素并不是屏幕上的像素，是不带有颜色的。接下来的片段着色器完成上色的工作。

**5**.**FragmentShader（片段着色器）：**

片段着色器为片段（像素）上的操作实现了通用的可编程方法，光栅化输出的每个片段都执行一遍片段着色器，对光栅化阶段生成每个片段执行这个着色器，生成一个或多个（多重渲染）颜色值作为输出。

**6.Per-Fragment Operations（逐片段操作）**

在此阶段，每个片段上执行如下功能:



**（1）pixelOwnershipTest（像素归属测试）：**

这个用来确定帧缓冲区中位置（x，y）的像素是不是归当前上下文所有。例如，如果一个显示帧缓冲区窗口被另一个窗口所遮蔽，则窗口系统可以确定被遮蔽的像素不属于此opengl的上下文，从而不显示这些像素。

**（2）ScissorTest（剪裁测试）：**

如果该片段位于剪裁区域外，则被抛弃

**（3）StencilTest and DepthTest（模板和深度测试）：**

深度测试比较好理解，若片段着色器返回的深度小于缓冲区中的深度，则舍弃。模板测试没有用过，不清楚具体功能，猜测功能应该和名字一样，模板形状内可通过。

**（4）Blending（混合）：**

将新生成的片段颜色值与保存在帧缓冲区的颜色值组合起来，产生新的RGBA。

**（5）dithering（抖动）：**

不知道这个是神马作用？

**最后把产生的片段放到帧缓冲区（前缓冲区或后缓冲区或FBO）中，若不是FBO，则屏幕绘制缓冲区中的片段，产生屏幕上的像素。**