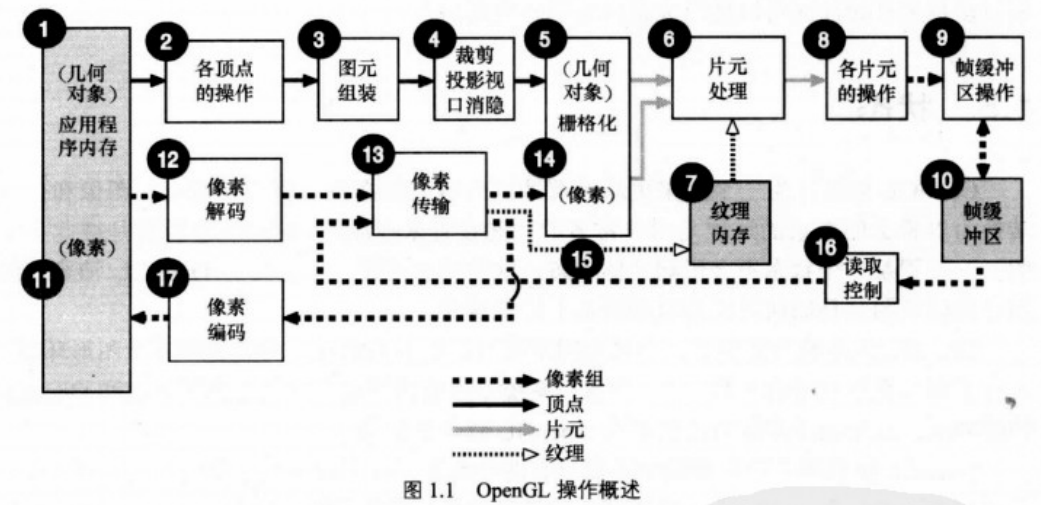
一次一个顶点与顶点数组都是立即模式，这是因为图元是在指定后立即渲染的。

显示列表：显示列表从glNewList开始，以glEndList结束，在这两个调用之间发送的所有命令都将成为显示列表的一部分，



**片元处理**

在进行光栅化之后，片元上会发生许多操作，这些操作被统成为“片元处理”，在这个阶段发现的可能最重要的操作是被称为“上纹理”。在这个操作中，与片元相关联的纹理坐标会被用来访问一个名为“纹理内存”的图形内存区域

在这个阶段发现的其他操作是“雾化”（根据片元距离当前视点的位置修改其颜色）和“颜色汇总”（将片元的朱颜色和次颜色的值结合在一起）。雾化参数是通过调用glFog设置的。而次颜色则是可以通过使用顶点属性命令glSecondaryColor来传递。或者通过光照阶段来计算的顶点属性

**片元测试**

像素所有权测试、裁剪测试、alpha测试、模板测试、深度测试、混合、抖动、逻辑操作以及遮蔽

一个程序通常会包含两个着色器：一个顶点着色器和一个片元着色器。每一种着色器都可以有不止一个。为了处理一个单独的图元，前面的着色器会多次执行，针对每个顶点都会执行一次顶点着色器，针对每个片元都会执行一次片元着色器。同一个着色器的多个执行是可以并行发生的。总的来说，着色器的执行之间没有直接的限制或顺序。信息不能在顶点之间传输，也不能在片元之间传输。

**标量**

bool float int

glsl中不支持类型提升，比如float a = 4就是错误的

**矢量**

float、int或bool矢量是内置的基本变量，它们可以有2个、3个或者4个

vec2(3,4) 包含2（3、4）个浮点数的矢量

ivec2(3,4) 包含2（3、4）个整数的矢量

bvec2(3,4) 包含2（3、4）个bool的矢量

我们可以使用下面的名称来选择矢量的各个部分：

x y z w 将矢量看做一个位置或者方向

r g b a 将矢量看做一种颜色

s t p q 将矢量看做一个纹理坐标

可以像从0开始的数组那样对矢量使用索引来获得各个分量，比如position[2]返回的是position的第三个分量



**矩阵**

mat2(3,4) 2\*2（3\*3、4\*4）的浮点数矩阵

用户可以将矩阵作为列矢量的数组来访问，也就是说，如果一个transform是一个mat4，那么transform[2]就是第三列，transform[2]的最终类型就是vec4，因此transform[2][1]是第三列第二分量

**取样器**

纹理查找需要指定哪一个纹理或者纹理单元将执行查找。Opengl着色语言实际上并不关心纹理单元的底层实现或者组织纹理查找硬件的其他方式。因此，它提供了一个简单而不透明的句柄来封装要查找的对象。这些句柄被称为“采样器”，可用的采样器有：

Sample1(2,3)D 访问一个一（二、三）维纹理

SampleCube 访问一个立方贴图纹理

Sample1DShadow 访问一个带对比的一维深度纹理

Sample2DShadow 访问一个带对比的二维深度纹理

当应用程序初始化一个采样器时，opengl实现将在其中存储说明要查找何种纹理所需要的信息。着色器本身不能初始化取样器，他们只能通过一个uniform限定的取样器从应用程序接受取样器，或者将它们传递给用户或内置函数。作为一个函数参数，取样器不能被修改，因此着色器不能更改一个取样器的值

例如：可以这样声明一个取样器

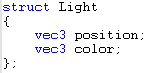
Uniform sample2D grass;

然后，可以将这个变量传递到对应纹理查找函数，以便访问一个纹理：

Vec4 color = texture2D(grass, coord)

Coord是一个vec2，存储了要用来作为草地纹理的索引的二维位置，color是执行纹理查找的结果。编译器和opengl api会一起验证grass确实封装了一个二维纹理，并且会验证grass只被传递到二维纹理查找中。

**结构**



目前，结构是唯一的用户定义的类型，关键字union、enum和class得以保留，将来可能会用到它们。

**数组**

该语言可以创建任意类型的数组，声明

Vec4 points[10];

创建的是一个包含10个vec4的数组，其索引是从0开始的。此处不存在指针。声明数组的唯一方式就是使用中括号。

在声明数组时不必指定大小，下面这样的声明是可行的：

Vec4 points[]

只要符合下面的两种情况之一：

1）在引用数组之前，要使用大小再次声明它，其他类型与第一次声明时相同



如上面的定义是正确的

但是在这之后不能再次声明它

2）静态引用数组的所有索引都是编译时的常量，在这种情况下，编译器会使数组足够大，易变存储它看到的是被使用的最大索引，例如：

Vec4 points[]; // points是一个未知大小的数组

Points[2] = vec4(1.0) // points是一个大小为3的数组

Points[7] = vec4(2.0) // points现在是一个大小为8的数组

这种情况下，在运行时数组只有一个大小，这是由编译器看到的最大索引决定的

共享同一个数组的多个着色器可以分别使用不同的大小来声明它，连接器会将数组的大小指定为在连接到一起的所有着色器中出现的最大大小

**Void**

类型void用来声明一个不反悔任何值的函数，除了用于不反悔任何值的函数之外，void类型没有其他用途

**类型匹配与提升**

Opengl着色语言对类型匹配是非常严格的。总的来说，要赋值的类型必须相互匹配，传递到函数的参数类型必须与形参声明相匹配，且在其上执行运算的类型必须与运算符的要求相匹配。从一种类型到另一种类型的自动提升是不存在的，有时，这可能会使着色器执行一次额外的明确转换，不过，这也简化了着色语言，避免了某些形式的混乱代码和某些类型的缺陷。例如，对于一个指定的函数调用，应该选择哪一个重载的函数是非常明确的。

**初始化器和构造函数**

着色器变量可以在声明时进行初始化，常量限定的变量必须初始化，在声明时不能初始化属性变量、一致变量和易变变量

该语言中没有c中使用大括号语法“{…}”的初始化器，而只有构造函数。从语法上说，构造函数看起来就像是在将出现函数名称的位置上使用了类型名称的函数调用。例如：

Vec4 v = vec4(1.0, 2.0, 3.0, 4.0)

矢量的内置构造函数也可以接受一个单独的参数，这个参数将被复制到各个部分中：

Vec3 v = vec3(0.6)

等价于：

Vec3 v = vec3(0.6, 0.6, 0.6)

只有矢量才是这样的，结构必须对要构成的每一个成员都接受一个参数。矩阵构造函数也有一种单一参数的形式，但对于这种情况，它只是初始化矩阵的对角线，其余部分将初始化为0.0

Mat2 m = mat2(1.0) // 创建一个2\*2的单位矩阵

等价于：

Mat2 m = mat2(1.0, 0.0, 1.0, 0.0)

**类型转换**

明确类型转换是使用构造函数完成的，例如



在该语言中，只可以使用构造函数来执行类型转换，构造函数的参数会被转换为它们要构造的类型。

**限定符和着色器接口**

变量和函数形参都可以使用限定符前缀，使用限定符可以修饰函数的形参（const、in、out和inout）

Attribute：用于经常更改的信息，从应用程序到顶点着色器

Uniform：用于不经常更改的信息，用于顶点着色器和片元着色器

Varying：用于从顶点着色器传递到片元着色器的差值得到的信息

Const：用来声明非可写的便是时常量变量（就像在C中那样）

将信息传入和传出着色器与更典型的编程环境有很大的不同。针对着色器的信息传入和传出是通过读取和写入内置变量以及用户定义的attribute、uniform和varying变量来实现的。

使用attribute、uniform和varying限定的变量必须是在全局范围内声明的。其合理之处在于，他们在着色器外部是可见的，而且对于一个单独的程序来说，它们都共享单一的名称空间。限定符总是在变量类型之前指定的，且因为不存在默认类型，所以限定的变量声明的形式总是包括一种类型。

**缺少限定符**

如果在声明变量（而非函数参数）时没有指定限定符，那么着色器可以读取和写入这个变量。在全局范围上声明的未限定变量可以在同一个程序中链接的同种着色器之间共享。顶点着色器和片元着色器分别具有它们自己针对未限定全局变量的全局名称空间。不过，未限定的 用户定义变量在程序外部是不可见的，这种特权是为使用attribute或者uniform限定的变量以及表示opengl状态的内置变量而保留的。

对于顶点着色器，因为未限定的全局变量具有与片元着色器不同的名称空间，所以通过这种变量在顶点着色器和片元着色器之间共享信息是不可能的。如果将只读变量声明为uniform，就可以共享它们，并且只有通过varying机制，片元着色器才能读取由顶点着色器写入的变量。

**流控制**

与c和c++的大部分规则一样，有一个特殊的分支discard可以避免片元更新帧缓冲区。当流控制遇到这个关键字时，正在处理的片元就会被标记为将要丢弃。实现可能会（也可能不会）继续执行着色器，但是可以确保对帧缓冲区没有影响。

该语言不存在goto关键字或者等价的功能，不存在标签，也没有提供使用switch关键字进行进行切换的功能。

**函数**

函数调用的操作与在C++中是一样的，函数名可以通过参数类型重载，但是不能仅仅通过返回类型进行重载。在调用一个函数之前，作用域中必须有函数定义（主体）或者声明。

注意：函数不能被递归调用（无论世界之的还是间接的）

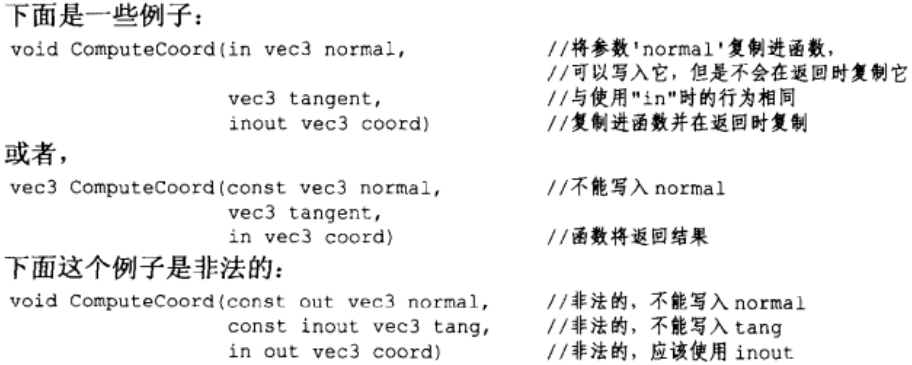
Opengl着色语言使用按值-返回调用作为其调用约定。因为不存在指针，所以函数不需要担心起参数是同一块内存的别名。按值-返回调用的按返回调用部分意味着限定为输出参数的参数将会返回到调用程序，这是通过在函数返回时将其从被调用函数复制到调用函数来实现的。

要想指定复制哪些参数，可以在它们前面使用限定符关键字in、out或inout

in 复制进函数但不在返回时复制，在函数内部仍然是可写的

out 只在返回时复制，是可读的，但是在进入函数时是未定义的

inout 复制进函数并在返回时复制



函数可以返回一个值，也可以不返回任何职，如果函数返回一个值，那么其类型可以是除了数组之外的任何类型，不过可以返回结构，且结构可以包含数组。

着色器中有大量的内置函数，而且着色器可以覆盖这些函数，提供它自己的定义，要想覆盖一个函数，需要提供一个在调用时处于有效作用域的原型或者定义。

**顶点属性**

顶点着色器会使用以下内置名称来访问标准属性。如果在片元着色器中使用了这些名称，则会出现编译器错误：

输入：

Attribute vec4 gl\_Color

Attribute vec4 gl\_SecondaryColor

Attribute vec3 gl\_Normal

Attribute vec4 gl\_Vertex

Attribute float gl\_FogCoord

Attribute vec4 gl\_MultiTexCoordn(其中n=gl\_MaxTextureCoords)

输出：

gl\_Position（变换后的顶点位置，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（用于用户裁剪平面的裁剪）

gl\_PointSize（点的大小）

gl\_FrontColor（正面的主颜色的varying输出）

gl\_BackColor（背面的主颜色的varying输出）

gl\_FrontSecondaryColor（正面的辅助颜色的varying输出）

gl\_BackSecondaryColor（背面的辅助颜色的varying输出）

gl\_TexCoord[]（纹理坐标的数组varying输出）

gl\_FogFragCoord（雾坐标的varying输出）

关于gl\_ClipVertex：

如果启用了用户裁剪，那么在执行了顶点着色器之后，它就会作为固定功能操作来发生作用，为了使用户裁剪能够与顶点着色器的使用正常地协同工作，顶点着色器必须计算出相对于用户定义的裁剪面板的顶点位置。然后，这个值必须存储在内置变量gl\_ClipVertex中。应用程序负责确保顶点着色器计算的裁剪顶点值与用于裁剪面板是在同一个坐标空间中定义的。用户裁剪面板只有在线性变换下才能正常工作。

这些变量都具有全局作用域。他们可以在顶点着色器的执行过程中的任何时刻被写入，写入之后它们就可以被读回，在写入它们之前进行读取将导致不确定的行为，如果多次写入它们，那么后面的操作将采用最后一次写入的值

这些变量只能在顶点着色器的内部引用，而且他们实质上是使用下面的类型声明的：

Vec4 gl\_Position // 必须写入

Float gk\_PointSize // 可以写入

Vec4 gl\_ClipVertex // 可以写入

**内置的易变变量**

如前所描述，易变变量是用来描述在图元上发生变化属性的。顶点着色器负责将需要插值的的值写入易变变量。片元着色器会从易变变量中读取插值的结果并在这些值上进行操作，从而为每个片元生成一个结果值。对片元着色器实际使用的各个用户定义的易变变量来说，必须在顶点着色器中声明一个匹配的易变变量，否则就会出现一个链接错误。

在顶点着色器中可以写入下列内置易变变量。顶点着色器应该写入固定功能片元处理所需要的变量（如果不打算使用片元着色器的话）或者对应的片元着色器所需要的那些变量。

Varying vec4 gl\_FrontColor

Varying vec4 gl\_BackColor

Varying vec4 gl\_FontSecondaryColor

Varying vec4 gl\_BackSecondaryColor

Varying gl\_TexCoord[gl\_MaxTextureCoords]

Varying gl\_FogFragCoord

固定功能退出顶点着色器时，会将写入gl\_FrontColor、gl\_BackColor、gl\_FrontSecondColor和gl\_BackSecondaryColor的值限定在范围[0,1]中。这四个值会与固定功能一起确定图元是正面的还是背面的，以便计算可以在片元着色器中使用的两个易变变量gl\_Color和gl\_SecondaryColor。

使用gl\_TexCoord数组，可以从一个顶点着色器中传递一组或多组纹理坐标。因此如果不存在片元着色器，固定功能处理也可以使用它们，或者，也可以使用gl\_TexCoord易变变量从片元着色器内部访问它们。

对于gl\_FogFragCoord写入的值应该是上一次调用glFog所设置的当前雾化坐标源所需要的值。如果将雾化坐标源设置为GL\_FRAGMENT\_DEPTH，那么写入gl\_FogFragCoord的值就应该是眼睛坐标中从眼睛到顶点的距离（其中眼睛位置假定为（0,0，0,1））。如果将雾化坐标源设置为GL\_FOG\_COORDINATE，那么写入gl\_FogFragCoord的值就应该是要在图元上进行插值的雾化坐标值（也就是内置的属性变量gl\_FogCoord）

**片元着色器**

与顶点着色器相似，当帧缓冲区被配置为一个颜色索引缓冲区而不是RGBA缓冲区（也就时说，opengl处于颜色索引模式）时，片元着色器的行为是不确定的。

下面的内置变量可以在片元着色器中读取。Gl\_Color和gl\_SecondaryColor名称与可以在顶点着色器中使用的内置属性变量名称是相同的。但是，这不会导致名称冲突，因为属性只有在顶点着色器中才能看到，而下面这些变量只有在片元着色器中才能看到。

Varying vec4 gl\_Color

Varying vec4 gl\_SecondaryColor

Varying vec4 gl\_TexCoord[gl\_MaxTextureCoords]

Varying float gl\_FogFragCoord

Gl\_Color和gl\_SecondaryColor的值将从gl\_FrongColor、gl\_BackColor、gl\_SecondFrontColor和gl\_BackSecondaryColor中自动导出，这是固定功能处理的一部分，用来确定片元属于正面的图元还是背面的图元。

Gl\_TexCoord[]数组将包含对来自顶点着色器的gl\_TexCoord[]值进行插值得到的值，或从固定功能能顶点处理得到的纹理坐标，它们的q部分不会对纹理坐标执行自动划分。

当片元着色器处理由于光栅化像素矩形或位图而生成的片元时，如果片元着色器使用的是一个非内置易变变量时，那么结果是不确定的，在这种情况下，因为没有执行顶点着色器，所以内置变易变变量是由当前光栅位置提供的。

变量gl\_FragCoord可以在片元着色器中作为只读变量使用，它含有片元的窗口相对坐标x、y、z和1/w，这个窗口位置值是在顶点处理之后对图元进行插值以生成片元的固定功能的结果。这个内置变量可以用来实现与窗口位置有关的操作，如纱窗透明（例如，在gl\_FragCoord.x为奇数或者gl\_FragCoord.y为奇数，但不是二者都为奇数的任何片元上使用discard）

片元还可以访问只读的内置变量gl\_FrontFacing，如果片元属于一个正面的图元，那么gl\_FrontFacing的值为true，反之为false

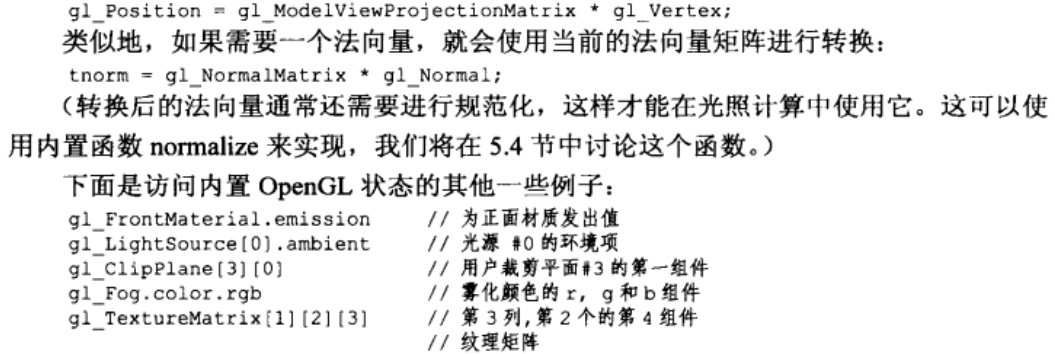
片元着色器的主要作用是计算最终将写入帧缓冲区的值，随后，片元着色器的输出将由位于opengl管线末端的固定功能操作进行处理。片元着色器会使用内置变量gl\_FragColor和gl\_FragDepth将它们的结果发送到opengl管线的末端。这些内置的片元着色器变量具有全局作用域，并且可以被片元着色器写入多次，如果它们被写入了多次，那么最后一次赋予的值就是在后续操作中使用的值。

大多数着色器都将计算gl\_FragColor的一个值，但并不是所有片元着色器都必须计算这个值。着色器只计算gl\_FragDepth也是完全可以的。着色器还可以使用discard关键字将片元标记为将要丢弃的，而不是用来更新帧缓冲区。需要注意的是，如果后续的固定功能使用了片元颜色，但执行的片元着色器没有向gl\_FragColor写入一个值，那么其行为是不确定的。

如果启动了深度缓存，而且着色器没有写入gl\_FragDepth，那么深度的固定函数将用作片元的深度值，否则，写入gl\_FragDepth就会确定正在处理的片元的深度值。退出片元处理器时，这个值将被限定在[0,1]范围内，并转换为固定点，其位数至少会与帧缓冲区中的位数一样多。对于通过着色器的每一条执行路径，写入了gl\_GradDepth的片元着色器都应该一一进行写入。如果在条件语句的一个分支中写入了，但却没有在其他分支写入，那么这个深度值对于某些执行路径而言就是不确定的。

不能保证着色器能够计算出与固定功能值相同的深度值，着色器生成的值可能不会与固定功能生成的值完全相等，即使是通过把gl\_FragCoord.z复制到gl\_FragDepth中来给片元的深度值赋值也是如此。深度恒定性的唯一保证就是：将gl\_FragCoord.z有条件或无条件地赋值给gl\_FragDepth的片元着色器彼此之间深度值不会发生变化。

内置的一致变量的列表还包括一些派生的状态，这些状态值不是由应用程序直接传递的，而是由opengl实现根据传递的值派生的，由于各种原因，让opengl实现计算这些派生值并允许着色器访问它们是非常方便的。法向量矩阵（gl\_NOrmalMatrix）就是这样一个例子，它只不过是模型矩阵左上角的3\*3子集的逆转置，因为它的使用非常频繁，所以要求着色器随时根据模型矩阵计算它是没有意义的，取而代之的是，opengl实现负责在任何需要的时候计算这个值，而且着色器可以通过一个内置的一致变量访问它。



在下面的例子中：

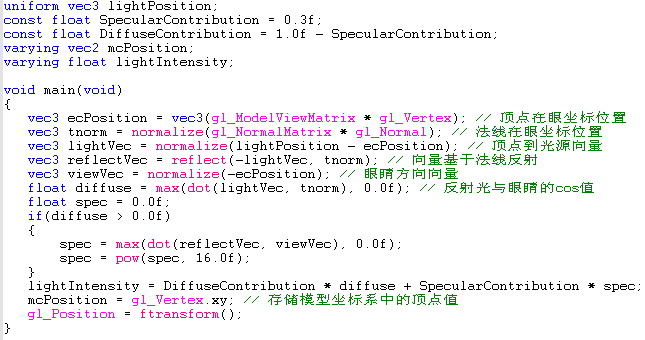
lightPosition = {150.0f, 100.0f, 20.0f}

brickColor = {1.0f, 0.0f, 0.0f}

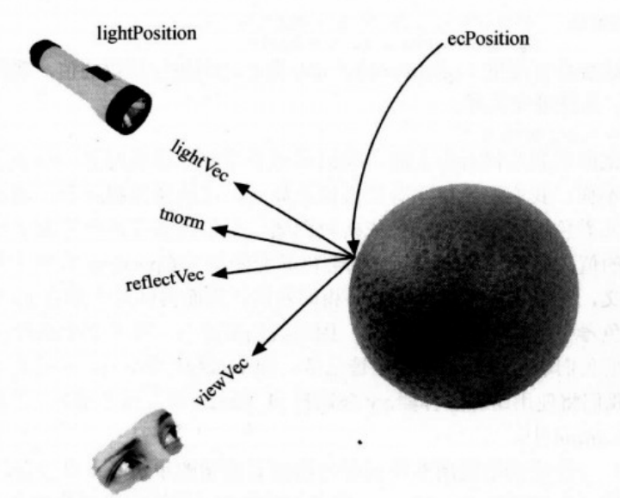
mortarColor = {0.0f, 1.0f, 0.0f}

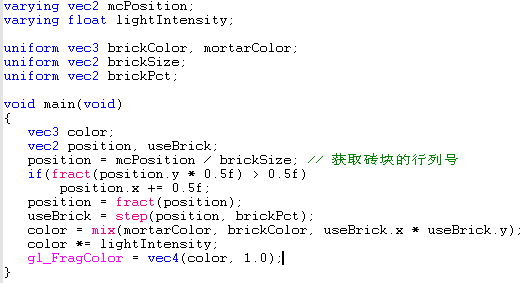
brickSize = {4.5f, 2.3f}

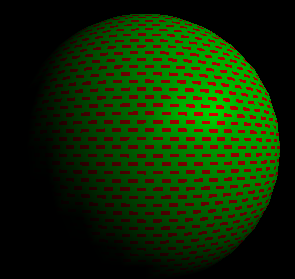
brickPct = {0.568f, 0.44f}



注意：在上面求解viewVec过程中，为-ecPosition，因为这些坐标均存于眼坐标系中







在上面的例子中，我们使用易变变量MCposition保存了gl\_Vertex.xy，将砖块图案应用到几何对象时，我们希望不管如何移动对象，砖块图案相对于对象表面都保持不变。我们还希望不管查看位置如何，砖块图案相对于对象表面都保持不变。为了在片元着色器中通过算法生成砖块图案，我们需要在各个片元上提供一个表示表面上的位置的值。

片元着色器的目的是计算要应用到片元上的颜色或者计算片元的深度值，或者二者都计算。在这个例子中（实际上多数片元着色器都是这样的），我们只关心片元的颜色，由opengl的光栅化阶段计算的深度值就已经很令人满意了，因此，这个着色器的全部用途就是计算当前片元的颜色。

**Opengl着色语言API**

Opengl着色语言API的设计是特意用来模拟c或c++应用程序的过程，第一步是创建源代码，然后必须编译源代码，编译的各个模块必须链接到一起，最后，目标处理器就可以执行得到的代码了。

**glCreateShaderObjectARB**

**glShaderSourceARB**

**glCompileShaderARB**

**glCreateProgramObjectARB**

**glAttachObjectARB**

可以附加到程序对象的着色器对象的数量没有内在的限制，不论着色器对象是否附加到程序对象上，可以在着色器对象上执行的所有操作都是有效的，我们可以再将源代码加载到着色器对象之前或者在编译着色器对象之前将着色器对象附加到程序对象上，我们也可以将一个着色器对象附加到不止 一个程序对象上。换句话说，glAttachObjectARB只是用来指定要链接的着色器对象的集合。

**glLinkProgramARB**

为了创建一个有效的程序，附加到程序对象的所有着色器对象都必须经过编译，且必须链接程序对象本身，链接操作会分配一致变量的位置，解析独立编译的着色器对象之间的引用，并进行检查以确保顶点着色器和片元着色器彼此兼容。

作为一次成功的链接操作，所有属于program的活动的用户定义的一致变量都将初始化为0，且程序对象的每一个活动的一致变量都将被分配一个位置，该位置可以通过调用glGetUniformLocationARB来查询，此外，尚未绑定到一个通用顶点属性索引的所有活动的用户定义的属性变量都将在此时被绑定到一个索引上。

如果链接操作成功，则将生成一个程序，它可能包含一个用于顶点处理器的可执行程序、一个用于片元处理器的可执行程序，或者二者都包含。无论链接操作成功还是失败，上一次链接操作的信息和可执行程序都将丢失。如果program含有类型为GL\_VERTEX\_SHADER\_ARB的着色器对象，但不包含类型为GL\_FRAGMENT\_SHADER\_ARB的着色器对象，那么顶点着色器将根据固定功能片元处理的隐含结构进行链接。类似的，如果program含有类型为GL\_FRAGMENT\_SHADER\_ARB的着色器对象，但是不包含类型为GL\_VERTEX\_SHADER\_ARB的着色器对象，那么片元着色器将根据固定功能顶点处理的隐含接口进行链接。

**glUseProgramObjectARB**

当链接操作成功完成时，它所包含的可执行程序可能作为当前渲染状态的一部分安装。如果program含有类型为GL\_VERTEX\_SHADER\_ARB的着色器对象，但是不包含类型为GL\_FRAGMENT\_SHADER\_ARB的着色器对象，那就会在顶点处理器上安装一个可执行程序，但是将使用固定功能进行片元处理。类似的，如果program含有类型为GL\_FRAGMENT\_SHADER\_ARB的着色器对象，但是不包含类型为GL\_VERTEX\_SHADER\_ARB的着色器对象，那就会在片元处理器上安装一个可执行程序，但是将使用固定功能进行顶点处理，如果program为0，那么可编程的处理器将禁用，顶点处理和片元处理都将使用固定功能。

在可编程的处理器上成功安装一个可执行程序将导致对应的opengl固定功能被禁用。

**glDeleteObjectARB**

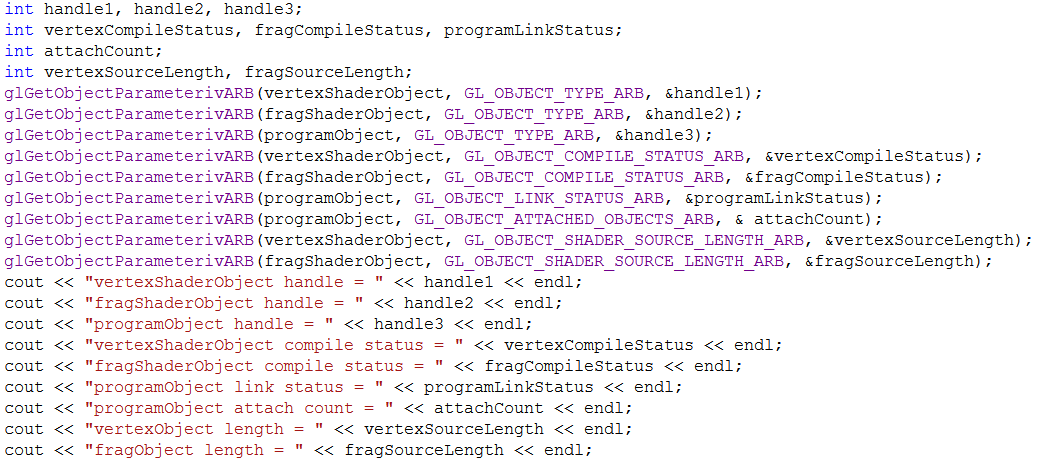
这会释放内存并使与object指定的由opengl管理的对象相关联的句柄无效，这个命令实际上是消除了调用glCreateShaderObjectARB或glCreateProgramObjectARB的效果。

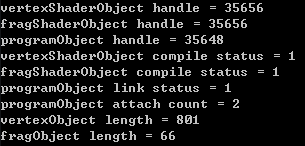
如果要删除的着色器对象被附加到了程序对象上，则它会带上删除标记，但是暂时不会删除它，直到它不再被附加到任何渲染环境的程序对象上为止（也就是说，在删除它之前，必须将它从所附加的对象上分离出来）。如果一个程序对象正被用作当前渲染状态的一部分，则它会带上删除标记，但是暂时不会删除它，直到它不再是任何渲染环境的当前状态的一部分为止。如果要删除的程序对象附加有着色器对象，则这些着色器对象会自动分离，但是不会被删除，除非之前的glDeleteObjectARB调用已经将它们标记为删除。

**glDetachObjectARB**

这会将由shader指定的着色器对象从由program执行的程序对象上分离出来，这个命令可以用来撤销命令glAttachObjectARB的效果。

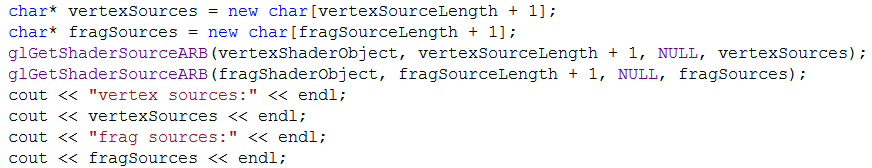
**glGetObjectParameterfvARB**

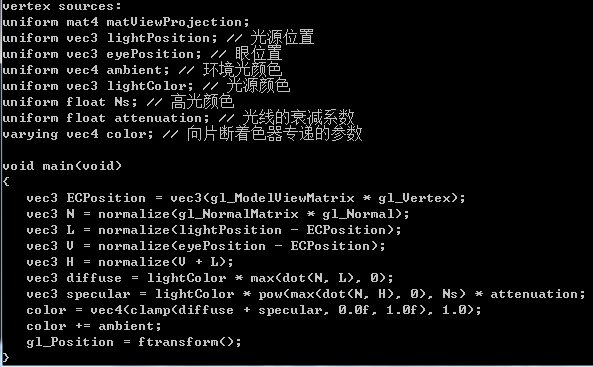


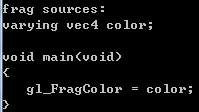


**glGetShaderSourceARB**

返回由shader指定的着色器对象中的源代码字符串

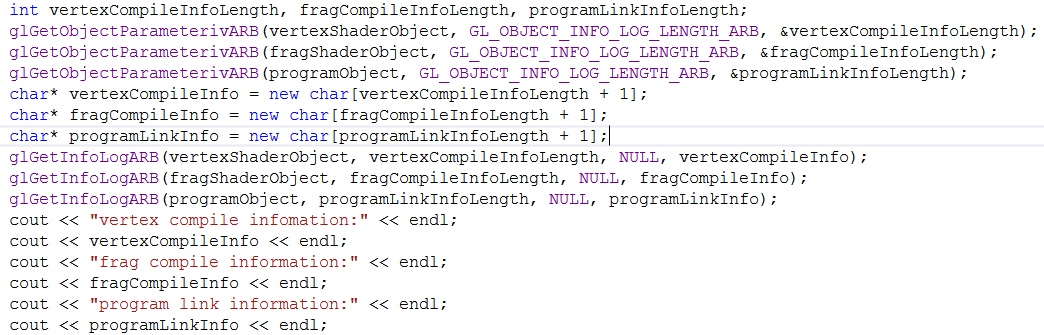


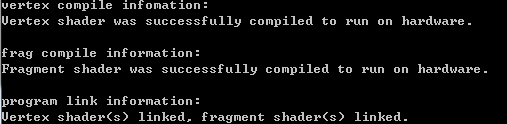




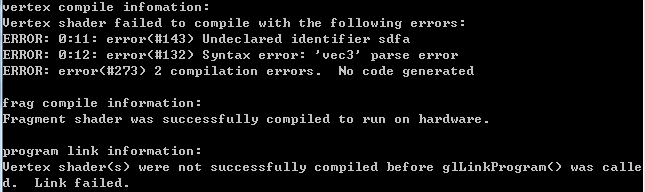
**glGetInfoLogARB**

有关编译操作的信息会存储在着色器对象的信息日志中，类似的，有关链接和验证操作的信息会存储在程序对象的信息日志中，信息日志是一个含有诊断信息和警告的字符串。着色器对象的信息日志是在编译着色器时修改的，而程序对象的信息日志是在连接或验证程序对象时修改的，返回的字符串将以null结束。





如果修改了顶点着色器，添加了一段字母得到的结果是：



Opengl的“一次一个顶点”接口是简单而强大的，但是在如今的系统上，它无疑不是将顶点数据传输到图形加速器的性能最高的方式。取而代之的是，在任何可能的时候，应用程序都应该使用顶点数组接口，使用这个接口可以将顶点数据存储在数组中并设置这些数组的指针，这就不再是一次下个opengl发送一个顶点，我们可以一次发送一个数组图元，有了顶点缓冲对象，甚至可以将顶点数组存储在图形板上的内存中，从而实现最高的性能。

顶点数组接口在opengl可编程环境中的工作方式一以前相同，将一个顶点数组发送到opengl时，顶点数组中的顶点数据会一次一个顶点地进行处理，就像“一次一个顶点”接口那样，如果顶点着色器时活动的，那么各个顶点将由顶点着色器而不是opengl的固定管线进行处理。