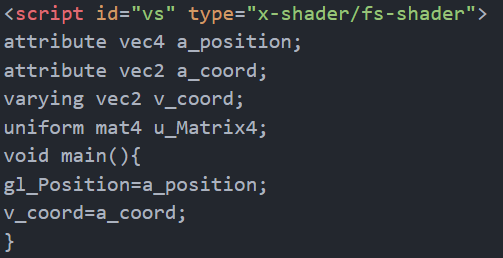
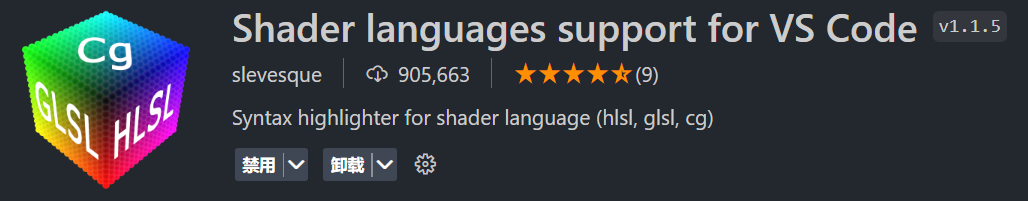
# 关于webGLshader

**着色器语法GLSL ES介绍与准备**

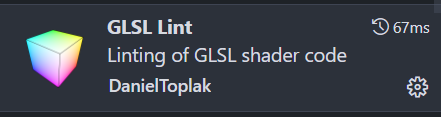
着色器语言GLSL（GL shader Language）与JS不同，GLSL的变量属于强制形变量，这是从变量的角度来说的，按照我们上面TS的话来说就是GLSL是一种强类型的语言。之前TS部分也说过了弱类型和强类型的区别，在这里就不进行过多的赘述。

Shader语言是独立于JS的一种新的编程语言，类似于C++，我们直接写GLSL在vscode里面是不会有高亮效果的，如果我们语法有问题也是不会报错的。和之前在script里面写着色器代码一样，通通都是灰色的像这样，写起来无疑是非常的难受的，没有高亮，没有报错，加上着色器的强类型特性，以及控制台打印全靠GPU数据回流，种种原因叠加起来，让我们报错时会摸不到头脑，其实对应的有很多解决方法，我们随着进度推荐会从这些角度一一解决代码编写难度的问题，让shader编写简单。对于没有高亮效果的这个问题，我们VScode提供了两个插件，我们安装这两个插件之后shader代码从此就有了颜色。他们分别是：

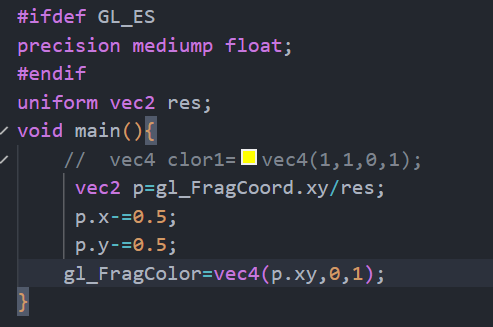
Shader language support for VS code



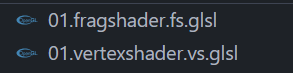
GLSL Lint

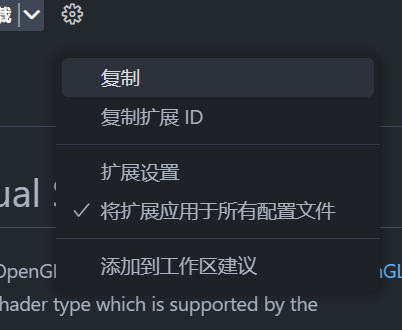
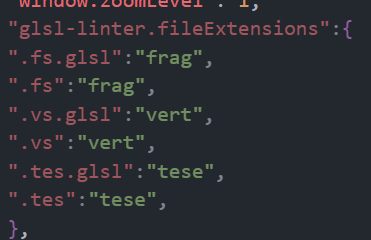


这两个插件安装后就可以实现高亮代码与代码报错功能。里面还有一些配置项要配置，这里说的话有点生硬，放在下面讲更加清晰一点。



**关于代码隔离问题**

在webGL之前的学习中着色器是与JS代码放在一起的，放在script标签里有一个很致命的问题就是写起来不方便，也就是说写在script标签里所有的代码都是一个颜色的，即使安装了上面的两个插件他也不会在代码上显示高亮与报错的效果。着色器语言存在单独的一个着色器文件。像.js,.vue一样，着色器语言也有一个自己的后缀.glsl，下面是一个定点着色器和一个片元着色器这里可以承接上面的一个插件扩展设置的知识点了，善于观察的孩子可以发现.glsl前还有一个点.fs和.vs这种的，其实这个就是关于这个着色器是顶点着色器还是片元着色器的设置。想设置这个我们要去GLSLLint的扩展里面

点进去之后就可以设置这个后缀文件扩展名了。

这个扩展名理论上按照自己的需求就行了，但事实上呢我们最好还是设置的像我一样，专业性强一点，这样的话会舒服很多的，自己写着舒服，主要是同事写的也舒服，毕竟代码不止是自己一个人写的，团队合作要考虑别人的感受。

**正式进行shader学习**

准备工作与shader概况都写了现在进入shader的学习，shader又叫GLSL，其分为顶点着色器和片元着色器两部分。之前的webGL中所有着色器都是通过无义script标签或ES6模板字符串实现。这种情况不方便debug，而GLSL装配插件配置路径后，我们可以将webGL，JS，GLSL代码进行代码隔离。提高效率，引入方法为经典的XMLHTTPREQUEST异步。这里写一下：

封装一个函数：



这里复习一下http请求协议

request.readystate

0为准备XHR

1为open

2为send

3为接中，

4为完成。

request.status，100是请求继续200是成功300重定向。400，失败。

**关于颜色计算：**范围是0-1，低于0为黑，高于0为白。当成向量加减乘除就行。

**shader的三个坐标：**

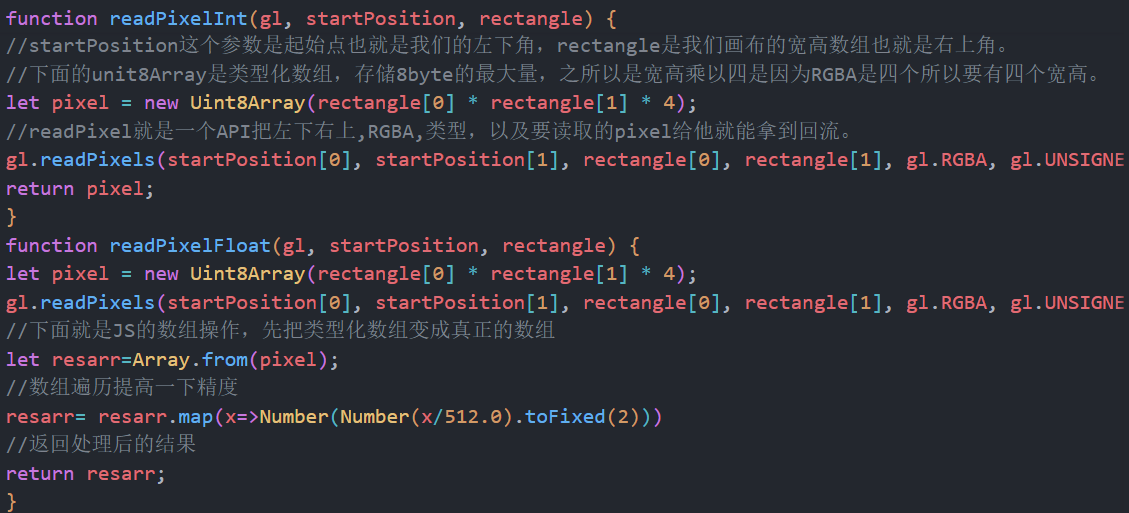
1.gl\_PointPosition:决定点在画布中的位置，满足宽高比。

2.gl\_PointCoord:是以左上为坐标原点，范围是0-1决定点内的位置。

3.gl\_FragCoord:左下为原点。通常x,y都减0.5到中间。其范围是0-255，需要除以canvas的宽高。其实是一个颜色坐标。

**webGLShader之像素颜色读取。**

在webGL中canvas画布每一个像素都有颜色

但是想打印颜色，并不容易。电脑GPU打印数据至今没有被突破，这也就意味着webGL无法像大多编程语言一样直接打印像JS的console.log，或者是Java的system.out.println这样控制台输出。而颜色打印利用的是GPU数据回流到CPU，在CPU中读取。用的API为gl.readpixels,这个提供的参数与gl.viewport非常类似，只要提供左下二点，右上二点即可。这里封装一个函数玩玩吧：这里直接把两种方法都封装了，根据精度以及实际需求自行选择

把他当成shader的console.log就行了，我们打印一块区域的颜色，通过颜色的表现来判断矢量计算，或者是其他的计算的问题。

**shader的变量：**和其他编程语言类似。包括强制类型转换都非常的像C语言。

shader的矢量，矢量既有大小又有方向，webGL的矢量有以下几点注意

1.vec4（1.）这种会自动填满1.但是如果输入的超过一个又没有4个就会是错误

2.vec4(vec2,1,1)这样他会把vec2解析出来放到前两个

3.vec2(vec4)这样他会解析vec4前两个放在vec2中

4.length(v1)可以算矢量的长度

5.normalize(v1)矢量归一化

6.dot(v1,v2)点乘

7cross(v1,v2)叉乘

8.向量取值可以通过color.r,color[0]这种方法取值

9.vec3有ivec,fvec3和vec3

10.向量加减乘除就不说了，在之前的webGL那里介绍的很清楚了。

11.float浮点类型的变量

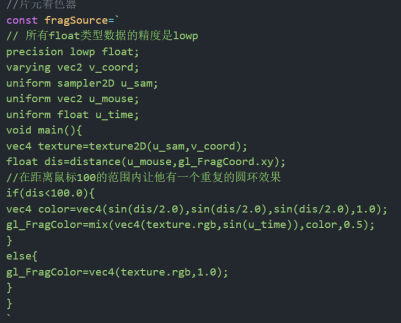
12.int整型变量，全是整数

13.加减乘除注意变量类型。

14.函数返回值要注意类型

15.内置变量gl\_PointSize，gl\_PointPosition,gl\_FragColor这些内置变量不用声明可以直接使用。

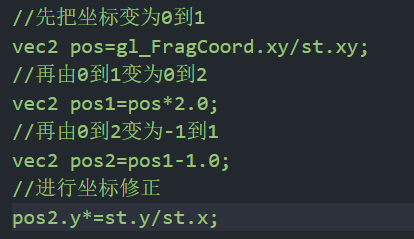
**外部变量输入**



这里用外部变量实现了一个动态的鼠标效果。其实就是不断更新数据,之前webGL部分的旋转立方体也用了类似于这个的方法,不过之前的旋转立方体里,我们传的是矩阵,其实都差不多。**不过注意传入的是动态变化的数据每次更新都要跟上drawArray方法,不然他不会更新渲染效果。**

**坐标修正**

坐标修正这个东西其实是解决在坐标转换时候出的一个问题的公式。如果细说他的原理就是:**canvas画布的长宽不一样，而我们坐标转换后的坐标不管x还是y都是存在于0,1这个范围的。这会导致坐标问题，具体就表现为圆形的坐标变成椭圆,正方形变成产方形这种拉伸问题。坐标修正其实就是通过控制解决拉伸问题。**

了解了每步的原理之后就可以进行一个连写。

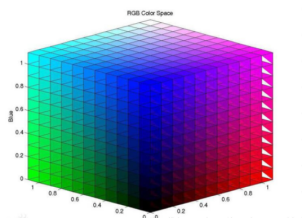
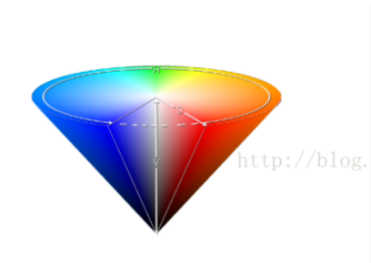


效果都是一样的。

**关于颜色转换**

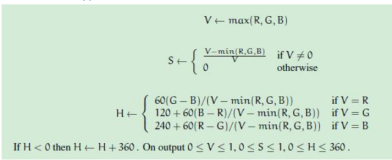
关于颜色模型来说大概是两种:RGB模型,HSV模型。通常来说:图形学中的RGB是在0到255之间的数。而在GLSL着色器语言之中,他给转成了0-1之间的。其主要原因就是GPU更适合浮点数计算,通过这样一个转换算力会得到很大的优化。

几何上来说:

这两个图形就是HSV和RGB空间的几何表示,正方体表示的是RGB，椎体表示的是HSV。RGB的空间是一个三维笛卡尔坐标系,每两个轴之间是垂直的。而HSV则是H:色相(0-360)。S：饱和度(0-1)。V:亮度(0-1)。于几何中表示大概就是H表示角度。S表示距圆心的距离。V表示的是高度。

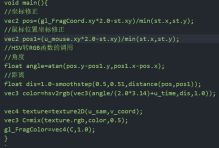
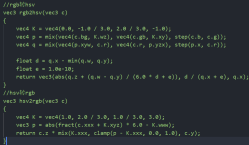
代数上来说：

代数上其实就是一个公式。这是RGB转HSV的原理。另一个可以通过逆推推出来。



代码上：

这玩意纯复制了,没必要自己再写一遍。就是下面这俩函数,其实老实来说实际开发多以HSV转RGB为主,很少用RGB转HSV的这样一个转换。所以这里就用HSV转RGB来实现一个色彩圆盘的效果。

这里就实现了一个随鼠标移动的彩色圆盘。

**webGLshader之矩阵，结构体，数组，函数，流程控制。**

**矩阵**

矩阵来说shader的矩阵实现列主序的，因为这样GPU可以进行高效率的运算。有的杠精可能会问为什么是列主序运算效率高效，如果你有这样的想法只能说无敌了孩子，人家就这么设计的你想要行主序计算快的就自己设计，别杠。

矩阵的变量为mat1,到mat4。拿到某行某列则用m2[0][1]这个就是m2的一列二行。开始可能会有点不舒服，但熟悉了列主序之后一切都会好起来的。剩下运算就是矩阵运算了，真没啥好说的，无非就是平移，缩放，旋转老三样，忘了去上面看。

**结构体**

结构体与C结构体一样，他更像是一个阉割版，他的结构体依然支持两种创建法，但是不支持嵌套，如果嵌套必须提前声明。且他不存在JS那样的Var变量提升。

例：struct Light{

vec2 position;

vec4 strong;

}Light1,Light2,//对象直接创建了

要么就是 Light Light1=Light(vec2(1.),vec4(1.))直接对号入座。

要么就是 Light.position=vec2(1.)这样就算实现结构体的创建了。

**数组：**

1.0版本的数组更是高手，不支持声明和赋值一体，必须先声明再赋值。还踏马得一个一个索引挨个赋值，这种反人类设计我想不管谁来了都得骂娘。在1.2版本后支持声明赋值一体化。索引只支持整形int和uniform变量。vec2 arr［2］这就声明了一个长度为2，每个元素都是vec2的数组。数组还不支持.length也没有push，shift这样的API，孩子，切记这些数组操作的方法只存在于JS里面，GLSL并不是JS，他是一门新的编程语言。。。切记，切记。

**函数：**

不支持全局与JS不同，他的形参要+数据类型，更类似于C，JAVA那一类的。

程序控制中discard更是一个寄吧。他与continue一样放弃这次，也就是放弃这个片元。还有break之类的，他的函数里面记得加分号，这也是所有强类型语言的默认规定了属于是。

Light li(float i){

if(i){

discard；

}

else{

}

return i；

},定义了一个返回值为Light，变量为i，名字为li的函数

**webGLshader的限定词：**

1. 参数限定词：in不写时的默认，输入的是in函数内部可以使用，但是对外面无影响，可以在内部更改。const in输入的参数在函数内部可以使用但是无法更改，out输入的参数内部使用可更改但是外部也要同时更改默认为null，inout默认不为null剩下的与out一致，这些都是在函数里面使用的。
2. 存储限定词：存储限定词之前webGL部分有提到过attribute只能在顶点着色器中且进行的是遍历无法做到单改一个数据，uniform二者都可且可以改变单个数据。
3. varying用于二者通信2.0版本，varying变成了in和out通常，是在顶点着色器里放一个out然后在片元着色器里用一个in接着，这样写起来更有逻辑感，但是如果用惯了varying可能会有点难受。需要适应一段时间。

3.精度限定词：precision mediump float， 除了片元着色器的float没有默认精度，剩下都有默认精度。顶点着色器都是hightp，片元着色器中int为mediump，纹理部分的simpler2D,samplercube顶点片元中都为Lowp

**webGL预处理指令**

类似于C中的宏 #开头，

#version

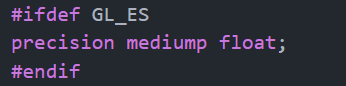


这个用于指定版本，防止版本出错。

#define

这个是定义一个变量，下面代码里的PI都代表3.14

#ifdef：



这个与#endif包裹的内容执行情况是后面的条件成立也就是GL\_ES是支持的就执行

#ifndef这个与上面那个正好是个相反的，不支持的话执行被包裹的代码。

#if #else #else if

这个就是ifelse语句，注意的是条件紧跟在#if和#elseif的后面就这么简单。

#extension扩展，  
都是在编译前先处理一下，防止处理时发生报错。主要是用于不同设备的适配和不同版本的适配。代码执行之前先判断一下，有一股子媒体查询的味。

**webGL内置变量**

gl\_FrontFace,

gl\_Pointsize，

gl\_FragColor,

gl\_FragCoord,

gl\_pointCoord，

gl\_drawData[]

内置常量，多是规定数量的，实际用的不多，也就不记录了

**比较常用的工具**

graphtoy简单化，轻量化

desmos复杂化

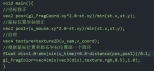
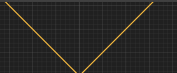
shadertoy多案例，好编辑

我比较喜欢用graphytoy他比较简单，所以下面介绍基本都用这个了。



**shader的常用函数以及图像和部分实例**：

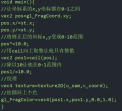
**Abs:**对数据取绝对值，可以是float也可以是vec2，vec3，vec4。对于vec2,vec3,vec4都是对每个数分别取整。这里用abs画了一个光圈：思路就是用一个数减去光圈中心点坐标和画布坐标之间的距离。会出现正负两种结果,取绝对值之后就会出现对称效果。顺手也可以画一下多个圆环，都差不多。



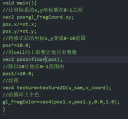
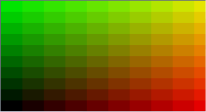
**Sign:**对数据取整，大于零取一，小于零取负一，等于零取零。对于vec2,vec3,vec4都是对每个数分别取整。这个案例的一个整体思路就是根据坐标系sign根据象限获得R和G的值,B的值则是由坐标的x,y实现。



**Ceil**:向上取整。对于vec2,vec3,vec4都是对每个数分别取整。

思路就是先乘十然后取整再除十得到方块。

**Floor:**向下取整。对于vec2,vec3,vec4都是对每个数分别取

思路与Ceil一致。

**step(1，x):**大于1为1，小于1为0，遇到vec每个分量单独看。

这个的思路就没那么复杂了:0左边是0,0右边是1。

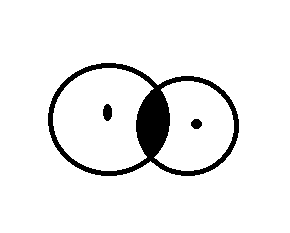
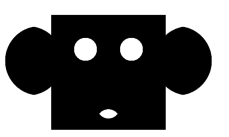
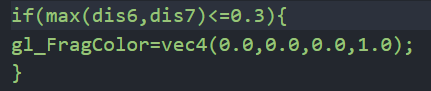
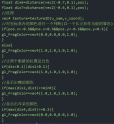
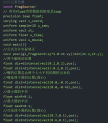
**smothstep(0.1,0.3,x)**:在大于0.3时取1，小于0.1时取零，smothstep(0.3,0.1,x)大于0.3是0，小于0.1取1，遇到vec就把每个分量单独来看



其实对于这个来说就是step带一个过渡。

**max（x,y):**取最大值，遇到vec就是每个分量分别取最大的

**min(x,y)**取最小值，遇到vec就是每个分量分别取最小的



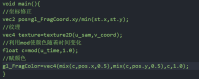
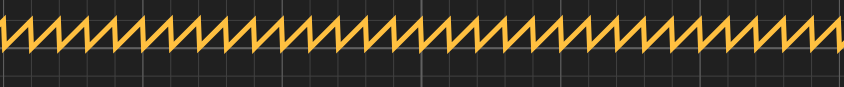
**这里用到min和max的地方其实就是耳朵和嘴巴。思路上是一个很简单的几何问题。就是两个圆相交的部分当做耳朵。那么问题就转换为确定两个圆重合部分了。重合部分有一个特点:他到两个圆圆心距离中最大的那一个比最小圆的半径小。根据这个就可以做出这个效果。**

**mod(x,y):**函数取余，左边被除数，右边除数。符号看第二个参数

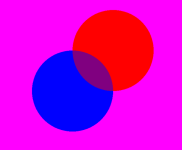
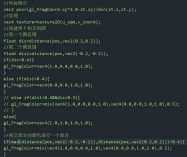
整数取余：左边除以右边＋1再乘以右边最后减去左边。

小数取余：左边除以右边再乘以右边-左边。被除数比除数小的话，直接取被除数，也就是左边。**从函数图像上来看会发现他是一个周期性的,所以他可以跟三角函数坐一桌实现一个**

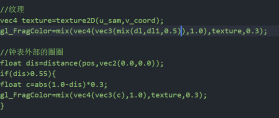
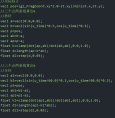
**周期性变化的结果。**



**mix(a,b,c)b-a\*c+a:**这个东西就和他的英语一样,就是一个混合。这里做了两个圆,相交部分采用颜色混合。让他既像红色又像蓝色,但是两个都不是。

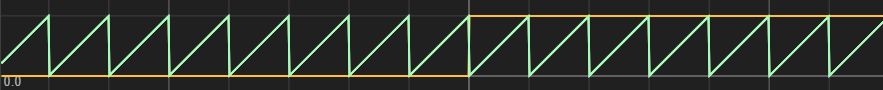


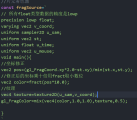
**clamp(x,a,b):**将x限定在a-b之间。x比a小取a,x比b大取b。如果在中间取插值。



这里用clamp限制了k的值让距离中心线A,B一定范围内的点显示白色,非常好。

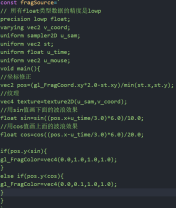
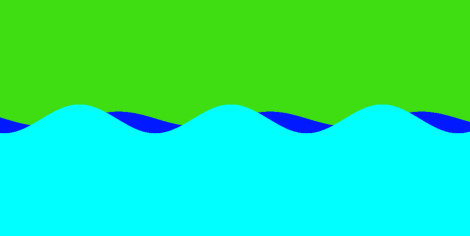
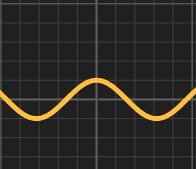
**Fract:**取小数位，正数直接取，小数原来的-小数向下取整，

这是fract（x）这玩意跟取余mod能坐一桌了函数图象都一样。

用fract实现的效果，乘以10之后fract就行了，和之前的Ceil和floor一样。

**三角函数**

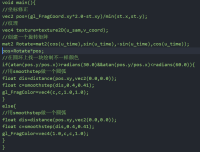
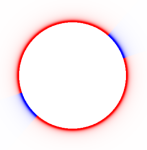
**sin,cos:**

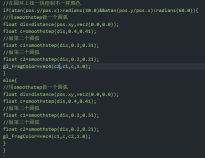


这个思路上其实就是一个正弦函数,一个是余弦函数,随着时间变化并赋予颜色。非常简单。

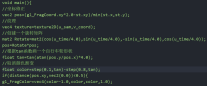
r**adians(),弧度degrees()度**

这俩就是弧度角度之间互相转换的方。radians角度转弧度,degrees弧度转角度。

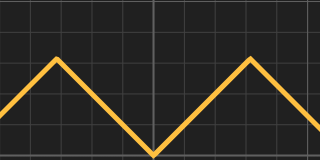
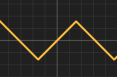
**其实就是用atan通过角度对颜色进行区分，之后用旋转矩阵转。**

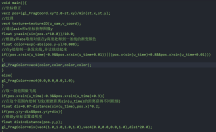
**多个圆环套在一起其实就是用不同距离得到不同的RGB。分别用一下就行了。还有一点就是这里的smoothstep用法,非常好用。Smoothstep(a,b,x)我们不一定要把x变量放到最后一个,放到别的地方也会有特殊的效果。**

**Tan:**正切函数

这里就是用tan画一堆线然后把他限制在一个圆里面而已。这里用了两个step组合的方法得到了纯色,去掉了渐变效果。

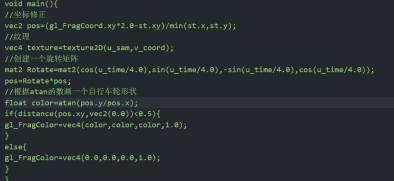
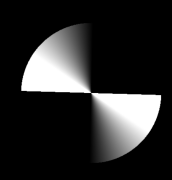
**asin(),acos():反正弦,反余弦。**





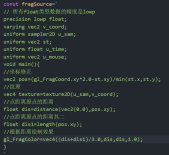
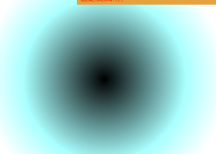
**Atan(y,x)**

这玩意之前也是一直用没啥好说的,做个案例得了。

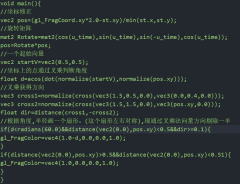


**几何函数**

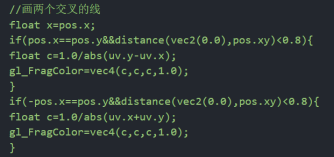
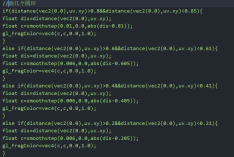
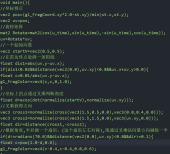
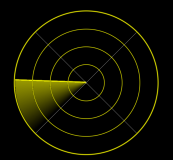
**length计算向量的长度distance（a,b）计算两点之间的距离**

这里用distance和length简单写了一个demo。

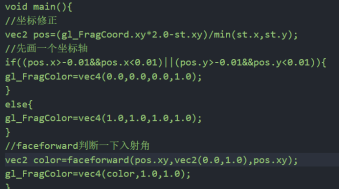
**dot(a,b)点乘。cross(a,b)叉乘。normalize()归一化。**

点乘叉乘归一化三个方法做了一个简单的雷达。

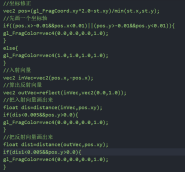
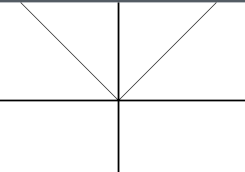
**可以再优化一下。**



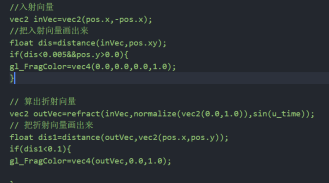
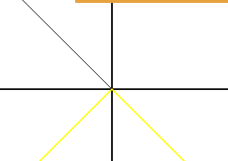
**faceforward(N,I,ner)I和ner的夹角大于90度不变否则N取反。这个判断入射角。**



**reflect（I,N）入射光线和法线输入就行。这个类似于反射。**

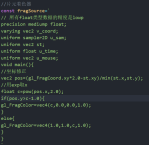
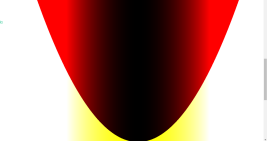


**refract(I,N,折射率)入射光线，法线和折射率。这个类似于折射。**

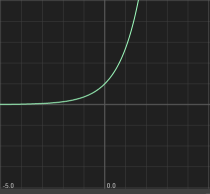
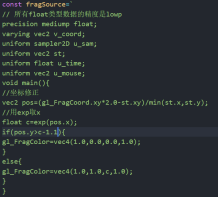


**指数对数函数**

**pow(a,x)a的x次方**

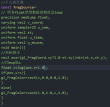
这便是这个函数的图像。真没啥好说的就是一个数学方法。

**exp(x)e的x次方**

这便是这个函数的图像。真没啥好说的就是一个数学方法。

**log(x)这是默认10为底的对数，如果不是10就用Log2（x）以二为底等等。**

主要用来平滑过渡与超级亮度

这便是这个函数的图像。真没啥好说的就是一个数学方法。

**矢量函数**

bvec2 any(bvec2)判断是否有。

bvec2 all(bvec2)判断是否所有。

bvec2 greaterthan(vec2,vec2)判断是否大。

bvec2 lessthan(vec2,vec2)判断是否小。

equal(vec2 ,vec2)判断是否相等。

**感觉这部分不自己封装底层的架构是不会用到的,了解一下就行了。**

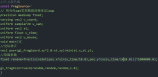
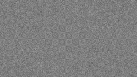
**矩阵**

mat2,mat3,mat4,**在webGL里已经比较详细的写过了,这里就不会多说了**,Martixcompmult(mat2,mat2)返回两个矩阵对位乘得到的mat2矩阵，同时也支持，mat3，mat4。**输入的其实都是行主序的，不过用的时候webGL会给他转乘列主序的。**

**随机数**

在shader中并没有固定的随机数函数，所以要自己写。**本质其实就是fract(sinx\*100000)得到一个伪随机效果。**后面的数不一定要是100000只要够大就行。雪花效果就是这么做的。

可以传一个一维浮点数，自然也能传vec2矢量，就比如说传入一个矢量返回浮点数我们就用fract(sin(dot(vec,vec))\*大数)这样return 的值是一个随机的浮点数。也可以返回vec2矢量，只需要把sinx中的x换成vec2就能达到这种要求，例如fract(sin(vec2(dot(),dot())\*10000）这样返回的就是一个vec2的矢量。**下面是用随机数生成的一个效果。**



**噪音**

之所以叫噪音是因为他的函数图像为噪音。

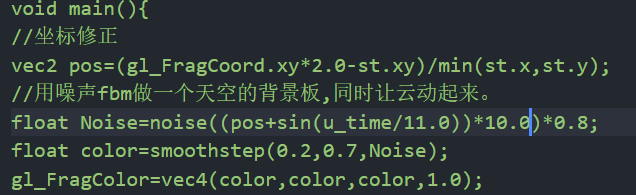
**noise函数是基于随机数基础上的有比较有规律的随机数。如何做到有规律的随机，那就需要传入值分别用fract和floor取到小数和正数，通过小数和整数来取到随机值，在有随机值的基础上也要有一个有规律的值，然后通过mix函数将数进行混合，就可以实现噪音效果。**就比如mix(random(int),random(float)，u)这样就实现了一个噪音，1D，2D，3D噪音的本质都是一样的，水面，云雾，这种都是靠的噪音。

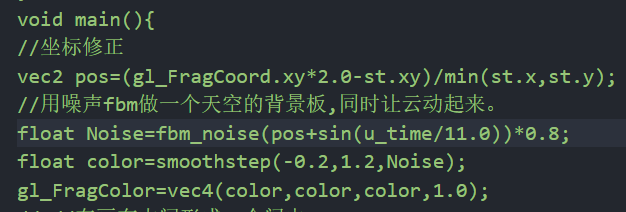
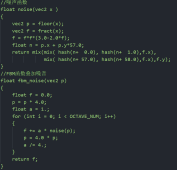
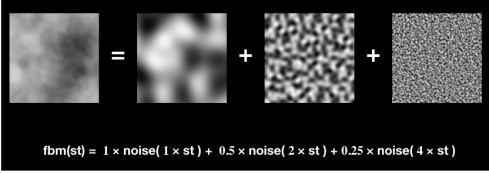
**同样的有时候需要多层噪音叠加，就会用到FBM，FBM本质就是噪音的一个叠加，其核心就是一个累加for循环与累乘变数据，在这里就不过多进行赘述了。**

**个人认为噪音可以由写成y=a\*noise(bx)这样的。经查询任何有周期的函数其实都是可以用这个公式来进行粗略的表述的。因为有周期,所以说他是有规律的随机数。**

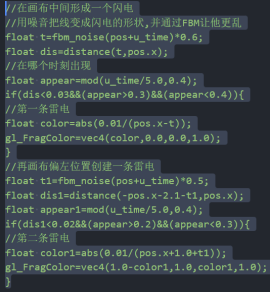
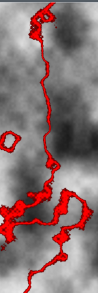
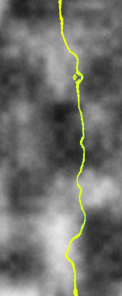
**下面是一个噪音的示例:**

**我们只用噪音生成的结果不是很好。**

查了一下资料要用FBM效果才好。FBM其实就是多个noise的叠加。



**然后用FBM做个闪电效果：其实就是FBM的随机性对直线实现一个变形。**

本来线是一条直的,FBM是一个随机的数,用在线的X上面就让他变成了歪歪扭扭的闪电形状了。就是这么简单。

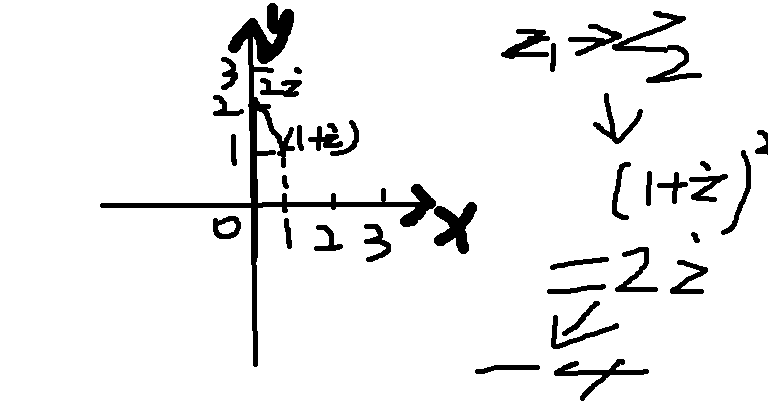
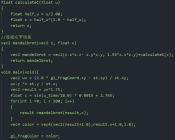
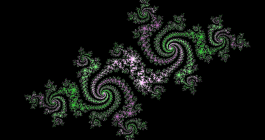
**分型**

**分型是噪声的一种**

其实就是通过一定的函数关系将形状抽象化，并用if语句来实现一个**类似递归的效果**，他达成的目的是在一定范围内实现收敛，一定范围内实现发散，比如说吧，一个圆他的各个点经过一个函数方程操作会变形成一个不是圆的图型。**而这个图型的边界是收敛到这里的，超过这个范围就是发散的了没有规律。**比较抽象，暂时理解到这里就够了。比较形象的就是说海边的石头，一直细分，无限收敛，永远分不完，但是越收敛越细。

分型大概有**曼德布罗特集**和**茱莉亚集**两种。**其本质上可以说就是一个递归复数运算。**

**下面是一个分型的实例：用到的时候就明白了。**

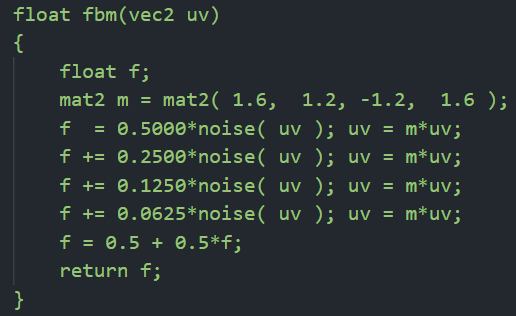
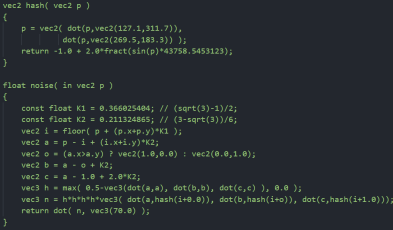
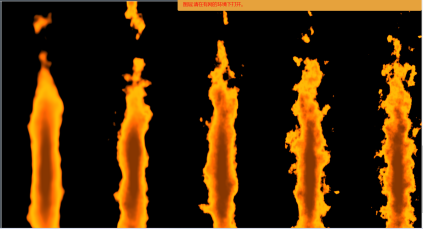


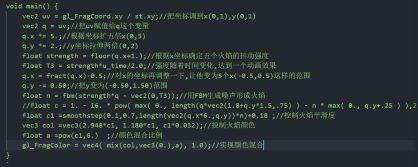
**纹理**

**在webGL里已经比较详细的写过了,这里就不会多说了**，所以他的本质意思其实就是一个coord的UV坐标，一个uniform传入的纹理，通过texture2D(uniform,coord)或者，textuerCube(uniform,normalize())三弟纹理用到的是法向量。

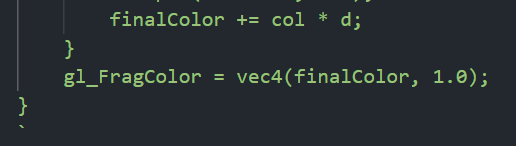
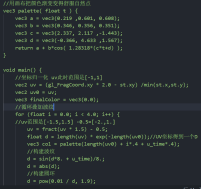
**着色器的案例：**

**火焰效果:**



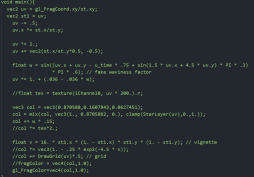
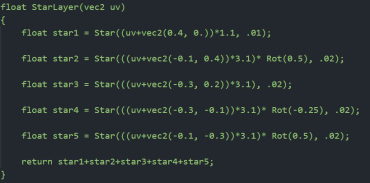
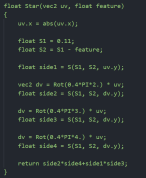
**大概思路:把整个坐标系修正为x(0,5）,y(0,2）这个范围。之后根据坐标确定一个火焰抖动强度。强度确定后,再根据噪音FBM创建基础火焰效果。再用smoothstep把火焰变得平滑，之后调一下火焰的颜色。最后把火焰颜根据距离与背景色进行一个混合就行了。**

**圆环叠加特效：**



**大概思路:其实他的核心代码是下面的这一段。用abs和length构建圆环,然后再用fract实现多个圆环的效果。For循环创建不同圆环就成了现在这个样子。**

**五星红旗**



**大概思路:星星的绘制其实非常的典型就是用dot判断角度画两个三角形之后合起来,封装成函数之后创建五次星星，之后调UV坐标把五个星星转到正确角度。之后用mix把颜色混和成黄红色。最后用sIn函数配合时间实现一个迎风飘的效果。**

**充电特效**

**大概思路：就是sdf拼接矩形三角形形成了这么一个特效。**

**实践是检验真理的唯一标准,有了上面的基础基本上就算是会shader了。可以去shadertoy上去抄点自己喜欢的shader用一下。当做一个实践。**

