

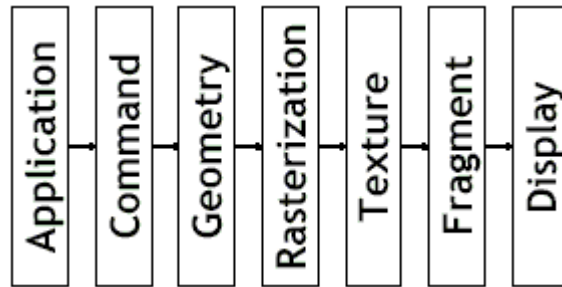
OpenGL 低级着色语言与高级着色语言

节选自偶的毕业论文，主要是对可编程图形处理器， ARB_VERTEX_PROGRAM 和 glslang 的简单介绍。抛砖引玉，希望大家不吝赐教：)

Octane3d@hotmail.com

第二章 可编程图形处理器

现代图形流水线如图-2.1 所示。



(图-2.1, 图形流水线)

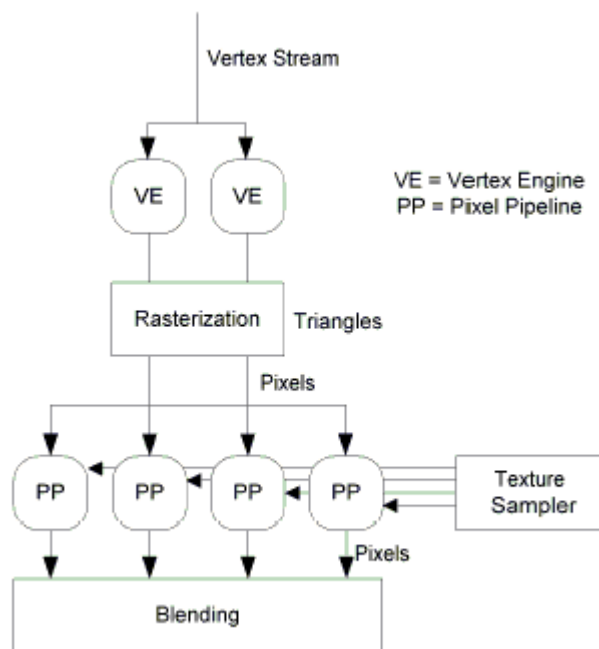
应用程序级 (Application) 实现物理模拟, 处理用户输入, 修改数据结构, 数据库访问, 几何基元生成等功能。命令级 (Command) 提供命令缓冲区, 解释执行命令和管理图形系统的状态。几何级 (Geometry) 对多项式表达的弯曲曲面求值, 进行几何变换, 光照计算, 纹理坐标生成, 实现剪切, 拣选与几何基元组装等功能。光栅化级 (Rasterization) 进行三角形设置, 对三角形进行抽样以生成片断, 并且对颜色和纹理坐标进行插值。纹理级 (Texture) 对纹理坐标进行变换, 进行纹理访问与过滤。片断级 (Fragment) 混合片断颜色与纹理, 进行雾化操作, 执行深度、透明以及模版测试, 最终生成要写入帧缓冲区的像素。显示级 (Display) 对象素颜色进行伽玛校正后写入帧缓冲区。命令与几何级属于对象空间的操作, 其上的操作是每顶点的, 主要是进行几何变换和光照计算, 它的特点是大量 (≥ 1000 万个顶点) 复杂的浮点操作。纹理与片断级输入图像空间的操作, 其上的操作是每片断的, 主要是进行纹理混合, 它的特点是进行海量 (≥ 10 亿个片断) 但是较简单的定点运算。

图形硬件的发展是一个逐渐对图-2.1 中的流水级增加硬件加速功能的过程。最早的图形硬件只是一个简单的帧缓冲区, 没有任何硬件加速功能。之后逐渐添加硬件光栅化, 硬件纹理映射, 硬件几何变换与光照计算, 硬件反走样等功能。按照图形硬件所支持的硬件加速功能的多少与性质, 可以把图形硬件分为几代。1984 年, SGI 公司发布了 IRIS 1400 图形系统, 其中集成的图形硬件可以实时光栅化平坦着色 (flat-shaded) 的多边形, 这是第一代图形加速硬件。第二代图形加速硬件以 HP 的 SPX 和 SGI 的 GT 为代表。它们提供 Phong 光照计算, Gouraud 着色和 Z-Buffer 功能, 光栅化性能也有了大大提高。1992 年, SGI 推出了 RealityEngine [Akeley93], 之后又于 1996 年推出了 InfiniteReality [Montrym97], 标志着图形加速硬件进入第三代。RealityEngine 在第二代图形加速硬件的基础上, 增加了纹理映射与与全场景反走样的功能, 大大提高了实时渲染的真实感。上述的图形加速硬件价格昂贵, 只有高档的图形工作站才能够配置, 但是 1996 年 3Dfx 公司推出的面向 PC 平台的 Voodoo 系列 3D 加速卡改变了这一局面。Voodoo 卡介于第二与第三代图形加速硬件之间, 它提供 Gouraud 着色, 硬件光栅化, 硬件纹理映射, Z 缓冲区等功能, 将 PC 带入了实时 3D 的时代。随后, PC 图形硬件蓬勃发展, 更高的三角形生成速率, 更高的填充速率, 多纹理功能, 硬件几何变换与光照计算, 碰撞与环境映射以及全屏反走样将其带入了第三代图形加速硬件的时代。此后, 随着 PC 游戏以及 PC 工作站的发展, 可以说, PC 图形硬件的发展代表了图形硬件技术发展的主流方向。

无论是传统工作站图形硬件, 还是 PC 图形硬件, 从第一代到第三代, 它们的一个共同特征就是都只实现了固定功能的渲染流水线, 而不具备可编程能力。与这些图形硬件相匹配, 主要的 3D API 例如 OpenGL 和 Direct3D 作为一个状态机实现, 用户通过 3D API 提供的函数设置好相应的状态, 例如变换矩阵、材质参数、光源参数、纹理混合模式等, 然后传入顶

点流。图形硬件则利用内置的固定渲染流水线和渲染算法对这些顶点进行几何变换、光照计算、光栅化、纹理混合、雾化操作、最终将处理结果写入帧缓冲区。这种渲染体系限制用户只能使用图形硬件中固化的各种渲染算法。这虽然可以很好的满足对渲染质量要求不高的应用，但难以满足那些需要更高的灵活性和更真实的渲染质量的实时图形应用。用户已经不再满足于基于顶点的近似 Phong 模型光照计算（这是 OpenGL 和 Direct3D 采用的光照计算模型，这两种 API 上的光照计算或者在支持硬件光照计算的的图形硬件之上进行，或者在 CPU 之上进行。）和简单的多纹理混合。用户需要硬件加速的角色动画支持，需要使用定制的光照模型（基于片断的光照模型、各向异性光照模型、基于 BRDF 的光照模型），需要非真实渲染（卡通渲染、素描渲染），需要每片断 Fresnel 效果，需要各种体积效果，需要过程纹理，以及各种以往只有在 Renderman [Upstill89] 之上才能看到的 3D 效果。对于这些需求，传统的图形硬件是无能为力的。满足这些需求的答案就是硬件可编程性。

2001 年 3 月，nVidia 公司推出了具有可编程能力的 GeForce 3 [Lindholm01]，从而将图形加速硬件带入了可编程的时代，即可编程图形处理器，我们不妨将它们视为第四代图形处理器。现代图形处理器的可编程引擎如图-2.2 所示。

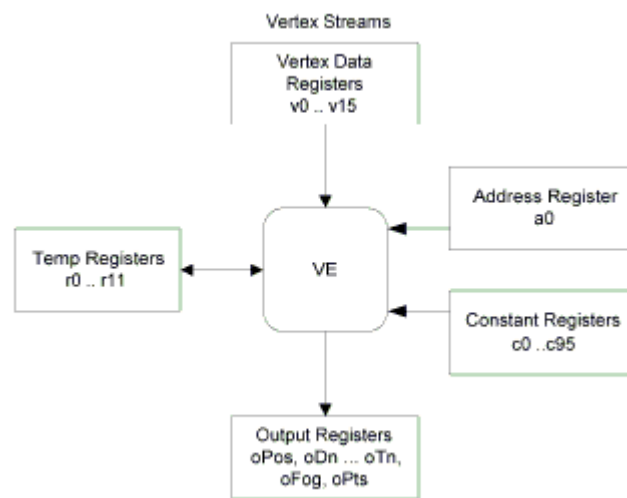


（图-2.2，现代图形处理器的可编程引擎）

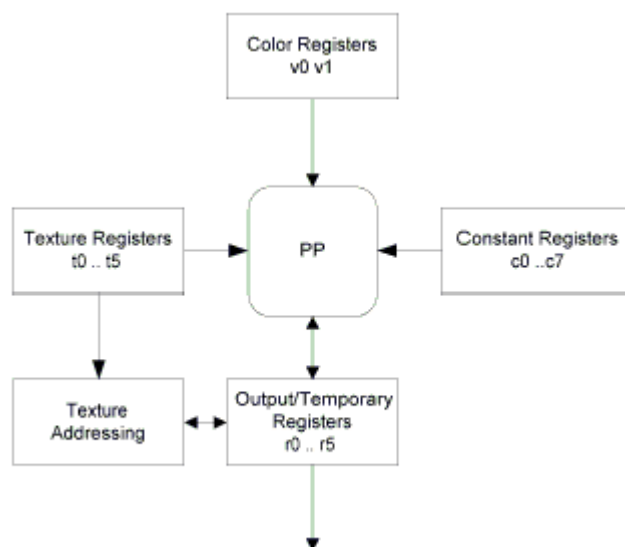
图-2.2 中的 VE（顶点引擎或者顶点处理器）和 PP（片断引擎或者片断处理器）就是可编程图形处理器中的可编程部件。运行于顶点处理器之上的程序称为顶点着色程序（Vertex Shader），它们的工作是进行几何变换和光照计算等操作。运行于片断处理器之上的程序称为片断着色程序（Fragment Shader），它们的工作是进行纹理混合等操作。顶点着色程序的输入是顶点流，输出是处理后的顶点流。后者经固定的光栅化模块进行三角形组装和光栅化处理用于生成片断流送入片断着色程序。片断着色程序访问纹理，进行纹理混合并最终计算出片断的颜色与深度信息送入后续流水级以进行透明、深度、模板与雾化等操作。一个图形处理器中一般包含多个顶点处理器和多个片断处理器。如图-2.2 所示的图形处理器包含 2 个顶点处理器，4 个片断处理器。同类型的处理器上运行的总是同一个着色程序的一个拷贝，这些着色程序并行运行，处理输入流中的不同数据元素。输入流中的数据元素类型相同，并且相互之间独立。Shader 在一个处理器上的每一次运行只能处理输入流中的一个元素。因此，顶点处理器或者片断处理器可以看作是使用同一指令对不同数据元素进行处理的单指令多

数据处理器 (SIMD Processor)。

图-2.3 是顶点处理器的体系结构。图-2.4 是片断处理器的体系结构 [Michael02]。这两种处理器都没



(图-2.3, 顶点处理器体系结构)



(图-2.4, 片断处理器体系结构)

有内存的概念，所有的运算都在寄存器之上进行。它们的每一个寄存器都是四分量浮点寄存器，指令集中的指令可以同时四个分量进行运算，因此可以把顶点处理器看作是一个基于寄存器的向量处理器。图-2.3 中的顶点数据寄存器用于只读访问当前被处理的顶点的属性，例如顶点位置、法向量、颜色和纹理坐标。常量寄存器用于为顶点程序提供只读常量参数或者程序参数。只写的地址寄存器用于间接访问常量寄存器。临时寄存器用于保存中间运算结果。只写的输出寄存器则用于输出处理后的顶点属性。图-2.4 中的颜色寄存器用于只读访问片断的颜色。只读的纹理坐标寄存器用于访问纹理坐标和对纹理进行抽样。常量寄存器用于为片断程序提供只读常量参数或者程序参数。临时寄存器用于保存中间运算结构，临时寄存器 r0 也是输出寄存器。图-2.3 和图-2.4 是 DirectX 所定义的顶点/片断处理器体系结构，但是它们也恰当的抽象了例如 OpenGL 等其它类型的顶点/片断处理器。

更详细的寄存器与指令集说明将放在下一章进行介绍。这里重点讨论一下顶点/片断处

理器区别于微处理器的一些特殊之处。微处理器的特点是标量运算、无内在向量并行性、算术运算单元少、有完善控制流指令、延迟低、带宽小。而图形应用的特点确是大运算量、大规模并行性、允许较长的延迟与深度前向流水。这就决定了图形处理器和微处理器在体系结构上存在着很大的差别。首先图形处理器要尽可能的实现高度并行性。这种并行性分为两种：数据并行性与流水线并行性。为了充分利用数据并行性，图形处理器在两个层次上进行并行处理。第一层利用输入数据流中数据元素之间的无关性，多个顶点（或片断）处理器运行一个顶点（或者片断）着色程序的多个拷贝，同时作用于输入数据流中的多个数据元素之上。从这个意义上说，图形处理器是一个流处理器。第二层是利用图形运算包含大量向量运算的特点，实现指令级并行性。具体地说就是把多个同类型的标量运算合并到一个向量运算之中。因此，图形处理器又是一个向量处理器。其次流水并行性允许顶点着色程序与片断着色程序同时运行，前者的输出恰为后者的输入，构成一种生产者消费者的关系。再次为了保证大规模并行运算，图形处理器基本上不支持转移指令或者只提供有限的支持，着色程序的静态长度和动态可执行长度也非常短。最后图形处理器有着比微处理器多的多的算术逻辑运算单元。图形处理器的所有指令都具有相同的延迟，不存在各类流水竞争。

现在图形处理器向着通用流处理器的方向发展，虽然它的重点应用仍然是图形应用，通用性也远不是微处理器概念上的通用性。可以预测，未来的图形处理器将支持功能更丰富的算术指令，提供更高级别的指令正交性和更强的转移指令功能。而它的应用范围除了实时图形应用外，也会扩展到全局光照计算、多体问题、分子动力学、弹性形变、流体模拟等科学计算领域 [Boltz03]。

第三章 低级着色语言

3.1 概述

要利用现代图形处理器的可编程性,图形程序员需要编写在图形处理器上运行的程序,这种程序称为着色程序(shader)。着色程序可以分为两类,一类是顶点着色程序(vertex shader),另一类是片断着色程序(fragment shader)。顶点着色程序运行于顶点处理器之上,每次激活处理一个顶点;片断着色程序运行于片断处理器之上,每次激活处理一个片断。目前绝大多数着色程序都由低级着色语言写成。图形处理器通过通用的3D API(DirectX 或者 OpenGL)向图形程序员提供可编程能力。

作为桌面PC上最主要的3D硬件接口,DirectX从8.0开始就提供了对低级着色语言的支持,并在9.0中进一步完善和发展。DirectX运行库分别定义了执行vertex shader和fragment shader的两个虚拟机,图形程序员针对这两个虚拟机编写shader,GPU厂商则在显卡驱动中实现这两个虚拟机。随着图形处理器硬件能力和DirectX的飞速发展,shader虚拟机的规范也在不断发展,目前DirectX 9.0支持多个版本的shader虚拟机。Vertex shader虚拟机有四个版本,分别是:vs_1_1、vs_2_0、vs_2_x和vs_3_0。Fragment shader(DirectX上称为pixel shader)虚拟机有六个版本,分别是:ps_1_1、ps_1_2、ps_1_4、ps_2_0、ps_2_x和ps_3_0。上述不同版本基本上可以分为三代。第一代是1.x版本,这一代的shader虚拟机对应于早期的图形处理器。这些图形处理器提供的可编程资源(寄存器、指令集)都有限,而且它们不提供转移指令。第二代是2.x版本,这一代虚拟机对应于目前的主流中档图形处理器。它们提供更多的寄存器和更大的指令集,而且提供基于常数的静态转移指令和有限的动态转移指令。第三代是3.x版本,这一代虚拟机对应于目前的主流高档和未来的图形处理器。它们提供更多的寄存器,包括静态与动态转移指令的更强大的指令集以及一个更一般的编程模型。

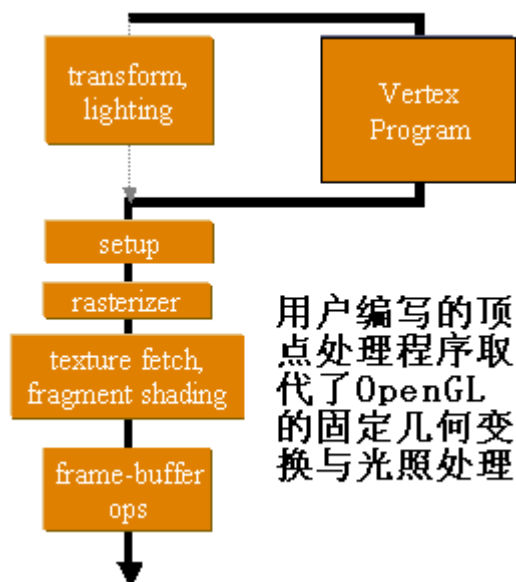
桌面PC的另一个主要3D硬件接口OpenGL接口也在不断的加强对低级着色语言的支持。这种支持最初是硬件厂商通过添加私有的扩展(openGL extension)[sgext]来实现的。例如nVidia公司通过添加NV_VERTEX_PROGRAM与NV_FRAGMENT_PROGRAM扩展来提供对vertex shader和fragment shader的支持;ATI公司通过添加EXT_VERTEX_SHADER与ATI_FRAGMENT_SHADER来提供对vertex Shader和fragmentsShader的支持。这种做法的一个巨大缺点是不同厂商提供的编程接口不同,图形程序员要想保证自己的应用能够在不同的图形处理器上都能顺利运行,必须针对每一个厂商的扩展编写不同的渲染代码。为了解决这个问题,OpenGL体系管理委员会于2002年6月和2002年9月分别通过了两个官方扩展:ARB_VERTEX_PROGRAM与ARB_FRAGMENT_PROGRAM来统一对低级着色语言的支持。这两个扩展同样定义了运行vertex shader与fragment shader的两个shader虚拟机,图形程序员和显卡厂商分别针对这两个虚拟机编写应用程序和驱动程序。目前这两个扩展的版本是1.0(arbvp10和arbf10),相当于DirectX中的vs_1_x和ps_1_x。这两个扩展的2.0的版本(arbvp20和arbf20,相当于DirectX中的vs_2_x和ps_2_x)现在也在制定之中。

本文的研究对象是OpenGL 2.0中提出的高级着色语言glslang及其优化编译技术。为此,本章的后续几节将重点介绍做为glslang编译器目标机器的arbvp10虚拟机[arbvp10]与arbf10虚拟机[arbf10]和arbvp20虚拟机与arbf20虚拟机。但由于目前arbvp20/arbf20的规范还未公布,我们只能在现有虚拟机的基础上设想这两种虚拟机的硬件能力。

3.2 arbv10 着色语言

3.2.1 概述

Vertex shader 在 OpenGL 渲染流水线中的位置和作用如图-3.1 所示。

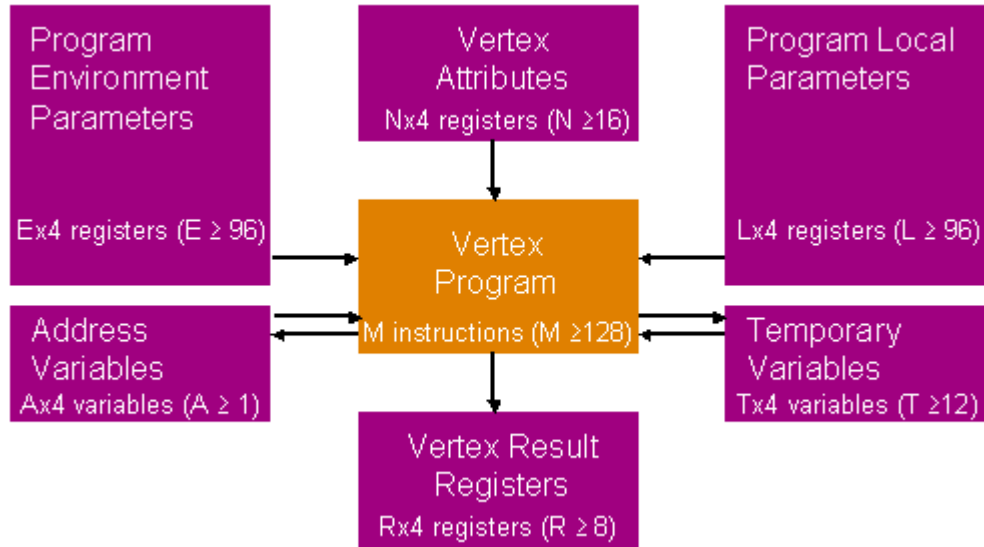


(图-3.1, OpenGL 中的顶点着色程序)

用户编写的 LSL vertex shader (在 OpenGL 中称为 vertex program) 可以在用户的控制下取代 OpenGL 的固定几何变换与光照处理模块。当用户允许 vertex program 模式时, vertex program 在每指定一个顶点坐标或者顶点属性 0 时执行。Vertex program 每次处理一个顶点, 对不同顶点的处理互不影响。它的输入是顶点的属性, 包括坐标、法向量、颜色、纹理坐标等; 输出是处理后的顶点属性, 包括齐次剪切空间顶点坐标、颜色、纹理坐标等。Vertex program 一般来说需要完成顶点世界-观察坐标变换与投影坐标变换、基于顶点的光照计算和纹理坐标生成等任务。

3.2.2 arbv10 虚拟机

arbv10 虚拟机的框图如图-3.2 所示。



(图-3.2, arbvp10 虚拟机)

粗略的说, arbvp10 虚拟机可以看作是一个基于寄存器的向量处理器。arbvp10 虚拟机中没有内存的概念, 所有的数据都保存在寄存器中。除地址寄存器外, 所有的寄存器都是四分量浮点寄存器, 也就是说, 每一个寄存器可以保存四个浮点数。arbvp10 虚拟机包含五种寄存器: 顶点属性寄存器、顶点结果寄存器、临时寄存器、地址寄存器与程序参数寄存器。

arbvp10 虚拟机支持至少 16 个顶点属性寄存器。这些寄存器可以通过表-3.1 中的寄存器的器名直接访问。

顶点属性传统寄存器名	分量	顶点属性一般寄存器名	说明
vertex.position	(x, y, z, w)	vertex.attrib [0]	对象坐标
vertex.weight	(w, w, w, w)	vertex.attrib [1]	顶点权重 0 - 3
vertex.wight [n]	(w, w, w, w)	vertex.attrib [1]	顶点权重 n - n + 3
vertex.normal	(x, y, z, 1)	vertex.attrib [2]	顶点法向量
vertex.color	(r, g, b, a)	vertex.attrib [3]	主颜色
vertex.color.primary	(r, g, b, a)	vertex.attrib [3]	主颜色
vertex.color.secondary	(r, g, b, a)	vertex.attrib [4]	次颜色
vertex.fogcoord	(f, 0, 0, 1)	vertex.attrib [5]	雾化坐标
vertex.texcoord	(s, t, r, q)	vertex.attrib [8]	纹理坐标, 纹理单元 0
vertex.texcoord [n]	(s, t, r, q)	vertex.attrib [8 + n]	纹理坐标, 纹理单元 n
vertex.matrixindex	(i, i, i, i)	N/A	顶点矩阵索引 0 - 3
vertex.matrixindex [n]	(i, i, i, i)	N/A	顶点矩阵索引 n - n + 3

(表-3.1, 顶点属性寄存器名)

表-3.1 中的顶点属性寄存器有两种命名方式。一种是第一列中的顶点属性传统寄存器名, 另一种是顶点属性一般寄存器名。前者对应于通过 glVertex, glNormal 等函数调用设置的顶点属性; 后者对应于通过 glVertexAttrib 函数调用设置的顶点属性。两种设置顶点属性的方式都是可行的, 但是不能混合使用。顶点属性寄存器也可以通过申明绑定到某一个顶点属性寄存器名的属性变量访问, 例如:

ATTRIB pos = vertex.position; # pos 等价于 vertex.position。

顶点属性寄存器是 vertex program 的输入, 是只读寄存器。

arbvp10 虚拟机支持至少 8 个顶点结果寄存器。这些寄存器可以通过表-3.2 中的寄存器名直接访问。

顶点结果寄存器名	分量	描述
result.position	(x, y, z, w)	剪切空间坐标
result.color	(r, g, b, a)	正面主颜色
result.color.primary	(r, g, b, a)	正面主颜色
result.color.secondary	(r, g, b, a)	正面次颜色
result.color.front	(r, g, b, a)	正面主颜色
result.color.front.primary	(r, g, b, a)	正面主颜色
result.color.front.secondary	(r, g, b, a)	正面次颜色
result.color.back	(r, g, b, a)	背面主颜色
result.color.back.primary	(r, g, b, a)	背面主颜色
result.color.back..secondary	(r, g, b, a)	背面次颜色
result.fogcoord	(f, *, *, *)	雾化坐标
result.pointsize	(s, *, *, *)	点大小
result.texcoord	(s, t, r, q)	纹理坐标
result.texcoord [n]	(s, t, r, q)	纹理坐标

(表-3.2 , 顶点结果寄存器)

也可以通过申明绑定到某一个顶点结果寄存器名的输出变量访问，例如：

```
OUTPUT clipPos = result.position; # clipPos 等价于 result.position
```

顶点结果寄存器是 vertex program 的输出，是只写寄存器。

arbvp10 虚拟机支持至少 12 个临时寄存器。临时寄存器的值在 vertex program 激活时无定义，也不能在 vertex program 的两次运行间共享。它们一般用于保存 vertex program 执行过程中的临时运算结果。临时寄存器通过申明临时变量使用：

```
TEMP temp0, temp1, temp2; # 申明了三个绑定到三个临时寄存器的临时变量。
```

arbvp10 虚拟机支持至少一个地址寄存器。地址寄存器是四分量整数寄存器，并且只有 x 分量能够访问。地址寄存器用作访问程序参数数组的下标变量。地址寄存器通过申明地址变量使用：

```
ADDRESS Areg; # 定义了一个绑定到地址寄存器的地址变量。
```

程序参数寄存器。arbvp10 虚拟机支持至少 96 个程序参数寄存器。程序参数寄存器用于为 vertex program 提供执行所需要的只读参数。程序参数寄存器通过申明 PARAM 变量使用。需要指出的是 PARAM 类型的变量必须在申明的時候进行初始化，这一个初始化过程称为程序参数寄存器的显式绑定，即将这一 PARAM 变量绑定到某一个常量。程序参数寄存器可以绑定到四种量（可绑定量）：即常量、程序局部参数、程序环境参数和 OpenGL 状态变量。下面具体讨论每一种可绑定量及其绑定规则。注意除了显式绑定外还有隐式绑定。即对于 shader 中直接使用的一个可绑定量，系统会自动申明一个匿名 PARAM 变量绑定到该可绑定量。

1) 常量绑定。PARAM 变量通过如下方式绑定到常量：

```
PARAM par1 = 1.0; # par1 的值成为 {1.0, 1.0, 1.0, 1.0}
```

```
PARAM par2 = {1.0, 2.0, 3.0, 4.0}; # par2 的值成为 {1.0, 2.0, 3.0, 4.0}
```

2) 程序局部参数。常量的值只能在 vertex program 内部设置。如果要从 vertex program 外部提供常量值，则需要通过程序局部参数实现。OpenGL 应用通过 glProgramLocalParameter

可以为某一个特定的 vertex program 设置最多达 96 个程序局部参数,这些参数只对该 vertex program 有效。vertex program 通过申明绑定到一个或多个程序局部参数的 PARAM 变量来访问该参数。

PARAM par = program.local [8];

PARAM b [2] = program.local [4..5];

3) 程序环境参数。程序环境参数与程序局部参数类似,但程序环境参数通过 glProgramEnvParameter 设置并且对所有的 vertex program 有效。Vertex program 通过申明绑定到一个或多个程序环境参数的 PARAM 变量来访问该参数。

PARAM a = program.env [8];

PARAM b [2] = program.env [4..5];

4) OpenGL 状态变量。用户可以通过申明绑定到表-3.3 中的状态变量访问 OpenGL 状态变量。

状态变量名	分量	描述
材质状态变量:		
state.material.ambient	(r, g, b, a)	正面材质环境反射系数
state.mateiral.diffuse	(r, g, b, a)	正面材质漫反射系数
state.material.specular	(r, g, b, a)	正面材质镜面反射系数
state.material.shininess	(r, g, b, a)	正面材质镜面高光系数
state.material.front.*	(r, g, b, a)	正面材质的相关系数 (* 同上)
state.mateiral.back.*	(r, g, b, a)	反面材质的相关系数 (* 同上)
光源状态变量:		
state.light [n].ambient	(r, g, b, a)	光源 n 的环境光强
state.llight [n].diffuse	(r, g, b, a)	光源 n 的满反射光强
state.light [n].specular	(r, g, b, a)	光源 n 的镜面反射光强
state.light [n].position	(x, y, z, w)	光源 n 的位置
state.light [n].attenuation	(a, b, c, e)	光源 n 的衰减系数和聚光灯指数
state.light [n].spot.direction	(x, y, z, c)	光源 n 的聚光灯方向和衰减角余弦
state.light [n].half	(x, y, z, l)	光源 n 的无限半角
state.lightmodel.ambient	(r, g, b, a)	光照模型的环境光强
state.lightmodel.scenecolor	(r, g, b, a)	光照模型的正面场景光强
state.lightmodel.front.*	(r, g, b, a)	光照模型的正面场景光强
state.lightmodel.back.*	(r, g, b, a)	光照模型的反面场景光强
state.lightprod [n].ambient	(r, g, b, a)	
state.lightprod [n].diffuse	(r, g, b, a)	
state.lightprod [n].specular	(r, g, b, a)	
state.lightprod [n].front.*	(r, g, b, a)	
state.lightprod [n].back.*	(r, g, b, a)	
纹理坐标生成状态变量:		
state.texgen [n].eye.s	(a, b, c, d)	纹理单元 n 的 TexGen 观察空间平面系数 s 坐标
state.texgen [n].eye.t	(a, b, c, d)	纹理单元 n 的 TexGen 观察空间

		平面系数 t 坐标
state.texgen [n].eye.r	(a, b, c, d)	纹理单元 n 的 TexGen 观察空间 平面系数 r 坐标
state.texgen [n].eye.q	(a, b, c, d)	纹理单元 n 的 TexGen 观察空间 平面系数 q 坐标
state.texgen [n].object.*	(a, b, c, d)	纹理单元 n 的 TexGen 对象空间 平面系数 s/t/r/q 坐标
雾化状态变量：		
state.fog.color	(r, g, b, a)	雾颜色
state.fog.params	(d, s, e, r)	雾化系数
剪切平面状态变量：		
state.clip [n].plane	(a, b, c, d)	剪切平面 n 的平面方程系数
点状态变量：		
state.point.size	(s, n, x, f)	点大小
state.point.attenuation	(a, b, c, l)	点大小衰减常数
矩阵状态变量：		
state.matrix.modelview [n]		模型观察矩阵 n
state.matrix.projection		投影矩阵
state.matrix.mvp		模型观察-投影矩阵
state.matrix.texture [n]		纹理矩阵 n
state.matrix.palette [n]		模型观察矩阵调色板 n
state.matrrix.program [n]		顶点程序矩阵 n

(表-3.3, 可绑定的 OpenGL 状态变量)

例如, 可以通过如下方式申明绑定到当前材质的环境反射系数状态变量的 PARAM 变量:

```
PARAM ambient = state.material.ambient;
```

隐式或者显式申明的与四种寄存器对应的变量的个数不能超过相应的资源限制。具体的数值可以通过 OpenGL 的查询函数获得。但是 OpenGL 实现必须支持上面提到的那些最小值。

3.2.3 arbv10 指令集

arbv10 虚拟机线性执行当前 vertex program。一个 vertex program 最多允许包含 128 条指令 (或更多, 由具体的 OpenGL 实现确定)。arbv10 虚拟机支持 27 条 SIMD 指令。这些指令的操作数可以是标量, 也可以是四分量向量。指令执行的结果可以是一般向量, 也可以是由标量重复扩展而成的向量。当操作数是向量时, 对向量四个分量的运算同时进行。arbv10 指令的一般格式是:

```
Opcode dst, [-]s0 [, [-]s1 [, [-]s2]; # 注释, [ ] 表示可选。
```

例如:

```
MOV R2, R2;
```

```
MAD R1, R2, R3, -R4;
```

表-3.4 列出了 27 条指令的格式和简单说明, 其中 v 表示向量, s 表示标量, ssss 表示由标

量重复扩展而成的向量，a 表示单一的地址寄存器分量：

指令	目的操作数	源操作数	说明
ABS	v	v	求绝对值
ADD	v	v, v	相加（按分量）
ARL	a	v	载入到地址寄存器 *
DP3	ssss	v, v	三分量点积
DP4	ssss	v, v	四分量点积
DPH	ssss	v, v	其次点积
DST	v	v, v	计算距离向量
EX2	ssss	s	计算底数为 2 的指数
EXP	v	s	计算底数为 2 的指数（近似）*
FLR	v	v	计算比 v 小的最大整数（按分量）
FRC	v	v	计算 v 的小数部分（按分量）
LG2	ssss	s	计算底数为 2 的对数
LIT	v	v	计算光照计算系数
LOG	v	s	计算底数为 2 的对数（近似）*
MAD	v	v, v, v	先乘后加
MAX	v	v, v	计算最大值（按分量）
MIN	v	v, v	计算最小值（按分量）
MOV	v	v	赋值
MUL	v	v, v	相乘（按分量）
POW	ssss	s, s	计算指数
RCP	ssss	s	计算倒数
RSQ	ssss	s	计算平方根的倒数
SGE	v	v, v	大于等于时置位（按分量）
SLT	v	v, v	小于时置位（按分量）
SUB	v	v, v	相减（按分量）
SWZ	v	v	扩展的分量下标置换
XPD	v	v, v	计算叉积

（表-3.4，带 * 的指令只能用于 Vertex Program）

源操作可以取负。例如，假设 $R1 = \{1, 2, 3, 4\}$ ，那么执行下面的指令后， $R0$ 的值为：

MOV R0, -R1; # R0 = $\{-1, -2, -3, -4\}$ 。

作为源操作数的的向量操作数可以进行分量下标置换操作。例如，假设 $R1 = \{1, 2, 3, 4\}$ ，那么执行下列指令后， $R0$ 的值分别为：

MOV R0, R1; # R0 = $\{1, 2, 3, 4\}$ 。

MOV R0, R1.y; # R0 = $\{2, 2, 2, 2\}$ 。

MOV R0, R1.yzwx; # R0 = $\{2, 3, 4, 1\}$ 。

作为目的操作数的向量操作数可以设置可写掩码。例如，假设 $R0 = \{0, 0, 0, 0\}$ ， $R1 = \{1, 2, 3, 4\}$ ，那么执行下面的指令后， $R0$ 的值为：

MOV R0.xy, R1; # R0 = $\{1, 2, 0, 0\}$ 。

3.2.4 一个简单的 arbvp10 程序

```
!! ARBvp1.0

ATTRIB pos = vertex.position;
PARAM mat [4] = {state.matrix.mvp};

# Transform by concatenation of the MODELVIEW and PROJECTION matrices.
DP4 result.position.x, mat [0], pos;
DP4 result.position.y, mat [0], pos;
DP4 result.position.z, mat [0], pos;
DP4 result.position.w, mat [0], pos;

# Pass the primary color through w/o lighting.
MOV result.color, vertex.color;

END
```

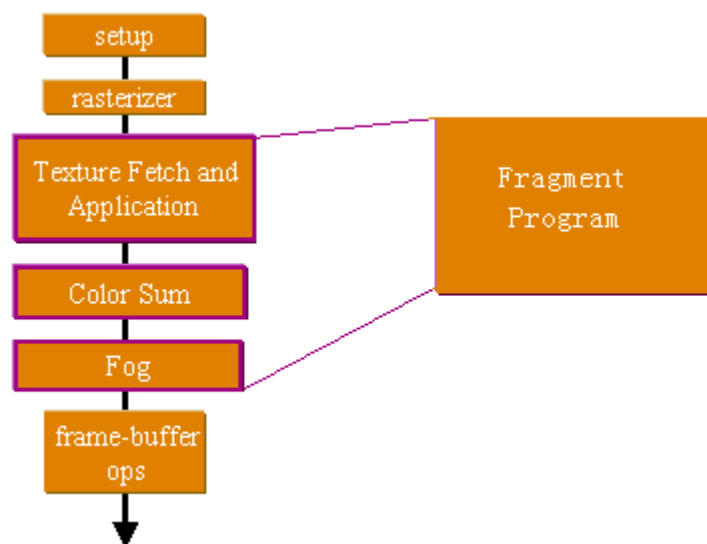
(图-3.3, arbvp10 着色程序)

上面的程序简单的将顶点的坐标由对象空间变换到齐次剪切空间,然后将顶点颜色不加修改的输出。

3.3 arbf10 着色语言

3.3.1 概述

Fragment shader 在 OpenGL 渲染流水线中的位置和作用如图-3.4 所示。



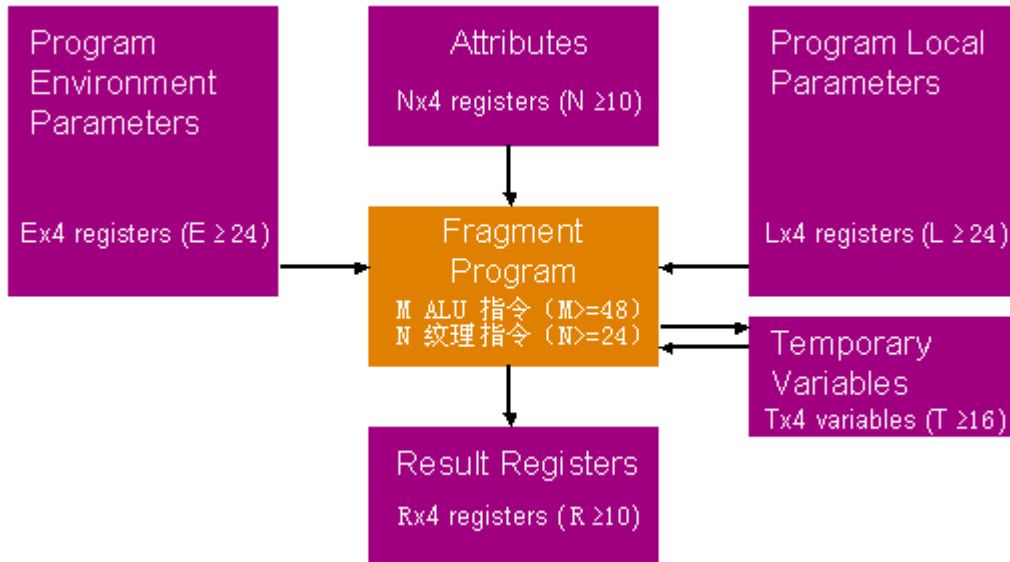
(图-3.4, OpenGL 中的片段着色程序)

用户编写的 LSL fragment shader (在 OpenGL 中称为 fragment program) 可以在用户的控制下取代 OpenGL 的固定片断处理模块。当用户允许 fragment program 模式时, fragment program 在光栅化模块每生成一个片断时执行。Fragment program 每次处理一个片断,对不同片断的处理互不影响。它的输入是片断的属性,包括片断颜色、纹理坐标、雾化坐标等;

输出是片断颜色和深度缓冲区深度值。Fragment program 一般来说需要完成纹理混合等任务。

3.3.2 arbf10 虚拟机

arbf10 虚拟机的框图如图-3.5 所示。



(图-3.5, arbf10 虚拟机)

与 arbv10 虚拟机类似, arbf10 虚拟机也可以看作是一个基于寄存器的向量处理器。arbf10 虚拟机中同样没有内存的概念, 所有的数据都保存在寄存器中。所有的寄存器都是四分量浮点寄存器。arbf10 虚拟机包括四种寄存器: 片断属性寄存器、片断结果寄存器、临时寄存器与程序参数寄存器。

arbf10 虚拟机至少支持 10 个片断属性寄存器。这些寄存器可以通过表-3.5 中的寄存器名直接访问。

片断属性寄存器名	分量	说明
fragment.color	(r, g, b, a)	片断主颜色
fragment.color.primary	(r, g, b, a)	片断主颜色
fragment.color.secondary	(r, g, b, a)	片断从颜色
fragment.texcoord	(s, t, r, q)	纹理坐标, 对应纹理单元 0
fragment.texcoord [n]	(s, t, r, q)	纹理坐标, 对应纹理单元 n
fragment.fogcoord	(f, 0, 0, 1)	片断雾化坐标
fragment.position	(x, y, z, 1/w)	片断在窗口中的位置

(表-3.5, 片断属性寄存器名)

片断属性寄存器也可以通过申明绑定到某一个片断属性寄存器的属性变量访问, 例如:

ATTRIB clr = fragment.color; # clr 等价于 fragment.color.

片断属性寄存器是 fragment program 的输入, 是只读寄存器。

arbf10 虚拟机至少支持 10 个片断结果寄存器。这些寄存器可以表-3.6 中的寄存器名直接访问。

片断结果寄存器名	分量	描述
result.color	(r, g, b, a)	片断输出颜色
result.depth	(* , * , d , *)	片断深度

(表-3.6, 片断结果寄存器)

也可以通过申明绑定到某一个片断结果寄存器的输出变量访问, 例如:

```
OUTPUT finalClr = result.color; # finalClr 等价于 result.color。
```

片断结果寄存器是 fragment program 的输出, 是只写寄存器。

arbf10 虚拟机至少支持 16 个临时寄存器。临时寄存器的值在 fragment program 激活时无定义, 也不能在 fragment program 的两次运行间共享。它们一般用于保存 fragment program 运行过程中的临时运算结果。临时寄存器通过申明临时变量使用:

```
TEMP temp0, temp1, temp2; # 申明了三个绑定到三个临时寄存器的临时变量。
```

arbf10 虚拟机至少支持 24 个程序参数寄存器。程序参数寄存用于为 fragment program 提供执行所需要的只读参数。程序参数寄存器通过申明 PARAM 变量使用。需要指出的是 PARAM 类型的变量必须在申明的時候进行初始化, 这一个初始化过程称为程序参数寄存器的显式绑定, 即将这一 PARAM 变量绑定到某一个常量。程序参数寄存器可以绑定到四种量(可绑定量): 即常量、程序局部参数、程序环境参数和 OpenGL 状态变量。下面具体讨论每一种可绑定量及其绑定规则。注意除了显式绑定外还有隐式绑定。即对于 shader 中直接使用的一个可绑定量, 系统会自动申明一个匿名 PARAM 变量绑定到该可绑定量。

1) 常量绑定。PARAM 变量通过如下方式绑定到常量:

```
PARAM par1 = 1.0; # par1 的值成为 {1.0, 1.0, 1.0, 1.0}
```

```
PARAM par2 = {1.0, 2.0, 3.0, 4.0}; # par2 的值成为 {1.0, 2.0, 3.0, 4.0}
```

2) 程序局部参数。常量的值只能在 fragment program 内部设置。如果要从 fragment program 外部提供常量值, 则需要通过程序局部参数实现。OpenGL 应用通过 glProgramLocalParameter 可以为某一个特定的 fragment program 设置最多达 24 个程序局部参数, 这些参数只对该 fragment program 有效。Fragment program 通过申明绑定到一个或多个程序局部参数的 PARAM 变量来访问该参数。

```
PARAM par = program.local [8];
```

```
PARAM b [2] = program.local [4..5];
```

3) 程序环境参数。程序环境参数与程序局部参数类似, 但程序环境参数通过 glProgramEnvParameter 设置并且对所有的 fragment program 有效。Fragment program 通过申明绑定到一个或多个程序环境参数的 PARAM 变量来访问该参数。

```
PARAM a = program.env [8];
```

```
PARAM b [2] = program.env [4..5];
```

4) OpenGL 状态变量。用户可以通过申明绑定到表-3.7 中的状态变量访问 OpenGL 状态变量。

状态变量名	分量	描述
材质状态变量:		
state.material.ambient	(r, g, b, a)	正面材质环境反射系数
state.mateiral.diffuse	(r, g, b, a)	正面材质漫反射系数
state.material.specular	(r, g, b, a)	正面材质镜面反射系数
state.material.shineness	(r, g, b, a)	正面材质镜面高光系数
state.material.front.*	(r, g, b, a)	正面材质的相关系数 (* 同上)
state.mateiral.back.*	(r, g, b, a)	反面材质的相关系数 (* 同上)
光源状态变量:		
state.light [n].ambient	(r, g, b, a)	光源 n 的环境光强

state.light [n].diffuse	(r, g, b, a)	光源 n 的满反射光强
state.light [n].specular	(r, g, b, a)	光源 n 的镜面反射光强
state.light [n].position	(x, y, z, w)	光源 n 的位置
state.light [n].attenuation	(a, b, c, e)	光源 n 的衰减系数和聚光灯指数
state.light [n].spot.direction	(x, y, z, c)	光源 n 的聚光灯方向和衰减角余弦
state.light [n].half	(x, y, z, l)	光源 n 的无限半角
state.lightmodel.ambient	(r, g, b, a)	光照模型的环境光强
state.lightmodel.scenecolor	(r, g, b, a)	光照模型的正面场景光强
state.lightmodel.front.*	(r, g, b, a)	光照模型的正面场景光强
state.lightmodel.back.*	(r, g, b, a)	光照模型的反面场景光强
state.lightprod [n].ambient	(r, g, b, a)	
state.lightprod [n].diffuse	(r, g, b, a)	
state.lightprod [n].specular	(r, g, b, a)	
state.lightprod [n].front.*	(r, g, b, a)	
state.lightprod [n].back.*	(r, g, b, a)	
雾化状态变量 :		
state.fog.color	(r, g, b, a)	雾颜色
state.fog.params	(d, s, e, r)	雾化系数
深度状态变量 :		
state.depth.range	(n, f, d, l)	深度范围
矩阵状态变量 :		
state.matrix.modelview [n]		模型观察矩阵 n
state.matrix.projection		投影矩阵
state.matrix.mvp		模型观察-投影矩阵
state.matrix.texture [n]		纹理矩阵 n
state.matrix.palette [n]		模型观察矩阵调色板 n
state.matrrix.program [n]		顶点程序矩阵 n

(表-3.7, 可绑定的 OpenGL 状态变量)

例如, 可以通过如下方式申明绑定到当前材质的环境反射系数状态变量的 PARAM 变量:

```
PARAM ambient = state.material.ambient;
```

隐式或者显式申明的与四种寄存器对应的变量的个数不能超过相应的资源限制。具体的数值可以通过 OpenGL 的查询函数获得。但是 OpenGL 实现必须支持上面提到的那些最小值。

3.3.3 arbf10 指令集

arbf10 虚拟机线性执行当前 fragment program。一个 fragment program 最多允许包含 72 条指令 (或更多, 由具体的 OpenGL 实现确定)。其中算术指令不能超过 48 条, 纹理指令不能超过 24 条。arbf10 虚拟机支持 33 条指令。其中 29 条是 SIMD 算是指令, 4 条是纹理指令。这些指令的操作数可以是标量, 也可以是四分量向量。指令执行的结果可以是一般向量, 也可以是由标量重复扩展而成的向量。当操作数是向量时, 对向量四个分量的运算同时

进行。arbf10 指令的一般格式是：

Opcode dst, [-]s0 [, [-]s1 [, [-]s2]; # 注释, [] 表示可选。

例如：

MOV R2, R2;

MAD R1, R2, R3, -R4;

表-3.8 列出了 33 条指令的格式和简单说明，其中 v 表示向量，s 表示标量，ssss 表示由标量重复扩展而成的向量，a 表示单一的地址寄存器分量：

指令	目的操作数	源操作数	说明
ABS	v	v	求绝对值
ADD	v	v, v	相加（按分量）
CMP	v	v, v, v	条件赋值（按分量）*
COS	ssss	s	计算余弦 *
DP3	ssss	v, v	三分量点积
DP4	ssss	v, v	四分量点积
DPH	ssss	v, v	齐次点积
DST	v	v, v	计算距离向量
EX2	ssss	s	计算底数为 2 的指数
FLR	v	v	计算比 v 小的最大整数（按分量）
FRC	v	v	计算 v 的小数部分（按分量）
LG2	ssss	s	计算底数为 2 的对数
LIT	v	v	计算光照计算系数
LRP	v	v, v, v	线性插值 *
MAD	v	v, v, v	先乘后加
MAX	v	v, v	计算最大值（按分量）
MIN	v	v, v	计算最小值（按分量）
MOV	v	v	赋值
MUL	v	v, v	相乘（按分量）
POW	ssss	s, s	计算指数
RCP	ssss	s	计算倒数
RSQ	ssss	s	计算平方根的倒数
SCS	ss--	s	同时计算正弦与余弦 *
SGE	v	v, v	大于等于时置位（按分量）
SIN	ssss	s	计算正弦 *
SLT	v	v, v	小于时置位（按分量）
SUB	v	v, v	相减（按分量）
SWZ	v	v	扩展的分量下标置换
XPD	v	v, v	计算叉积
KIL	v	v	终止 Fragment Program 运行 *
TEX	v	v, u, t	纹理抽样 *
TXB	v	v, u, t	带偏移的纹理抽样 *
TXP	v	v, u, t	纹理坐标投影后抽样 *

（表-3.8，带 * 的指令只能用于 Fragment Program）

源操作数可以取反。例如，假设 $R1 = \{1, 2, 3, 4\}$ ，那么执行下面的指令后， $R0$ 的值为：

```
MOV R0, -R1; # R0 = {-1, -2, -3, -4}。
```

作为源操作数的的向量操作数可以进行分量下标置换操作。例如，假设 $R1 = \{1, 2, 3, 4\}$ ，那么执行下列指令后， $R0$ 的值分别为：

```
MOV R0, R1; # R0 = {1, 2, 3, 4}。
```

```
MOV R0, R1.y; # R0 = {2, 2, 2, 2}。
```

```
MOV R0, R1.yzwx; # R0 = {2, 3, 4, 1}。
```

作为目的操作数的向量操作数可以设置可写掩码。例如，假设 $R0 = \{0, 0, 0, 0\}$ ， $R1 = \{1, 2, 3, 4\}$ ，那么执行下面的指令后， $R0$ 的值为：

```
MOV R0.xy, R1; # R0 = {1, 2, 0, 0}。
```

3.3.4 一个简单的 arbf10 程序

```
!! ARBfp1.0
```

```
TEMP temp; # temporary.
```

```
ATTRIB tex0 = fragment.texcoord [0];
```

```
OUTPUT out = result.color;
```

```
TEX temp, tex0, texture [0], 2D; # Fetch texture.
```

```
MOV out, temp; # Output result fragment color.
```

```
END
```

(图-3.6， arbf10 着色程序)

上面的程序根据纹理坐标访问纹理，然后将纹理抽样值作为片断的颜色输出。

3.4 arbv20 与 arbf20 着色语言

arbv10/arbfp10 虚拟机的一个重要的特点是它们不支持任何类型的转移指令，包括绝对转移指令、条件转移指令和过程调用与返回指令。这一特点准确的反映了 arbv10/arbfp10 虚拟机所面向的早期图形处理器较弱的硬件处理能力。对于简单的着色程序来说，线性执行的编程模型可以满足大多数需求。但是复杂的图形应用，特别是要达到电影集并渲染效果的实时图形应用需要编写复杂的着色程序。另一方面，图形处理器正在逐渐向着一般的流处理器发展，图形处理器可以支持的指令集也越来越接近微处理器指令集。为此，需要定义与之匹配的具有更强能力的虚拟机。与 arbv10/arbfp10 的发展类似，各个硬件厂商提出了自己的 OpenGL 扩展（例如 nVidia 的 NV_VERTEX_PROGRAM2 [nvvp20]）来定义这一虚拟机。为了解决图形程序员需要支持不同扩展的问题，OpenGL ARB 目前正在着手制定 arbv20/arbfp20 扩展来统一不同的扩展。由于 arbv20/arbfp20 标准还未公布，我们只能结合 NV_VERTEX_PROGRAM2 扩展和 arbv10/arbfp10 标准来预测可能的 arbv20/arbfp20 标准。

NV_VERTEX_PROGRAM2(下面简单表示为 nvvp20)是 nVidia 公司提出的面向 nVidia 最新图形处理器芯片 NV3X 的 OpenGL 扩展。考虑到 nVidia 公司在 arbv10 制订过程中所起的作用，可以相信 arbv20 将具有很多 nvvp20 的特性。Nvvp20 对于 arbv10 最重要的扩展是增加了无条件转移、条件转移和过程调用与返回指令。这样，用 nvvp20 编写的着色程序就可以具有非常复杂的结构和功能。

为了支持转移指令，nvvp20 增加了一个四分量条件码寄存器 CC，三条转移指令（BRA, CAL 和 RET）以及对标号语句的支持。同时对于每一条算术指令，nvvp20 还增加了一个会修改条件码寄存器 CC 的对应版本。

条件码寄存器 CC 的四个分量可以是下列枚举值之一：

GT（大于），EQ（等于），LT（小于），UN（无序）

这些分量的初始值都为 EQ。当执行一条会修改 CC 寄存器的指令时，图形处理器会根据运算结果的每一分量与 0 的大小关系设置 CC 寄存器的相应分量。之后转移指令就可以根据 CC 寄存器的内容确定是否进行转移。

转移指令的一般格式是：

BRA <标号> [<转移条件>];

其中标号是定义在着色程序中某处的标号，标号可以在转移语句之前，也可以在转移语句之后。下表是一些有效的转移条件及其含义。

转移条件	含义
GT	CC 寄存器的任一分量为 GT 时转移
EQ	CC 寄存器的任一分量为 EQ 时转移
LT.x	CC 寄存器的 x 分量为 LT 时转移
LT.xyzw	CC 寄存器的任一分量为 GT 时转移
LT.wyzw	CC 寄存器的 y, z, w 分量中任一为 LT 时转移
空	无条件转移

（表-3.9，转移条件）

下面是一个使用转移指令的例子：

MOVC CC, c [0]; # c [0] = (-2, 0, 2, NaN), CC 值为 (LT, EQ, GT, UN)。

BRA Label1 (LT.xyzw);

MOV R0, R1; # 不会执行。

Label1:

BRA Label2 (LT.wyzw);

MOV R0, R2; # 语句会执行。

Label2:

容易看出第一个条件转移指令会转移到 Label1，但第二个条件转移指令不会转移到 Label2。

过程调用语句的格式类似于转移指令，可以无条件调用过程也可以条件调用过程。同样过程返回指令也可以无条件或者条件返回。nvvp20 限制过程调用不能超过四层，一旦图形处理器检测到超过四层的过程调用就会自动结束 vertex program 的执行。

nvvp20 还限制一个 vertex program 程序的所占的指令槽数不能超过 256，一次运行执行的全部指令数不能超过 64K。

我们可以把 nvvp20 的上述指令添加到 arbvp10 与 arbf10 虚拟机中，构造一个假想的 arbvp20 与 arbf20 虚拟机做为 glslang 编译器 gcx 的编译目标机器。

3.5 小结

本章介绍的四个虚拟机 arbvp10、arbf10、arbvp20 与 arbf20 将是 glslang 编译器 gcx 的四个编译目标机器。arbvp10/arbf10 不支持转移指令，只能从头线性执行着色程序；而 arbvp20/arbf20 支持动态转移指令，能够支持复杂的控制流结构。我们将使用不同的技术将 HLSL shader 转换为可以在上述四个虚拟机上执行的 LLSL shader。

第四章 高级着色语言

4.1 概述

正如通用处理器上的软件开发从汇编语言程序设计逐渐过渡到高级语言程序设计一样，着色程序开发也在经历着同样的过程。低级着色语言虽然能够满足大多数图形应用的需要，允许开发者写出性能最好的着色程序，以最大程度的利用图形处理器的硬件能力，但是它不可避免的具有下述问题：

1) 对底层图形硬件具有强烈的依赖性。虽然 DirectX, OpenGL 等 3D API 可以通过定义虚拟机抽象底层硬件的体系结构，但是正如我们所看到的，用他们编写着色程序仍然不的不选择某一个具体版本的低级着色语言来匹配底层的图形硬件。

2) 低级着色语言难以学习，开发效率较低。难学习，难使用是汇编语言的通性，而图形处理器的向量体系更加增大了学习和使用的难度。

3) 无法直接利用大量现存的 Renderman shader。

4) 不能利用广泛使用的各种优化编译技术，开发者必须手工优化汇编代码。

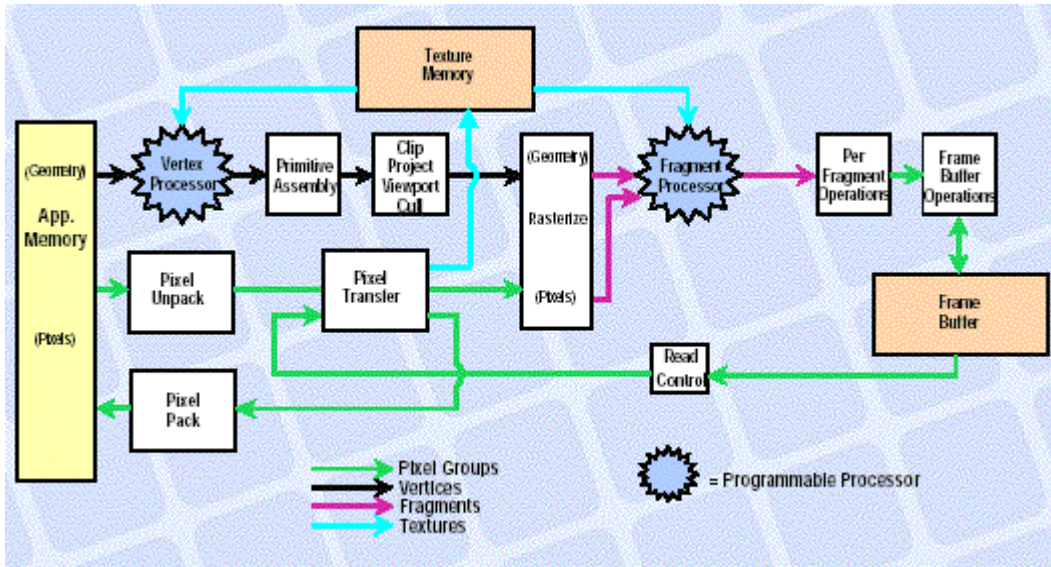
为此，高级着色语言进入研究者的视野。2001 年 Pat Hanrahan 等在 siggraph 上提出了一个实时过程着色系统：斯坦福着色系统 [Pat2001]。该系统由三部分组成，第一部分是对可编程图形流水线的一个抽象，第二部分是一个类似于 Renderman 着色语言的高级着色语言，第三部分是一个支持多后端的后端编译器。此后不久，商业化的高级着色语言相继出现，它们是 Cg、DirectX HLSL 与和 OpenGL glslang。Cg 由 nVidia 公司和微软公司联合开发而成，是一种增加了向量与矩阵处理能力的类 C 语言。Cg 的设计目标是成为一种通用的面向硬件的语言。它既可以用来开发用于渲染的着色程序，也可以用来开发在图形处理器上运行的科学计算程序。Cg 是一个跨平台的高级着色语言，通过定义不同的配置 (profile)，Cg 编译器可以为不同的低级着色语言后端生成代码，既支持 DirectX 的所有版本的低级着色语言，也支持 OpenGL 的所有版本的低级着色语言，同时还支持 nVidia 专有的所有版本的低级着色语言。DirectX HLSL 类似于 Cg，但是它的后端仅支持 DirectX 的各个版本的低级着色语言。

glslang 是 OpenGL Shading Language 的简称，它是 OpenGL 2.0 规范的一部分，是 OpenGL 平台上的高级着色语言。glslang 在 OpenGL 管理委员会的监督下，由 3D Labs 公司制定规范和提供参考实现。glslang 的设计目标是能够体现当前和未来图形硬件能力，易于使用，功能强大，可以明显减少日渐增加的 OpenGL 扩展数目。本文的主要研究对象就是 glslang 及其优化编译技术。

4.2 glslang 简介

4.2.1 glslang 程序运行环境

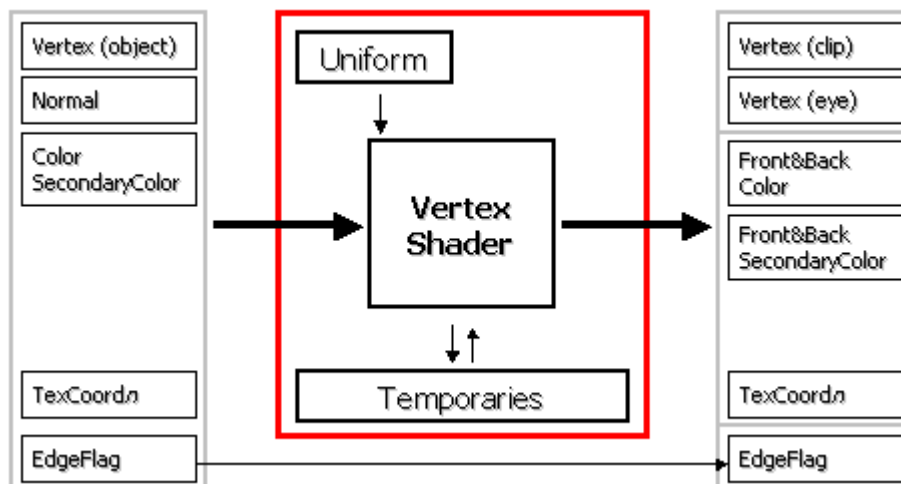
glslang 与 OpenGL 2.0 规范紧密联系。下面是制订中的 OpenGL 2.0 逻辑框图。



(图-4.1 OpenGL 2.0 逻辑框图)

在 OpenGL 2.0 中，可编程的顶点处理器取代了固定功能的顶点处理模块；可编程的片段处理器取代了固定功能的片段处理模块。用户可以用 glslang 编写运行在顶点处理器之上的程序，称为顶点着色程序 (vertex program)；或者用 glslang 编写运行在片段处理器之上的程序，称为片段着色程序 (fragment program)。当然，用户可以继续使用 OpenGL 内置的固定功能渲染模块。用于编写 vertex program 与 fragment program 的 glslang 略有区别，所以事是上存在两种 glslang 语言。根据运行的处理器不同，可以分为 vertex glslang 和 fragment glslang。当然，这两种语言仍然具有相当多的共性，除了一些微妙的区别，我们仍然可以把它们看作是同一种语言。为了管理 OpenGL 2.0 应用中用到的着色程序，OpenGL 2.0 规范引入了 program 对象和 shader 对象的概念。运行在顶点或者片段处理器上的一个完整的程序称为 program 对象，一个 program 对象由一个或多个 shader 对象组成。Shader 对象是一个独立的可编译单元，类似于 C 语言中的一个源文件。与 glslang 语言类似，program 对象与 shader 对象也有 vertex 和 fragment 的区别。组成 program 对象的各 shader 对象必须与被组成的 program 对象具有相同的类型。要在 OpenGL 2.0 应用中使用 program 对象，首先要把组成 program 对象的 shader 对象附着到 program 对象之上，然后对 program 对象进行编译，之后对编译后的 program 对象进行连接。

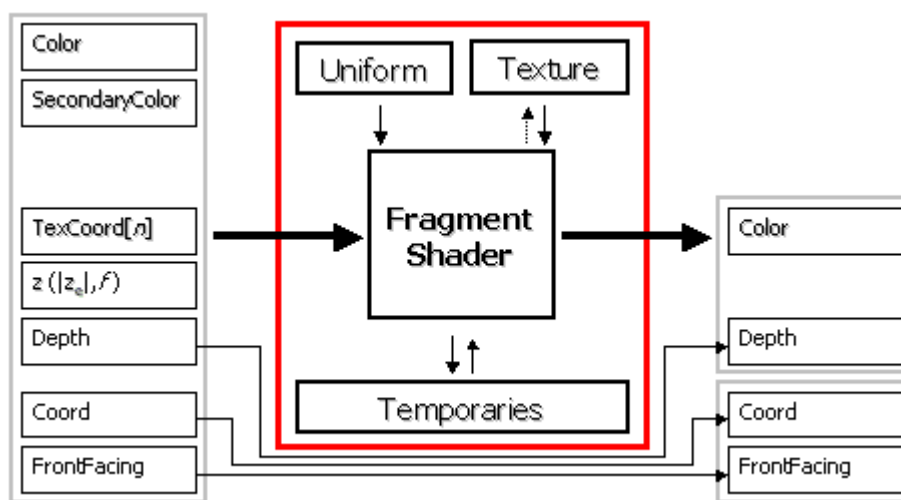
顶点处理器用于进行顶点变换、法向量变换与归一化、纹理坐标生成与变换、光照计算等操作。图-4.2 是顶点处理器的逻辑框图。



(图-4.2 顶点处理器逻辑框图)

图-4.2 的左边部分是 vertex program 的输入，也即顶点属性。OpenGL 2.0 定义了两种顶点属性：传统顶点属性和一般顶点属性。与 arbv10 语言不同，vertex glslang 中的这两种顶点属性是互不影响的。在 OpenGL 程序中，传统顶点属性通过 glVertex, glNormal 等设置，一般顶点属性通过 glVertexAttrib 设置。经优化编译后的 vertex program 中真正访问的顶点属性称为活跃属性（无论是传统还是一般顶点属性），一个 vertex program 中的活跃顶点总数不能超过 MAX_VERTEX_ATTRIBS_ARB (16)。在 vertex program 中，传统顶点属性通过内置的顶点属性变量访问，一般顶点属性通过用户定义的属性变量访问。图-4.2 的右边部分是 vertex program 的输出。vertex program 可以通过内置的特殊变量（special variables）和插值变量（varying variables）输出，也可以通过用户定义的插置变量输出，但是不管用什么方式输出，必须写内置特殊变量 gl_Position。图-4.2 中间上方的 uniform 表示顶点处理器中的 uniform 寄存器。这些寄存器用于设置在渲染一个或多个几何基元（primitive）时保持不变的常量值。这些值由 OpenGL 2.0 应用在 vertex program 之外设置。用户在 vertex program 中可以申明 uniform 变量来访问 uniform 寄存器中保存的值。OpenGL 2.0 也定义了内置 uniform 变量来提供对常量和 OpenGL 状态变量的访问。经优化编译后的 vertex program 中真正用到的 uniform 变量称为活跃 uniform 变量。活跃 uniform 变量所占单一浮点空间的总数不能超过 MAX_VERTEX_UNIFORM_COMPONENTS_ARB (512)。OpenGL 2.0 对 vertex program 的指令数和临时寄存器使用数没有限制。

片断处理器用于进行纹理访问、纹理混合、雾化等操作。图-4.3 是片断处理器的逻辑框图。



(图-4.3 片断处理器逻辑框图)

图-4.3 的左边部分是 fragment program 的输入，也即片断属性，片断属性都是插值变量。片断属性由光栅化模块对 vertex program 或者固定顶点处理模块生成的顶点数据进行插值生成。片断属性有两种：内置插值变量和自定义插值变量。内置插值变量通过表-4.7 中的插值变量名访问。自定义插值变量通过在 vertex program 和 fragment program 中定义同名插值变量访问。图-4.3 的右边部分是 fragment program 的输出，称为片断输出。片断输出通过表-4.8 中的特殊变量名访问。输出值将传递给 OpenGL 2.0 渲染流水线的后续模块进行处理。图-4.3 中间上方的 uniform 与 vertex program 中的 uniform 意义相同，但是最大活跃 uniform 变量数为 MAX_FRAGMENT_UNIFORM_COMPONENTS_ARB (64)。OpenGL 2.0 对 fragment program 的指令数和临时寄存器使用数也没有限制。

4.2.2 glslang 程序例子

图-4.4 是一个简单的 vertex program 的例子：

```
varying vec3 Normal; // output.
#define MVP gl_ModelViewProjectionMatrix
#define MV gl_ModelViewMatrix
#define MV_IT gl_NormalMatrix

void main ( void )
{
    gl_Position = MVP * gl_Vertex;
    Normal = normalize ( MV_IT * gl_Normal );
}
```

(图-4.4, Vertex Program)

图-4.5 是一个简单的 fragment program 的例子：

```
varying vec3 Normal; // input.
uniform vec3 LightColor; // Light color.
uniform vec3 LightPos; // Light position.

void main ( void )
{
    vec3 color = LightColor;
    color *= max ( 0.0, dot ( normalize ( Normal ), LightPos ) );
    gl_FragColor = vec4 ( color, 1.0 );
}
```

(图-4.5, Frgment Program)

容易看出, glslang 与 C 语言具有极大的类似性。事实上, glslang 规范就是在 C 语言和 Renderman 着色语言的基础上制定而成的, 这也是其它高级着色语言所采用的策略。

4.3 glslang 变量与数据类型

4.3.1 glslang 数据类型

表-4.1 列出了 glslang 支持的基本数据类型。

数据类型	说明
void	用于申明不返回任何数据的函数
bool	布尔类型, 值为 true 或者 false
int	有符号整数
float	浮点标量
vec2	二分量浮点向量
vec3	三分量浮点向量
vec4	四分量浮点向量

bvec2	二分量布尔向量
bvec3	三分量布尔向量
bvec4	四分量布尔向量
ivec2	二分量整数向量
ivec3	三分量整数向量
ivec4	四分量整数向量
mat2	2 × 2 浮点矩阵
mat3	3 × 3 浮点矩阵
mat4	4 × 4 浮点矩阵
sampler1D	用于访问一维纹理
sampler2D	用于访问二维纹理
sampler3D	用于访问 3 维纹理
samplerCube	用于访问立方映射纹理
sampler1DShadow	用于带比较访问一维深度纹理
sampler2DShadow	用于带比较访问二维深度纹理

(表-4.1, glslang 的基本数据类型)

除了表-4.1 中的基本数据类型外, glslang 还支持两种聚合数据类型: 结构和数组。glslang 中数据类型的语法和语义与 C 语言基本相同, 但也存在一些区别。在 glslang 中, 底层硬件也许不能直接支持 bool 和 int 类型。int 类型为 16 位精度, float 精度为 IEEE 单精度浮点。Matrix 类型按列存储。结构类型不支持匿名成员, 不支持前向引用, 也不支持位域, 同时内嵌定义的结构类型局限于包含它的结构类型之内。数组仅支持一维数组。Sampler 变量只能申明为 uniform 变量或者函数参数, 并且是只读的, 它的值由 OpenGL 2.0 应用设置以绑定到具体的纹理。变量可以用与 C 语言类似的语法在申明时初始化, 也可以用构造器进行初始化。下面是一些变量申明的例子:

```
bool bSuccess;
int i, j = 42;
float a, b = 1.5;
vec4 vColor = vec4 ( 0.3, 0.4, 0.4, 1 );
mat4 mView;
struct light
{
    float intensity;
    vec3 position;
};
float freqs [3];
light lights [4];
```

4.3.2 作用域

glslang 的作用域规则与 C 语言相同。

4.3.3 类型修饰符

表-4.2 列出了 glslang 支持的类型修饰符 (type qualifiers)。

类型修饰符	说明
<none, default>	用于申明局部变量或者函数形参
const	用于申明常量类型或者只读函数形参
attribute	用于申明顶点属性变量，只用于全局范围
varying	用于申明插值变量，只能用于全局范围
uniform	用于申明不变变量，只能用于全局范围
in	用于申明函数只读形参
out	用于申明函数只写形参
inout	用于申明函数读写形参

(表-4.2, 类型修饰符)

全局变量只能使用 const、attribute、uniform 或者 varying 修饰符。不使用或者使用 const 修饰符的全局变量可以在申明时初始化。未初始化的全局变量在进入 main 函数时其值无定义。Attribute 变量用于向 vertex program 传递顶点属性。Varying 变量由 vertex program 按每顶点设置，经光栅化模块插值后按每片断传递给 fragment program。Uniform 变量用于设置程序运行所需的外部参数和访问 OpenGL 常量、状态变量以及纹理抽样器。局部变量只能使用 const 修饰符。函数形参只能使用 in、out、inout 和 const 修饰符。

4.4 glslang 运算符与表达式

glslang 支持绝大多数 C 语言运算符，但是不支持与指针相关的运算符，也不支持强制类型转换 (cast) 运算符。glslang 数组的访问方法与支持的运算符和 C 语言类似，但是数组下标必须是整型表达式。glslang 结构的访问方法与支持的运算符和 C 语言类似，但是支持 == 与 != 运算符。

向量类型的分量用类似于访问结构成员变量的方式访问。向量的分量名如表-4.3 所示。三种分量名都可以用于访问同一个向量的分量，但是不能混合使用。也可以把向量看作是数组，用访问数组的方式访问。

分量名	说明
{x, y, z, w}	一般用于访问代表位置和法向量的向量
{r, g, b, a}	一般用于访问表示代表颜色的向量
{s, t, p, q}	一般用于访问代表纹理坐标的向量

(表-4.3, 向量分量名)

下面是一些正确和错误的访问向量分量的例子：

```
vec4 pos;
vec3 normal;
vec4 color;

pos.x;      // 正确
pos.xyzw;   // 正确
pos.wxyz;   // 正确，分量名可以置换
```

```

pos.xyy;    // 正确，分量名可以复制，但是这种形式只能用于右值
pos.rg;     // 正确
pos [3];    // 正确，访问 z 分量
pos.rgbw;   // 不正确，不能混合使用分量名
normal.xw;  // 不正确，normal 没有第四个分量。

```

矩阵的分量可以通过数组访问方式访问。例如：

```

mat4 m;

m [1];      // 访问矩阵的第二列向量
m [0][0];   // 访问矩阵的左上角元素

```

作用于向量和矩阵的一元运算都按分量运算。对于标量与向量或者矩阵相乘，运算结果是标量乘以向量或者矩阵的每一个分量。对于向量左乘矩阵、矩阵左乘向量和矩阵乘以矩阵，按照线性代数中的向量矩阵乘法运算。除此之外的二元运算都是按分量运算，且必须作用于同类型的量。

glslang 引入了 C++语言中的构造器（Constructor）来进行类型转换和对复杂类型进行初始化、赋值。下面是一些构造器使用的例子：

```

bool b = false;
int n = int ( b );           // bool 转换成整数
float f = 0.0;
vec3 v3 = vec3 ( f );       // 用 f 的值初始化 v3 的三个分量
vec4 v4 = vec4 ( v3 , f );   // v4 的前三个分量初始化成 v3 的相应分量，第四个分量初始化成 f

```

glslang 的表达式类似于 C 语言中的表达式，但是不支持隐式类型转换。

4.5 glslang 语句与程序结构

一个 shader（glslang 的编译单元）的结构与 C 语言的一个源文件类似。但是 glslang 也引入了一些 C++语言的特性，包括用时定义变量和函数重载。

函数调用按 value-return 规则传递参数。对于 in 参数，实参的值在调用函数时复制到函数的形参；对于 out 参数，形参的值在函数返回时复制到实参。实参按照从左向右的顺序求值。递归（直接或者间接）调用会导致未定义的结果。

glslang 支持选择语句（if, if-else），循环语句（for, while-do, do-while）和结构化转移语句（continue, break, return, discard），但是不支持 switch 语句。前述语句除了 discard 语句外，语法和语义与 C 语言中的相应语句相同。discard 语句只能用在 fragment program 中，执行该语句将立刻终止对当前片断的处理，也不会更新帧缓冲区。

4.6 glslang 内置变量与函数

4.6.1 glslang 内置顶点属性

表-4.4 所列是 glslang 的内置顶点属性，它们都是特殊变量。Vertex program 通过这些变量访问传统顶点属性。

内置顶点属性	类型	说明
gl_Color	vec4	顶点主颜色

gl_SecondaryColor	vec4	顶点从颜色
gl_Normal	Vec3	顶点法向量
gl_Vertex	vec4	顶点坐标
gl_MultiTexCoord0	vec4	顶点纹理坐标, 对应纹理单元 0
gl_MultiTexCoord7	vec4	顶点纹理坐标, 对应纹理单元 7
gl_ForCoord	vec4	顶点雾化坐标

(表-4.4, glslang 内置顶点属性)

4.6.2 glslang 内置顶点输出

glslang 的内置顶点输出有两类。一类是特殊变量, 见表-4.5, vertex program 通过这些变量名与 OpenGL 2.0 的固定流水线处理模块通讯。另一类是插值变量, 见表-4.6, vertex program 通过这些变量名与 OpenGL 2.0 的固定流水线处理模块和 fragment program 通讯。

内置顶点输出	类型	是否必须写	说明
gl_Position	vec4	是	定点齐此空间坐标
gl_PointSize	Float	否	点大小
gl_ClipVertex	vec4	否	

(表-4.5, glslang 内置顶点输出, 特殊变量)

内置顶点输出	类型	说明
gl_FrontColor	vec4	正面顶点主颜色
gl_BackColor	vec4	背面顶点主颜色
gl_FrontSecondaryColor	vec4	正面顶点从颜色
gl_BackSecondaryColor	vec4	背面顶点从颜色
gl_TexCoord []	Vec4	顶点纹理坐标
gl_FogFragCoord	float	顶点雾化坐标

(表-4.6, glslang 内置顶点输出, 插值变量)

4.6.3 glslang 内置片断属性

表-4.7 所列是内置片断属性, 它们都是插值变量。Fragment program 通过这些变量获得输入片断的属性。

内置片断属性	类型	说明
gl_Color	vec4	片断主颜色
gl_SecondaryColor	vec4	片断从颜色
gl_TexCoord []	vec4	片断纹理坐标
gl_FogFragCoord	float	片断雾化坐标

(表-4.7, glslang 内值片断输入)

4.6.4 glslang 内置片断输出

表-4.8 所列是 glslang 的内置片断输出，它们都是特殊变量。Fragment program 通过这些变量与后续的 OpenGL 2.0 处理模块通讯。

内置片断输出	类型	说明
gl_FragCoord	vec4	片断坐标
gl_FrontFacing	Bool	片断所在面的朝向
gl_FragColor	vec4	片断颜色
gl_FragDepth	Float	片断深度

(表-4.8, glslang 内置片断输出)

4.6.5 glslang 内置常量与内置 uniform 状态变量

glslang 还提供了内值常量和内值 uniform 状态变量以允许 glslang shader 程序访问 OpenGL 状态和实现相关常量。这里不详细列出这些变量，具体可以参考 glslang 规范。

4.6.6 glslang 内置函数

glslang 还提供了大量用于标量和向量操作的内置函数。glslang shader 应当尽量使用内置函数，因为底层的图形硬件可以提供对这些函数硬件加速。glslang 内置函数可以分为如下几类：三角函数、指数函数、通用函数、几何函数、矩阵函数、向量关系函数、纹理访问函数、片断函数和噪音函数。这里不详细列出，具体可以参考 glslang 规范。