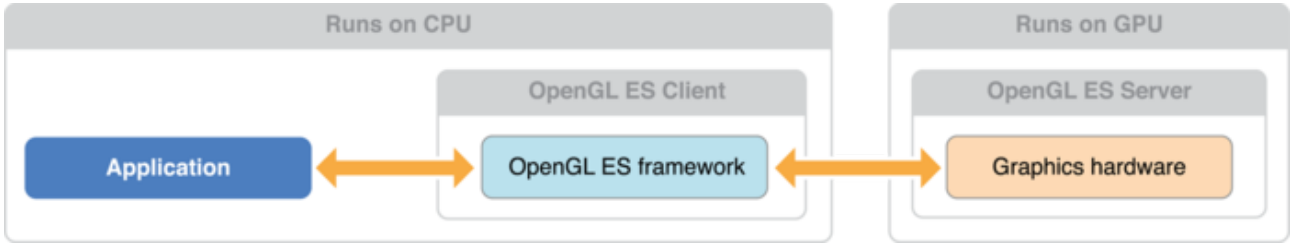
OpenGL ES For iOS

OpenGL是什么?

一般认为OpenGL是一个API，包含一系列可以操作图形、图像的函数。实际上，OpenGL是一个由“Khronos组织”制定并维护的规范。
OpenGL规范严格规定了每个函数该如何执行以及它们的输出值，至于内部实现由OpenGL库的开发者（通常是显卡生产厂商）自行决定。

OpenGL是如何工作的?

C/S架构



CPU、GPU之间的关系，类似于C/S模式，其中CPU相当于Client，GPU相当于Server，当应用需要图形的绘制以及渲染时，它会通过OpenGL ES框架，向GPU传输相应绘制指令（类似于client发起的请求）， GPU收到指令后会进行相应操作（类似于Server对请求进行响应），最终将渲染结果在屏幕上绘制出来。数据在CPU和GPU之间传递是耗时操作。

渲染管线

渲染管线接收一组3D坐标，然后把它们转变为在屏幕上显示的2D像素。

什么是管线?

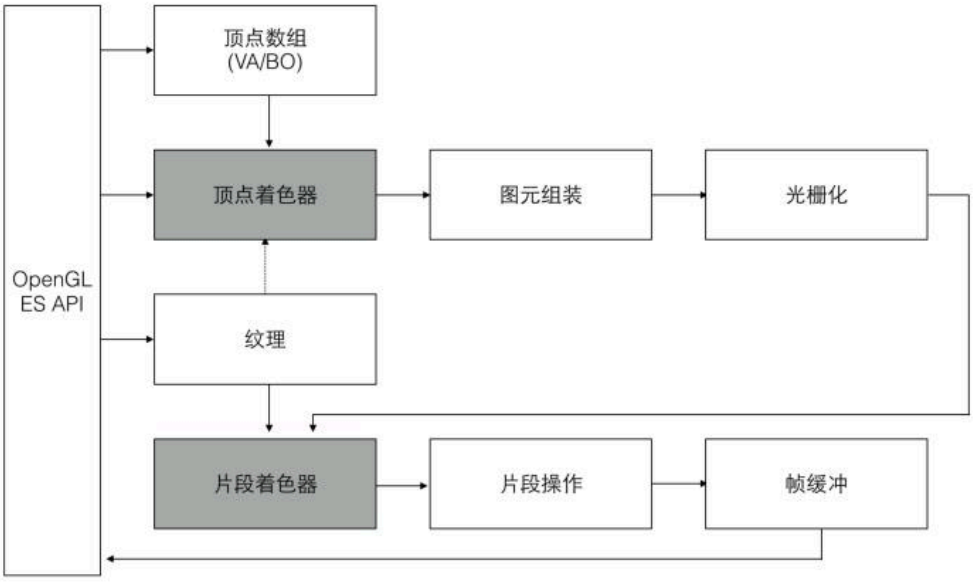
管线（渲染管线）即生产流水线，OpenGL按流水线形式工作，一个环节接着一个环节执行，上一个环节的输出是下一个环节的输入。

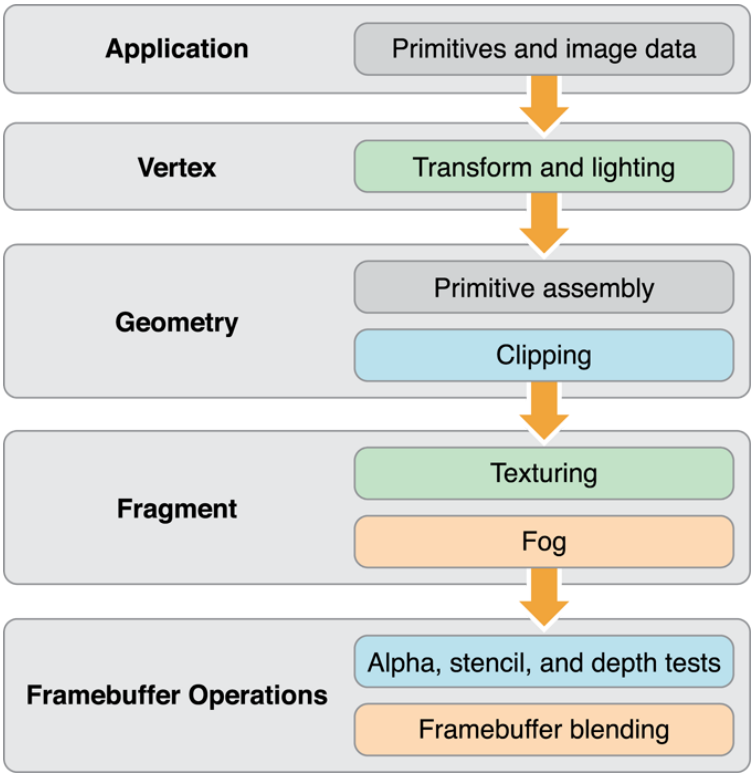
固定管线（立即渲染模式）：OpenGL3.2版本以前支持，3.2版本被废弃，OpenGL ES 2.0不再支持。容易使用，易于学习。低效。知道已被废弃即可。
可编程管线（核心模式）：灵活性、效率；难于学习；更深入的理解图形编程

顶点：一个顶点是一个3D坐标数据（位置、颜色、纹理坐标等）。顶点数据是一系列顶点的集合。此处的3D坐标不是指3维空间坐标?

图元：为了让OpenGL知道我们的坐标和颜色构成的是什么，我们需要指定这些数据所表示的渲染类型，比如一系列的点、一系列的三角形还是一条线，做出这些提示的叫做图元，每个图元包含一个或多个顶点。OpenGL支持三种图元：点、线、三角形。
注意：三角形顶点连线时按照统一（逆时针）顺序，因为面剔除时按连线顺序分辨正面和背面。

渲染管线的各个阶段





顶点着色器：
顶点着色器的主要工作是执行坐标转换，同时也可以对顶点做一些基本处理。

图元装配：
把顶点组装成图形，丢弃不在视锥体内的图元，剪裁与视锥体边界相交的图元等

几何着色器：（可选，OpenGL ES裁剪了这个）
它可以产生新的顶点构造出新的图元也可以减少顶点裁剪图元。

光栅化（像素化）：
把矢量图形转化成像素点

片段着色器：
主要目的是计算像素的最终颜色
片段着色器包含3D场景的数据（比如光照、阴影、光的颜色等），这些数据可以被用来计算最终像素的颜色

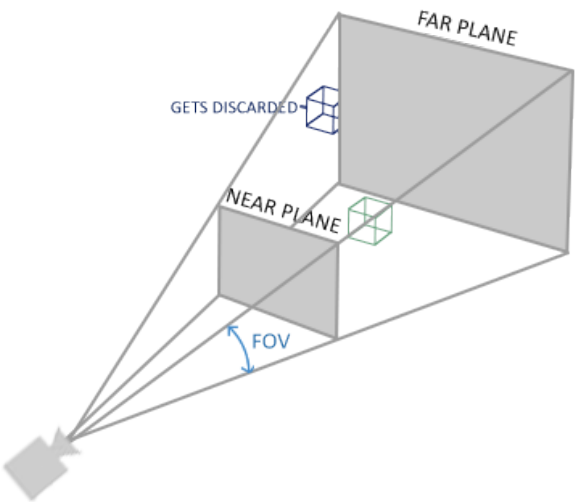
alpha测试和混合：
最终确定用于渲染的像素（根据alpha值混合、检测遮挡等等）

几个概念

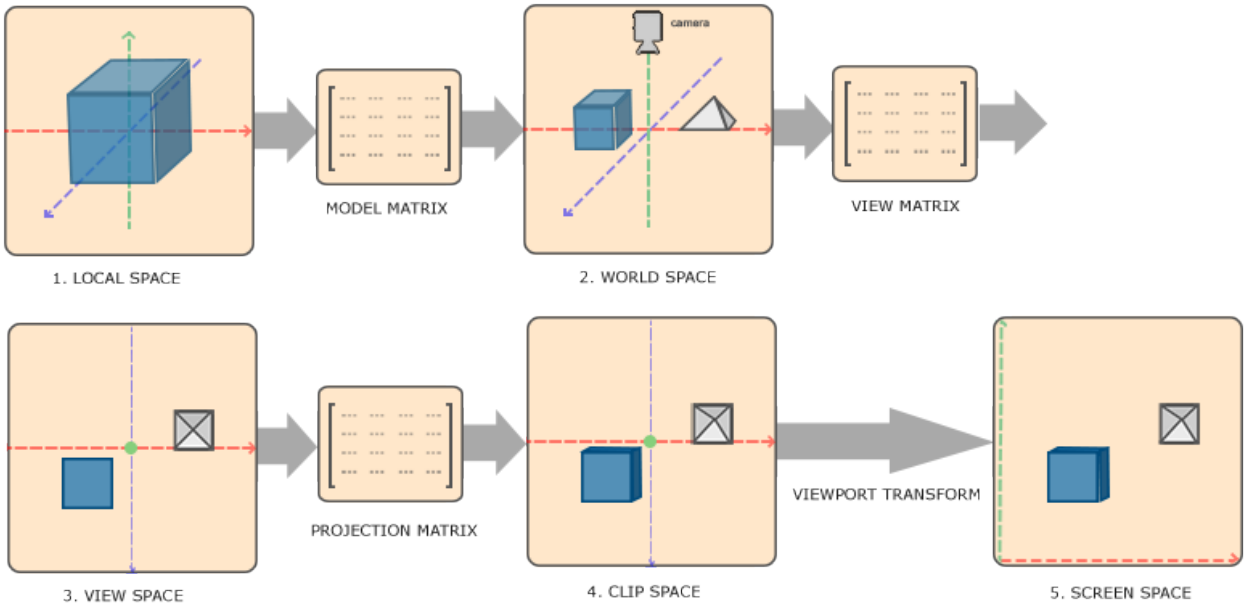
纹理坐标：
纹理即一张2D图片（也可以是大量数据，但这不在讨论范围内）。要把纹理映射到图元上，需要制定图元各个顶点对应纹理的哪个部分，即给每个顶点关联一个纹理坐标，纹理坐标通常在0~1之间，（0，0）左下角，（1，1）右上角，超出这个范围后OpenGL默认行为是重复这个纹理图像（这个默认行为可以更改）

framebuffer(FBO): frame buffer对象实际上不是一个缓冲区，而是一个包含多个附件的聚合对象，其附件是实际的缓冲区。纹理或RenderBuffer可以作为其附件。
RenderBuffer：是一个实际的缓冲区。

视锥体（平截头体）：远平面、近平面、视角边界组成的空间



坐标系



物体坐标系（局部坐标系）：是以物体某一点为原点而建立的“坐标系”，该坐标系仅对该物体适用，用来简化对物体各部分坐标的描述。物体放到场景中时，各部分经历的坐标变换相同，相对位置不变，所以可视为一个整体，与人类的思维习惯一致。

世界坐标系：是OpenGL中用来描述场景的坐标，Z+轴垂直屏幕向外，X+从左到右，Y+轴从下到上，是右手笛卡尔坐标系统

眼坐标系（观察坐标系）：是以视点为原点，以视线的方向为Z+轴正方向的坐标系。OpenGL管道会将世界坐标先变换到眼坐标，然后进行裁剪，只有在视线范围(视见体)之内的场景才会进入下一阶段的计算。

裁剪空间（裁剪坐标系）：OpenGL期望所有的坐标都能落在一个给定的范围内，且任何在这个范围之外的点都应该被裁剪掉，这个给定的范围就是裁剪空间。被裁剪掉的坐标就被忽略了，所以剩下的坐标就将变为屏幕上可见的片段。

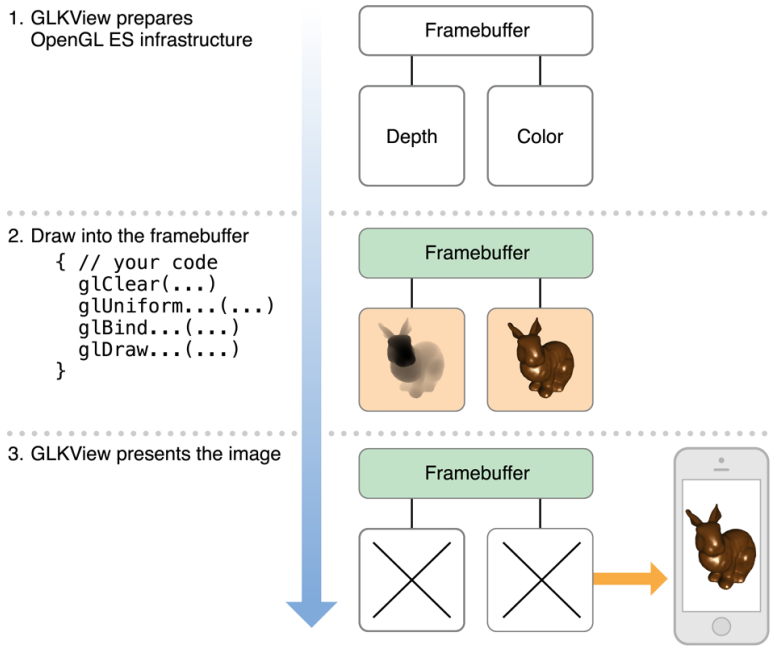
屏幕坐标系（视口坐标系）：屏幕左下角坐标为（-1，-1），右上角坐标为（1,1）。我们用这个坐标系来描述物体及光源的位置。

状态机

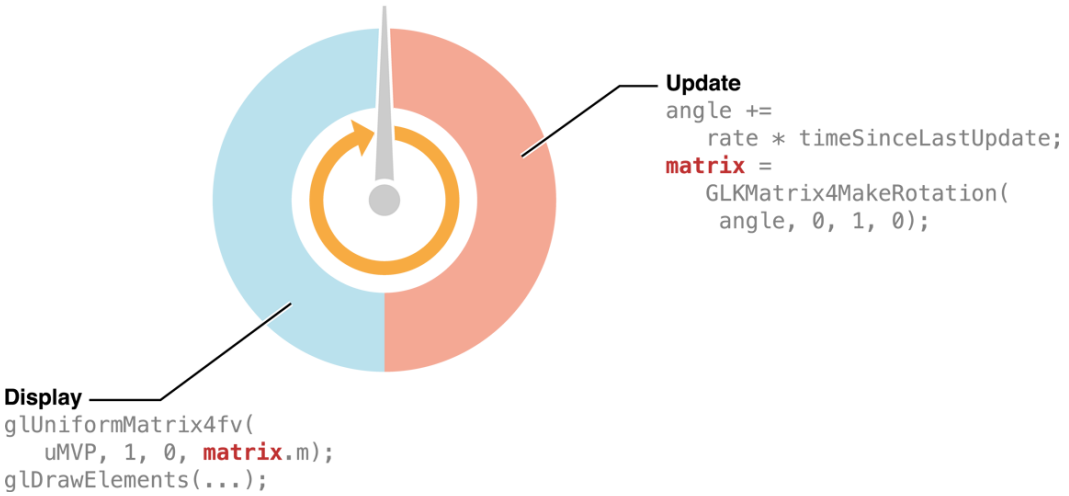
OpenGL是一个状态机，即一系列的变量描述OpenGL此刻该如何运行。OpenGL的状态称为OpenGL的上下文。
在对OpenGL做任何操作之前，应该先设置好当前上下文，所有OpenGL方法最终都是作用到这个上下文上。

GLKit

GLKView管理OpenGL ES的基础设施，用于绘制（继承、代理）
使用OpenGL渲染一个图像分三个部分：1、构建资源；2、绘制；3、把绘制结果显示出来。GLKView封装了第一步和第三步。



GLKViewController管理渲染循环(update + display)（使用CADisplayLink，目的是为了让CPU、GPU并行）
update阶段计算渲染所需数据，如根据 timeSinceLastUpdate 计算物体当前位置
display阶段渲染。为了最佳性能，应用应该先设置OpenGL的各个对象，后提交绘制命令



GLKBaseEffect一个简单的光照、着色系统，可以配置光照、纹理、材质等。省去了我们写着色器的工作

OpenGL缓冲区对象之VBO

简称：

顶点缓冲对象：Vertex Buffer Object, VBO
索引缓冲对象：Element Buffer Object, EBO　　(Index Buffer Object, IBO)

简介：

VBO是OpenGL提供的一种特性，主要是用于在非立即模式下用来保存顶点数据（包括位置、纹理、颜色等），同时提供了更新这些数据的方法。 VBO相比较立即模式的渲染来说效率更高，这主要是因为VBO的数据一般会放在显存中而不是内存中。通俗点说VBO就好像是显卡中开辟的一块存储区域，用来把以前放在内存中的数据放在了显存中，便于更加方便的传输处理。 VBO特性是在OpenGL1.5版本引入的。

相关函数：

glGenBuffers	创建顶点缓冲区对象
glBindBuffer	将顶点缓冲区对象设置为当前数组缓冲区对象(array buffer object)或当前元素(索引）缓冲区对象(element buffer object)
glBufferData	为顶点缓冲区对象申请内存空间，并进行初始化(视传入的参数而定)
glBufferSubData	初始化或更新顶点缓冲区对象
glDeleteBuffers	初始化或更新顶点缓冲区对象

创建VBO：

1. 新建一个新的缓冲区对象（调用 glGenBuffers）
2. 绑定缓冲区对象 （调用glBindBuffer）
3. 拷贝顶点数据到缓冲区对象中（调用glBufferData）

```
void glGenBuffers  (GLsizei n, GLuint * buffers);
```

n:创建多少个缓冲区对象
buffers:接收这些对象ID的地址（如果是多个需要传入数组）

```
void glBindBuffer (GLenum target,GLuint buffer);
```

target:缓冲区是用来存储什么东西的。顶点数组（GL_ARRAY_BUFFER) 、索引数组（GL_ELEMENT_ARRAY_BUFFER)。

注： 所有顶点属性（如顶点坐标值、顶点纹理坐标、顶点法线和颜色）都必须使用GL_ARRAY_BUFFER。索引数组GL_ELEMENT_ARRAY_BUFFER必须配合glDrawElements( )使用。

```
void glBufferData(GLenum  target,GLsizeiptr  size,const GLvoid * data,GLenum usage);
```

target：取值是GL_ARRAY_BUFFER或者是GL_ELEMENT_ARRAY
size：需要写入的数据大小（单位：字节）
data：写入数据的指针，如果data是空，那么VBO仅仅保留size大小的空间
usage：是一种优化的参数

链接顶点属性

```
glVertexAttribPointer(GLuint indx, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid *ptr)
```

在不使用VBO的情况下：ptr就是一个指针，指向的是需要上传到顶点数据指针。通常是数组名的偏移量。

在使用VBO的情况下：首先要glBindBuffer，以后ptr指向的就不是具体的数据了。因为数据已经缓存在缓冲区了。这里的ptr指向的是缓冲区数据的偏移量。这里的偏移量是整型，但是需要强制转换为const GLvoid *类型传入。注意的是，这里的偏移的意思是数据个数总宽度数值。
比如说：这里存放的数据前面有3个float类型数据，那么这里的偏移就是，3*sizeof(float).

工作原理：

1、通过index得到着色器对应的变量openGL会把数据复制给着色器的变量。

2、通过size和type知道当前数据什么类型，有几个。openGL会映射到float，vec2，vec3 。

3、由于每次上传的顶点数据不止一个。那么通过stride就是在数组中间隔多少byte字节拿到下个顶点此类型数据。

4、最后，通过ptr的指针在迭代中获得所有数据。

5、openGL无法通过属性

那么，最最后openGL如何知道ptr指向的数组有多长，读取几次呢。是的，openGL不知道。所以在调用绘制的时候，需要传入一个count数值，就是告诉openGL绘制的时候迭代几次glVertexAttribPointer调用。

glDrawArrays( ) 与glDrawElements( ) 功能与区别

1.共同点：

都是从一个数据数组中提取数据渲染基本图元

2.不同点：

glDrawElements( )能够对数据数组进行随机存取，支持顶点索引机制。

glDrawArrays( )只能按照顺序访问。

注：

顶点索引含义：把输入的顶点坐标值从0开始编号，并在一个单独的无符号类型数组中保存多个索引值组成的图元信息，从而进一步避免了重复指定顶点数据造成的冗余。

绘制模式：

GL_POINTS： 单独的将顶点画出来。

GL_LINES：单独地将直线画出来。行为和 GL_TRIANGLES 类似。

GL_LINE_STRIP：连贯地将直线画出来。行为和 GL_TRIANGLE_STRIP 类似。

GL_LINE_LOOP：连贯地将直线画出来。行为和 GL_LINE_STRIP 类似，但是会自动将最后一个顶点和第一个顶点通过直线连接起来。

GL_TRIANGLES：这个参数意味着OpenGL使用三个顶点来组成图形。所以，在开始的三个顶点，将用顶点1，顶点2，顶点3来组成一个三角形。完成后，在用下一组的三个顶点(顶点4，5，6)来组成三角形，直到数组结束。

GL_TRIANGLE_STRIP： OpenGL的使用将最开始的两个顶点出发，然后遍历每个顶点，这些顶点将使用前2个顶点一起组成一个三角形。如P0，P1，P2这三个点组成一个三角形，P1，P2，P3这三个点也组成一个三角形。

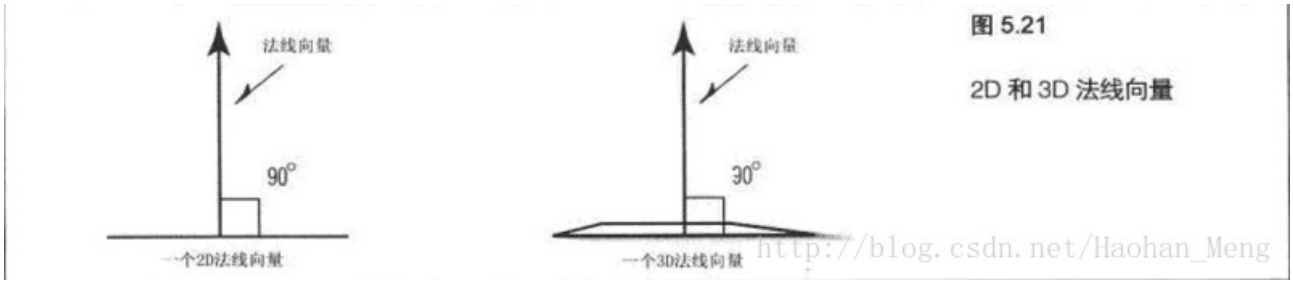
GL_TRIANGLE_FAN - 在跳过开始的2个顶点，然后遍历每个顶点，让OpenGL将这些顶点于它们前一个，以及数组的第一个顶点一起组成一个三角形。 如P2、P1、P0 ; P3、P2、P0 这2个三角形。

法线：

一个多边形被一束来自某个光源的光线所照射，当这束光线照射到多边形的表面时，它与平面会形成一个角度（A）；然后，光线按照一个角度（B）向观察者反射（观察者不一定能够看到它）。通过这些角度，再加上我们之前所讨论的光照参数和材料属性，就可以计算这个位置的外观颜色了

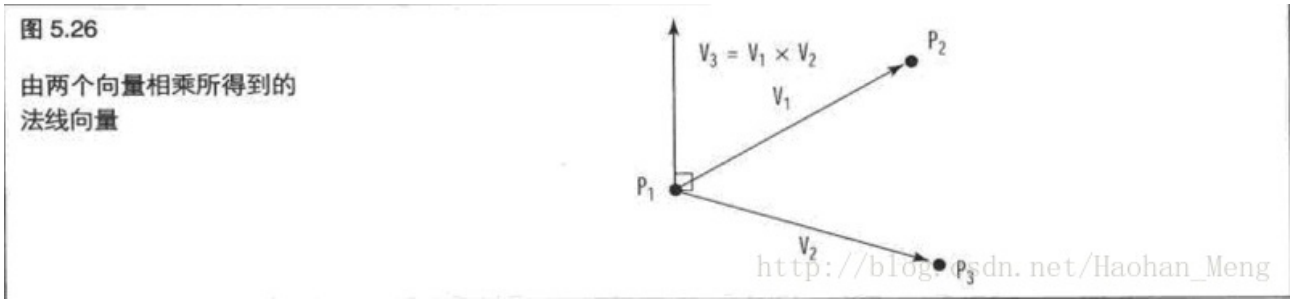
表面法线

在一个假想的平面（或多边形）上，一条垂直向上的向量称为法线向量。实际上，它就是一条指向某个方向的直线（向量），它与多边形的表面呈90度角



生成法线

平面上的3个点P1、P2、P3，我们可以定义两个向量，从P1至p2的向量V1以及从P1至P3的向量V2。从数学的角度而言，三维空间中的两个向量定义了一个平面。我们对这两个向量求叉积（V1×V2），其结果所产生的向量与这个平面垂直，也就是我们所需的法线向量



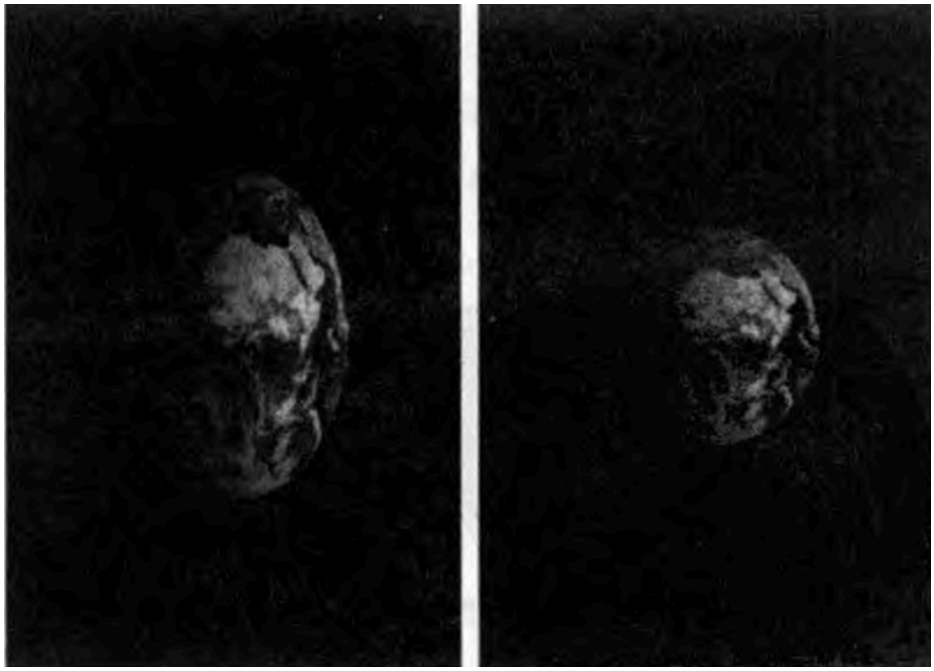
深度渲染缓存（Depth Render Buffer）

图元是按照GPU处理顺序被渲染的，如果没有深度缓存，为最后一个对象的绘制而产生的片元总是会覆盖以前渲染的片元，那么必须小心的控制绘图的顺序。视点变化时，排序三角形经常会导致重写顶点缓存的内容，并且击溃缓存带来的内存访问优化。深度渲染缓存是一个可选的输出缓存，并且与像素颜色渲染缓存相似，几乎所有的OpenGL ES应用都使用深度缓存。每次渲染一个片元时，片元的深度（片元与视点之间的距离）被计算出来并与在深度缓存中的那个片元保存的值进行对比，如果这个片元的深度值更小（更接近视点），那么就用这个片元来替换在像素颜色渲染缓存中的那个片元位置的任何颜色，并用刚刚渲染的片元的深度值来更新深度缓存。如果一个片元的深度值比在深度缓存中保存的值更大，这就意味着某些已经渲染的片元更接近于视点，在这种情况下，新的片元在还没有更新像素颜色渲染缓存的情况下就会被丢弃。

设置方式：drawableDepthFormat：

GLKViewDrawableDepthFormat16 65536个不同的深度
GLKViewDrawableDepthFormat24 1700多万个不同的深度

为什么对称的球形坐标会拉伸：
projectionMatrix（投影矩阵，定义一个正射投影）
modelviewMatrix（坐标系矩阵，定义一个用于控制对象限制位置的坐标系）



变换方式：

平移：GLKMatrix4MakeTranslation

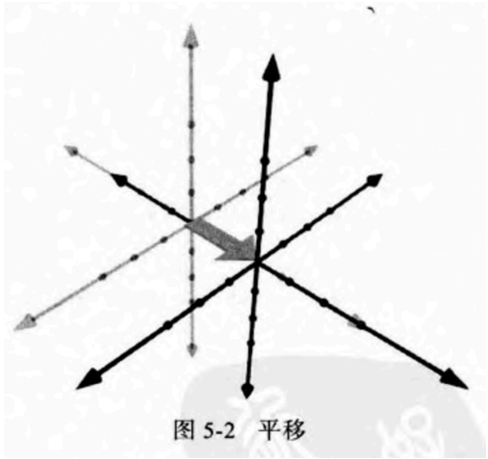


图 5-2 平移

旋转：GLKMatrix4MakeRotation
缩放：GLKMatrix4MakeScale

图 5-4　缩放

透视：GLKMatrix4MakeFrustum

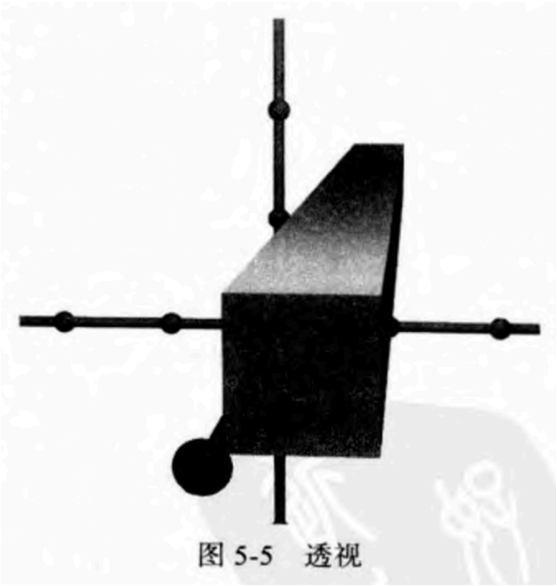


图 5-5　透视

projectionMatrix（投影矩阵）和modelviewMatrix（坐标系矩阵）相互作用关系：

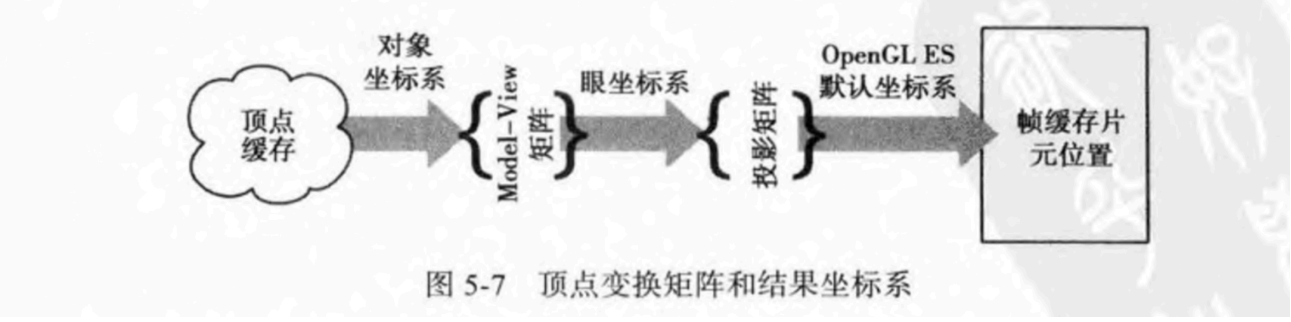


图 5-7　顶点变换矩阵和结果坐标系

GLKMatrixStack

为了通过复制矩阵到一个临时的变量来保存并回复modelviewMatrix，GLKit提供了这样一个方便的数据类型GLKMatrixStackPush，还提供了一个用来向栈数据结构保存矩阵的函数操作集合。

GLKMatrixStackPush()函数会复制最顶部的矩阵并加入到矩阵栈的顶部，GLKit同时为修改矩阵栈顶部的矩阵提供了一系列的操作函数，操作完成后，利用GLKMatrixStackGetMatrix4()获取最顶部的矩阵，GLKMatrixStackPop()函数会移除堆栈最顶部的矩阵，并把前一个顶部矩阵恢复到最顶部的位置。

在iOS APP中每一个线程都有一个当前上下文

准备写一个APP

APP需要做三件事：使用着色器语言编写着色器程序，组织顶点数据和纹理数据，配置OpenGL状态机。

OpenGL不会立即执行每一条命令，而是把命令放到队列中，等必要的时候交由硬件执行。但有些方法会导致命令立刻被执行甚至会阻塞APP运行直到命令执行完毕，这类方法非必要时尽量不要使用。

glFlush把所有命令立刻发送到硬件并阻塞直到所有命令发送完成。
glFinish推送所有命令并阻塞直到所有命令执行完成
从OpenGL取数据的那类方法（如glReadPixels）也会等待所有命令执行完成
命令队列满了会发送所有命令

iOS平台通常只有两个情况需调用glFlush
1、应用进入后台时（后台执行OpenGL命令会被杀死）
2、切换当前上下文时

参考资料

OpenGL教程 https://learnopengl-cn.github.io/

OpenGL ES Programming Guide https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008793-CH1-SW1

OpenGL ES 框架详细解析 https://www.jianshu.com/p/5989b045430f

Khronos 组织规范 https://www.khronos.org/registry/OpenGL-Refpages/es2.0/