# WebGL部分

Cesium实在是憋不出什么好屁了，所以写写webGL的吧。毕竟cesium也是基于webGL的所以写webGL也相当于写cesium了，毕竟一个爹一个儿子都是一家人。

关于webGL的一些知识点是基于有webGL基础知识的基础上写的，如果没有基础看不懂，基础东西没啥营养自己找资料学吧，如果已经掌握了webGL的基础语法那么看完我下面的内容会对webGL会有一个新的更深的理解。

**WebGL核心：渲染管线**

1. **顶点数据：就是我们JS里面写的东西类型化数组，像Unit32Array这种的。放入顶点缓冲区里面。**
2. **顶点着色器：顶点数据放在顶点着色器里面进行逐顶点渲染。**
3. **图元装配：把顶点装配出来。变成一个个三角形。**
4. **光栅化：把图元用光栅填充起来。形成一个个片元。**
5. **片元着色器：给片元填充颜色。**
6. **颜色缓存：把颜色存储起来。**
7. **把我们要绘制 图形画出来。**

**关于webGL画图的逻辑**

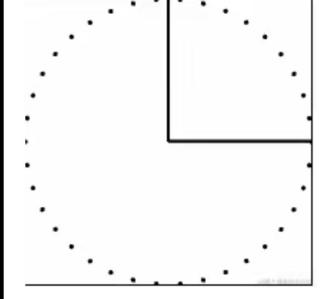
学习一样东西最好的办法就是搞明白他的原理，把复杂的东西用自己的理解吸收掉了，下次再遇到时能够描述出来，这也就是自己学明白了。

在我看来webGL其实就是一个笔在浏览器上画画，仅此而已。决定了哪个像素，什么时候是什么样的颜色，只不过他具体操作的是UV坐标而不是像素，仅此而已。至于怎么画图，画什么图，那就是API做的事了，API也是用来实现原理了，原理懂了API只是一个工具，就像是用笔做题，不需要了解笔是怎么制作的，只需要知道他是铅笔还是碳素笔，就是这样。

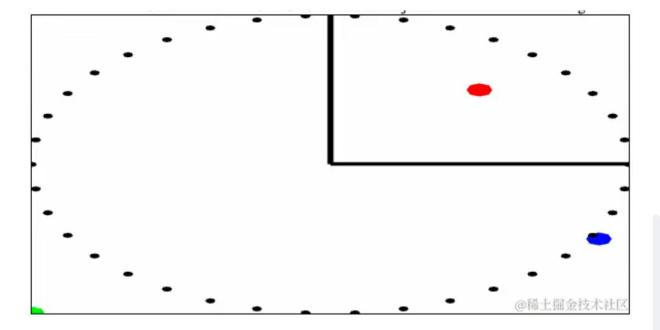
**关于webGL绘图中注意的问题**

上面说了webGL就是在浏览器中画图，不过他不是通过操作像素来实现的而是通过操作坐标来实现的，这也就导致了我们绘制出来的图象与画布大小有关系。

比如说我要画一个圆，如果我们的画布是一个正方形画布我们只要给定坐标位就可以画出来。



但是如果我们的画布是一个长方形，由于我们是根据坐标绘图的，必定造成图型的一个不准确的问题，像下面这样圆变椭圆。



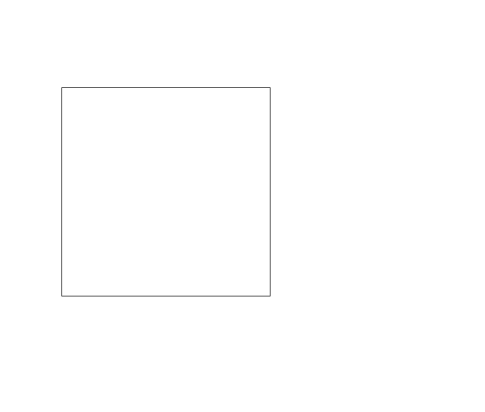
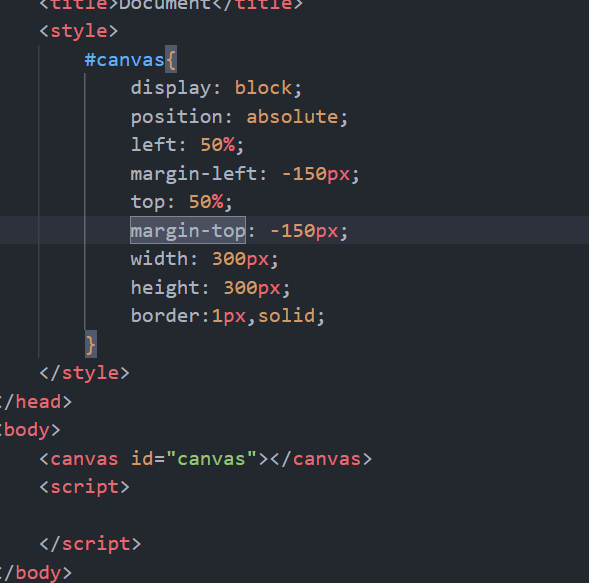
虽然画出来的结果不一样，但是在webGL中执行的代码都是完全一样的，只是我们的画布设置不一样罢了。

**最后注意的一点是关于webGL画布坐标系问题，gl\_Position的二维范围是从-1,1的也就是说左下位-1，-1，右上为1，1**

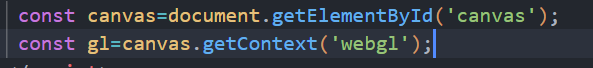
**关于webGL画图的准备**

虽然前面说了要求基础但是还是写一下吧，不然总觉的少点东西。WebGL的画图流程如下：

先准备一个canvas画布放在浏览器上，这相当于有了纸。



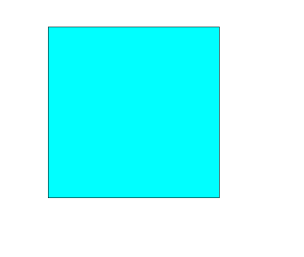
下面我们在canvas上面画画就行了，首先拿到webGL上下文。



拿到了webGL的上下文。下面对gl的上下文操作就可以把JS里的数据传给WebGL，实现GPU编程了。

白色不好看我们刷个底色。





**关于顶点着色器和片元着色器**

顶点着色器（vertex shader）和片元着色器（fragment shader）这两个东西是webGL画图的核心，我们webGL画图就依靠着这两着色器。

**顶点着色器**

顶点着色器负责的就是一个形状的问题，物体什么形状就是由顶点着色器根据buffer里的数据给gl\_position赋值来进行确定的，他也可以用gl\_PointSize来控制顶点的大小，同时他还能把一些buffer里的数据通过varying传给片元着色器。

**片元着色器**

片元着色器是拿到数据之后确定颜色的，正如我们webGL的绘制流程一般先是顶点着色器，然后是图元装配，再是光栅化，最后给片元着色器进行渲染以制作出我们所需要的图形，就是这么简单。

这里我们只讲道理，具体代码自己去学得了，懒得打，就简单描述一下吧

**关于shader代码**，多是以字符串的形式拿到的，初学的时候我们用script标签写然后通过id和innerText获取并放入shader容器里，后期会通过封装XHR来请求shader来实现一个代码隔离的效果。

**创建shader程序：**

我们创建顶点着色器和片元着色器用gl.createShader(type),gl.shaderSource(shader名，shader代码)，gl.compileShader(shader名)这样我们就创建了一个shader，这样的shader我们一个webGL代码里要写两段一段顶点着色器，一段片元着色器。

**创建program程序：**

//一个canvas画布或者说webGL画布中会有多个program。

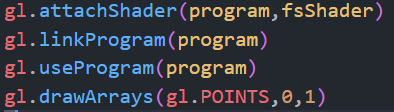
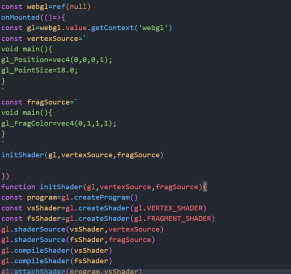
//哪个program被使用了,就会画哪个，如果都用了会画后一个。

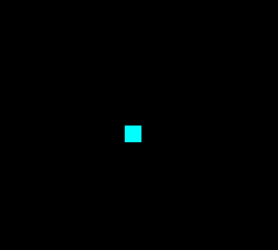
Shader创建完毕后我们要有一个**program**把两个shader结合起来，最后我们实现的webGL的就是这个program。Program的流程是这样的:gl.createProgram(),gl.attachShader(program,shader名）写两遍把顶点片元都加进去，最后gl.LinkProgram然后gl.useProgram()这也就实现了shader在GPU里面挂载。剩下的就是通过JS的代码API往shader里面加东西就行了。

 gl.program = program;

这一步也挺重要的，在我们通过JS传数据的时候会用到。

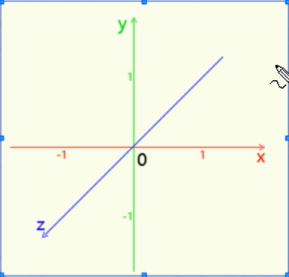
**简单创个点代码和效果如下：**



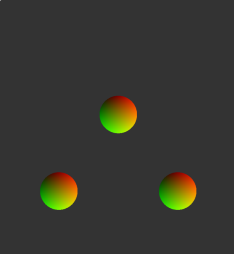
以上就是我们简单在webGL里创建了一个点。

**关于webGL坐标系：**

我们创建的点在0,0,0坐标处。他在屏幕中心。这就是webGL坐标系的圆心。可以理解为webGL的标准设备坐标系是一个按比例设置的相对坐标。不是多少多少px的绝对坐标。而是左右，前后，上下都是-1到1的范围的一个正方体空间投影出来的。

这其实就是裁剪空间，所有超出这个空间的图形都会被裁剪掉。

**下面对基本的API进行一个介绍并加入个人理解：**

这是我用webGL画的三个简单的圆点，我们就结合这个圆来对webGL的API进行一个介绍。

1. **gl\_Position：**这个是我们vertexShader里面的一个核心的东西，他决定了我们的点的位置在-1，-1到1,1这之间的哪个位置。简单说就是决定了点在canvas画布里面的位置。
2. **gl\_PointSize：**这个也是在vertexShader里面的东西，他决定了我们点的大小，比如说我们图上的点大小就是设置的50到150的随机值，我们每次打开点的大小都是不一样的。
3. **gl\_FragColor:**这个决定的是我们渲染的颜色，我们这个点里用的是gl\_PointCoord做的点的颜色。
4. **gl\_PointCoord：**点内部的一个UV左边，自左上0,0至右下1,1。我们把它给gl\_FragColor的前两个参数就实现了这种渐变色的效果。
5. **gl.FragCoord:**这个坐标为画布的宽高，画布多少宽高，这个东西的范围就有多少。
6. **discard:**这个东西是我用来画圆的。我们把gl\_PointCoord和vec2(.5,.5)这个进行一个distance距离计算（这个是shader里的语法，后面会提）如果在.5范围内就渲染，如果不在，我们就discard不让他渲染这样就成了一个圆，这个API更接近shader看不懂学完shader回来就懂了。

**如果我们只通过这些API其实就可以画出一个死的webGL的图形，可以长得和上面那个实例一模一样。但是我们想要改的话只能改着色器，不能通过JS改这样就失去了webGL通过浏览器改shader的目的。所以我们下面就介绍一下如何实现JS传数据，以及相对应的API。**

**存储限定符**

1. **Attribute：**

这个存储限定符是只有顶点着色器内部有的，他负责把JS传递给顶点缓冲区里的数据拿到，并且赋值给gl\_Position或者赋值给varying的参数传给片元着色器。他可以是很长一段数据，通常都是用类型化数组像Float32Array传的，至于为什么不用普通array只能说JS的类型number不支持底层的二进制，也就是说规定不了字节的精度。我们在顶点着色器里基本都是用这个存储限定符，但也不全是用，像mat4这种矩阵就不是用uniform。

1. **Uniform**

这个是顶点着色器和片元着色器都可以用的存储限定符，通常是片元用的多，我们的attribute里面数据是迭代遍历执行的，所以我们只要改了数据执行的逻辑所有的数据迭代时都会执行，不可以改单个数据。Uniform就可以实现改单个数据,根据不同情况来选择。

1. **Varying**

这个存储限定符可以理解为两个着色器之前的桥梁，实现的是两个着色器数据之间的通信。

**存储限定符介绍完了，就要说如何通过JS把数据传给这些存储限定符以实现对shader数据的操作，从而实现GPU的编程**。

**我们JS传的数据只有uniform和attribute两种数据，varying只是一个桥梁。**

**1.getAttribLocation:**gl.getAttribLocation（gl.program,变量在着色器里的名字）他会返回我们这个变量。

**2.gl.vertexAtrribnf:**这个API实现的都是传单一顶点的数据，基本上都不用，因为我们画图不可能只画一个点，这个里面的n可以是1,2,3,4了解一下就行了。

**3.gl.vertexAttribPointer:**gl.vertexAttribPointer(getAttribLocation拿到的变量，几个数据一组，数组是float或者是别的类型数据，偏移量，起始量)。这个通常配合缓冲区来使用。这个API就把我们数组里面的数据都给了着色器，实现了数据的传输，完成了JS操作GPU。

**4.gl.enableVertexAttribArray(getAttribLocation拿到的变量)**

这个配合上面那个使用，只要写了就行了，原理不用多管，知道不写会出错就行了。

1. **gl.createBuffer**

gl.createBuffer()这个API在于创建缓冲区。

1. **gl.bindBuffer**

gl.bindBuffer(gl.ARRAY\_BUFFFER，上面创建的缓冲区)这样我们就实现了绑定缓冲区了。除了ARRAY\_BUFEER外还有索引缓冲区，这里不说，后面会有介绍的。

1. **gl.bufferData**

这个主要是实现赋值的，就像下面一样的写法。



1. **gl.getUniformLocation()**

语法与顶点类似不说了

1. **gl.uniform4fv(上面那个拿到的变量，数据）**

与顶点类似也不说了

1. **gl.drawArray()**

gl.drawArray(绘制的类型，start，count)，从哪组数据开始，绘制几个点。这里注意一下从哪组数据开始和绘制几个点。都是按照上面vertexAttribPointer里面几个数据一组那个属性来决定的。

1. **precision mediump**

这个API是给着色器限制精度的，顶点着色器不加可能不会报错，但是片元着色器不加是一定会报错的。最好都加上精度，int或者是float

**注：这里只写了传递数据，数据操作是着色器代码。通过数据实现炫酷颜色效果为shader内容，此处不解释，此处只写简单画图。**

**webGL基础篇的总结**  
**WebGL能干什么**在canvas上画点，画线画三角形。进而模拟出丰富多彩的2D 和 3D 世界。

**WebGL运行在哪里**运行在GPU上，使用者需要自己提供可以运行的代码传到 GPU。WebGL运行什么形式的代码vertex shader和fragment shader，这两个成对出现，合起来称之为program。

**WebGL可以运行的东西就是program。**用户使用什么语言编写program使用的语言叫做GLShaderLanguage，简称GLSL。

**vertex shader**的功能是什么计算顶点的最终位置(gl\_Position)传递一些数据给fragment shader (例如：buffer中的数据)

**fragment shader**的功能是什么根据顶点的最终位置 (gl\_Position)和其他数据, 最终计算出一个颜色(gl\_FragColor)。然后绘制到canvas。

**用户如何向 program 中传递数据buffer**： buffer 是一个二进制的连续的平坦的一维数组。用户需要在绘制之前，将数据传到GPU中。这部分数据通常包含：顶点坐标、顶点颜色等等。但是实际上，只要你愿意，你可以将任何东西放进来，以备后续之用。指的一提的是，只有vertex shader才能访问 buffer，访问的方式就是 利用attribute 变量去接收。例如： attribute vec2 a\_PointVertex; 。

**varying :** vertex shader 可以向 fragment shader 传递一些数据，传递的方式就是在 两个程序里都声明同名的 varying 变量：例如varying vec3 color; 。 然后在 vertex shader main 函数里，对这个变量进行赋值即可。

**uniform :** buffer 数据通常很大，我们不会经常的去修改 buffer 中的数据。而且 buffer 中的数据一般是逐点的。也就是说，通常情况下，只能全部改掉，而不会发生只改其中一个数据的情况。例如将所有坐标全部变成以前的一半。此时，我们可以利用 uniform变量来完成这个事情,uniform他可以是逐片元的。计算光照我们通常都用uniform传光照方向和法线方向。。uniform 变量是存在于vertex shader 和 fragment shader中的，两者都可以使用。并且，WebGL提供了随时修改 uniform 变量的 api。uniform texture ：这也是一大块需要事先上传到显卡的数据。

刚才说到 buffer 是用来逐点计算的。也就是说，buffer不能随机访问，只能是 WebGL 进行顺序的，迭代访问。那texture的区别就是可以随机访问，原理上讲，你可以像使用数组那样，去根据索引，访问其中的数据。我们一般这样来声明texture: uniform sampler2D u\_texture;。由于大多数情况下，这种类型的变量都是用来进行纹理贴图，所以大部分情况下，我们都是根据一个坐标点，去到这张图中去采样一个颜色。

**GLSL 数据类型**

float：浮点数。

vec2:含有两个float元素的向量类型。x,y分别就是其中的第一个分量和第二个分量。

vec3:同上。第三个分量叫z。

vec4:同上。第四个分量叫w。

mat3:三阶方阵，行数，列数都是三的矩阵。

**js、WebGL、显卡、buffer、vertex shader 、fragment shader、 canvas 等等等等的关系是什么**

js中利用WebGL提供的api，将数据传递到显卡中。显卡中存储用户数据的东西叫buffer（顺序逐点由vertex shader进行访问）。另一种叫texture(随机，采样访问)。vertex shader可以接受buffer中的数据(attribute变量)，进行计算或者传递给fragment shader。fragment shader 可以计算决定最终的颜色是什么。vertex shader和fragment shader 成对出现，合成program。vertex shade可以利用varying变量，往fragment shader传递数据。显卡中可以有多个buffer多个program。uniform属于某一个program，并不是显卡全局可用的。不同的program中可以拥有不同数量，不同类型的uniform。js中可以随时切换program,然后进行绘制。js中可以随时设置某一个program中的uniform变量。js中可以随时切换某一个program中的 buffer来源。（如果你有多个buffer）js中可以执行绘制draw指令,在canvas中画画。等等等等等等。

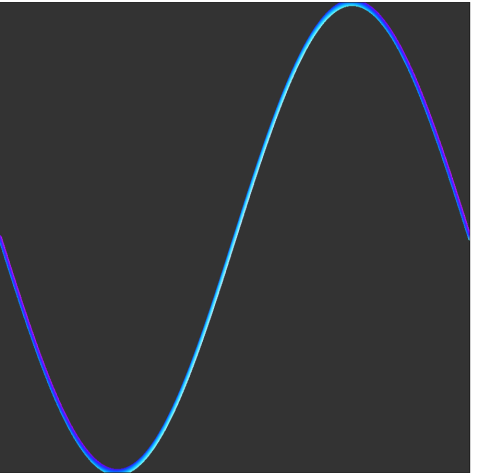
**浅谈一下vertexshader**

Vertexshader顶点着色器，我们画出图型的形状都是他定的。他是通过对buffer的最终计算获得顶点的位置然后赋值给gl\_Position来把我们画图形的形状规定出来，后面的图元装配光栅化与片元着色器的渲染都是得基于这步。

这块我决定用点实例来把这个玩意搞明白。

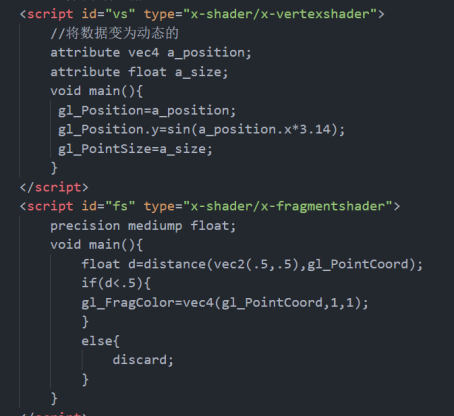
之前在上面写了很多API我们就借助上面的部分API在vertexshader里面画一个正弦函数玩玩吧。。。

先把画出来的正弦函数效果摆出来，有点意思吧。

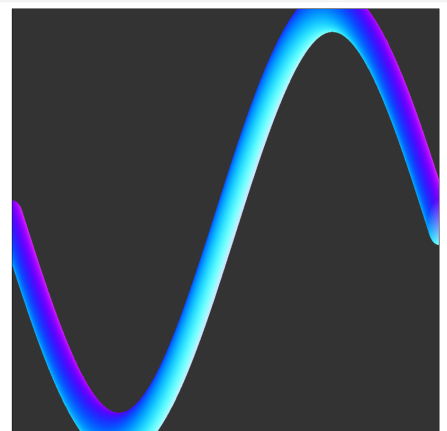
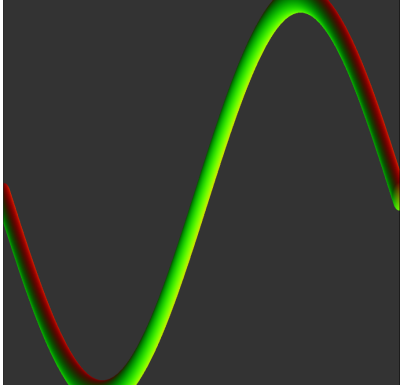
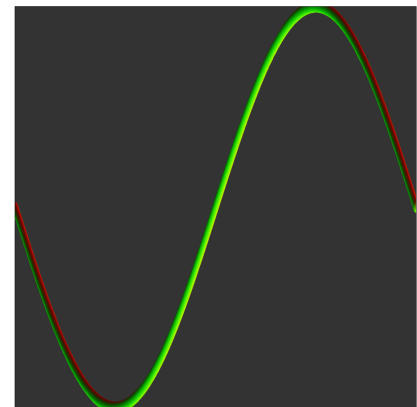
其实就是通过多个点的组合连接来实现的。

Vertexshader代码与JS代码如下。

其核心逻辑就是通过JS创建点的数据，数据变成Float32的数组，放入显卡显存的buffer里面。然后交给vertexshader迭代，shader会在GPU里进行一个计算，多个点数据就会连接成一个图型，而下面的fragmentshader则是控制颜色并执行渲染的工具。



其实如果真的理解了的话还可以给shader传时间参数，让绘制动起来，实现一个逐渐变粗到一定粗度之后停止，然后颜色变化。



效果摆在这里，具体实现代码自己写，如果能写出来也就说明你明白了。

片元着色器涉及GLSL比较多，在GLSL再着重讲解。

**关于向量的问题**

从实际使用的角度来看向量，其实就是一个x,y,z,w四个分量的加减乘除。其中最常用的就是向量的加法以及数乘向量。

比如说我这里有两个向量：向量A【2,3,4,5】和向量B【6,7,8,9】我下面会用这两个向量进行加减乘除的计算。

向量加法：D=A+B=【8,10,12,14】每个对应的分量相加即可。

向量减法：E=B-A=【4,4,4,4】对应的分量相减即可。

向量数乘：G=A\*2=[4,6,8,10]

向量的线性运算：H=x\*A+y\*B，A和B相当于H的一组基地，很好的一个方法，并不一定所有的线性运算都有意义，但特定场景下的向量线性运算是很好用的，很有意义的。

正如向量是一堆数组合在一起。

矩阵其实就是向量的一个组合，下面介绍的就是矩阵。

**关于webGL的MVP矩阵变换**

WebGL里画在canvas画布上的东西都是二维，只是通过了一系列的数学计算形成了三维的效果，最基本的就是通过矩阵变换，来决定物体在画布里的位置，决定以什么视角来看物体，视线的范围等等，这也就是我们MVP矩阵的作用，专业点来说就是实现仿射变换。下面对MVP矩阵进行一个介绍。

**M：Model模型矩阵**

**模型矩阵负责的主要是把物体自身的坐标系转换为世界坐标系。**

平移：在原有的x,y,z,w坐标上加一个u\_offset这种偏移参数就行

x1=x+u\_offset。

缩放：缩放又叫拉伸，在原有的x,y,z基础上乘以一个数值u\_scale就行了

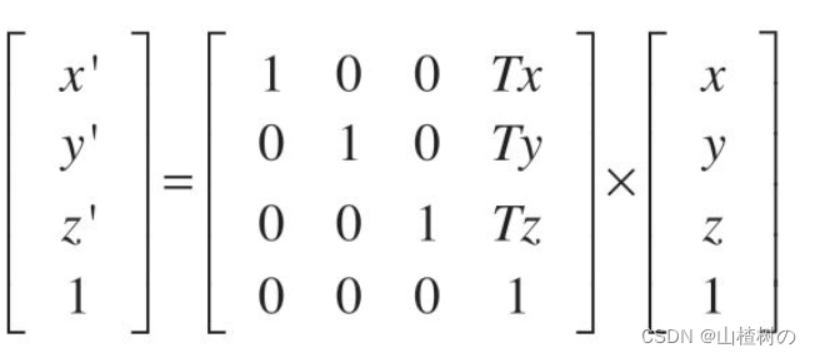
x1=x\*u\_scale。

旋转：x1=x\*cos(u\_rotate)-y\*sin(u\_rotate)

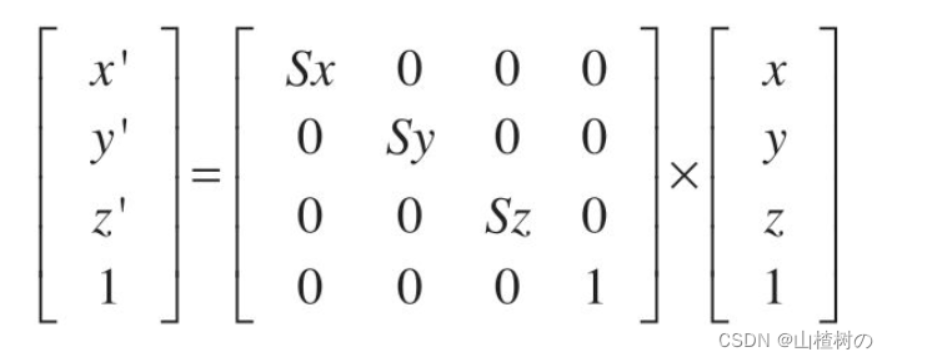
Y1=y\*sin(u\_rotate)+y\*cos(u\_rotate)

**其核心大概是三部分：**这里其实就是一个矩阵计算所以直接把各种矩阵罗列一下。

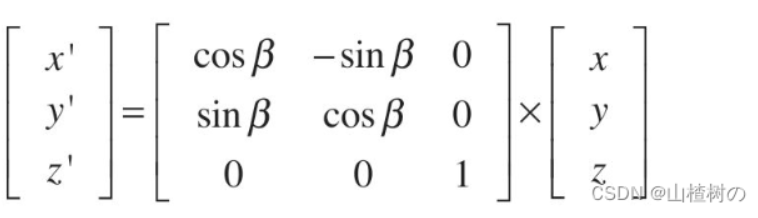
**平移**：这是一个平移矩阵，并且是三维的平移矩阵因为他不止有X,Y，Z还有W



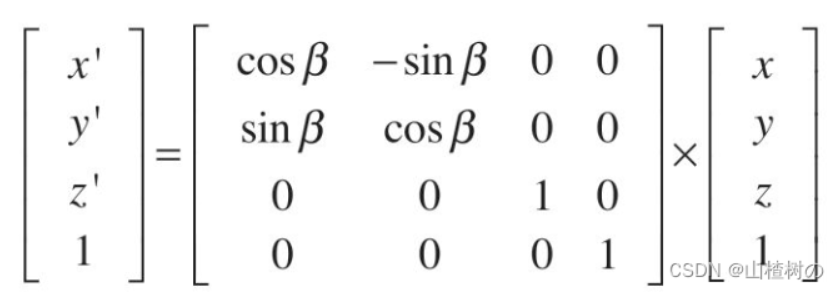
**缩放**：下面这个就是缩放矩阵Sx,Sy,Sz就



**旋转**：旋转矩阵大概就是这样的，不过注意下面这个是一个二维的旋转矩阵，为什么是二维的在下面的齐次坐标会讲到。



三维的旋转矩阵先给出来，防止好奇。这个其实就是绕着z轴转的，因为z处没动。哪个轴数据没变化就是绕着哪个轴转的，至于说w那纯纯是一个齐次坐标，



**V:View视图矩阵**

**视图矩阵主要负责的是把世界坐标系转换为相机坐标系。让我们能以人眼的视角看到他，具体实现在下面THREE.js讲解。（类似于一个camera的作用）**

视图矩阵：他也有三个核心的东西

**视点**：视线的起点，也就是眼睛所在三维空间中的位置（eyeX，eyeY，eyeZ）。

**目标点**：被观察目标所在的点，当确立目标点和视点时，视线方向也随着确立。目标点的坐标用（atX，atY，atZ）表示。

**上方向**：最终绘制在屏幕上的影像中的向上的方向，即正方向。当视线方向确定时，观察者还能够以视线为轴旋转的。当指定上方向时，整个坐标就彻底固定住。上方向是具有3 个分量的矢量（upX，upY，upZ）。

在webgl的默认规则中，**视点位于坐标系统原点（0，0，0），视线为Z轴负方向（屏幕内），上方向为Y轴负方向（向下）。**

通常我们用glMatrix矩阵库来实现这种矩阵的定义，不用自己写。

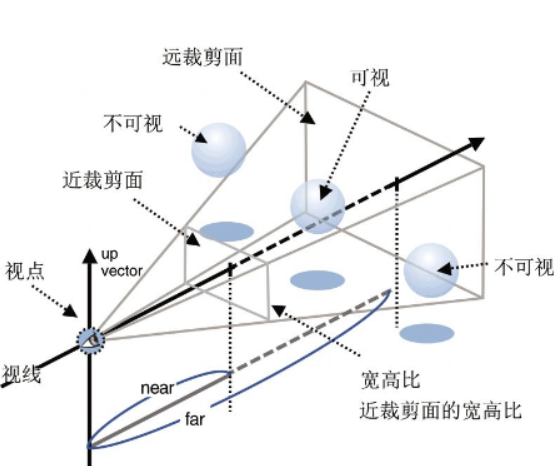
**P:Project投影矩阵**

**投影矩阵主要负责的是把相机坐标系转换到裁剪坐标系里。实现了三维空间到二维空间的投影。得到2维的webGL标准坐标系。**

主要是做了一个裁剪空间。我们常用的是透视投影矩阵，还有一个投影矩阵是正交投影矩阵，是一个盒子，物体不会随着距离导致投影变小。与透视投影相比透视投影更符合人类眼睛所以我们通常都用透视投影。

投影矩阵：他有四个核心的东西

1. 宽高比：近裁剪面的宽高比
2. 近裁剪面：近裁剪面距离视点的距离
3. 远裁剪面：远裁剪面距离视点的距离。
4. Fov垂直视角：也就是上面与下面之间的夹角，必须大于0

**在投影矩阵得到标准坐标系的坐标后我们通过视口变换把标准坐标系坐标转换为设备坐标系。**

**齐次坐标系**

上面的矩阵里不管是二维矩阵还是三维矩阵都多了一个维度，比如二维的会用三维矩阵，三维的会用思维矩阵。之所以多了一个维度是因为其次坐标的关系。

何谓其次坐标，这么说吧。这里有一个向量x=[2,4]。我们想要将这个向量进行一个平移x1=x+u\_offset。X1=[2+u\_offset.x,4+u\_offset.y]这样我们的平移就完成了。如果要缩放就乘以就行了。但是我们通常不喜欢用这种复杂的方法，平移写一次，缩放写一次，旋转还要在写一次，我们通常喜欢把所有的变化都写在一个矩阵里之后x1=x\*矩阵。如果继续按照上面的方法写会发现一个问题。

就是式子只能表示一个效果我已经表示平移了，式子就没办法缩放了，如果硬要表示缩放数据就混了平移和缩放参数会混在一起，这种情况我们选择的是直接升维，升维之后就会发现可以同时实现拉伸和位移了。下面就用式子对比一下来把其次坐标的用途显示出来，下面是一个一维向量的矩阵。

如果我们只用一维矩阵实现平移和缩放。

平移：x1=x+A

那缩放怎么办呢？  
缩放:x2=x1\*B吗？

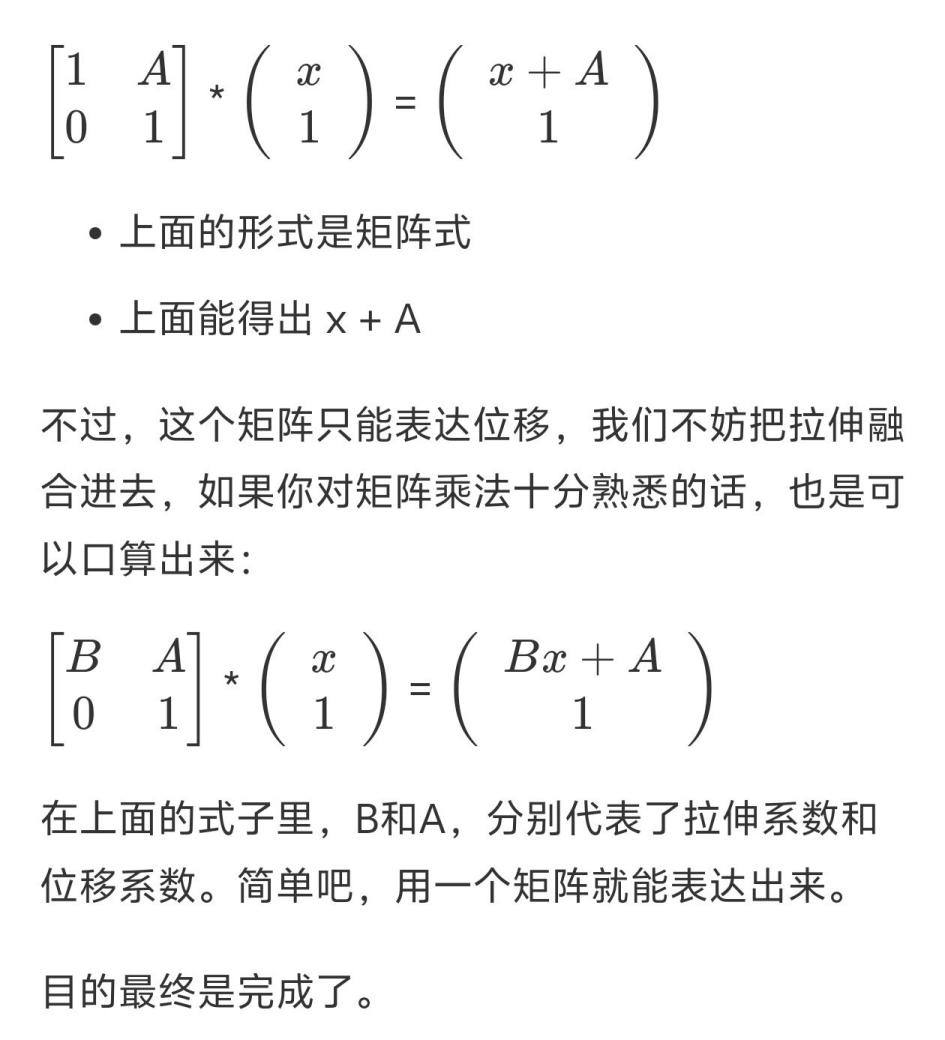
那我们的平移系数A也被缩放了。结果就是x2=BX+AB.

可能会有人问：那先缩放呢？

缩放：x1=Bx;

平移：x2=B(x+A)结果还是一样的，到这里来看，这个问题好像是无解的。

**下面我们用二维矩阵来解决这个问题。**



齐次坐标系最重要的一点就是在这里。

**还有一个比较牛逼的点就是齐次坐标可以省去除法操作**，我们通常用1就是没除，如果齐次坐标变成其他数了，比如说0.7，那其他的坐标也会按比例缩放，减少了很多除法操作。

**总结：**

齐次坐标可以统一的表示点和向量（齐次坐标为0时是点，1时是向量，0表示的是无穷远，**利用这一点可以实现透视投影**），同时可以方便的进行仿射变换。如：平移，缩放，旋转这样的。

可以简化运算，我们只要改齐次坐标，其他的数都会按比例缩小。用其次坐标方便了使用矩阵运算，就像上面的一样，可以用一个矩阵实现平移缩放和旋转。

**glMatrix矩阵库.gl.matrix.net**

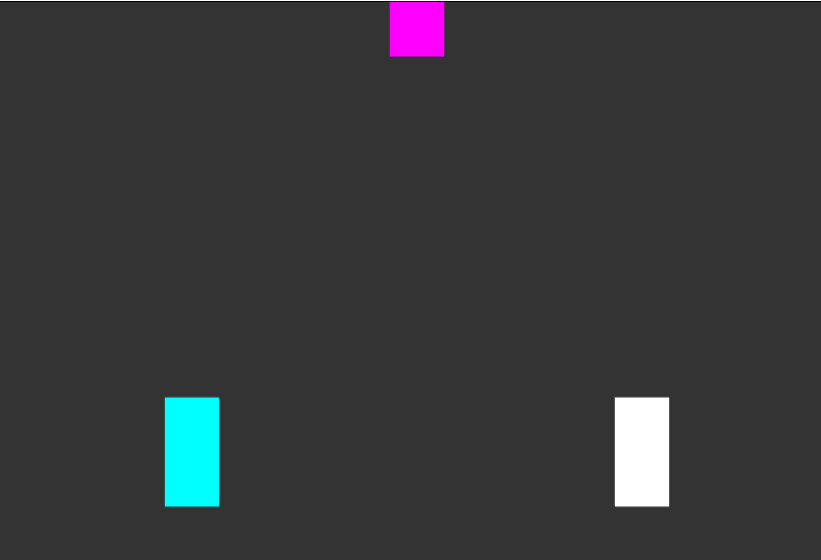
**关于webGL绘制图型**

WebGL绘制图形大致分为三种，点，线，三角型，这三种往下还可以进行一个细分。绘制他们之中的哪个主要是在drawArray这个API里面的第一个参数决定的

下面分别介绍一下API并用具体的效果帮助理解。

**点：**gl.POINTS,点





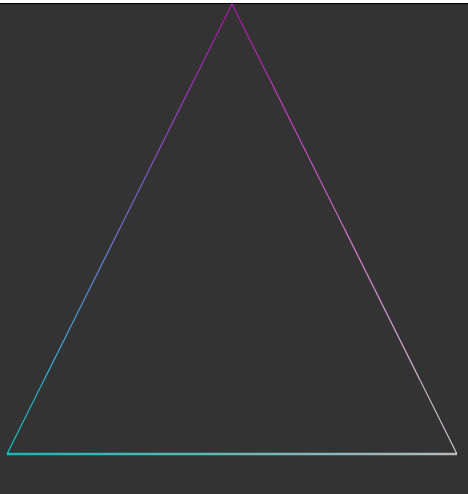
**线：**gl.LINES,线 点数除以2就是线数





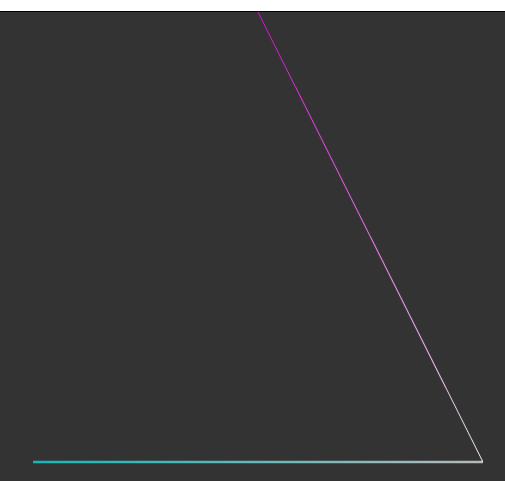
gl.LINE\_LOOP,点数就是线数，封闭线



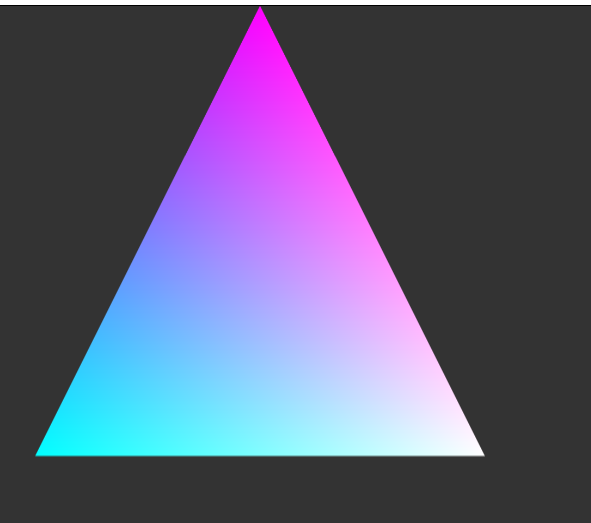


gl.lINE\_STRIP,点数减一是线数



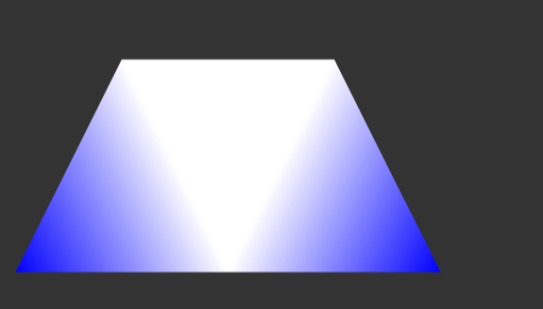


**面：**gl.TRIANGLES ，线数=点数

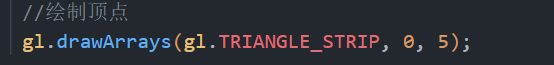
 

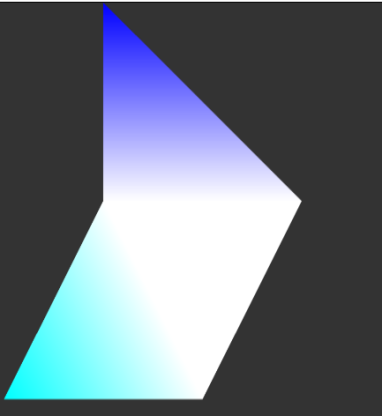
gl.TRIANGLE\_FAN,点数+2



这个画法是以第三个边为首边往后画。并不是一个扇形。

gl.TRIANGLE\_STRIP,点数+2，

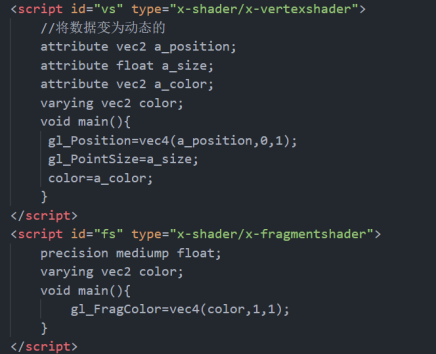


这个解释一下就是说他会以第三个点为圆心，最后一条边为底边一直往后画，与上面那个形成了明显的对比。

**上面的三角形是有颜色的，其实这个用到了一个插值运算的原理。只要给三个顶点颜色，webGL就会根据顶点的颜色和与顶点的距离来对图形进行一个填充。这就是插值的原理，我们webGL就是用这种算法来实现对三角面填充颜色。**

**实际开发中根据需要使用。**

**关于传数据问题：在开始就说过了JS是往显卡的内存里的buffer里面传数据的，这里就用到喽，虽然之前浅谈vertexshader的时候也用了，但不如这里的全面。因为在这里作者李某为了让图形看起来有辨别性，给每个顶点加了颜色。这里就用到了vertexAttribPointer和varying这个存储限定符。下面就是传数据部分的代码。**

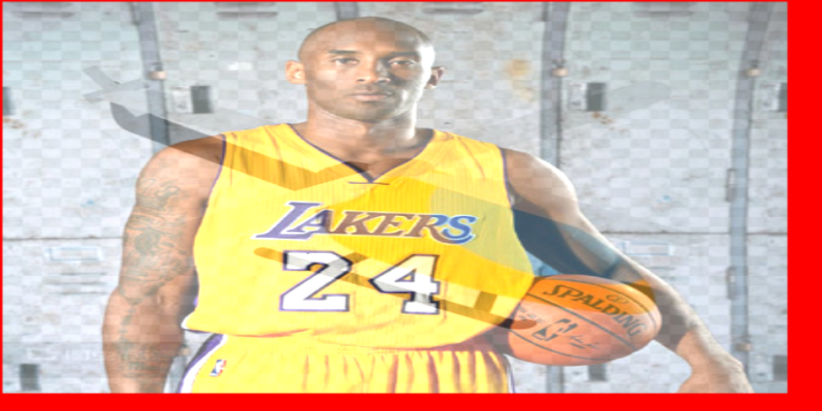


**还有一件事：上面所有的代码基本都说过，但是在JS部分有一块代码是没有说过的，就是那个gl.CULL\_FACE,背面剔除功能，这个玩意是为了节省效率的，玩过FPS游戏的卡过地图bug应该都明白吧，我当年玩生死狙击的时候就存在这个问题，有时候操作出问题会意外的卡到地图外，地图外看不见地面墙体，这个就是背面剔除的作用，他只会看到一面，也就是正面，或者说是用户看到的那面，用户看不到的那面我们就不渲染他了，这样可以节约一半的效率，是一种很典型的提高效率的办法。比如我把1点和3点换一下位置。有的部分就不会被画出来。这其实就是我们把背面呈现给用户，同时还开启了背面剔除功能。**

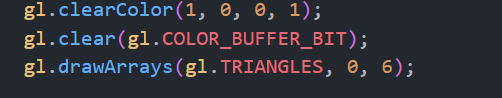
**到这里有的孩子就会问了，什么是正面什么是反面，我只能说：无敌了孩子，我们顺时针绘制的三角面都为反面，逆时针绘制的面都是正面。记住逆正顺反就行了，这个东西记住就行了，也没有什么营养。**

**关于webGL的纹理贴图**

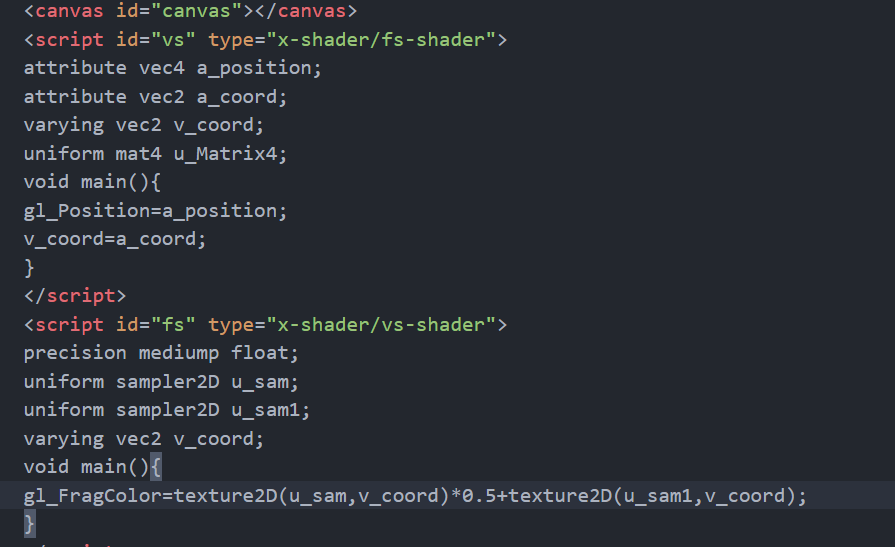
上面讲了绘制三角形，并且各种图形我们都用webGL画出来了，并且给予了颜色。但我们很容易的就发现，我们画出来的虽然很炫，但是好像吧，嗯。。。我们实际很少见到这种土炫土酷的3D效果。我们通常见到的都是二次元贴图之类的，其实我们webGL也提供了这种贴图方法，他叫做texture纹理贴图。通过这个东西我们可以把图片贴到webGL图形里面。其实他本质也是一种插值算法，只不过他是根据uv坐标把每个uv坐标对应的图片赋值给每个实际的坐标，就是这么简单的一个道理。这里作者李某也想偷懒不写例子了，但是不写例子总感觉不完美，李某也有点强迫症，所以从之前的demo里面找到了我纹理贴图部分的demo。



Man!我把直升机贴给了牢大，其实形成了一个双重纹理贴图，一个像素贴了两个纹理，一个直升机，一个科比。下面对贴图的逻辑进行一个阐述。

这部分就是JS代码的核心了。

这部分是GLSL着色器部分的核心：



这里一些API是新的，下面就着逻辑部分顺便对这些API进行一个介绍罢！

我们创建纹理贴图首先要有一个图片，图片添加事件监听，图片添加完毕之后再进入正常纹理的操作里面。

**创建纹理(gl.createTexture):**这个API用于创建纹理的。

**激活纹理(gl.activeTexture)：**我们纹理分了32层，也就是说最多可激活32次从0-31这样的一个激活我们激活纹理只要gl.activeTexture(gl.TEXTURE0)就算是激活了。

**绑定纹理(gl.bindTexture):**gl.bindTexture(gl.TEXTURE\_2D,texture)纹理绑定。

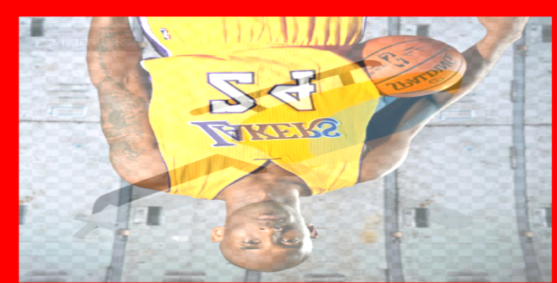
这三步就是一个流程，与上面buffer的异曲同工。下面才是比较重要的部分，因为下面负责的是数据方面的。

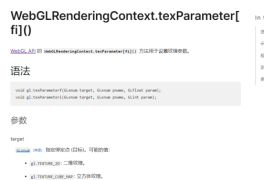
**提供纹理图片：**gl.texImage2D（gl.TEXTURE\_2D,贴图层数，gl.RGB,gl.RGB,gl.UNSIGN\_BYTE,图片源）里面参数细节就不提了。

**图片上传：**gl.uniform1i(sampler2D变量名，贴图层数)

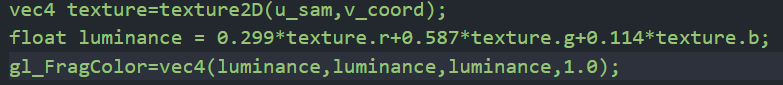
最后在shader里面用texture2D（sampler2D，UV坐标)接受颜色赋值给fragColor即可。。其实没啥难度。

**细节方面：gl.pixelStorei()这个是调节图片坐标和UV坐标一致的，因为UV和图片坐标是反的y图片=-yUV这样的一个关系。如果我们不加图片就是反的**

**还有就是那一堆textParameter也是细节的一个体现。规定了水平数值渲染，这种细节。具体的话去MDN可以找到每个参数的意思，这里不过多赘述。**

**还有细节就是MipMap这个东西在下面详细说。**

**关于贴图的灰度转换:就是把图片变黑白这个其实就是对贴图的RGB分别进行操作。下面就是灰度转换的方法。**



**纹理贴图之MipMap**

这个问题在webGL2中得到了解决，但是webGL1里面还是有一点问题的，就是说如果我们的图片大小中的宽长不是二的次方倍会出事，这个就是mipmap的一个作用，要么让美术乖乖做2的次方倍的图，要么就是WebGL开启generateMipMap之后webGL会帮你做图，可以想象成webGL里有一个美术在干活，就这么理解一下就行，了解了解。

**WebGL模拟光照**

3D环境要逼真不止要有图形，还有有光照，环境光，有了光照，我们的三维场景才足够真实。光照其实就是在shader里面经过一系列的计算实现不同的明暗效果，提高几何体的一个对比度的方法。

添加场景光照

光照分为：**环境光，平行光，点光源**

环境光：当光源发出光以后经过墙壁等物体的反射最后反射到物体上的情况叫环境光。

平行光:当光源距离物体过远时光线趋近于平行，这种光称为平行光。

点光源：类似于灯泡一般发出的光呈现出的是发散状】

**反射理论**：

光打到物体后会发生反射。分为镜面反射和漫反射

漫反射算法：反射光的颜色=入射光颜色\*物体底色\*cosØ

环境光反射的算法：反射光的颜色=入射光的颜色\*物体的颜色，这种算法是基于光线直线舍入，直线反射出的

这两种算法都是基于现实世界的一种计算方法。其目的是更好的实现三维场景。

**关于光照部分：**

**环境光：**我们只需要向着色器传光照颜色，通过光照颜色和基色相乘实现颜色的计算。

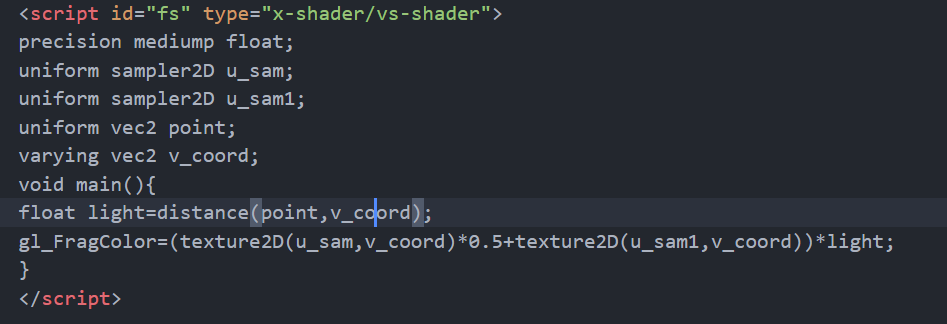
**平行光：**需要normal，光照direction，二者归一化相乘得到COS，通过光照颜色和基色和cos相乘实现颜色的计算。

**点光源**：比平行光多了一步计算光线方向，他传的是光的点，不是光的方向，与a\_position相减才能得到光的方向。

逐片段着色器：不在顶点着色器里面执行。

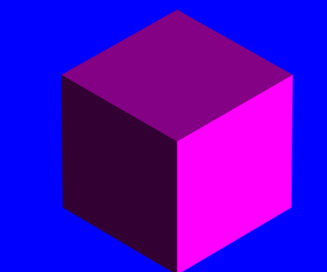
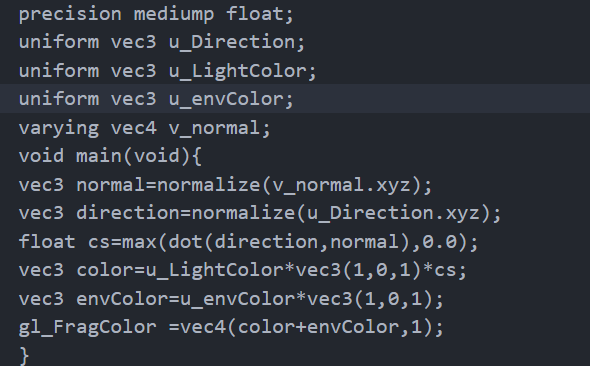
下面以上面的牢大和直升机的纹理进行一个2D光照的模拟。

**镜面光：**物体表面基色\*光源颜色\*入射光线与反射光线夹角的余弦值。



2D光照很简单，就是一个光照点和UV坐标直接距离的一个计算，然后再乘进颜色里。

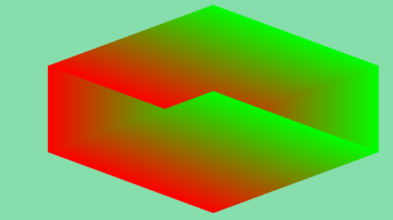
3D光照要考虑的就多了，要考虑光的光源方向，光源与光的目标方向形成一个光线，法线方向，之后根据数学计算来模拟出光照效果。原来光的颜色要加入环境光的颜色和本身物体的颜色与光照角度进行计算（光照角度的余弦值不得小于0）。下面是之前写的环境光示例看看多练。

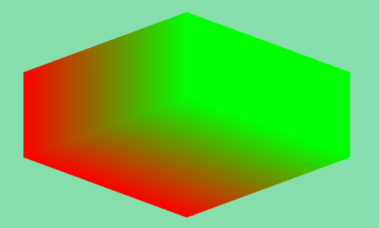


这是3D光的效果。

**关于深度测试**

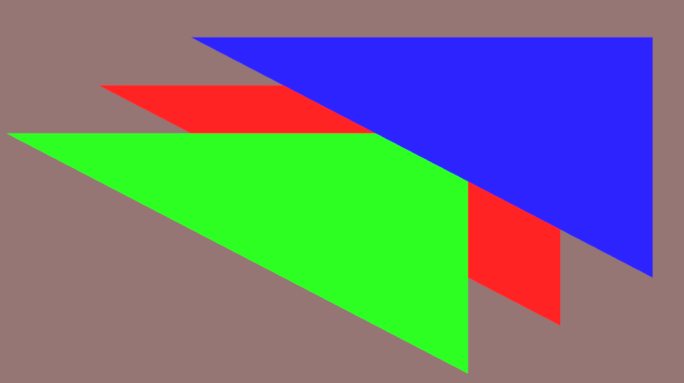
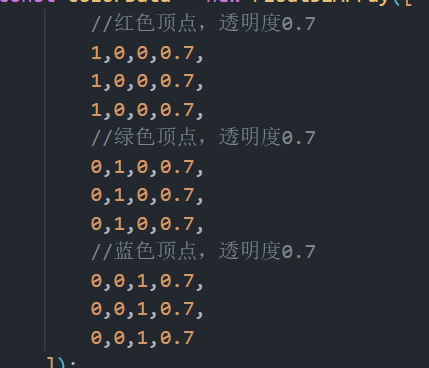
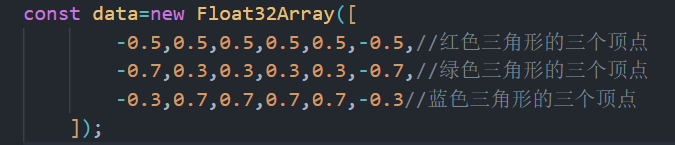
每个顶点都会有一个深度，默认按照绘制顺序来，按照绘制顺序来会存在深度无法识别，绘制出来的效果很差的情况，这种时候我们就要开启深度测试。

这里没开深度测试颜色显示乱。

开启深度测试颜色就正常了。

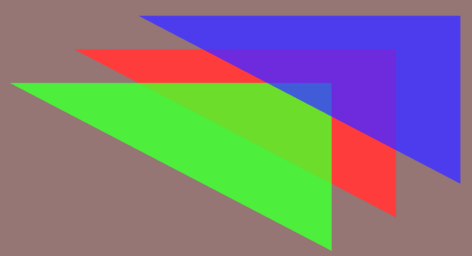
**关于α融合**

这也是在webGL变换流水线中的一步，他会对透明度进行操作，实现颜色混合，从而达到透明效果。他是在深度测试之后进行的。

这分别是顶点的数据和，颜色数据，我们可以看到如果没开启颜色透明度混合他就不会显示透明效果。透明度一点用也没有了就。



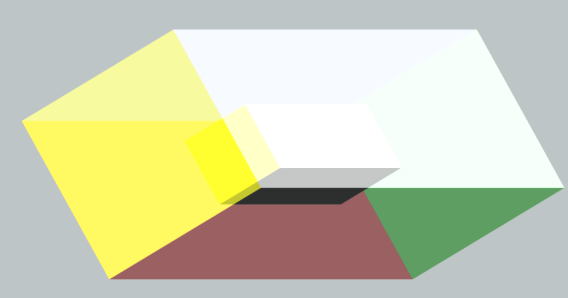
开启透明度测试后就有透明度效果了。

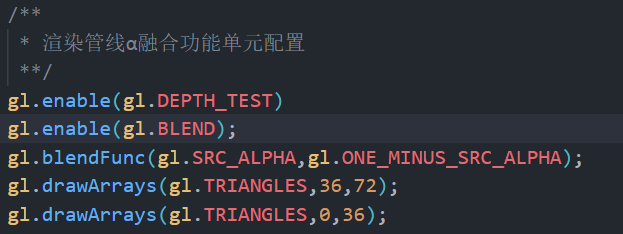
具体混合方法可以查文档。

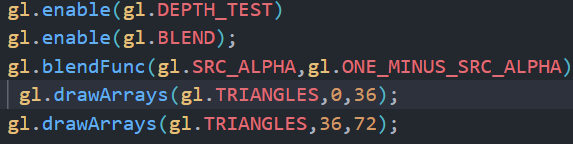
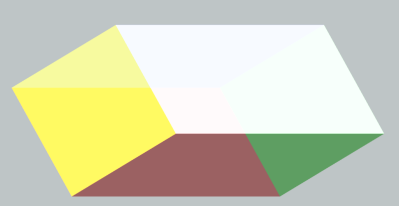
文档里有很多关于混合方法的介绍。

**很多时候都是深度测试与α混合结合使用**

这种情况下如果深度测试不通过就会被替换掉，透明度混合也就不好用了。所以要注意绘制顺序，使深度测试和透明度混合能正常结合。

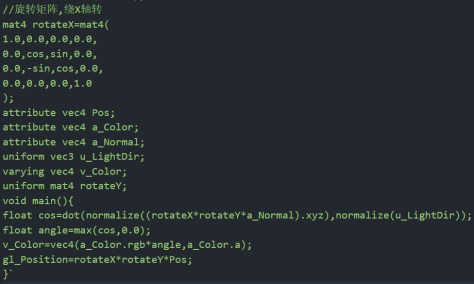
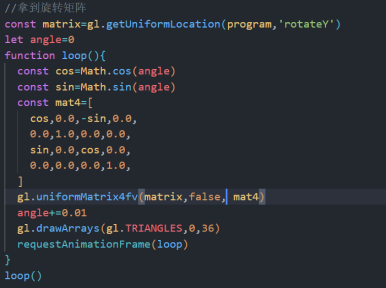
这就是深度测试和α混合同时执行的结果。

一定是不透明的先绘制。如果透明的先绘制的话，里面的再绘制时深度测试不通过，就会放弃绘制成这样了。



**关于渲染循环**

其实就是不断更新矩阵放入渲染循环里面，然后重新绘制就能实现关键帧动画效果。

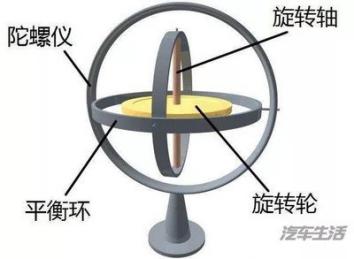
x的方向写死，然后通过CPU上的JS不断更新Y轴的旋转矩阵给GPU，GPU不断跟着CPU上的数据走，不断绘制就能实现动画效果了。

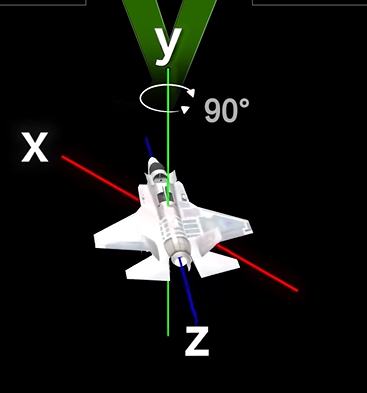
**WebGL欧拉旋转**

欧拉旋转他的作用多用于旋转，我们用传统的矩阵旋转，一个两个可以多了就会出现一个误差累计的问题，误差越累计越大，旋转不准确。欧拉旋转可以精确的将物体绕某个坐标系的X,Y,Z分别绕多少度。如果以自身坐标系旋转就是内旋，如果以世界坐标系旋转则是外旋。我们通常用的都是内旋。

下面用一个具象点的例子。你在写代码，同学过来看，同学在你的旁边，且他是站着的那你就要把电脑左旋40度，再往上转20度让他看见。这个就是欧拉旋转，注意的是欧拉旋转每次旋转后要继续旋转不能在上一次旋转的基础上进行旋转，也就是说如果右边也有人要看。你不能直接转到右边应该先回归原来的位置，再转给右边的人看。**欧拉旋转很重顺序。顺序不同旋转结果也不同，所以用欧拉角要严格遵循顺序。**

**欧拉旋转也有一个比较严重的问题就是万向死锁，这个问题也和上面的欧拉旋转的顺序有关系。在网上搜出来的教程里解释万向死锁问题都用的平衡环来解释。**

**这就是一个平衡环。我们也从平衡环来进行解释，平衡环的旋转外部环的旋转会带着内部环的旋转，但内部环的旋转不会带动外部环的旋转，这限制于他物理的结构。这样的结构导致我们一直旋转，转到一定的角度之后会出现两个轴重合的问题，如此我们只剩下了两个轴，无法再正常的控制旋转。其实用这个来讲欧拉角是很巧合的，与平衡环的物理结构问题类似，我们的欧拉角只负责变换，并不是运动，每一次变换都是从初始值开始的。用户的操作顺序无法改变这个算法，这也就导致和平衡环一样，转90度后会出现两个坐标轴的重合导致少了一个轴，专业点就叫做失去自由度，真要解决这个问题就少用欧拉角，如果一定要用就把顺序设置成很少旋转90度的角在中间。这样可以少bug，真要解决还得靠四元数。死锁的根本原因就是中间轴旋转了90度。导致两轴重合。**



**这里讲个笑话，当年李某生死狙击王者晋级赛就被欧拉角坑了**

**人物成这样了，一个枪战游戏怎么打！**

**WebGL四元数旋转**

四元数就是一个向量，绕着哪个轴，转了多少度，就这么记。

向量（x,y,z）轴（u,v,w）,角为β，

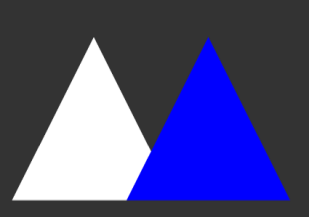
v=xi+yi+zk;

p=cos(β/2)+sin(β/2)ui+sin(β/2)vj+sin(β/2)zk

P1=p\*v\*p-1得到的就是一个只有ijk三项有值的新四元数，i,j,k的系数就是旋转后的结果。

**WebGL之z-value**

这个东西涉及到了一个比较专业的名词：深度冲突，何谓深度冲突就是说你画了两个三角形，哪个在上哪个在下的问题。

默认符合的是画家算法，哪个后画哪个在上面，如果要人为的控制哪个在上这个时候就要用到z-value这个东西了，其实这玩意是通过Z坐标算出来的。你可以把他看成Z坐标，但他不是Z坐标他是Z坐标+1后除2得到的。

到这里原理看明白了就不得不提出一个API了那就是gl.enable(gl.DEPTH\_TEST)这玩意就是开启深度测试，开启后每个店默认会有一个z-value。谁的大谁在上，就是这么简单的一个道理。注意一下如果开启了深度测试我们clear里面也要对应的加入深度测试。Gl.clear（gl.DEPTH\_BUFFER\_BIT）

**WebGL之drawElement**

这个玩意看起来很像drawArray吧，事实上他们还真有点关系在里面，在drawarray画矩形时，我们用drawArray需要画六个点，浪费了两个点，drawElement解决的就是多画点的问题，他只需要创建一个索引缓冲区然后用点的代号画退化三角形就可以解决drawArray多需要点的问题。

**关于webGL的性能优化问题以及一些细节。**

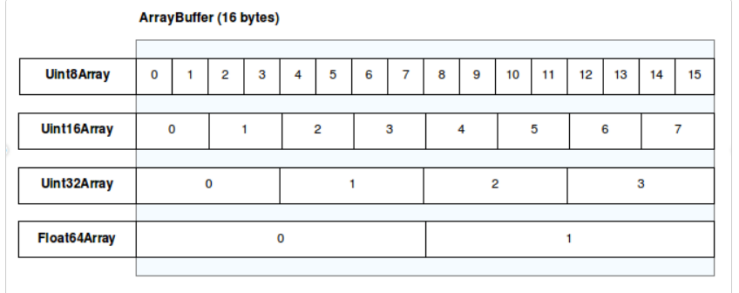
**关于类型化数组**

FLOAT，INT,UNIT表示的是类型存储什么数据用什么类型

32,64，表示的是字节数，32代表的是**4byte**，64是**8byte**，如果你不知道，array. BYTES\_ PER\_ELEMENT可以看到的,这在我们用vertexAttribPointer时很有用。

类型化数组的数字其实就是最大存储的二进制位数，超出位数会被裁剪掉，然后转换为十进制，这样数就会少一节，所以要选择合适的类型化数组，这样数据才不会失真。其实这是JS的内容，官方给提供了一个图片，每个字节都是8位，这个8位是计算机的基础知识，1位是一个0或者1。

**特别注意:以32为例，这里的32有4个字节也就是也就是32个零一，他的限制对象是数组里面的元素（如Array【0】）如果元素大于这个32个零一就会失真，甚至报错，因此选择正确的类型化数组变得尤为重要。**



**提高性能**

在数组方面优化，尽量用一个数组，属性分开放，用vertexattribpointer来进行传递给着色器

绘制方面，尽量用TRIANGLE\_STRIP的退化三角形和drawelement配合索引来优化性能，减少缓存使用。

**webGL变换流水线**

局部坐标系→模型矩阵->世界坐标系→视图矩阵→相机坐标系->投影矩阵→投影坐标系->透视剔除→设备坐标系->视口变换→屏幕坐标系

矩阵.plus

**齐次坐标则是，**透视空间的根本，他能用有限的表示无穷，也可以说是一种比例关系。0是点，1是向量。

**模型变换**，也就是模型矩阵，控制的是平移缩放和旋转。平移就是在第四列＋上TX,TY,TZ，缩放直接在1上动手，旋转则是cs-sc。

**WebGL的性能优化**

webGPU与webGL相比渲染效果是好，但存在安全性问题

CPU的主要作用体现在，进行中断如if else Switch的时候进行判断，相当于古代的皇帝。

GPU的主要作用体现在，进行大量连续计算的时候进行计算，他的效率是很高的。相当于现在的打工人。

渲染管线软硬件版：首先是CPU进行一个control命令，需要计算的计算，放到内存中，显存后，放入状态机，再进行绘制，GPU中进行的渲染缓存，顶点缓存，纹理缓存，帧缓存（颜色，深度，魔板），离屏渲染，视频控制器，屏幕显示器，顶点着色器，，图元装配，片元着色器光栅化，裁剪分析，模型分析，多重采样，深度测试，融合，抖动。