**Opengl的固定管线**

最终完成渲染的图像是由在屏幕上绘制的像素组成的。在内存中，和像素有关的信息（如像素的颜色）组织成位平面的形式。位平面是一块内存区域，保存了屏幕上每个像素的一个位的信息。例如，它指定了一个特定像素的颜色中红色成分的强度。位平面又可以组织成帧缓冲区（framebuffer）的形式，后者保存了图形硬件为了控制屏幕上所有像素的颜色和强度所需要的全部信息。

（1）指定几何对象

如：点、线、三角形等一些几何图元，opengl绘制几何图元的方法有以下三种：

1）一次一个顶点，即使用glBegin、glVertex、glEnd指定几何对象

2）使用顶点数组，如glDrawArray、glDrawElements等，一次性绘制大量图元

上面两种模式是立即模式，即指定完图元之后会被立即渲染，即按所有数据发往渲染管线后立即被渲染

3）显示列表模式，它存储于opengl服务端（接收opengl命令的一端），操作函数有glNewList、glEndList、glCallList

（2）顶点处理操作

不管以上的几何对象是如何指定的，所有的几何数据都将会经历这个阶段，这个阶段负责的是逐个顶点的操作

1）顶点变换：根据模型视图和投影矩阵变换

2）光照计算和法线变换（法线矩阵是模型矩阵的左上角3\*3的逆矩阵）和法线规格化

3）纹理坐标变换（纹理矩阵）

4）材质状态：纹理坐标生成

最重要的是变换以及光照，每个顶点在这个阶段分别是单独处理的，这个阶段所接收到的数据则是每个顶点的属性特征，输出则是变换后的顶点数据

（3）图元组装

在顶点处理之后，顶点的全部属性都已经被确定，这个阶段顶点加你工会根据应用程序发送到图元规则GL\_POINTS、GL\_TRIANGLES等将会被组装成图元

（4）图元处理（裁剪、消隐）

1）这个步骤第一个所做的应当是裁剪操作，会将图元与用户定义的裁剪平面，即glClipPlane和模型投影矩阵所建立的视景比较，这将会裁剪且丢掉位于视景和裁剪平面外部的图元，不在予以处理

2）其次，若采用透视投影，那么，将会对每个顶点的xyz坐标分别除以w(1和2颠倒？)

3）紧接着，则是由视口变换将顶点坐标变换到窗口坐标

4）执行消隐操作

（5）栅格化操作

1）由图元处理传递过来的图元数据，在此将会被分解成更小单元并对应帧缓冲区的各个像素，这个单元被称之为片元，一个片元可能包含窗口左边、深度、颜色、纹理坐标等属性

2）片元的属性则是图元上顶点数据等经过插值而确定的，这里生成的片元将会包含主颜色和次颜色，glShaderMode函数的作用将会在这里体现，即使用插值（平滑着色）或者使用最后一个顶点的颜色（平面着色）

3）点宽、线宽、多边形模式、正面背面等一些特征也将会这阶段发生作用

4）反走样也是这个阶段起作用

（6）片元处理

1）纹理；通过纹理坐标取得纹理内存中下对应的颜色

2）雾化：通过片元距离当前视点位置修改颜色

3）颜色汇总：这个与混合完全不同的概念，将纹理，主定义的颜色、雾化的颜色、次颜色光照阶段计算的颜色汇总在一起

（7）逐个片元操作

1）所有的一些测试，像素所有权、剪切、alpha测试、模板测试、深度测试、混合

这些操作将会最后影响其在帧缓冲区的颜色值

（8）帧缓冲操作

1）这个阶段执行帧缓冲的写入等操作，最后产生了显示出来的像素

glColorMask、glStrnciilMask、glDepthMash、glClearDepth、glClearStencil、glClearColor等将在这个阶段影响写入的值

以上只是讨论opengl图元绘制的基本过程，那么基于像素图像的绘制，几乎和上面一样，只是在光栅化处理前的操作不一样，即经过像素解码、像素传输、栅格化最后形成片元，片元之后的处理完全一样

在着色器编程领域，可以实现：

1）Vertex Shader（顶点着色器）：替换顶点处理阶段

2）Fragment Shader（片元着色器）：替换片元处理阶段

3）Geometry Shader（几何着色器）：替换图元组装阶段

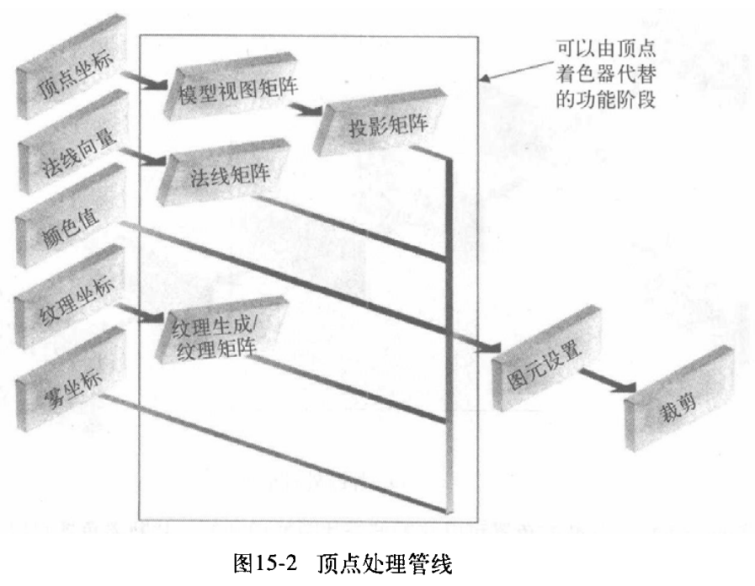
在前面的固定管线中提到，在阶段5：栅格化操作过程中，片元的属性会由图元上顶点数据等经过插值而确定。在定点着色器处理完毕后，opengl都会将顶点与顶点之间的片元（基本上可以理解为像素）的属性（如位置坐标、纹理坐标等）进行线性插值。然后经这些经过插值处理之后的片元交给片元着色器处理。片元着色器确定最终的片元颜色。

<http://blog.csdn.net/racehorse/article/details/6593719>

<http://www.ituring.com.cn/article/851>（实践）

用glsl模仿opengl中的光照

**顶点处理**



当opengl使用固定功能的管线处理顶点时，它负责提供下面这些值，用于后面的光栅化处理：

1）视觉空间坐标

2）主颜色和辅助颜色

3）纹理坐标

4）雾坐标

5）点的大小

并不是顶点管线中的所有操作都可以由顶点着色器代替。在执行着色器之后，下面的操作仍然会出现，就像之前的顶点处理是由固定功能的管线完成那样：

1）透视除法

2）视口映射

3）图元装配

4）平截头体和用户裁剪

5）背面剔除

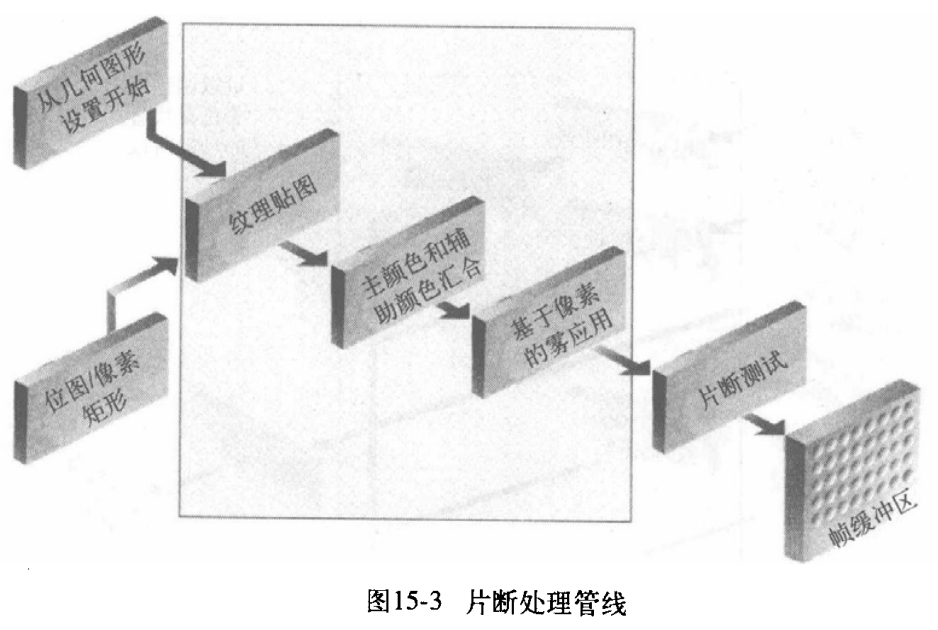
6）双面光照选择

7）多边形模式处理

8）多边形偏移

9）深度范围截取

**片段处理**



与顶点着色器类似，片断着色器可以接管部分片断处理功能，片断着色器可以处理的操作包括：

1）纹理应用

2）雾

3）主颜色和辅助颜色汇合

不论是否使用片断着色器，opengl总会执行下面这些操作：

1）单调或平滑着色（控制片断之间的差值）

2）像素覆盖计算

3）像素所有权测试

4）裁剪操作

5）点面模式应用（在opengl 3.0废弃，在opengl3.1删除）

6）alpha测试（知道opengl3.0都存在，在opengl3.1删除了这一测试，并用片断着色器中的可用功能替代了它）

7）深度测试

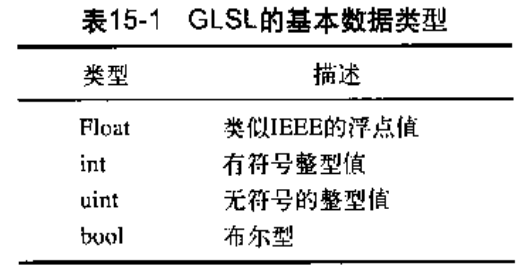
8）模板测试

9）alpha混合

10）对像素进行逻辑操作

11）颜色值的抖动

12）颜色掩码操作



**使用glsl创建着色器**

着色器程序就像C语言一样，是从main函数开始执行的，但是返回类型为void

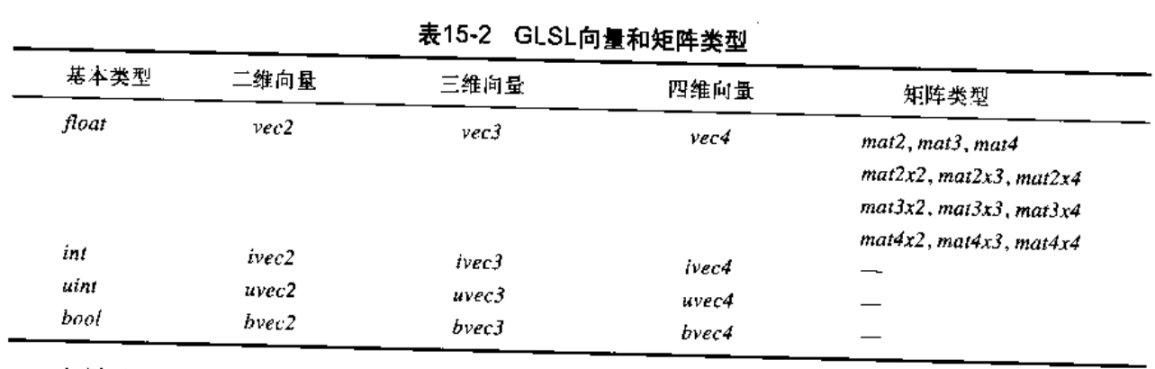
Glsl是一种强类型语言，也就是说，每个变量都必须进行声明，

变量的作用域：在所有函数定义之外声明的变量具有全局作用域，他们在着色器程序的所有函数中均可见

如前所述，glsl是一种强类型语言，甚至比c++更为严格，不同类型的值不存在隐式类型转换，例如：

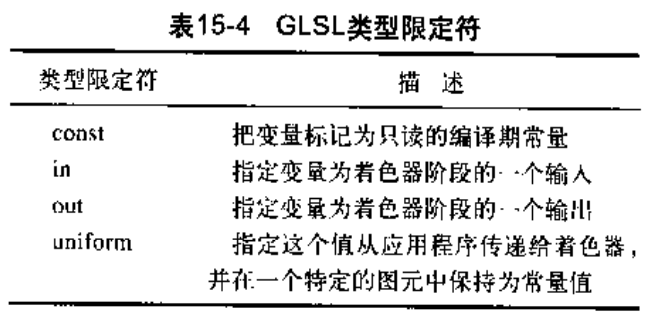
int f = 10.0

将会导致编译错误



**类型限定符**

类型也可以使用限定符，对他们的行为进行限定，glsl定义了4种限定符



在glsl1.30版以前，顶点着色器的输入变量用关键字attribute来限定，类似地，片断着色器的输入用关键字varying来限定，预料到潜在的增加更多着色阶段，这些关键字都由更加通用的in和out形式来代替，在glsl1.40中，attribute和varying都删除了。

Const类型限定符

所有的存储类型限定符都适用于全局作用域的变量，另外const还适用于局部变量和函数的形参。和c一样，const类型限定符表示变量是只读的。

In类型限定符

In限定符用来限定一个顶点着色器阶段的输入，这些输入可能是顶点属性（用于顶点着色器），或者插值变量（用于片断着色器），它们必须是全局范围声明的，不能在函数内部，一个in可以是浮点数类型的标量、向量、或者矩阵，不可以是数组或者结构体

Out类型限定符

Out限定符用来限定一个顶点着色器阶段的输出，例如，颜色或者纹理坐标，来自一个顶点着色器的齐次坐标变换，或者来自一个片断着色去的最终片段颜色，作为片断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、向量或者矩阵，不可以是数组或者结构体，centorid carying在没有多重采用的情况下与varying是一样的，在多重采样的情况下，centorid varying在光栅化的图形内部进行求值而不是在片断中心的固定位求值

Uniform类型限定符

Uniform限定了表示一个变量的值将由应用程序在着色器执行之前指定，并且在图元的处理过程中不会发生变化。Uniform变量是由顶点着色器和片断着色器共享的，他们必须声明为全局变量，任何类型的变量，包括结构和数组，都可以声明为uniform变量

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

1）顶点着色器输入属性：

gl\_Color gl\_SecondaryColor gl\_Normal gl\_Vertex gl\_MultiTexCoordn（顶点的第n个纹理的坐标） gl\_FogCoord

2）顶点着色器输出属性

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）

3）片断着色器输入属性

Discard

片断着色器中有一种特殊的控制流称为discard，使用discard会退出片断着色器，不执行后面的片断着色操作

Glsl中的函数递归是不被允许的

Glsl的变量命名与C语言类似，变量名称可以使用字母、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。

Glsl中只可以使用一维数组，数组的类型可以是一切基本类型或者结构体

下面是创建和使用

OpenGL shader

的简介：

用glCreateShaderObjectARB创建一个或多个(空的)shader对象

调用glShaderSourceARB给shader提供源代码。

为每个shader调用glCompileShaderARB来编译它。

调用glCreateProgramObjectARB创建一个program对象.

通过调用glAttachObjectARB把所有的shader对象和program对象关联起来。

调用glLinkProgramARB把program对象连接起来。

调用glUseProgramObjectARB把可执行的program设置成作为OpenGL当前状态

的一部分。

如果shader使用了uniform变量。如果想要对这些uniform变量进行处理，需

要通过调用glLinkProgramARB和glGetUniformLocationARB查询得到该变量，

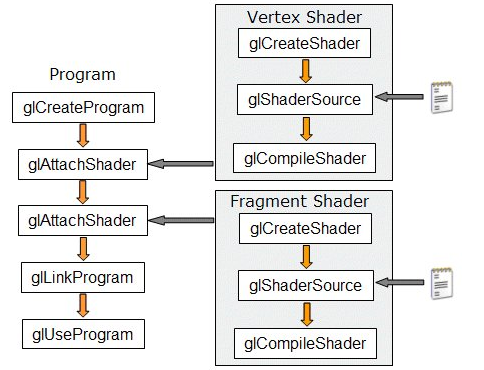
然后再通过glUniformARB来赋值。 如果vertex shader自定义了attribute变量，有两种方式指定该属性对应的索引值，首先可以通过在链接shader前调用glBindAttribLocationARB来设定，也可以由OpenGL自动指定，在连接后可以通过调用函数 glGetAttribLocationARB来查询得到使用的索引值。顶点的一般属性

（Generic vertex attributes）也有两种方式送入vertex shader，第一种

是通过调用glVertexAttribARB，第二种是一起调用

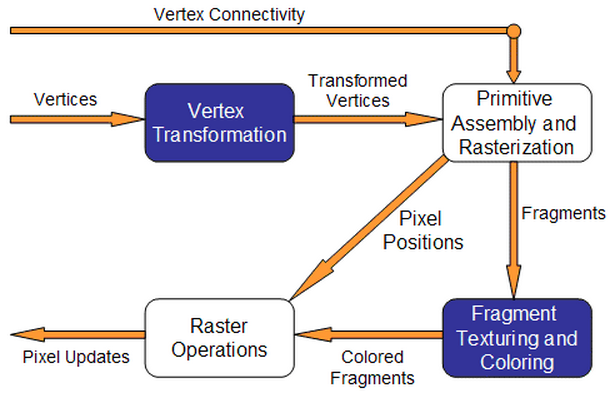
glVertexAttribPointerARB 和 glEnableVertexArrayPointer，通过这些指

令就可以绘制顶点数组了。



**学习教程**

**http://blog.csdn.net/racehorse/article/details/6593719**



上图是一个简化的图形处理流水线，功能包括：

1）顶点变换

这里一个顶点是一个信息几何，包括空间中的位置、顶点的颜色、法线、纹理坐标等。这一阶段的输入是独立的顶点信息，固定功能流水线在这一阶段通常进行如下工作：

顶点位置的变换

为每个顶点计算光照

纹理坐标的生成与变换

2）图元组合与光栅化

此阶段的输入时变换后的顶点和连接信息（比如三角形、多边形等），此阶段还负责视景体裁剪和背面剔除

光栅化决定了片断，以及图形的像素位置。这里的片断是指一块数据，用来更新帧缓存中特定位置的一个像素。一个片断除了包含颜色，还有法线和纹理坐标等属性，这些属性用来计算新的像素颜色值，本阶段的输出包括：

帧缓存中片断的位置

在顶点变换阶段计算出的信息对每个片断的插值

这个阶段利用在顶点变换阶段算出的数据，结合连接信息（三角形、多边形等）计算出片断数据，比如位置和颜色

3）片断纹理化和色彩化

此阶段的输入是经过插值的片断信息。在前一阶段已经通过插值计算了纹理坐标和一个颜色值。这个颜色在本阶段可以用来和纹理元素进行组合，此外，这一阶段还可以进行雾化处理，通常最后的输出是片断的颜色值以及深度信息

4）片断测试（即上面的raster Operations）

进行各种测试：

裁剪测试（scissor test）

Alpha测试

模板测试

深度测试

如果测试成功，则根据当前的混合模式（blend mode）用片断信息来更新像素值。注意混合只能在此阶段进行，因为片断纹理话和颜色花阶段不能访问帧缓存，帧缓存只能在此阶段访问。

个人的理解：

顶点变换 --- 图元装配与裁剪 --- 光栅化 --- 片断测试

顶点变换：Model View 逐顶点或逐像素（？？？）光照 纹理坐标计算与变换

图元装配与裁剪：Projection 裁剪 透视除法 背面剔除

光删化：位置与颜色插值

片断测试

对于上面的片断纹理化与色彩化是必须的还是可选的，且具体在哪个地方？？？

shader取代固定的功能

1）顶点shader实现顶点变换阶段的功能

2）片断shader取代片断纹理化和色彩化的功能

一个顶点shader可以编写代码实现如下功能：

1）使用模型视图矩阵以及投影矩阵进行顶点变换

2）发现变换及归一化

3）纹理坐标生成和变换

4）逐顶点或逐像素光照计算

5）颜色计算

不一定要完成上面的所有操作，例如你的程序可能不需要光照。但是，一旦你使用了顶点shader，顶点处理器的所有固定功能都将被替换，所以，你不能只编写发现变换的shader而指望固定功能帮你完成纹理坐标生成。

从上一节知道，顶点处理器并不知道连接信息，因此这里不能执行拓扑信息有关的操作。比如顶点处理器不能进行背面剔除，它只是操作点而不是面

顶点shader至少需要一个变量：gl\_Position，通常要用模型视图矩阵以及投影矩阵进行变换，顶点处理器可以访问opengl状态，所以可以用来处理材质和光照，最新的设备还可以访问纹理。

一个片断处理器可以进行如下操作：

1）逐像素计算颜色和纹理坐标

2）应用纹理

3）雾化计算

如果需要逐像素光照，可以用来计算法线

片断处理器的输入是顶点坐标、颜色、法线等极端插值得到的结果。在顶点shader中对每个顶点的属性值进行了计算，现在将对图元中的每个片断进行处理，因此需要插值的结果

如同顶点处理器一样，当你编写片断shader后，所有固定功能将被取代，所以不能使用片断shader对片断材质化，同时用固定功能进行雾化，程序猿必须编写程序实现需要的所有效果

片断处理器只对每个片断独立进行操作，并不知道相邻片断的内容，类似顶点shader，我们必须访问opengl状态，才可能知道应用程序中设置的雾颜色等内容。

一个片断shader有两种输出：

抛弃片断内容，什么也不输出

计算片断的最终颜色gl\_FragColor，当要渲染到多个目标时计算gl\_FlagData

还可以写入深度信息，但上一阶段已经算做了，所以没有必要

需要强调的是片断shader不能访问帧缓存，所以混合（blend）这样的操作只能发生在这之后。

传递参数

一个opengl程序可以用多种方式和shader进行通信，注意，这种通信是单向的，因为shader的输出只能是渲染到某些目标，比如颜色和深度缓存

Opengl的部分状态可以被shader访问，因此程序改变opengl某些状态就可以与shader进行通信了，例如，一个程序想将光的颜色传递给shader，可以直接调用opengl接口，就像使用固定功能流水线做的那样。Glsl允许用户自定义变量。

上面的讨论设计到glsl提供的两种类型修饰符（更多的将在后面提到）

1）一致变量（Uniform）

2）属性（Attribute）

在shader中定义的变量如果用这两种类型修饰符，表示对shader来说，它们是只读的。下面将详细讲述怎样使用这些类型的变量

还有一种将变量送给shader的方法：使用纹理。一个纹理不止可以表示一张图片，还可以表示一个数组。事实上，你完全可以决定如何在shader中解释纹理数据，即使它真是一幅图片。

下面是glsl中的基本数据类型：

Float

Bool

Int

浮点类型与c中leis，布尔类型可以为true或false。这些基本类型可以组成2、3或4维向量。如下所示：

Vec{2,3, 4} a vector of 2,3,or 4 floats

Bvec{2,3, 4} bool vector

ivec{2,3, 4} vector of integers

glsl中还包含2\*2,3\*3或4\*4型矩阵没因为这些矩阵类型在图形处理中很常用

mat2 mat3 mat4

此外，还有一组用来实现纹理访问的特殊类型，他们被称为采样器（sampler），在读取纹理值（也称为texel）时用到，下面就是纹理采样用到的数据类型：

Sampler1D - for 1D textures

Sampler2D - for 2D textures

Sampler3D - for 3D textures

samplerCube - for cube map textures

sampler1DShadow - for shadow maps

sampler2DShadow - for shadow maps

在glsl中，可以像C一样声明和访问数组，但是不能声明时初始化数组，glsl还可以定义结构体

声明一个基本类型变量方法与c类似，你可以在声明它的同时进行初始化

在glsl中还有一些实用例子（selector），可以简化我们的操作并让代码更简洁。访问一个向量可以使用如下的方法：

1. vec4 a = vec4(1.0,2.0,3.0,4.0);
2. **float** posX = a.x;
3. **float** posY = a[1];
4. vec2 posXY = a.xy;
5. **float** depth = a.w

在上面的代码中，可以使用x，y，z，w来访问向量成员，如果是颜色的话可以使用r，g，g，a，如果是纹理坐标的话可以使用s，t，p，q，注意表示纹理坐标通常是使用s，t，r，q，但r已经表示颜色中的红色（red）了，所以纹理坐标中需要使用p来代替

变量修饰符

Const – 声明一个编译器常量

Attribute – 随不同顶点变化的全局变量，由opengl应用程序传给顶点shader，这个修饰符只能用在顶点shader上，在shader中它是一个只读变量

Uniform – 随不同图元变化的全局变量（即不能在glBegin/glEnd中设置），由opengl应用程序传给shader，这个修饰符能在顶点和片断shader中，在shader中它是一个只读变量

Varying – 用于顶点shader和片断shader间传递的差值数据，在顶点shader中可写，在片断shader中只读

Uniform变量

不同于顶点属性在每个顶点有其自己的值，一个一致变量在一个图元的绘制中是不会改变的，所以其值不能在glBegin与glEnd中设置，一致变量适合描述在一个图元中，一帧中甚至一个场景中都不变的值，一致变量在顶点shader和片断shader中都是只读的。

首先你需要获得变量在内存中的位置，这个信息只有在连接程序之后才可获得，注意，对某些驱动程序，在获得存储位置前还必须使用程序（调用glUdeProgram）

1. GLint loc1,loc2,loc3,loc4;
2. **float** specIntensity = 0.98;
3. **float** sc[4] = {0.8,0.8,0.8,1.0};
4. **float** threshold[2] = {0.5,0.25};
5. **float** colors[12] = {0.4,0.4,0.8,1.0,
6. 0.2,0.2,0.4,1.0,
7. 0.1,0.1,0.1,1.0};
9. loc1 = glGetUniformLocation(p,"specIntensity");
10. glUniform1f(loc1,specIntensity);
12. loc2 = glGetUniformLocation(p,"specColor");
13. glUniform4fv(loc2,1,sc);
15. loc3 = glGetUniformLocation(p,"t");
16. glUniform1fv(loc3,2,threshold);
18. loc4 = glGetUniformLocation(p,"colors");
19. glUniform4fv(loc4,3,colors);

属性变量

上面提到的一致变量只能针对一个图元全体，就不说不能在glBegin与glEnd之间改变

如果要针对每个顶点设置变量，那就要用到属性变量了。事实上属性变量可以在任何时刻更新，在顶点shader中属性变量是只读的，因为它包含的是顶点数据，所以在片断shader中不能直接应用。

loc = glGetAttribLocation(p,"height");

1. glBegin(GL\_TRIANGLE\_STRIP);
2. glVertexAttrib1f(loc,2.0);
3. glVertex2f(-1,1);
4. glVertexAttrib1f(loc,2.0);
5. glVertex2f(1,1);
6. glVertexAttrib1f(loc,-2.0);
7. glVertex2f(-1,-1);
8. glVertexAttrib1f(loc,-2.0);
9. glVertex2f(1,-1);
10. glEnd();

顶点数组和属性变量也可以一起使用，首先需要能使用数组，如下：

1. **void** glEnableVertexAttribArray(GLint loc);
2. 参数：
3. loc – the location of the variable.

接下来使用函数提交包含数据的数组指针：

1. **float** vertices[8] = {-1,1, 1,1, -1,-1, 1,-1};
2. **float** heights[4] = {2,2,-2,-2};
3. ...
4. loc = glGetAttribLocation(p,"height");
6. glEnableClientState(GL\_VERTEX\_ARRAY);
7. glEnableVertexAttribArray(loc);
8. glVertexPointer(2,GL\_FLOAT,0,vertices);
9. glVertexAttribPointer(loc,1,GL\_FLOAT,0,0,heights);

易变变量（varying variables）

如前面所说，shader包括两种类型：顶点shader与片断shader。为了计算片断的值，往往需要访问顶点的差值数据。例如，当使用逐片断光照时，我们需要知道当前片断的法线，但是在opengl中只为每个顶点指定了法线，顶点shader可以访问这些法线，而片断shader不能，因为法线是在opengl程序作为属性变量指定的。

顶点变换后的数据移动到流水线的下一个阶段，在这个阶段通过使用连接信息，生成了所有图元并完成片断化。对每个片断，有一组变量会被自动黄金星插值并提供给片断shader，这些都是固定功能，片断的颜色就是这么处理的，到达片断shader的颜色就是组成图元的顶点颜色插值的结果。

像片断shader接收到的这种插值产生的变量，就是“易变变量”类型。Glsl包含一些预先定义的易变变量，例如，前面提到的颜色，用户也可以自己定义易变变量，他们必须同时在顶点shader与片断shader中声明

1. varying **float** intensity;

一个易变变量必须在顶点shader中声明，然后计算每个顶点的变量值。在片断shader中，接受这个变量通过插值得到的结果，注意此时这个变量是只读的。

控制流语句

与c 语言类似，glsl中有类似if-else的条件语句，for，while，do-while等

Glsl中也有跳转语句：

Continue break 都和c语言类似

Discard：只能在片断shader中使用，它将在不写入帧缓存或者深度缓存的情况下，终止当前片断的shader程序

Glsl中可自定义函数，这些函数和c函数一样，一般都会有返回值，返回值类型没有限制，但不能是数组

允许函数重载，但标准中没有定义递归行为

**顶点shader**

我们都知道glsl可以获取某些opengl状态信息，在renderMonkey中得到（已验证）



其中的gl\_ProjectionMatrix与gl\_ModelViewMatrix都是在glsl直接从opengl中访问得到的，为一致变量，gl\_Vertex也是glsl提供的，为属性变量。注意所有其他功能都将散失，比如没有光照计算。

上面的代码中变换每个顶点时，投影矩阵都将乘上模型视图矩阵，这显然非常浪费时间，因为这些矩阵不是每个顶点变化的，注意这些矩阵是一致变量

Glsl提供一些派生的矩阵，也就是说gl\_ModelViewProjectionMatrix是上面两个矩阵的乘积，所以顶点shader也可以写成下面这样（已验证）：



上面的操作能够获得和固定功能流水线相同的结果吗？理论上如此，但实际上对顶点变换操作的顺序可能会不同，顶点变换通常在显卡中是高度优化的任务，所有有一个利用了这种优化的特定函数用来处理这个任务，这个神奇的函数就是ftransform（已验证）



使用这个函数的另一个原因是float数据类型的精度限制。由于数据精度的限制，当使用不同的顺序计算时，可能得到不同的结果，因此glsl提供这个函数保证获得最佳性能的同事，还能得到与固定功能流水线相同的结果。

**片断shader**

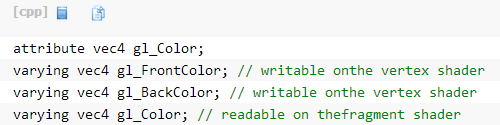
片断shader也有预定义的变量gl\_FragColor，可以向体重写入片断的颜色值，比如将所有的片断颜色变为绿色



**颜色例子**

Glsl可以读取一些opengl状态，在这里我们将学习如何访问在opengl中设置的glColor变量

Glsl有一个属性变量记录当前颜色，也提供易变变量从顶点shader向片断shader传递颜色值



变量使用细想如下：

1）opengl程序通过glColor传送颜色信息

2）顶点shader通过属性gl\_Color接受颜色值

3）顶点shader计算正面和反面的颜色，然后分别保存在gl\_FrontColor和gl\_BackColor中

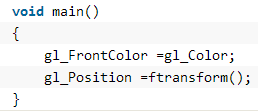
4）片断shader接受易变变量gl\_Color中存储的插值产生颜色，由当前图元的方向决定颜色是gl\_FrontColor还是gl\_BackColor插值产生的

5）片断shader根据易变变量gl\_Color设置gl\_FragColor

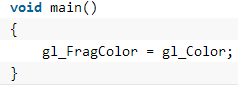
前面说过顶点shader和片断shader中传递的易变变量要有相同的名字，但这里是个例外，顶点shader中的gl\_FrontColor和gl\_BackColor会根据图元的方向，自动转换为片断shader中的gl\_Colot。还要注意属性变量gl\_Color和易变变量gl\_Color没有冲突，因此前者只存在于顶点shader，后者只存在于片断shader

注意：上面的gl\_Color、gl\_FrontColor、gl\_BackColor都是glsl中的默认值

顶点shader：

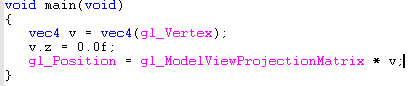


片断shder：



**形状例子**

我们在顶点shader中要改变gl\_Vertex时候，需要先用一个临时变量保存然后改变，因为gl\_Vertex是一个glsl提供的属性变量，所以在顶点shader中它是只读的



**卡通着色**

需要访问顶点法线，顶点shader通过属性变量gl\_Normal来访问opengl中指定的法线，这些法线在opengl中通过glNornal函数定义，因此位于模型空间

如果在opengl中没有对模型进行旋转或缩放等操作，那么传给顶点shader的位于世界空间的gl\_Normal正好等于模型空间中定义的法线。另外法线只包含方向，所以不受移动变换的影响。

1）逐顶点计算片断强度

顶点shader：

vec3 lightDirection = vec3(0.0f, 0.0f, -1.0f);

varying float cos;

void main(void)

{

cos = dot(gl\_Normal, lightDirection);

gl\_Position = ftransform();

}

片断shader

varying float cos;

void main(void)

{

vec4 color;

if(cos >= 0.95f)

{

color = vec4(1.0,0.5,0.5,1.0);

}

else if(cos > 0.5f)

{

color = vec4(0.6,0.3,0.3,1.0);

}

else if(cos > 0.25f)

{

color = vec4(0.4,0.2,0.2,1.0);

}

else

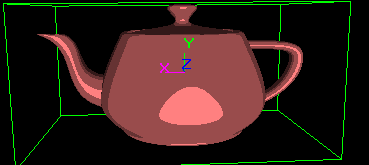
{

color = vec4(0.2,0.1,0.1,1.0);

}

gl\_FragColor = color;

}



上面的着色效果不太好，主要原因是因为我们对cos进行插值，插值的结果与用片断法线算出的cos有区别，接下来我们采用逐片断着色

2）逐片断计算片断强度

为了达到这个目的，我们需要访问每个片断的法线，顶点shader中需要将顶点的法线写入一个易变变量，这样在片断shader中就可以得到经过插值的法线。

顶点shader：

varying vec3 normal;

void main(void)

{

normal = gl\_Normal;

gl\_Position = ftransform();

}

片断shader

vec3 lightDirection = vec3(0.0f, 0.0f, -1.0f);

varying vec3 normal;

void main(void)

{

vec4 color;

float cos = dot(normal, lightDirection);

if(cos >= 0.95f)

{

color = vec4(1.0,0.5,0.5,1.0);

}

else if(cos > 0.5f)

{

color = vec4(0.6,0.3,0.3,1.0);

}

else if(cos > 0.25f)

{

color = vec4(0.4,0.2,0.2,1.0);

}

else

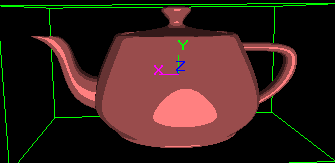
{

color = vec4(0.2,0.1,0.1,1.0);

}

gl\_FragColor = color;

}



令人吃惊的是新的渲染结果居然和之前的版本1一模一样，这是为什么呢？

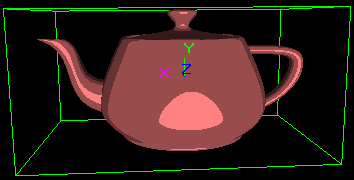
仔细看看这两个版本的区别，在版本1中，我们在定点shader中计算出一个cos值，然后在片断shader中使用这个值的插值结果。在版本2中，我们先对法线插值，然后再片断shader中计算点积。插值和点积都是线性运算，所以两者运算的顺序并不影响结果。

真正的问题在于片断shader中对插值后的法线进行点积运算的时候，尽管这时法线的方向是对的，但是它并没有归一化。

我们说法线方向是对的，因为我们假定传入顶点shader的法线是经过归一化的，对法线插值可以得到一个方向正确的向量，但是，这个向量的长度在发部分情况下都是错的，**因为归一化法线进行插值的时候，只有在所有法线的方向一致时才会得到一个单位长度的向量。**

综上所述我们修改为：

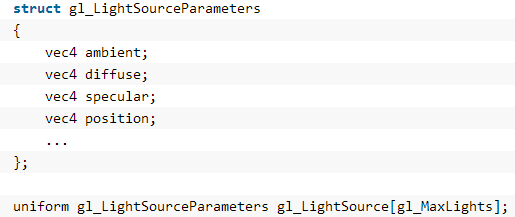
float cos = dot(**normalize**(normal), lightDirection);



肉眼看不出效果，实际上两者比较的时候确实会发现有些像素的 颜色发生了变化（已验证）

结束卡通着色的内容之前，还有一件事情需要解决：使用opengl中的光来代替变量lightDirection。我们需要在opengl中定义一个光源，然后在我们的shader中使用这个光源的方向数据。注意：不需要用glEnable打开这个光源，因为我们使用了shader

我们假设opengl中定义了1号光源（GL\_LIGHT0）是方向光，glsl已经声明了一个C语言形式的结构体描述光源属性，这些结构体组成一个数组，保存所有光源信息



Opengl标准中规定，当一个光源的位置确定后，将自动转换到视点空间（eye space）的坐标系中，例如摄像机坐标系。如果模型视图矩阵的左上（3\*3）是正交的（如果使用gluLootAt并且不适用缩放变换就可以满足这一点），便能保证光线方向向量在地洞变换到视点空间之后保持归一化。

我们必须将发现变换到视点空间，然后计算器与光线的点积，只有在相同空间，计算两个向量的点积得到的余弦值才有意义。

为了将法线转换到视点空间，我们必须使用预先定义的mat3型的一致变量gl\_NormalMatrix，这个矩阵是模型视图矩阵的左上3\*3子阵的逆矩阵的转置矩阵，需要对每个法线进行这个变换，现在顶点shader变为如下形式：

varying vec3 normal;

void main(void)

{

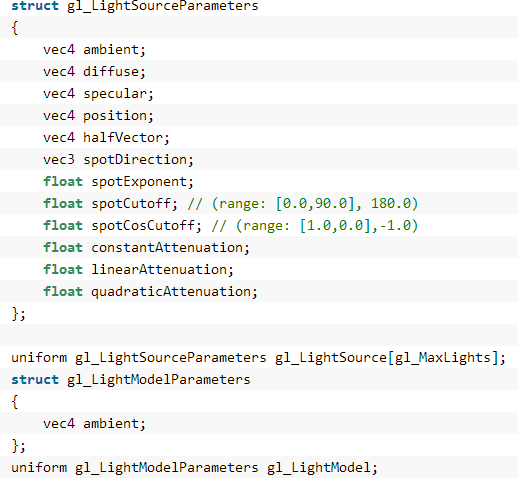
normal = gl\_NormalMatrix \* gl\_Normal;

gl\_Position = ftransform();

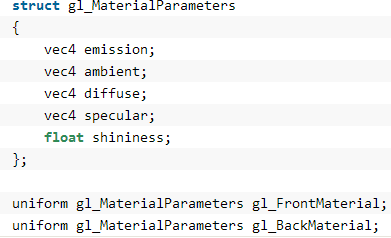
}

**逐顶点光照**

在之前我们接触过了的glsl中如何访问opengl状态中关于光源的部分，这些数据描述了每个光源的参数



在glsl中也同样可以访问材质参数



我们需要计算漫反射的cos值，要计算这个值，首先要确保光线方向向量(gl\_LightSource[0].position)与发现向量都是归一化的，然后就可以使用点积得到余弦值。注意：对方向光，opengl中保存的方向是从顶点指向光源

Opengl将光源的方向保存在视点空间坐标系中，因此我们需要将法线也变换到视点空间，完成这个变换可以用预先定义的一致变量gl\_NormalMatrix，这个矩阵是模型视图变换矩阵的左上3\*3子矩阵的逆矩阵的转置

顶点shader：

void main(void)

{

// transform the normal into eye space and normalize the result

vec3 normal = normalize(gl\_NormalMatrix \* gl\_Normal);

// normalize the light's direction, since we'are talking about direction

// light, the position field is actually direction

vec3 lightDirection = normalize(vec3(gl\_LightSource[0].position));

// calculate the cos of angle between the normal the light direction

float angleCos = max(dot(normal, lightDirection), 0.0f);

vec4 diffuse = gl\_FrontMaterial.diffuse \* gl\_LightSource[0].diffuse;

gl\_FrontColor = angleCos \* diffuse;

gl\_Position = ftransform();

}

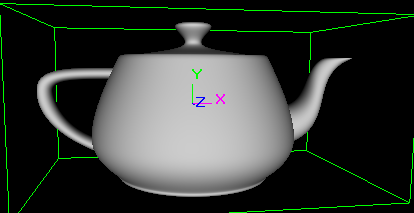
片断shader

void main(void)

{

gl\_FragColor = gl\_Color;

}



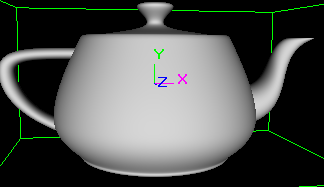
添加全局光照

// calculate ambient and global ambient

vec4 ambient = gl\_FrontMaterial.ambient \* gl\_LightSource[0].ambient;

vec4 globalAmbient = gl\_FrontMaterial.ambient \* gl\_LightModel.ambient;

gl\_FrontColor = angleCos \* diffuse + ambient + globalAmbient;

（其实通过对比会发现变亮了一些）

// calculate specular with blin-phong;

float dotHV = max(dot(normal, gl\_LightSource[0].halfVector.xyz), 0.0f);

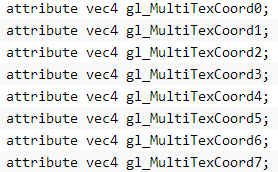
vec4 specular = gl\_FrontMaterial.specular \* gl\_LightSource[0].specular \* pow(dotHV, gl\_FrontMaterial.shininess);

gl\_FrontColor = angleCos \* diffuse + ambient + globalAmbient + specular;

glsl雾的实现

**纹理贴图**

为了在glsl中应用纹理，我们需要访问每个顶点的纹理坐标。Glsl中提供了一些属性变量，每个纹理单元一个：



Glsl还未访问每个纹理的纹理矩阵提供了一个一致变量数组

