[**GLSL语言基础**](http://www.cnblogs.com/kex1n/p/3941680.html)

from http://www.kankanews.com/ICkengine/archives/120870.shtml

**变量**

GLSL的变量命名方式与C语言类似。变量的名称可以使用字母，数字以及下划线，但变量名不能以数字开头，还有变量名不能以gl\_作为前缀，这个是GLSL保留的前缀，用于GLSL的内部变量。当然还有一些GLSL保留的名称是不能够作为变量的名称的。

**基本类型**

除了布尔型，整型，浮点型基本类型外，GLSL还引入了一些在着色器中经常用到的类型作为基本类型。这些基本类型都可以作为结构体内部的类型。如下表:

|  |  |
| --- | --- |
| 类型 | 描述 |
| void | 跟C语言的void类似，表示空类型。作为函数的返回类型，表示这个函数不返回值。 |
| bool | 布尔类型，可以是true 和false，以及可以产生布尔型的表达式。 |
| int | 整型 代表至少包含16位的有符号的整数。可以是十进制的，十六进制的，八进制的。 |
| float | 浮点型 |
| bvec2 | 包含2个布尔成分的向量 |
| bvec3 | 包含3个布尔成分的向量 |
| bvec4 | 包含4个布尔成分的向量 |
| ivec2 | 包含2个整型成分的向量 |
| ivec3 | 包含3个整型成分的向量 |
| ivec4 | 包含4个整型成分的向量 |
| mat2 或者 mat2x2 | 2×2的浮点数矩阵类型 |
| mat3或者mat3x3 | 3×3的浮点数矩阵类型 |
| mat4x4 | 4×4的浮点矩阵 |
| mat2x3 | 2列3行的浮点矩阵（OpenGL的矩阵是列主顺序的） |
| mat2x4 | 2列4行的浮点矩阵 |
| mat3x2 | 3列2行的浮点矩阵 |
| mat3x4 | 3列4行的浮点矩阵 |
| mat4x2 | 4列2行的浮点矩阵 |
| mat4x3 | 4列3行的浮点矩阵 |
| sampler1D | 用于内建的纹理函数中引用指定的1D纹理的句柄。只可以作为一致变量或者函数参数使用 |
| sampler2D | 二维纹理句柄 |
| sampler3D | 三维纹理句柄 |
| samplerCube | cube map纹理句柄 |
| sampler1DShadow | 一维深度纹理句柄 |
| sampler2DShadow | 二维深度纹理句柄 |

结构体

**结构体**

结构体可以组合基本类型和数组来形成用户自定义的类型。在定义一个结构体的同时，你可以定义一个结构体实例。或者后面再定义。

struct surface {float indexOfRefraction;

vec3 color;float turbulence;

} mySurface;

surface secondeSurface;

你可以通过=为结构体赋值，或者使用 ==，!=来判断两个结构体是否相等。

mySurface = secondSurface;

mySurface == secondSurface;

只有结构体中的每个成分都相等，那么这两个结构体才是相等的。访问结构体的内部成员使用. 来访问。

vec3 color = mySurface.color + secondSurface.color;

结构体至少包含一个成员。固定大小的数组也可以被包含在结构体中。GLSL的结构体不支持嵌套定义。只有预先声明的结构体可以嵌套其中。

struct myStruct {

  vec3 points[3]; //固定大小的数组是合法的

  surface surf;  //可以，之前已经定义了

  struct velocity {  //不合法float speed;

    vec3 direction;

  } velo;

  subSurface sub; //不合法，没有预先声明；};struct subSurface {  int id;

};

**数组**

GLSL中只可以使用一维的数组。数组的类型可以是一切基本类型或者结构体。下面的几种数组声明是合法的：

surface mySurfaces[];

vec4 lightPositions[8];

vec4 lightPos[] = lightPositions;const int numSurfaces = 5;

surface myFiveSurfaces[numSurfaces];float[5] values;

指定显示大小的数组可以作为函数的参数或者使返回值,也可以作为结构体的成员.数组类型内建了一个length()函数，可以返回数组的长度。

lightPositions.length() //返回数组的大小 8

最后，你不能定义数组的数组。

**修饰符**

变量的声明可以使用如下的修饰符。

|  |  |
| --- | --- |
| 修饰符 | 描述 |
| const | 常量值必须在声明是初始化。它是只读的不可修改的。 |
| attribute | 表示只读的顶点数据，只用在顶点着色器中。数据来自当前的顶点状态或者顶点数组。它必须是全局范围声明的，不能再函数内部。一个attribute可以是浮点数类型的标量，向量，或者矩阵。不可以是数组或则结构体 |
| uniform | 一致变量。在着色器执行期间一致变量的值是不变的。与const常量不同的是，这个值在编译时期是未知的是由着色器外部初始化的。一致变量在顶点着色器和片段着色器之间是共享的。它也只能在全局范围进行声明。 |
| varying | 顶点着色器的输出。例如颜色或者纹理坐标，（插值后的数据）作为片段着色器的只读输入数据。必须是全局范围声明的全局变量。可以是浮点数类型的标量，向量，矩阵。不能是数组或者结构体。 |
| centorid varying | 在没有多重采样的情况下，与varying是一样的意思。在多重采样时，centorid varying在光栅化的图形内部进行求值而不是在片段中心的固定位置求值。 |
| invariant | (不变量)用于表示顶点着色器的输出和任何匹配片段着色器的输入，在不同的着色器中计算产生的值必须是一致的。所有的数据流和控制流，写入一个invariant变量的是一致的。编译器为了保证结果是完全一致的，需要放弃那些可能会导致不一致值的潜在的优化。除非必要，不要使用这个修饰符。在多通道渲染中避免z-fighting可能会使用到。 |
| in | 用在函数的参数中，表示这个参数是输入的，在函数中改变这个值，并不会影响对调用的函数产生副作用。（相当于C语言的传值），这个是函数参数默认的修饰符 |
| out | 用在函数的参数中，表示该参数是输出参数，值是会改变的。 |
| inout | 用在函数的参数，表示这个参数即是输入参数也是输出参数。 |

**内置变量**

内置变量可以与固定函数功能进行交互。在使用前不需要声明。顶点着色器可用的内置变量如下表：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| gl\_Color | vec4 | 输入属性-表示顶点的主颜色 |
| gl\_SecondaryColor | vec4 | 输入属性-表示顶点的辅助颜色 |
| gl\_Normal | vec3 | 输入属性-表示顶点的法线值 |
| gl\_Vertex | vec4 | 输入属性-表示物体空间的顶点位置 |
| gl\_MultiTexCoordn | vec4 | 输入属性-表示顶点的第n个纹理的坐标 |
| gl\_FogCoord | float | 输入属性-表示顶点的雾坐标 |
| gl\_Position | vec4 | 输出属性-变换后的顶点的位置，用于后面的固定的裁剪等操作。所有的顶点着色器都必须写这个值。 |
| gl\_ClipVertex | vec4 | 输出坐标，用于用户裁剪平面的裁剪 |
| gl\_PointSize | float | 点的大小 |
| gl\_FrontColor | vec4 | 正面的主颜色的varying输出 |
| gl\_BackColor | vec4 | 背面主颜色的varying输出 |
| gl\_FrontSecondaryColor | vec4 | 正面的辅助颜色的varying输出 |
| gl\_BackSecondaryColor | vec4 | 背面的辅助颜色的varying输出 |
| gl\_TexCoord[] | vec4 | 纹理坐标的数组varying输出 |
| gl\_FogFragCoord | float | 雾坐标的varying输出 |

片段着色器的内置变量如下表：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| gl\_Color | vec4 | 包含主颜色的插值只读输入 |
| gl\_SecondaryColor | vec4 | 包含辅助颜色的插值只读输入 |
| gl\_TexCoord[] | vec4 | 包含纹理坐标数组的插值只读输入 |
| gl\_FogFragCoord | float | 包含雾坐标的插值只读输入 |
| gl\_FragCoord | vec4 | 只读输入，窗口的x,y,z和1/w |
| gl\_FrontFacing | bool | 只读输入，如果是窗口正面图元的一部分，则这个值为true |
| gl\_PointCoord | vec2 | 点精灵的二维空间坐标范围在(0.0, 0.0)到(1.0, 1.0)之间，仅用于点图元和点精灵开启的情况下。 |
| gl\_FragData[] | vec4 | 使用glDrawBuffers输出的数据数组。不能与gl\_FragColor结合使用。 |
| gl\_FragColor | vec4 | 输出的颜色用于随后的像素操作 |
| gl\_FragDepth | float | 输出的深度用于随后的像素操作，如果这个值没有被写，则使用固定功能管线的深度值代替 |

**表达式**

**操作符**

GLSL语言的操作符与C语言相似。如下表（操作符的优先级从高到低排列）

|  |  |
| --- | --- |
| 操作符 | 描述 |
| () | 用于表达式组合，函数调用，构造 |
| [] | 数组下标，向量或矩阵的选择器 |
| . | 结构体和向量的成员选择 |
| ++ – | 前缀或后缀的自增自减操作符 |
| + – ! | 一元操作符，表示正 负 逻辑非 |
| \* / | 乘 除操作符 |
| + - | 二元操作符 表示加 减操作 |
| <> <= >= == != | 小于，大于，小于等于， 大于等于，等于，不等于 判断符 |
| && || ^^ | 逻辑与 ，或，  异或 |
| ?: | 条件判断符 |
| = += –= \*=  /= | 赋值操作符 |
| , | 表示序列 |

像 求地址的& 和 解引用的 \* 操作符不再GLSL中出现，因为GLSL不能直接操作地址。类型转换操作也是不允许的。 位操作符(&,|,^,~, <<, >> ,&=, |=, ^=, <<=, >>=)是GLSL保留的操作符，将来可能会被使用。还有求模操作（%，%=)也是保留的。

**数组访问**

数组的下标从0开始。合理的范围是[0, size - 1]。跟C语言一样。如果数组访问越界了，那行为是未定义的。如果着色器的编译器在编译时知道数组访问越界了，就会提示编译失败。

vec4 myColor, ambient, diffuse[6], specular[6];

myColor = ambient + diffuse[4] + specular[4];

**构造函数**

构造函数可以用于初始化包含多个成员的变量，包括数组和结构体。构造函数也可以用在表达式中。调用方式如下：

vec3 myNormal = vec3(1.0, 1.0, 1.0);

greenTint = myColor + vec3(0.0, 1.0, 0.0);

ivec4 myColor = ivec4(255);

还可以使用混合标量和向量的方式来构造，只要你的元素足以填满该向量。

vec4 color = vec4(1.0, vec2(0.0, 1.0), 1.0);

vec3 v = vec3(1.0, 10.0, 1.0);

vec3 v1 = vec3(v);

vec2 fv = vec2(5.0, 6.0);

float f = float(fv); //用x值2.5构造，y值被舍弃

对于矩阵，OpenGL中矩阵是列主顺序的。如果只传了一个值，则会构造成对角矩阵，其余的元素为0.

mat3 m3 = mat3(1.0);

构造出来的矩阵式：

1.0 0.0 0.0

0.0 1.0 0.0

0.0 0.0 1.0

mat2 matrix1 = mat2(1.0, 0.0, 0.0, 1.0);

mat2 matrix2 = mat2(vec2(1.0, 0.0), vec2(0.0, 1.0));

mat2 matrix3 = mat2(1.0);

mat2 matrix4 = mat2(mat4(2.0)); //会取 4×4矩阵左上角的2×2矩阵。

构造函数可以用于标量数据类型的转换。GLSL不支持隐式或显示的转换，只能通过构造函数来转。其中int转为float值是一样的。float转为int则小数部分被丢弃。int或float转为bool，0和0.0转为false，其余的值转为true. bool转为int或float，false值转为0和0.0，true转为1和1.0.

float f = 1.7;

int I = int(f); // I = 1

数组的初始化，可以在构造函数中传入值来初始化数组中对应的每一个值。

ivec2 position[3] = ivec2[3]((0,0), (1,1), (2,2));

ivec2 pos2[3] = ivec2[]((3,3), (2,1), (3,1));

构造函数也可以对结构体进行初始化。其中顺序和类型要一一对应。

struct surface {  int  index;

  vec3 color;  float rotate;

};

surface mySurface = surface(3, vec3(red, green, blue), 0.5);

**成分选择**

向量中单独的成分可以通过{x,y,z,w},{r,g,b,a}或者{s,t,p,q}的记法来表示。这些不同的记法用于顶点，颜色，纹理坐标。在成分选择中，你不可以混合使用这些记法。其中{s,t,p,q}中的p替换了纹理的r坐标，因为与颜色r重复了。下面是用法举例：

vec3 myVec = {0.5, 0.35, 0.7};float r = myVec.r;float myYz = myVec.yz;float myQ = myVec.q;//出错，数组越界访问，q代表第四个元素float myRY = myVec.ry; //不合法，混合使用记法

较特殊的使用方式，你可以重复向量中的元素，或者颠倒其顺序。如：

vec3 yxz = myVec.yxz; //调换顺序vec4 mySSTT = myVec.sstt; //重复其中的值

在赋值是，也可以选择你想要的顺序，但是不能重复其中的成分。

vec4 myColor = {0.0, 1.0, 2.0, 1.0};

myColor.x = -1.0;

myColor.yz = vec2(3.0, 5.0);

myColor.wx = vec2(1.0, 3.0);

myColor.zz = vec2(2.0, 3.0); //不合法

我们也可以通过使用下标来访问向量或矩阵中的元素。如果越界那行为将是未定义的。

float myY = myVec[1];

在矩阵中，可以通过一维的下标来获得该列的向量(OpenGL的矩阵是列主顺序的)。二维的小标来获得向量中的元素。

mat3 myMat = mat3(1.0);

vec3 myVec = myMat[0]; //获得第一列向量 1.0, 0.0, 0.0float f = myMat[0][0]; // 第一列的第一个向量。

**控制流**

**循环**

与C和C++相似，GLSL语言也提供了for, while, do/while的循环方式。使用continue跳入下一次循环，break结束循环。

for (l = 0; l < numLights; l++)

{if (!lightExists[l])continue;

    color += light[l];

}while (i < num)

{

    sum += color[i];

    i++;

}do{

    color += light[lightNum];

    lightNum--;

}while (lightNum > 0)

**if/else**

color = unlitColor;if (numLights > 0)

{

    color = litColor;

}else{

    color = unlitColor;

}

**discard**

片段着色器中有一种特殊的控制流成为discard。使用discard会退出片段着色器，不执行后面的片段着色操作。片段也不会写入帧缓冲区。

if (color.a < 0.9)

discard;

**函数**

在每个shader中必须有一个main函数。main函数中的void参数是可选的，但返回值是void时必须的。

void main(void)

{

 ...

}

GLSL中的函数，必须是在全局范围定义和声明的。不能在函数定义中声明或定义函数。函数必须有返回类型，参数是可选的。参数的修饰符(in, out, inout, const等）是可选的。

//函数声明bool isAnyNegative(const vec4 v);//函数调用void main(void)

{bool isNegative = isAnyNegative(gl\_Color);

    ...

}//定义bool isAnyNegative(const vec4 v)

{if (v.x < 0.0 || v.y < 0.0 || v.z < 0.0 || v.w < 0.0)return true;elsereturn false;

}

结构体和数组也可以作为函数的参数。如果是数组作为函数的参数，则必须制定其大小。在调用传参时，只传数组名就可以了。

vec4 sumVectors(int sumSize, vec4 v[10]);void main()

{

    vec4 myColors[10];

    ...

    vec4 sumColor = sumVectors(5, myColors);

}

vec4 sumVectors(int sumSize, vec4 v[10])

{int i = 0;

    vec4 sum = vec4(0.0);for(; i < sumSize; ++i)

    {

        sum += v[i];

    }return sum;

}

GLSL的函数是支持重载的。函数可以同名但其参数类型或者参数个数不同即可。

float sum(float a, float b)

{return a + b;

}

vec3 sum(vec3 v1, vec3 v2)

{return v1 + v2;

}

GLSL中函数递归是不被允许的。其行为是未定义的。