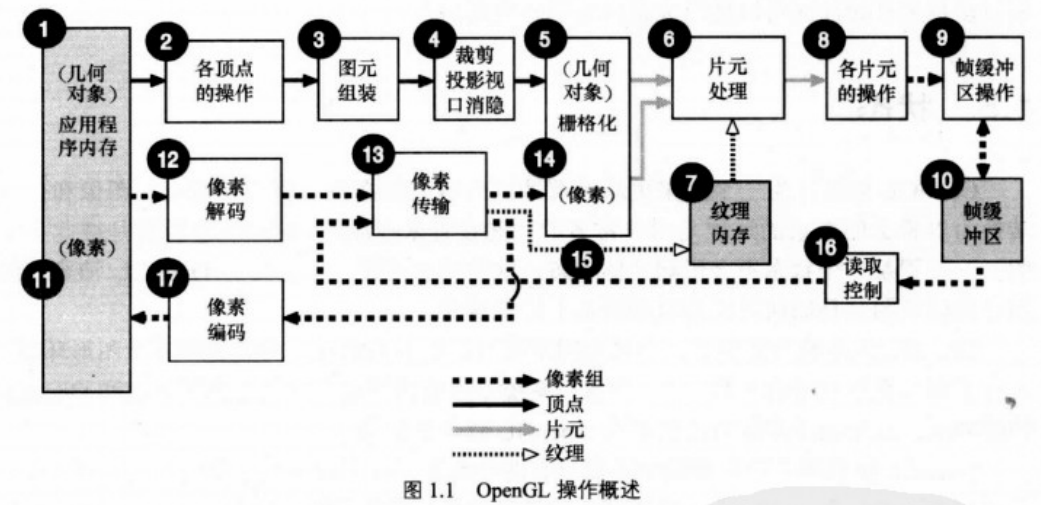
一次一个顶点与顶点数组都是立即模式，这是因为图元是在指定后立即渲染的。

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

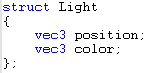
Uniform sample2D grass;

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。