

OPENGL® 着色语言

约翰·凯塞尼奇 戴夫·鲍德温 兰

迪·罗斯 语言版本 1.10

文档修订版 59

2004年4月30日

版权所有 © 2002-2004 3Dlabs, Inc. Ltd.

本文档包含 3Dlabs, Inc. Ltd. 未公开的信息。

本文件受版权保护，包含3Dlabs, Inc. Ltd.的专有信息。未经3Dlabs, Inc. Ltd.明确书面许可，严禁复制、改编、分发、公开表演或公开展示本文件。接收或持有本文件并不授予任何权利，包括复制、披露或分发其内容，或制造、使用或销售其中描述的任何全部或部分内容。

本文件包含3Dlabs Inc. Ltd.的知识产权，但不授予任何来自3Dlabs或任何第三方的知识产权许可。若OpenGL 2.0 API规范经ARB批准并采纳本文件全部或部分内容，3Dlabs将依据ARB章程向Silicon Graphics, Inc.授予免版税许可，但仅限于实现符合规范所需的3Dlabs知识产权。

本规范按“原样”提供，不作任何明示、默示、法定或其他形式的保证。3DLABS特此声明不承担任何适销性、不侵权、特定用途适用性或其他因任何提案、规范或样本而产生的保证责任。

美国政府限制权利声明

政府对本文件的使用、复制或披露受《联邦采购条例》(FAR)第52.227.19(c)(2)条或《国防联邦采购条例》(DFARS)第252.227-7013条“技术数据和计算机软件权利条款”第(c)(1)(ii)项及/或《联邦采购条例》(FAR)、国防部(DOD)或美国国家航空航天局(NASA)《联邦采购条例补充条款》(FAR Supplement)中类似或后续条款的限制。根据美国版权法保留未公开权利。

承包商/制造商为3Dlabs, Inc. Ltd.，地址：阿拉巴马州麦迪逊市麦迪逊大道9668号，邮编35758。

OpenGL 是 Silicon Graphics Inc. 的注册商标。

目录

1 简介	1
1.1 自 1.051 版以来的变更	1
1.2 概述	2
1.3 动机	2
1.4 设计考量	3
1.5 错误处理	3
1.6 排版规范	4
2 OpenGL着色概述	5
2.1 顶点处理器	5
2.2 片段处理器	6
3 基础知识	8
3.1 字符集	8
3.2 源字符串	8
3.3 预处理器	9
3.4 注释	13
3.5 标记	13
3.6 关键词	14
3.7 标识符	14
4 变量与类型	16
4.1 基本类型	16
4.1.1 空	17
4.1.2 布尔值	17
4.1.3 整数	17
4.1.4 浮点数	18
4.1.5 向量	19
4.1.6 矩阵	19
4.1.7 采样器	19
4.1.8 结构	20
4.1.9 数组	21
4.2 作用域	21
4.3 类型限定符	22

4.3.1	默认限定符	22
4.3.2	构造函数	23
4.3.3	积分常数表达式	23
4.3.4	属性	23
4.3.5	统一	24
4.3.6	变化	24
5	运算符与表达式	26
5.1	运算符	26
5.2	数组下标	27
5.3	函数调用	27
5.4	构造函数	27
5.4.1	转换与标量构造函数	27
5.4.2	向量与矩阵构造函数	28
5.4.3	结构构造函数	29
5.5	向量组件	29
5.6	矩阵分量	31
5.7	结构与场	31
5.8	作业	31
5.9	表达式	32
5.10	向量与矩阵运算	34
6	陈述与结构	36
6.1	函数定义	37
6.1.1	函数调用约定	38
6.2	选择	39
6.3	迭代	40
6.4	跳转	41
7	内置变量	42
7.1	顶点着色器特殊变量	42
7.2	片段着色器特殊变量	43
7.3	顶点着色器内置属性	44
7.4	内置常量	44
7.5	内置统一状态	45
7.6	变化变量	48
8	内置函数	50
8.1	角度与三角函数	51
8.2	指数函数	52
8.3	常用函数	52
8.4	几何函数	54

.....
.....
.....
.....

8.5 矩阵函数 55

8.6	向量关系函数	55
8.7	纹理查找函数	56
8.8	片段处理函数	58
8.9	噪声函数	60
9	着色语法规	62
10	问题.....	73
11	致谢.....	104

1 引言

注：本文件所指定语言的文档修订版本与语言版本分开追踪。更改文档修订版本不会改变语言版本。本文档规定OpenGL着色语言版本为1.10，文档修订号为59。要求VERSION必须为110，且#version需支持110版本。

1.1 自版本1.0以来的变更 51

- 新增问题101至105。基于这些问题的规范变更包括：要求数组参数指定大小，并在构造函数中添加限制。详见第4.2、5.4.2、6.1、6.1.1节。
- 新增与 ATI_draw_buffers 和 ARB_color_clamp_control 的交互机制，特别是输出变量 `gl_FragData[n]`。
- 3.3 新增 #version 和 #extension 用于声明版本和扩展。
- 7.5 为矩阵的逆矩阵和转置矩阵添加了内置状态。
- 8 新增内置函数 `refract`、`exp` 和 `log`。
- 新增以下说明与修正：
 - 2.1 从顶点处理器功能列表中移除“颜色钳位”项，该内容已过时。
 - 2.1 将“透视投影”改为更明确地指出射影变换和透视除法，二者应归入不同列表。
 - 3.3 保留以“GL_”开头的预处理器宏。
 - 3.6 新增保留字 `packed`、`this`、`interface`、`sampler2DRectShadow`。同时明确说明列表中列出的关键词和保留字即为全部。
 - 4.1.5 删除“整数向量可用于从纹理读取中获取多个整数”的说明。该内容已过时。
 - 4.3.5 明确结构体可作为常量，并说明 `const` 必须通过初始化赋值。
 - 5.8 明确解释 `*=`、`+=` 等运算符的实际含义，并说明 `?:` 不是左值。
 - 5.9 明确说明：与浮点数类似，整数也可进行标量与向量间的运算，且列表应完整罗列所有运算符与表达式。
 - 6.1 修正点积原型的示例。这些示例与原型列表存在不一致，而原型列表本身已长期保持正确。
 - 6.1 添加说明：“若在调用内置函数前于着色器中重新声明该函数（即函数原型可见），则链接器仅尝试在与其链接的着色器集中解析该调用。”
 - 7.2 删除过时文本：“实现将在使用相同源级表达式计算深度的着色器内部提供不变结果，但着色器与固定功能单元之间不保证不变性。”

- 7.4 修正内置常量名称列表：移除后缀并更新数值。
- 8.2 明确指数函数的定义域。
- 8.3 将`step()`的比较条件修改为`x < edge`（小于边界）而非`x <= edge`（小于等于边界）。
- 8.7 明确关于何时影子查找未定义的讨论。
- 8.9 进一步明确噪声的范围与频率限制。
- 语法：将`MOD_ASSIGN`改为保留字（以匹配规范文本）。
- 语法：修改为要求函数参数中指定数组大小。
- 修正若干拼写错误。

12 Overview

本文档描述了一种名为*OpenGL着色语言* (*glslang*) 的编程语言。图形硬件的最新趋势是在日益复杂的领域（如顶点处理和片段处理）用可编程性取代固定功能。

OpenGL着色语言的设计旨在让应用程序开发者能够表达在OpenGL管道可编程节点处发生的处理过程。

用此语言编写的独立可编译单元称为着色器。程序是由一组经过编译和链接的着色器组成的集合。本文档旨在全面规范该编程语言。用于操控程序和着色器并与之通信的OpenGL入口点，其定义独立于本语言规范之外。

OpenGL着色语言基于ANSI C，保留了该语言的多数特性，仅在与性能或实现便捷性冲突时予以调整。为使三维图形中常见操作更简洁，C语言新增了向量与矩阵类型（含硬件加速修饰符），并借鉴C++机制，如基于参数类型实现函数重载，以及支持在变量首次使用处声明而非块首声明。

13 Motivation

半导体技术已发展到这样的程度：每个顶点或每个片段所能完成的计算量，已超出传统OpenGL机制（通过设置状态来影响固定管道阶段的操作）所能描述的范围。

为充分释放硬件扩展能力，大量扩展功能应运而生，但其不幸的后果是削弱甚至丧失了应用程序的可移植性，这动摇了OpenGL的核心设计初衷。

驯服这种复杂性与扩展泛滥的自然之道，是允许用户编程阶段替代部分管道环节。近期扩展虽已实现此功能，但编程仍需使用汇编语言——这种直接映射硬件的表达方式缺乏前瞻性。主流程序员早已从汇编语言转向高级语言，以提升生产力、可移植性与易用性。这些目标同样适用于着色器编程。

本工作的目标是创建一种前瞻性的硬件无关高级语言：既易于使用又足够强大，能够经受时间考验并大幅减少扩展需求。但需注意，该语言必须在当前硬件世代或下一代硬件内实现快速运行。

14 设计考虑事项

我们将介绍的各类可编程处理器将替代OpenGL管道中的部分组件，作为起点，它们必须具备替代对象的所有功能。这仅仅是开端，来自RenderMan社区和新一代游戏的实例已为我们揭示了未来令人振奋的可能性。

为实现这一目标，着色语言应具备足够高的抽象层次，并针对我们所处理的问题领域提供相应的抽象层。在图形领域，这意味着向量与矩阵运算构成语言的基础部分。其功能涵盖从表达式中直接指定标量/向量/矩阵运算，到高效操作与分组向量及矩阵分量的方法。该语言包含丰富的内置函数集，这些函数对向量和标量的操作同样便捷。

我们很幸运能以C语言为基础进行构建，并以现有的RenderMan着色语言为学习范本。OpenGL与“实时”图形处理相关（区别于离线图形处理），因此C语言和RenderMan中任何妨碍高效编译或硬件实现的特性均已被舍弃，但这些改动在多数情况下不会造成明显影响。

OpenGL着色语言专为OpenGL环境设计，旨在为OpenGL固定功能的特定部分提供可编程替代方案。其设计使着色器内部能够轻松调用现有OpenGL状态参数。同时，该设计也支持在OpenGL处理管道的不同环节分别采用固定功能与可编程处理，操作简便。其设计初衷是使着色器生成的目标代码独立于其他OpenGL状态，从而无需重新编译或管理多个目标代码副本。

图形硬件在顶点和片段处理层级正不断提升并行能力。OpenGL着色语言的定义经过精心设计，以支持更高层次的并行处理。

最终目标是为OpenGL管道所有可编程部分采用统一的高级编程语言。虽然某些可编程处理器不支持特定类型和内置函数，但绝大多数语言特性在所有可编程处理器上保持一致。这极大简化了应用程序开发者掌握着色语言的过程，使其能更高效地解决OpenGL渲染难题。

15 编译器中的错误处理

编译器通常会接受格式错误的程序，因为无法检测所有此类程序。例如，完全准确地检测未初始化变量的使用是不可能的。本规范描述的格式正确程序才能确保可移植性。

编译器应努力检测格式错误程序并输出诊断信息，但并非

要求在所有情况下都这样做。编译器必须针对词法、语法或语义错误的着色器返回消息。

16 Typographic Conventions

本规范中斜体、粗体及字体选择主要用于提升可读性。代码片段采用等宽字体。嵌入文本中的标识符以斜体显示。 嵌入文本的关键词采用粗体。运算符以名称标注，其符号以粗体括号形式跟随。文本中用于阐释语法的片段采用粗体表示字面量，斜体表示非终结符。第9节“着色语言语法”中的正式语法采用全大写表示终结符，小写表示非终结符。

2 OPENGL着色语言概述

OpenGL着色语言实为两种密切相关的语言。这些语言用于为OpenGL处理管道中的可编程处理器创建着色器。这些可编程单元的精确定义留待其他规范说明。本文仅对其进行足够定义以提供语言定义的背景。

除非本文另有说明，否则语言特性均适用于所有语言，且通常会将这些语言视为单一语言。具体语言将以其目标处理器名称进行区分：顶点着色器或片段着色器。

2.1 顶点处理器

顶点处理器是一个可编程单元，用于处理输入的顶点值及其关联数据。该处理器旨在执行传统图形操作，例如：

- 顶点变换（模型视图与投影矩阵）
- 法线变换与归一化
- 纹理坐标生成
- 纹理坐标变换
- 光照
- 材质贴图应用

使用OpenGL着色语言编写的、旨在在此处理器上运行的程序称为顶点着色器。顶点着色器可用于指定对每个顶点及其关联数据执行的完全通用的操作序列。执行上述列表中部分计算的顶点着色器负责编写实现上述所有所需功能的代码。例如，无法仅使用现有固定功能实现顶点与法线变换，同时让顶点着色器执行特定光照功能。顶点着色器必须编写为同时执行这三项功能。

顶点处理器无法替代需要同时处理多个顶点信息或需要拓扑知识的图形操作，例如：

- 透视分割
- 视口映射
- 基元组合
- 截断锥与用户裁剪
- 背面剔除
- 双面光照选择
- 多模式处理
- 多边形偏移

- 深度范围

着色器使用的任何OpenGL状态都会被自动追踪并提供给着色器。这种自动状态追踪机制使应用程序能够使用现有的OpenGL状态命令进行状态管理，并将这些状态的当前值自动提供给顶点着色器使用。

顶点处理器每次处理一个顶点。其设计重点在于实现对单个顶点进行变换和光照所需的功能。顶点着色器必须计算坐标的齐次位置，还可计算颜色、纹理坐标及其他任意值，这些值将传递给片段处理器。顶点处理器的输出将进入后续处理阶段，其定义与OpenGL 1.4完全一致：基元组合、用户裁剪、视锥裁剪、透视投影、视口映射、多边形偏移、多边形模式、着色模式及剔除。该可编程单元不具备从帧缓冲区读取的能力，但支持纹理查找功能。细节级别（LOD）并非由顶点着色器实现计算，而可通过着色器进行指定。纹理贴图的OpenGL参数定义了过滤操作、边界处理及卷绕模式的行为。

2 片元处理器

片段处理器是一个可编程单元，用于处理片段值及其关联数据。该处理器旨在执行传统图形操作，例如：

- 插值值操作
- 纹理访问
- 纹理应用
- 雾效
- 颜色求和

使用OpenGL着色语言编写并旨在在此处理器上运行的程序称为**片段着色器**。片段着色器可用于指定对每个片段执行的完全通用的操作序列。执行上述列表中部分计算的片段着色器必须实现该列表中的全部所需功能。例如，无法仅使用现有固定功能计算雾效，同时让片段着色器执行专用的纹理访问与纹理应用操作。片段着色器必须编写为同时执行这三项功能。

片段处理器不会取代OpenGL像素处理管道后端执行的固定功能图形操作，例如：

- 着色模型
- 覆盖率
- 像素所有权测试
- 剪裁
- 点画
- 剪裁
- 深度测试
- 模板测试
- Alpha混合

- 逻辑运算
- 抖动
- 平面遮罩

若着色器使用相关OpenGL状态，系统将自动追踪该状态。片段着色器无法改变片段的x/y坐标。为支持片段处理层级的并行性，片段着色器需以表达单个片段所需计算的方式编写，且禁止访问相邻片段。片段着色器可自由读取单张纹理的多个值，或多张纹理的多个值。片段着色器计算出的值最终用于更新帧缓冲区内存或纹理内存，具体取决于当前OpenGL状态及触发片段生成的OpenGL命令。

纹理贴图的OpenGL参数持续定义过滤操作、边界处理及卷绕模式的行为。这些操作在访问纹理时被应用。片段着色器可自由处理读取到的纹理像素。它既能从纹理中读取多个值并执行自定义过滤操作，也可利用纹理执行查找表运算。这两种情况下，纹理参数均应设置为在访问操作中应用最近邻过滤。

对于每个片段，片段着色器可计算颜色和/或深度，或直接丢弃该片段。

片段着色器的结果随后被传递至后续处理环节。OpenGL 管道的其余部分仍遵循 OpenGL 1.4 规范定义。片段需依次经过覆盖应用、像素所有权测试、剪裁测试、透明度测试、模板测试、深度测试、混合、抖动、逻辑运算及遮罩处理，最终写入帧缓冲区。将固定功能模块保留在处理管道末端的首要原因在于：固定功能模块硬件实现成本低且易于部署。而将这些功能可编程化则更为复杂，因为读取/修改/写入操作可能引发严重的指令调度问题和管道停滞。大多数固定功能操作均可禁用，若需替代操作，可在片段着色器中执行。

3 基础知识

3.1 字符集

OpenGL着色语言使用的源字符集是ASCII字符集的子集，包含以下字符：

字母 a-z、A-Z 以及下划线(_)。数字 0-9。

符号点号(.)、加号(+)、短横线(-)、斜杠(/)、星号(*)、百分号(%)、尖括号(<和>)，方括号([和])、圆括号((和))、大括号({和})、插入符(^)、竖线(|)、连字符(&)、波浪号(~)、等号(=)、感叹号(!)、冒号(:)、分号(;)、逗号(,) 和问号(?)。

预处理器使用的井号(#)。

空白字符：空格字符、水平制表符、垂直制表符、换页符、回车符和换行符。

行符用于编译器诊断信息和预处理器。行符以回车或换行符结束。若同时使用两者，则视为单一行符。本文档后续内容中，此类组合统称为换行符。

通常情况下，该语言对字符集的使用区分大小写。

本语言未定义字符或字符串数据类型，因此未包含引号字符。

源字符串的结尾由长度而非字符表示，不存在文件结束字符。

3.2 源字符串 gs

单个着色器的源代码是由字符集中的字符组成的字符串数组。单个着色器通过连接这些字符串生成。每个字符串可包含多行内容，以换行符分隔。字符串中无需包含换行符；单行可由多个字符串组成。实现将字符串连接成单个着色器时，不会插入换行符或其他字符。相同语言（顶点或片段）的多个着色器可链接成单个程序。

着色器编译返回的诊断信息必须同时标识字符串内的行号以及该信息所对应的源字符串。源字符串按顺序编号，首个字符串为字符串0。行号比已处理的换行符数量多1。

33 Preprocessor

存在一个在编译前处理源字符串的预处理器。完整的预处理指令列表如下。

```
#  
#define #undef  
  
#if #ifdef  
#ifndef #else  
#elif #endif  
  
#error #pragma  
  
#extension #version  
  
#line
```

以下运算符也可用

定义

每个井号 (#) 在其所在行中只能由空格或水平制表符前置。它也可以由空格和水平制表符后置，位于指令之前。每个指令以换行符结束。预处理不会改变源字符串中换行符的数量或相对位置。

单独出现在行首的井号 (#) 将被忽略。任何未在上述列表中列出的指令都会触发诊断信息，并导致实现将着色器视为格式错误。

`#define` 与 `#undef` 功能遵循 C++ 预处理器的标准定义，适用于带参数与不带参数的宏定义。

以下预定义宏可用

```
__LINE__  
__FILE__  
__VERSION__
```

`__LINE__` 将替换为十进制整数常量，该常量比当前源字符串中前面的换行符数量多 1。

`__FILE__` 将替换为一个十进制整数常量，该常量表示当前正在处理的源字符串编号。

`_VERSION` 将替换为十进制整数，反映 OpenGL 着色语言的版本号。本文档所述着色语言版本中，`VERSION` 将替换为十进制整数 110。

所有包含两个连续下划线（`_`）的宏名均预留为未来预定义宏名。所有以“`GL_`”（即“`GL`”后跟单个下划线）为前缀的宏名同样预留。

`#if`、`#ifdef`、`#ifndef`、`#else`、`#elif` 和 `#endif` 的定义遵循 C++ 预处理器的标准规范。`#if` 和 `#elif` 后的表达式仅限于操作字面整数常量，以及由 **定义** 运算符消耗的标识符。字符常量不被支持。可用的运算符包括

运算符优先级	运算符类别	运算符	结合性
1 (最高)	括号分组	()	NA
2	一元	定义 + - ~ !	从右到左
3	乘法	* / %	从左到右
4	加法运算	+ -	从左到右
5	位移运算	<< >>	从左到右
6	关系运算符	< > <= >=	左至右
7	等号	== !=	从左到右
8	位与	&	从左到右
9	位运算异或	^	从左到右
10	位运算包含或		从左到右
11	逻辑与	&&	从左到右
12	逻辑或		从左到右

定义 的运算符可通过以下任一方式使用：

```
定义的标识符
defined ( 标识符 )
```

不存在基于数字符号的运算符（无 #、#@、## 等），也不存在 `sizeof` 运算符。

在预处理器中对整数字面量应用运算符的语义符合 C++ 预处理器的标准规范，而非 OpenGL 着色语言的规范。

预处理器表达式的求值行为遵循宿主处理器的特性，而非着色器目标处理器的特性。

`#error` 将导致实现将诊断信息写入着色器的信息日志（访问着色器信息日志的方法请参阅外部文档中的 API）。该信息将包含 `#error` 指令后跟随的令牌，直至首个换行符。此时实现必须将该着色器视为格式错误。

#pragma 指令允许实现依赖的编译器控制。**#pragma** 之后的令牌不受预处理器宏展开影响。若实现无法识别 **#pragma** 后的令牌，则会忽略该指令。下列 **pragma** 作为语言规范定义：

```
#pragma STDGL
```

STDGL 指令用于为该语言未来版本预留指令空间。任何实现均不得使用以 **STDGL** 为首字符的指令。

```
#pragma optimize(on) #pragma  
optimize(off)
```

可用于关闭优化功能，以辅助着色器的开发和调试。该指令仅可在函数定义外部使用。默认情况下，所有着色器均启用优化功能。调试指令

```
#pragma debug(on) #pragma  
debug(off)
```

可用于启用带调试信息的着色器编译与注释，以便配合调试器使用。该指令仅可在函数定义外部使用。默认情况下调试功能处于关闭状态。

着色器应声明其编写的语言版本。着色器所采用的语言版本通过

```
#version number
```

其中数字必须为110（遵循与上述VERSION相同的约定），表示本规范所采用的语言版本。此时指令将被接受且不会产生错误或警告。任何小于110的数字都会导致错误生成。任何大于编译器支持的最新语言版本的数字同样会导致错误生成。语言版本110不强制要求着色器包含此指令，未包含 **#version** 指令的着色器将被视为针对版本110编写。后续版本的编译器在着色器中遇到“**#version 110**”指令时，必须确保要么支持该版本，要么明确报错声明不支持。

#version 指令必须出现在着色器文件中所有内容之前，仅允许在注释和空白符之后出现。

默认情况下，本语言的编译器必须对不符合本规范的着色器发出编译时语法、语法和语义错误。任何扩展行为都必须先启用。用于控制编译器对扩展行为的指令需通过 **#extension** 指令声明

```
#extension 扩展名 : 行为  
#extension all : 行为
```

其中 `extension_name` 为扩展名称。扩展名称未在本规范中记录。`"all"` 表示该行为适用于编译器支持的所有扩展。行为可选取以下任一形式：

行为	效果
<code>require</code>	<p>按扩展 <code>extension_name</code> 的规定进行行为。</p> <p>如果扩展 <code>extension_name</code> 不被支持，或者指定了 <code>all</code>，则在 <code>#extension</code> 上报错。</p>
<code>启用</code>	<p>按 <code>extension_name</code> 扩展指定的行为处理。</p> <p>如果扩展 <code>extension_name</code> 不被支持，则在 <code>#extension</code> 上发出警告。如果指定了 <code>all</code>，则在 <code>#extension</code> 上发出错误。</p>
<code>warn</code>	<p>按扩展 <code>extension_name</code> 的规定行为，但对该扩展在其他已启用或必需扩展中未被支持的任何可检测使用发出警告。</p> <p>如果指定了 <code>all</code>，则对所有可检测到的扩展使用发出警告。如果扩展 <code>extension_name</code> 不被支持，则在 <code>#extension</code> 上发出警告。</p>
<code>禁用</code>	<p>行为（包括发出错误和警告）如同扩展 <code>extension_name</code> 不属于语言定义的一部分。</p> <p>若指定了 <code>all</code>，则行为必须恢复为所编译语言的非扩展核心版本的行为。</p> <p>若扩展 <code>extension_name</code> 不被支持，则对 <code>#extension</code> 发出警告。</p>

扩展指令是一种简单的基础机制，用于设置每个扩展的行为。它不定义策略（如哪些组合是合适的），这些策略必须在其他地方定义。指令的顺序对设置每个扩展的行为很重要：后出现的指令会覆盖先出现的指令。`"all"` 变体设置所有扩展的行为，覆盖之前发布的所有扩展指令，但仅限于警告和禁用行为。

编译器的初始状态如同指令

```
#extension all : disable
```

已生效，要求编译器必须严格遵循本规范进行所有错误与警告报告，忽略任何扩展功能。

每个扩展功能可定义其允许的作用域粒度。若未特别说明，则粒度为着色器（即单个编译单元），且扩展指令必须出现在任何非预处理符之前。必要时，链接器可强制执行大于单个编译单元的粒度，此时每个涉及的着色器都必须包含必要的扩展指令。

宏展开不作用于包含 `#extension` 和 `#version` 指令的行。

宏替换后的`#line`指令必须符合以下两种形式之一：

```
#line 行号  
#line line source-string-number
```

其中行号和源字符串编号均为常量整数表达式。处理此指令（包括其换行符）后，实现将表现为正在行号`行+1`处编译源字符串编号为`source-string-number`的代码。后续源字符串将按顺序编号，直至另一条`#line`指令覆盖该编号。

34 注释 s

注释由 `/*` 和 `*/` 或 `//` 及换行符分隔。注释起始分隔符 (`/*` 或 `//`) 在注释内部不被识别为分隔符，因此注释不能嵌套。若注释完全位于单行内，则语法上视为单个空格。

35 Tokens

该语言由令牌序列构成。令牌可为

token:
关键字 标识符 整数常量
浮点常量运算符

36**s**

以下是该语言中的关键字，除本文档定义的用途外不得用于其他目的：

```
attribute const uniform varying break continue do for while
if else
in out inout
浮点数 整数 无效 逻辑值 真 假 丢弃 返回
mat2 mat3 mat4
vec2 vec3 vec4   ivec2 ivec3 ivec4    bvec2 bvec3 bvec4
sampler1D sampler2D sampler3D samplerCube sampler1DShadow sampler2DShadow struct
```

以下为预留关键词，禁止使用，否则将导致错误：

asm						
类 联合体		enum	typedef	template this packed goto switch		
默认						
内联	noinline	volatile	public	静态外部	外部接口长短双精度	半精度 固定精度 无符号
输入	输出					
hvec2	hvec3	hvec4dvec2	dvec3dvec4	fvec2	fvec3	fvec4
sampler2D矩形		sampler3D矩形sampler2D矩形阴影				
sizeof 转换命名空间 using						

此外，所有包含两个连续下划线（_）的标识符均被保留为可能的未来关键字。

37**标识符**

标识符用于变量名、函数名、结构体名以及字段选择器（字段选择器用于选择向量和矩阵的分量，类似于结构体字段，详见第5.5节“向量分量”和第5.6节“矩阵分量”）。标识符采用以下形式：

标识符**非数字****标识符 非数字 标识符 数字****非数字:** 以下任一形式**_a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z****数字:** 以下之一**0 1 2 3 4 5 6 7 8 9**

以“gl_”开头的标识符是OpenGL专用的保留标识符，不得在着色器中作为变量或函数进行声明。

4 变量与类型

所有变量和函数必须在使用前声明。变量名和函数名均属于标识符。

不存在默认类型。所有变量和函数声明必须指定类型，并可选地添加修饰符。声明变量时需先指定类型，后跟一个或多个以逗号分隔的名称。多数情况下，可通过赋值运算符(=)在声明时初始化变量。本文件末尾的语法部分提供了变量声明语法的完整参考。

用户定义类型可通过 `struct` 将现有类型的列表聚合为单一名称。OpenGL着色语言具有类型安全性，类型之间不存在隐式转换。

4.1 基本类型

OpenGL着色语言支持以下基本数据类型。

<code>void</code>	用于不返回值的函数
<code>bool</code>	条件类型，取值为 <code>true</code> 或 <code>false</code>
<code>int</code>	带符号整数
<code>float</code>	单精度浮点标量
<code>vec2</code>	一个两分量的浮点向量
<code>vec3</code>	三维浮点向量
<code>vec4</code>	一个四分量浮点数向量
<code>bvec2</code>	一个两分量的布尔向量
<code>bvec3</code>	三元布尔向量
<code>bvec4</code>	一个四组分布布尔向量
<code>ivec2</code>	一个两分量的整数向量
<code>ivec3</code>	三元整数向量
<code>ivec4</code>	一个四分量整数向量
<code>mat2</code>	一个 2×2 浮点矩阵
<code>mat3</code>	一个 3×3 浮点矩阵
<code>mat4</code>	一个 4×4 浮点矩阵
<code>sampler1D</code>	用于访问1D纹理的句柄
<code>sampler2D</code>	用于访问二维纹理的句柄
<code>sampler3D</code>	用于访问三维纹理的句柄
<code>samplerCube</code>	用于访问立方体贴图纹理的句柄

sampler1DShadow	用于访问带比较功能的1D深度纹理的句柄
sampler2DShadow	用于访问带比较功能的二维深度纹理的句柄

此外，着色器可通过数组和结构体聚合这些类型以构建更复杂的类型。不存在指针类型。

4.1.1 Void

不返回值的函数必须声明为 **void**。函数没有默认返回类型。

4.1.2 布尔值

为便于表达代码的条件执行，本语言支持**bool**类型。硬件无需直接支持此类变量，它是一种纯粹的布尔类型，仅能取两个值之一，分别表示真或假。**true**和**false**两个关键字可作为布尔常量使用。布尔变量的声明及可选初始化方式如下例所示：

```
bool success; // 声明布尔变量"success"bool done = false; // 声明并初始化"done"
```

赋值运算符右侧(=)可为任意类型为**bool**的表达式。用于条件跳转的表达式(**if**, **for**, **?:**, **while**, **do-while**)必须评估为**bool**类型。

4.1.3 整数

整数主要作为编程辅助手段被支持。在硬件层面，真正的整数能有效实现循环和数组索引，并辅助纹理单元引用。但语言中的整数并不要求映射到硬件的整数类型，底层硬件也不必全面支持广泛的整数运算。基于其限定用途，顶点语言和片段语言中的整数精度均限制为16位加符号位。OpenGL着色语言实现可将整数转换为浮点数进行运算，亦允许使用超过16位的精度处理整数。因此整数不存在可移植的溢出处理机制，超过16位精度的着色器可能存在移植性问题。

整型变量声明时可选地使用整型表达式初始化，示例如下：

```
int i, j = 42;
```

字面量整数常量可采用十进制（基数10）、八进制（基数8）或十六进制（基数16）表示，格式如下：

整数常量: 十进制常量 | 八进制常量

 十六进制常量

十进制常量: 非零数字

 十进制常量 位数

```

八进制常量:
  0
八进制常量八进制数字

十六进制常量:
  0x 十六进制数字
  0X 十六进制数字
  十六进制常量 十六进制数字

数字:
  0
非零位

非零位数: 取值为
  1 2 3 4 5 6 7 8 9

八进制数字: 以下之一
  0 1 2 3 4 5 6 7 8 9

十六进制数字: 以下之一
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

```

整数常量中的数字之间不允许有空格，包括常量前导0之后或前导0x/0X之后。前导单一减号(-)被解释为算术单一取反运算符，而非常量的一部分。不存在字母后缀。

4.1.4 浮点数

浮点数可用于各类标量计算。浮点变量定义示例如下：

```
float a, b = 1.5;
```

作为处理单元之一的输入值，浮点变量应符合IEEE单精度浮点数的精度与动态范围定义。内部处理的精度无需完全符合IEEE浮点数规范，但必须满足OpenGL 1.4规范中确立的精度准则。同样地，对除以零等条件的处理可能导致未指定的结果，但此类情况绝不应导致处理中断或终止。

浮点常量定义如下：

```

浮点常量:
  小数常量 指数部分 可选 有效数字序列 指数部分

小数常量:
  位序列. 位序列 位序列.
  . 位序列

```

指数部分:

e 符号 可选 数字序列

E 符号 可选 数字序列

符号: 取值为

+ -

数字序列: 数字

数字序列/数字

若存在指数部分，则无需小数点(.)。

4.1.5 向量

OpenGL着色语言包含通用2、3、4分量向量的数据类型，支持浮点数、整数或布尔值。浮点向量变量可存储计算机图形学中多种实用数据：颜色、法线、位置、纹理坐标、纹理查找结果等。布尔向量可用于数值向量的分量比较。在着色语言中定义向量，可将向量运算直接映射至支持向量处理的图形硬件。通常，应用程序通过向量计算而非标量计算，能更充分地利用图形硬件的并行处理能力。向量声明示例如下：

```
vec2 texcoord1, texcoord2; vec3 position;
vec4 myRGBA;
ivec2 textureLookup; bvec3
lessThan;
```

向量初始化可通过构造函数实现，相关内容将在后续章节详述。

4.1.6 矩阵

矩阵是计算机图形学中另一种有用的数据类型，OpenGL着色语言定义了对 2×2 、 3×3 和 4×4 浮点数矩阵的支持。矩阵采用列优先顺序进行读写。矩阵声明示例：

```
mat2 mat2D; mat3
optMatrix;
mat4 view, projection;
```

矩阵值的初始化通过构造函数实现（详见第5.4节“构造函数”）。

4.1.7 采样器

采样器类型（如sampler2D）实质上是纹理的透明句柄。它们配合内置纹理函数（详见第8.7节“纹理查找函数”）指定需访问的纹理。采样器仅可声明为函数参数或统一变量（参见第4.3.5节“统一变量”）。

采样器不得作为表达式操作数，亦不可被赋值。作为统一变量时，需通过OpenGL API初始化；作为函数参数时，仅可向采样器传递采样器。

匹配类型。这使得在着色器运行前能够检查着色器纹理访问与OpenGL纹理状态之间的一致性。

4.1.8 结构体

用户可通过聚合其他已定义类型创建自定义结构体，使用

`struct` 关键字将其他已定义类型聚合为结构体。例如：

```
struct 光照 {
    float 强度;vec3 位置;
} lightVar;
```

在此示例中，`light`成为新类型的名称，`lightVar`成为类型为`light`的变量。声明新类型变量时需使用其名称（无需`struct`关键字）。

```
light lightVar2;
```

更正式的结构体声明格式如下。但完整的语法规则详见第9节“着色语言语法”。

结构体定义：

限定符 可选 结构体名称 可选 { 成员列表 } 声明符 可选 ;

成员列表：

成员声明；

成员声明 成员列表；

成员声明；

基本类型声明符; 嵌套结构体定义

嵌套结构体定义：

struct 名称 可选 { 成员列表 } 声明符;

其中名称将成为用户定义的类型，可用于声明此新类型的变量。该名称与其他变量和类型共享相同命名空间，遵循相同的作用域规则。可选限定符仅适用于声明符，不属于名称所定义的类型的一部分。

结构体必须至少包含一个成员声明。成员声明符不包含任何限定符，也不包含位域。成员类型必须是已定义的（不存在前向引用），或通过嵌套另一个结构体定义在原地定义。成员声明不能包含初始化表达式。成员声明符可包含数组。此类数组必须指定大小，且大小必须为大于零的整数常量表达式（参见第4.3.3节“整数常量表达式”）。结构的每层都为成员声明符中的名称提供独立命名空间；此类名称仅需在该命名空间内唯一。

不支持匿名结构体，因此嵌套结构体必须有声明符。赋予嵌套结构体的名称作用域与所嵌套结构体相同。

结构体可在声明时使用构造函数初始化，详见第5.4.3节“结构体构造函数”。

4.1.9 数组

同类型的变量可通过声明名称后跟方括号 ([]) 并可选地包含大小来聚合为数组。当数组声明中指定大小时，该大小必须是大于零的整常量表达式（参见第4.3.3节“整常量表达式”）。若数组被非整常量表达式索引，或作为函数参数传递，则必须在使用前声明其大小。允许先声明无尺寸的数组，随后重新声明同名数组并指定尺寸。以下情况均属非法：在声明数组时指定大小，随后（在同一着色器中）使用大于或等于声明大小的整常量表达式对该数组进行索引；使用负常量表达式对数组进行索引。函数声明中作为形式参数声明的数组必须指定大小。

使用大于或等于数组大小或小于0的非常量表达式对数组进行索引将导致未定义行为。仅允许声明一维数组。所有基本类型和结构体均可构成数组。示例如下：

```
float frequencies[3];
uniform vec4 lightPosition[4]; light lights[];
const int numLights = 2; light
lights[numLights];
```

着色器内部不支持在声明时初始化数组。

42 S 应对

变量的作用域由其声明位置决定。若在所有函数定义之外声明，则具有全局作用域，该作用域从声明处开始，持续至声明所在着色器的结尾。若在while测试或for语句中声明，则作用域限定至后续子语句的结尾。若作为复合语句内的语句声明，则作用域至该复合语句结束。若作为函数定义中的参数声明，则作用域至该函数定义结束。函数体具有嵌套于函数定义内的作用域。if语句的表达式不允许声明新变量，因此不会形成新作用域。

声明为空数组的变量可重新声明为相同基类型的数组。否则，在同一编译单元内，相同名称的变量不能在同一作用域内重新声明。

然而，嵌套作用域可以覆盖外部作用域对特定变量名的声明。嵌套作用域中的声明与被覆盖名称关联的存储空间相互独立。

同一作用域内的所有变量共享相同命名空间。函数名始终可通过上下文识别为函数名，且拥有独立命名空间。

共享全局变量是指在同一语言（顶点或片段着色器）下，由独立编译单元（着色器）声明的同名全局变量，这些单元经链接后构成单一程序。共享全局变量共享相同命名空间，必须声明为相同类型，并将共享同一存储空间。共享全局数组必须具有相同基类型和相同大小。标量必须拥有完全相同的类型名称和类型定义。结构体必须具有相同名称、类型序列

名称、类型定义和字段名称被视为相同类型。此规则对嵌套或嵌入类型递归适用。共享全局的初始化器必须具有相同值，否则将导致链接错误。

4.3 类型修饰符

变量声明可在类型前指定一个或多个限定符，其总结如下：

<无限定符：默认>	本地读写内存，或函数的输入参数
const	编译时常量，或函数的只读参数
attribute	顶点着色器与OpenGL之间的链接，用于顶点级数据
uniform	值在处理的基本元中保持不变，统一变量构成着色器、OpenGL和应用程序之间的关联
变量	顶点着色器与片段着色器之间用于插值数据的关联
in	用于传递给函数的函数参数
输出	用于从函数返回但未初始化以供传入使用的函数参数
inout	对于函数中传入和传出的函数参数

全局变量仅可使用 **const**、**attribute**、**uniform** 或 **varying** 修饰符，且只能指定其中一种。

局部变量仅可使用 **const** 修饰符。

函数参数仅可使用 **in**、**out**、**inout** 或 **const** 修饰符。参数修饰符的详细说明参见第 6.1.1 节“函数调用约定”。

函数返回类型和结构体字段不使用限定符。

用于着色器不同运行阶段间数据传递（如片段间或顶点间通信）的数据类型不存在。此设计可防止同一着色器在多个顶点或片段上并行执行。

未带限定符或仅带**const**限定符的全局变量声明可包含初始化表达式，此时该变量将在**main()**函数首行执行前完成初始化。此类初始化表达式必须为常量类型。未在声明中初始化且应用程序也未初始化的无限定符全局变量，OpenGL不会进行初始化，该变量将以未定义值进入**main()**函数。

4.3.1 默认限定符

若全局变量未声明限定符，则该变量与应用程序或其他处理器上运行的着色器之间不存在链接关系。对于全局或局部未限定变量，其声明将为目标处理器分配关联内存，该变量可读写访问此分配内存。

4.3.2 常量

命名编译时常量可通过 `const` 修饰符声明。任何被标记为常量的变量在该着色器中均为只读变量。相较于硬编码数值常量，声明常量变量能使着色器更具描述性。`const` 修饰符可用于任何基本数据类型。在声明范围外对 `const` 变量进行写入操作将导致错误，因此必须在声明时初始化。例如：

```
const vec3 zAxis = vec3(0.0, 0.0, 1.0);
```

结构体字段不可添加 `const` 修饰。结构体变量可声明为 `const`，并通过结构体构造函数初始化。

`const` 声明的初始化表达式必须由字面量、其他 `const` 变量（不包括函数调用参数）或这些元素的组合构成。

构造函数可在该表达式中使用，但函数调用不可用。

4.3.3 整数常量表达式

一个整数常量表达式可以是以下形式之一：

- 字面整数值
- 全局或局部标记为 `const` 的标量整型变量（不包括标记为 `const` 的函数参数）
- 其操作数为整数常量表达式的表达式，包括构造函数，但不包括函数调用。

4.3.4 属性

属性限定符用于声明从 OpenGL 传递至顶点着色器的变量，这些变量以每个顶点为单位进行传递。在顶点着色器之外的任何着色器类型中声明属性变量均视为错误。

对于顶点着色器而言，属性变量均为只读属性。属性变量的值可通过 OpenGL 顶点 API 或作为顶点数组的一部分传递至顶点着色器。它们向顶点着色器传递顶点属性，且预期在每次顶点着色器运行时发生变化。属性修饰符仅可与浮点型（`float`）、向量型（`vec2/vec3/vec4`）、矩阵型（`mat2/ma3t4`）数据类型配合使用。属性变量不可声明为数组或结构体。

声明示例：

```
attribute vec4 position;attribute vec3
normal;attribute vec2 texCoord;
```

所有标准 OpenGL 顶点属性均内置变量名，以便用户程序与 OpenGL 顶点函数轻松集成。内置属性名称列表详见第 7 节“内置变量”。

图形硬件预计会为传递顶点属性保留少量固定位置。因此，OpenGL 着色语言将每个非矩阵属性变量定义为最多可容纳四个浮点值（即 `vec4` 类型）。可使用的属性变量数量存在实现依赖性限制，若超出该限制将导致链接错误。（未使用的声明属性变量不计入此限制。）浮点属性变量占用的限制空间与 `vec4` 属性变量相同。

可使用的属性变量数量存在实现依赖的限制，若超出该限制将导致链接错误。（未使用的声明属性变量不计入此限制。）浮点属性与vec4属性占用相同数量的限制空间，因此应用程序可考虑将四个无关的浮点属性打包为vec4，以更充分地利用底层硬件能力。矩阵属性变量占用空间的计算规则如下：- mat4 占用相当于 4 个 vec4 属性变量位置- mat3 占用相当于 3 个属性变量位置- mat2 占用 2 个属性变量位置矩阵对空间的具体占用方式通过 API 和语言机制被实现层隐藏。

属性变量必须具有全局作用域，且必须在函数体外部声明，并在首次使用前完成声明。

4.3.5 统一变量

`uniform`修饰符用于声明全局变量，其值在整个处理中的原始图形中保持一致。**所有uniform变量均为只读**，其初始化可由应用程序通过API命令直接完成，或由OpenGL间接完成。

声明示例如下：

```
uniform vec4 lightPosition;
```

`uniform`修饰符可用于任何基本数据类型，或声明结构体类型变量及上述类型的数组。

每种着色器可使用的统一变量存储空间存在实现相关的限制，若超过该限制将导致编译时或链接时错误。声明但未使用的统一变量不计入此限制。着色器内部使用的用户定义统一变量数量与内置统一变量数量将相加计算，以此判定可用统一存储空间是否超限。

若多个着色器被链接在一起，它们将共享单一的全局统一变量命名空间。因此，在链接到单个可执行文件的所有着色器中，同名统一变量类型必须保持一致。

4.3.6 变量

变量变量（Varying variables）在顶点着色器、片段着色器及其间固定功能模块之间提供接口。顶点着色器将为每个顶点计算值（如颜色、纹理坐标等），并将其写入声明时带有`变量`修饰符的变量。顶点着色器也可读取`变量`变量，获取其写入的相同值。若在写入前读取顶点着色器中的变量变量，将返回未定义值。

根据定义，变量变量按顶点设置，并在渲染的基元上以透视校正的方式进行插值。若采用单采样，插值值对应于片段中心。若采用多采样，插值值可在像素内的任意位置，包括片段中心或其中一个片段采样点。

片段着色器可读取变化变量，其读取值即为根据原始图形内片段位置计算的插值结果。片段着色器无法向变化变量写入数据。

顶点着色器和片段着色器中声明的同名变量类型必须一致，否则链接命令将失败。顶点着色器仅可写入片段着色器中使用的（即读取的）变量；在顶点着色器中声明冗余变量是允许的。

变量声明示例如下：

```
varying vec3 normal;
```

varying修饰符仅可用于以下数据类型：**float**、**vec2**、**vec3**、**vec4**、**mat2**、**mat3**、**mat4** 数据类型或其数组。结构体不能作为变量使用。

若未激活顶点着色器，OpenGL的固定功能管道将计算内置变量值供片段着色器使用。同样地，若未激活片段着色器，顶点着色器则负责计算并写入OpenGL固定功能片段管道所需的变量值。

变量必须具有全局作用域，且必须在函数体外部声明，并在首次使用前完成声明。

5 运算符与表达式

5.1 运算符

OpenGL着色语言包含以下运算符。标记为保留的运算符非法。

运算符优先级	运算符类别	运算符	结合律
1 (最高)	括号分组	()	NA
2	数组下标 函数调用与构造函数 结构体字段选择器、交换器后缀递增与递减	[] () . . ++ -	左至右
3	前缀递增与递减一元运算符 (波浪号保留)	++ - + - ~ !	从右到左
4	乘法运算符 (模运算符保留)	* / %	左至右
5	加法	+ -	从左到右
6	位移 (保留)	<< >>	从左到右
7	关系运算符	< > <= >=	左至右
8	等号	== !=	从左到右
9	位与 (保留)	&	从左到右
10	位异或 (保留)	^	从左到右
11	位运算包含或 (保留)		从左到右
12	逻辑与	&&	从左到右
13	逻辑异或	^^	从左到右
14	逻辑或		从左到右
15	选择	? :	从右到左
16	赋值 算术赋值 (模运算、移位运算和位运算保留)	= += -= *= /= %= <<= >>= &= ^= =	从右到左
17 (最低)	序列	,	左至右

没有地址运算符，也没有解引用运算符。没有类型转换运算符，取而代之的是构造函数。

52 数组下标操作

数组元素通过数组下标运算符 (`[]`) 访问。这是唯一作用于数组的运算符。访问数组元素的示例如下：

```
diffuseColor += lightIntensity[3] * NdotL;
```

数组索引从零开始。访问数组元素时需使用类型为整数的表达式。

若着色器使用小于0或大于/等于数组声明大小的下标访问数组，则行为未定义。

53 函数调用

若函数返回值，则对该函数的调用可作为表达式使用，其类型即为声明或定义该函数时使用的类型。

函数定义和调用约定详见第6.1节“函数定义”。

54 构造函数

构造函数采用函数调用语法，其中函数名是基本类型关键字或结构体名称，用于生成所需类型的值以供初始化器或表达式使用。（详见第9节“着色语言语法”）参数用于初始化构造值。构造函数可用于请求数据类型转换，实现标量类型间的转换，或通过组合小类型构建大类型，亦可将大类型转换为小类型。

构造函数原型没有固定列表。构造函数并非内置函数。从语法上讲，所有词法正确的参数列表均有效。从语义上讲，参数数量必须足够且类型正确才能完成初始化。若构造函数包含过多参数以致无法全部使用，则视为错误。具体规则如下。下文列出的原型仅为示例子集。

5.4.1 转换与标量构造函数

标量类型间的转换遵循下列原型示例：

```
int(bool)           // 将布尔值转换为整型 int(float) // 将浮点值转换为整型 float(bool) // 将布尔  
值转换为浮点型 float(int) // 将整型值转换为浮点型 bool(float) // 将浮点值转换为布尔型 bool(int)  //  
将整数值转换为布尔值
```

当使用构造函数将浮点数转换为整数时，浮点数的小数部分将被舍弃。

当构造函数用于将 `int` 或 `float` 转换为 `bool` 时，`0` 和 `0.0` 将转换为 `false`，非零值将转换为 `true`。当构造函数用于将 `bool` 转换为 `int` 或 `float` 时，`false` 将转换为 `0` 或 `0.0`，`true` 将转换为 `1` 或 `1.0`。

恒等构造函数（如 `float(float)`）同样合法，但实用性有限。

带非标量参数的标量构造函数可用于提取非标量对象的首个元素。例如构造函数 `float(vec3)` 将选取 `vec3` 参数的首个分量。

5.4.2 向量与矩阵构造函数

构造函数可用于从一组标量、向量或矩阵创建向量或矩阵，其中包含缩短向量的功能。

若向量构造函数仅有一个标量参数，则该参数值将用于初始化构造向量的所有分量。若矩阵构造函数仅有一个标量参数，则该参数值将用于初始化矩阵对角线上的所有分量，其余分量初始化为 `0.0`。若存在非标量参数和/或多个标量参数，则按从左至右顺序依次赋值给构造值的各分量。此时参数提供的分量数量必须足以初始化构造值的所有分量。若构造函数最后使用的参数中提供的元素数量超过构造值所需，则仅使用该参数最左侧的元素，其余元素将被忽略。在最后使用的参数之后提供额外参数将导致错误。矩阵将按列优先顺序构造。禁止通过其他矩阵构造矩阵，此功能保留用于未来扩展。

若构造函数参数的基本类型（`bool`、`int` 或 `float`）与被构造对象的基本类型不匹配，则采用标量构造规则（如上所述）转换参数。

以下为若干常用向量构造函数示例：

```
vec3(float)           // 使用浮点数初始化vec3的每个分量vec4(ivec4) // 从ivec4创建vec4，进行分量级转换

vec2(float, float)    // 用两个浮点数初始化vec2 ivec3(int, int, int) // 用3个整数初始化ivec3类型
变量bvec4(int, int, float, float) // 通过4个布尔值转换初始化

vec2(vec3) // 舍弃vec3的第三个分量vec3(vec4) // 舍弃vec4的第四个分量

vec3(vec2, float) // vec3.x = vec2.x, vec3.y = vec2.y, vec3.z = floatvec3(float, vec2) // vec3.x = float, vec3.y =
vec2.x, vec3.z = vec2.yvec4(vec3, float)
vec4(float, vec3) vec4(vec2,
vec2)
```

这些运算的示例如下：

```
vec4color = vec4(0.0, 1.0, 0.0, 1.0);
vec4 rgba = vec4(1.0);      // 将每个分量设为1.0
```

```
vec3 rgb = vec3(color); // 舍弃第四个分量
```

若需初始化矩阵对角线元素，同时将其他所有元素设为零：

```
mat2(float)
mat3(float)
mat4(float)
```

通过指定向量初始化矩阵，或分别使用4个、9个或16个浮点数初始化mat2、mat3和mat4。浮点数按列优先顺序分配给元素。

```
mat2(vec2, vec2); mat3(vec3, vec3, vec3);
mat4(vec4, vec4, vec4, vec4);

mat2(float, float, float, float);

mat3(float, float, float, float, float, float,
      float, float, float);

mat4(float, float, float, float, float, float,
      float, float, float, float, float, float,
      float, float, float);
```

存在多种其他构造方式，只要组件数量足以初始化矩阵即可。但通过其他矩阵构造矩阵的功能目前保留用于未来扩展。

5.4.3 结构体构造函数

定义结构体并为其类型命名后，即可使用同名构造函数创建该结构体的实例。例如：

```
struct 光 {
    浮点数 强度; 向量3 位置;
};

light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));
```

构造函数的参数顺序和类型必须与结构体声明时一致。

结构体构造函数可作为初始化器或在表达式中使用。

55 向量到光照组件

向量分量的名称由单个字母表示。为便于记号，根据位置、颜色或纹理坐标向量的常用分量，每个分量关联多个字母。

坐标向量。向量的各个分量可通过在变量名后跟随句点 (.) 再接分量名来选取。

支持的分量名称包括：

{x, y, z, w}	在访问表示点或法线的向量时非常有用
{r, g, b, a}	适用于访问表示颜色的向量
{s, t, p, q}	适用于访问表示纹理坐标的向量

例如，分量名称 *x*、*r* 和 *s* 是向量中同一（第一个）分量的同义词。

请注意，纹理的第三个分量（在OpenGL中为*r*）已被重命名为*p*，以避免与颜色中的*r*（代表红色）产生混淆。

访问超出向量类型声明范围的分量将导致错误，例如：

```
vec2 pos;
pos.x           // 合法 pos.z      // 非法
```

通过在点号 (.) 后追加多个同名组件名称，可实现组件组合选择。

```
vec4 v4;
v4.rgb;          // 属于vec4类型，等同于直接使用v4或v4.rgb;           // 表示vec3类型
v4.b;            // 表示浮点数，v4.xy;
                 // 表示vec2类型,
v4.xgba;         // 非法 - 组件名称未来自
                 // 同一组。
```

组件顺序可通过交换实现混排，或进行复制：

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz = pos.wzyx; // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy; // dup = (1.0, 1.0, 2.0, 2.0)
```

这种表示法比构造函数语法更简洁。要形成右值，可将其应用于任何产生向量右值的表达式。

分量组表示法可出现在表达式的左侧。

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
pos.xw = vec2(5.0, 6.0);                  // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0);                  // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0);                  // 非法 - 'x' 被重复使用两次
pos.xy = vec3(1.0, 2.0, 3.0); // 非法 - vec2 与 vec3 不匹配
```

要形成左值，必须对向量类型的左值进行数据交换操作，且该左值不得包含重复分量。根据指定的分量数量，操作结果将生成标量或向量类型的左值。

数组下标语法也可应用于向量以实现数值索引。因此在

```
vec4 pos;
```

`pos[2]` 表示 `pos` 的第三个元素，等同于 `pos.z`。这允许对向量进行变量索引，并提供了一种通用组件访问方式。任何整数表达式均可作为下标使用。第一个组件位于索引零处。若索引大于或等于向量大小，则行为未定义。

56 矩阵分量

矩阵的分量可通过数组下标语法访问。对矩阵应用单个下标时，将矩阵视为列向量数组，并选取单列分量，其类型为与矩阵同尺寸的向量。最左列为第0列。第二个下标将作用于该列向量，其操作规则与前述向量定义一致。因此双下标操作可依次选取列分量与行分量。

```
mat4 m;
m[1] = vec4(2.0);           // 将第二列全部设为2.0m[0][0] = 1.0; // 将左上角元素设为1.0
m[2][3] = 2.0;              // 将第三列的第四个元素设为2.0
```

当访问矩阵边界外的分量时（例如 `mat3` 的 `[3][3]` 分量），行为未定义。

57 结构体与字段

与向量分量及交换规则类似，结构体的字段同样通过点号（`.`）进行选择。结构体支持以下操作符：

结构体字段选择器	<code>.</code>
等值	<code>== !=</code>
赋值	<code>=</code>

等值运算符和赋值运算符仅在两个操作数的类型为相同声明结构时有效。使用等值运算符时，两个结构体仅当所有字段逐元素相等时才视为相等。

58 符号

将值赋给变量名时使用赋值运算符（`=`），例如：

```
lvalue = expression
```

赋值运算符将表达式的值存储到左值中。仅当表达式和左值具有相同类型时才会编译通过。所有所需的类型转换都必须通过构造函数显式指定。

lvalue类型相同时才会通过编译。所有所需的类型转换都必须通过构造函数显式指定。L-

值必须可写。内置类型变量、完整结构体、结构体字段、通过字段选择器`(.)`选取组件的左值、不含重复字段的交换操作，以及使用数组下标运算符`([])`解引的左值，均属于左值。其他二元或一元表达式、未解引用的数组、函数名、含重复字段的交换式以及常量均不能作为左值。三元运算符`(?:)`同样不允许作为左值。

赋值表达式左侧的表达式优先于右侧表达式求值。其他赋值运算符包括：

- 算术赋值包含加法赋值`(+=)`、减法赋值`(-=)`、乘法赋值`(*=)`和除法赋值`(/=)`。表达式

```
lvalue op= 表达式
```

等同于

```
左值 = 左值 运算符 表达式
```

且左值与表达式必须同时满足运算符`op`与等号`(=)`的语义要求。

- 赋值运算符的模运算`(% =)`、左移运算`(<<=)`、右移运算`(>>=)`、包含或运算运算符`(|=)`和异或运算符`(^=)`。这些运算符预留用于未来扩展。

在写入（或初始化）变量之前读取该变量是合法的，但其值未定义。

59 表达式

着色语言中的表达式由以下元素构成：

- `bool`、`int`、`float`类型的常量，所有向量类型及矩阵类型。
- 所有类型的构造函数。
- 除末跟随下标的数组名称外，所有类型的变量名称。
- 带下标的数组名称。
- 返回值的函数调用。
- 组件字段选择器和数组下标结果。
- 括号表达式。括号可用于分组运算，括号内的运算优先于括号外的运算。
- 算术二元运算符包括加法`(+)`、减法`(-)`、乘法`(*)`和除法`(/)`，这些运算符适用于整数型和浮点型表达式（包括向量与矩阵）。两个操作数必须同类型，或一方为标量浮点数而另一方为浮点向量/矩阵，或一方为标量整数而另一方为整数向量。此外，对于乘法运算`(*)`，一方可为向量，另一方可为与该向量维度相同的矩阵。这些运算的结果类型与操作对象相同（整型或浮点型）。若一个操作数为标量，另一个为向量或矩阵，则标量将按分量与向量或矩阵相乘，结果类型与向量或矩阵相同。除以零不会引发异常，但会导致未指定的行为。

值。对两个向量应用乘法运算符(*)将执行分量级乘法。对两个矩阵应用乘法运算符(*)将执行线性代数矩阵乘法，而非分量级乘法。使用内置函数dot、cross和matrixCompMult分别获取向量点积、向量叉积和矩阵分量级乘积。

- 取模运算符（%）保留用于未来扩展。
- 算术一元运算符包括取反（-）、后递增/递减（++）、前递增/递减（--）以及 $(++)$ 运算符可作用于整数或浮点数值（包括向量和矩阵）。其结果类型与操作数相同。对于后递增/递减和前递增/递减运算符，表达式必须是可赋值的（即左值）。前置递增与前置递减操作会对表达式内容加减1或1.0，其返回值即为修改后的结果值。后递增与后递减运算符对操作对象进行加减1或1.0的操作，但返回值为操作执行前的原始表达式值。
- 关系运算符大于(>)、小于(<)、大于等于(>=)和小于等于(<=)

小于或等于(<=)运算符仅适用于标量整数和标量浮点表达式。结果为标量布尔值。操作数的类型必须匹配。若需对向量进行分量比较，请使用内置函数lessThan、lessThanEqual、greaterThan和greaterThanEqual。

- 等于运算符 equal (==) 和不等于运算符 not equal (!=) 可用于除数组外的所有类型。

运算结果为标量布尔值。对于向量、矩阵和结构体，所有操作数的分量必须完全相等才视为相等。若需获取向量的分量级相等结果，请使用内置函数equal 和 notEqual。

- 逻辑二元运算符包括与运算符(&&)、或运算符(||)及异或运算符(^)。这些运算符仅作用于对两个布尔表达式进行运算并产生布尔表达式。与运算符(&&)仅在左操作数评估为真时才评估右操作数。或运算符(||)仅在左操作数评估为假时才评估右操作数。异或运算符(^)始终评估两个操作数。
- 逻辑单目运算符非(!)。它仅作用于布尔表达式，并返回一个布尔表达式。若需对向量操作，请使用内置函数not。
- 序列运算符(,)对表达式进行操作，返回以逗号分隔的表达式列表中最右侧表达式的类型和数值。所有表达式均按从左到右的顺序依次求值。
- 三元选择运算符(?:)。它对三个表达式进行操作($exp1 ? exp2 : exp3$)。

该运算符首先评估第一个表达式，该表达式必须返回标量布尔值。若结果为真，则选择评估第二个表达式；否则选择评估第三个表达式。第二个和第三个表达式中仅评估其中一个。第二个和第三个表达式必须是相同类型，但可以是除数组以外的任何类型。结果类型与第二个和第三个表达式的类型相同。

- 运算符：与(&)、或(||)、异或(^)、非(~)、右移(>>)、左移(<<)。这些运算符保留用于未来扩展。

有关表达式语法的完整规范，请参阅第9节“着色语言语法”。当操作数类型不同时，必须符合以下规则之一：

- 其中一个参数为浮点数（即标量），此时结果等同于将标量值复制为向量或矩阵后再进行运算。
- 左侧参数为浮点数向量，右侧参数为维度兼容的矩阵时，*运算符将执行行向量与矩阵的乘法运算。
- 左侧参数为矩阵，右侧参数为浮点向量且维度兼容时，*运算符将执行列向量与矩阵的乘法运算。

5.10 向量与矩阵运算

除少数例外情况外，运算均为分量级操作。当运算符作用于向量或矩阵时，会以分量级方式独立处理向量或矩阵的每个分量。

例如：

```
vec3 v, u; float f;
```

```
v = u + f;
```

等同于

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
```

以及

```
vec3 v, u, w; w = v +
u;
```

将等效于

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
```

此规则适用于大多数运算符以及所有整型、浮点型向量和矩阵类型。例外情况包括矩阵与向量相乘、向量与矩阵相乘以及矩阵与矩阵相乘。这些运算不按分量进行，而是执行正确的线性代数乘法。它们要求操作数的大小匹配。

```
vec3 v, u; mat3 m;
```

```
u = v * m;
```

等效于

```
u.x = dot(v, m[0]); // m[0] 是矩阵 m 的左列向量
u.y = dot(v, m[1]); // dot(a,b) 表示 a 与 b 的内积 (点积)
u.z = dot(v, m[2]);
```

以及

```
u = m * v;
```

等同于

```
u.x = m[0].x * v.x + m[1].x * v.y + m[2].x * v.z;
u.y = m[0].y * v.x + m[1].y * v.y + m[2].y * v.z;
u.z = m[0].z * v.x + m[1].z * v.y + m[2].z * v.z;
```

以及

```
mat m, n, r; r = m *
n;
```

等同于

```
r[0].x = m[0].x * n[0].x + m[1].x * n[0].y + m[2].x * n[0].z;
r[1].x = m[0].x * n[1].x + m[1].x * n[1].y + m[2].x * n[1].z;
r[2].x = m[0].x * n[2].x + m[1].x * n[2].y + m[2].x * n[2].z;
```

```
r[0].y = m[0].y * n[0].x + m[1].y * n[0].y + m[2].y * n[0].z;
r[1].y = m[0].y * n[1].x + m[1].y * n[1].y + m[2].y * n[1].z;
r[2].y = m[0].y * n[2].x + m[1].y * n[2].y + m[2].y * n[2].z;
```

```
r[0].z = m[0].z * n[0].x + m[1].z * n[0].y + m[2].z * n[0].z;
r[1].z = m[0].z * n[1].x + m[1].z * n[1].y + m[2].z * n[1].z;
r[2].z = m[0].z * n[2].x + m[1].z * n[2].y + m[2].z * n[2].z;
```

同理适用于2维和4维的向量与矩阵。

所有一元运算都按分量方式对操作数进行操作。对于二元算术运算，若两个操作数类型相同，则按分量方式执行运算，并产生与操作数同类型的结果。若一个操作数是标量浮点数，另一个操作数是向量或矩阵，则运算过程会将标量值复制为匹配的向量或矩阵操作数。

6 语句与结构

OpenGL着色语言的基本构建模块包括：

- 语句与声明
- 函数定义
- 选择语句（`if-else`）
- 迭代（`for`、`while` 和 `do-while`）
- 跳转语句（`discard`、`return`、`break` 和 `continue`）

着色器的整体结构如下

翻译单元：

全局声明

翻译单元全局声明

全局声明：函数定义声明

即着色器是由声明序列与函数体组成的序列。函数体定义为

函数定义：

函数原型 { 语句列表 }

语句列表：

语句

语句列表 语句

语句：

复合语句 简单语句

大括号用于将一系列语句组合成复合语句。

复合语句：

{ 语句列表 }

简单语句：声明语句 表达式语句 选择语句

迭代语句 跳转语句

简单的声明语句、表达式语句和跳转语句以分号结束。

上述内容略有简化，应以第9节“着色语言语法”中规定的完整语法作为最终规范。

声明和表达式已在前文讨论。

61

函数定义

如上文语法所示，有效的着色器由全局声明与函数定义序列构成。函数声明格式如下例所示：

```
// 原型声明
returnType functionName (type0 arg0, type1 arg1, ..., typen argn);
```

函数定义格式如下：

```
// 定义
returnType functionName (type0 arg0, type1 arg1, ..., typen argn)
{
    // 执行计算 return 返回值;
}
```

其中 `returnType` 必须存在并包含类型声明。每个 `typeN` 声明必须包含类型声明，并可选地包含 `in`、`out`、`inout` 和/或 `const` 修饰符。

函数调用时需使用其名称，后跟括号内的参数列表。

数组可作为参数传递，但不能作为返回类型。当数组作为形式参数声明时，必须包含其大小。向函数传递数组时，只需使用数组名称（无需下标或方括号），且传递的数组参数大小必须与形式参数声明中指定的大小一致。

结构体同样可作为参数使用。返回类型亦可为结构体。

有关函数声明与定义语法的权威参考，请参阅第9节“着色语言语法”。

所有函数在调用前都必须声明原型或定义函数体。例如：

```
float myfunc (float f, // f 为输入参数 out float g); // g 为输出参数
```

不返回值的函数必须声明为 `void`。不接受输入参数的函数在参数列表中无需使用 `void`，因为函数原型是必需的，因此声明空参数列表 “()” 时不会产生歧义。参数列表中的 “(`void`)” 表达式仅为方便起见而提供。

函数名可以重载。这允许同一个函数名用于多个函数，只要参数列表类型不同即可。若函数名称和参数类型匹配，则其返回类型和参数限定符也必须匹配。内置函数大量使用了重载机制。

解析重载函数（或任何函数）时，系统会精确匹配函数签名，包括数组尺寸的精确匹配。不会对返回类型或输入参数类型进行类型提升或降级。所有预期的输入输出组合都必须定义为独立函数。

例如内置点积函数具有以下原型：

```
float dot(float x, float y); float dot(vec2 x, vec2 y);
float dot(vec3 x, vec3 y); float dot(vec4 x, vec4 y);
```

用户定义函数可以有多个声明，但只能有一个定义。着色器可以重新定义内置函数。如果在调用内置函数之前，该函数已在着色器中重新声明（即其原型可见），则链接器仅尝试在与其链接的着色器集中解析该调用。

*main*函数作为着色器的入口点。单个着色器无需包含名为*main*的函数，但由多个着色器链接形成的单一程序中必须存在该函数。此函数不接受参数、不返回值，且必须声明为**void**类型：

```
void main()
{
    ...
}
```

*main*函数中可包含**return**语句的使用。更多细节请参阅第6.4节“跳转”。

6.1.1 函数调用约定

函数采用值传递方式调用。这意味着输入参数在调用时被复制到函数内部，输出参数在函数退出前被复制回调用方。由于函数操作的是参数的局部副本，因此函数内部不存在变量别名问题。调用时，输入参数按从左到右的顺序依次求值。但输出参数回传给调用方的顺序未作定义。若需通过函数定义或声明控制参数的传入/传出行为：

- 使用关键字 **in** 作为限定符，表示参数仅被复制进入函数，而不被复制出函数。
- 关键字 **out** 用作修饰符，表示参数仅需复制输出而不需复制输入。应尽可能使用此修饰符，以避免不必要的参数复制操作。
- 关键字 **inout** 用作修饰符，表示该参数既需复制输入也需复制输出。
- 未声明此类修饰符的函数参数，其含义等同于指定为 **in**。

在函数中允许对只输入参数进行写操作，但仅修改函数内部的副本。若需禁止此行为，可使用 `const` 修饰符声明参数。

在调用函数时，无法将未评估为左值的表达式传递给声明为 `out` 或 `inout` 的参数。

函数的返回类型不允许使用限定符。

函数原型：

类型 函数名(`const`限定符 参数限定符 类型 名称 数组指定符 ...)

类型：

任何基本类型、结构体名称或结构体定义

`const`限定符：空 `const`

参数修饰符：

空

`in out`

`inout`

名称：

空标识符

数组指定符：

空

[整数常量表达式]

然而，`const`限定符不能与`out`或`inout`同时使用。上述规则适用于函数声明（即原型声明）和函数定义。因此，函数定义可以包含未命名的参数。

若使用递归，行为将未定义。递归指在运行时函数调用栈中，任何函数在任意时刻出现超过一次的情况。即函数不得直接或间接调用自身。编译器可在编译时检测到此类情况时输出诊断信息，但并非所有此类情况都能在编译时被检测到。

62

选择

着色语言中的条件控制流通过 `if` 或 `if-else` 实现：

```
if (布尔表达式) true-语句
```

或

```
if (布尔表达式) 真语句
else
```

false-语句

如果表达式评估为真，则执行**真语句**。如果评估为假且存在
else 部分，则执行**false-statement**。

任何类型可评估为布尔值的表达式均可作为条件表达式`bool-expression`使用。向量类型不可作为**if语句**的表达式。

条件语句可嵌套使用。

63

迭代

允许使用以下形式的 `for`、`while` 和 `do` 循环：

```
for (初始化表达式; 条件表达式; 循环表达式) 子语句

while (条件表达式) 子语句

do
    语句
    while (条件表达式)
```

有关循环的权威规范，请参阅第9节“着色语言语法”。

for循环首先评估**初始化表达式**，随后评估**条件表达式**。若**条件表达式**评估结果为真，则执行**循环体**。**循环体**执行完毕后，**for**循环将评估**循环表达式**，然后返回评估**条件表达式**，如此循环往复直至**条件表达式**评估结果为假。此时循环终止，跳过**循环体**和**循环表达式**。由**循环表达式**修改的变量在循环退出后仍保持其值（前提是它们仍在作用域内）。而在**初始化表达式**或**条件表达式**中声明的变量，其作用域仅持续到**for**循环子语句结束为止。

while循环首先评估**条件表达式**。若为真，则执行**循环体**。此过程将持续重复，直至**条件表达式**评估为假，此时循环退出并跳过**循环体**。在**条件表达式**中声明的变量仅在**while**循环子语句结束前有效。

do-while循环先执行**主体**，再执行**条件表达式**。**循环**重复执行直至
条件表达式评估为假时，**循环**才终止。**条件表达式**必须评估为布尔值。

条件表达式和**初始化表达式**均可声明并初始化变量，但**do-while**循环除外——其**条件表达式**中不可声明变量。该变量的作用域仅持续至构成**循环主体**的子语句结束。

循环可以嵌套。

允许使用无限循环。过长或无限循环的后果取决于具体平台。

64 跳转

跳转语句包括：

```
跳转语句: continue; break;
return;
return 表达式;
discard; // 仅限于片段着色器语言中不存在"goto"或其他非结构化控制流。
```

continue跳转仅用于循环结构。它会跳过当前所在最内层循环主体的剩余部分。对于**while**和**do-while**循环，该跳转将跳至**循环条件表达式**的下一次评估，循环将按先前定义的方式继续执行。对于**for**循环，跳转将跳至**循环表达式**，随后执行**条件表达式**。

break跳转同样仅限于循环结构。它会立即退出包含**break**语句的最内层循环，不再执行后续的**条件表达式**或**循环表达式**。

discard关键字仅允许在片段着色器中使用。它可在片段着色器中用于放弃对当前片段的操作。该关键字会导致片段被丢弃，且不会对任何缓冲区进行更新。通常在条件语句中使用，例如：

```
if (intensity < 0.0) discard;
```

片段着色器可检测片段的透明度值，并根据检测结果丢弃该片段。但需注意：覆盖率测试发生在片段着色器运行之后，且覆盖率测试可能改变透明度值。

返回跳转会立即退出当前函数。若其带有**表达式**，则该表达式即为函数的返回值。

主函数(*main*)可使用**return**语句。这仅会使*main*函数像执行到末尾时那样退出，并不意味着片段着色器中使用了**discard**操作。在*main*中定义输出前使用**return**，其行为等同于在定义输出前执行到*main*末尾。

7 内置变量

7.1

顶点着色器特殊变量

某些OpenGL操作仍在顶点处理器与片段处理器之间的固定功能单元中执行。其他OpenGL操作则在片段处理器之后继续于固定功能单元中运行。着色器通过内置变量与OpenGL的固定功能单元进行通信。

变量 `gl_Position` 仅在顶点语言中可用，用于写入齐次顶点位置。所有结构良好的顶点着色器执行时都必须向该变量写入值。着色器执行期间可随时写入该变量，写入后着色器也可读取该值。该值将被用于顶点处理后的基元组装、裁剪、剔除及其他基于基元的固定功能操作。若编译器检测到未写入 `gl_Position` 或在写入前读取该变量，可能会生成诊断信息，但并非所有此类情况均可检测。若顶点着色器执行时未写入 `gl_Position`，结果将未定义。

变量 `gl_PointSize` 仅在顶点语言中可用，用于顶点着色器写入待光栅化点的尺寸。该尺寸以像素为单位。

变量 `gl_ClipVertex` 仅在顶点着色器语言中可用，为顶点着色器提供存储坐标的位置，该坐标将用于用户裁剪平面。用户必须确保裁剪顶点与用户裁剪平面定义在同一坐标空间中。用户裁剪平面仅在线性变换下正常工作，在非线性变换下的行为未定义。

这些用于与固定功能单元通信的内置顶点着色器变量，其内在声明类型如下：

```
vec4 gl_Position;           // 必须写入浮点型变量 gl_PointSize;  
                           // 可能被写入 vec4 gl_ClipVertex; //  
可能被写入
```

若未向 `gl_PointSize` 或 `gl_ClipVertex` 写入值，则其值未定义。着色器可在写入后读取这些变量以获取写入内容。在写入前读取这些变量将导致未定义行为。若多次写入，后续操作将采用最后写入的值。

这些内置变量具有全局作用域。

72

片段着色器特殊变量

片段着色器的输出由OpenGL管道后端的固定功能操作处理。除非执行**discard**关键字，否则片段着色器通过内置变量*gl_FragColor*、*gl_FragData*和*gl_FragDepth*向OpenGL管道输出值。

这些变量在片段着色器中可能被多次写入。若出现这种情况，则最后赋予的值将在后续的固定功能管道中使用。写入这些变量的值可在写入后被读取。若在写入前读取这些变量，将导致未定义值。通过读取下文所述的*gl_FragCoord.z*，可获取片段的固定功能计算深度。

向*gl_FragColor*写入值将指定后续固定功能管道使用的片段颜色。若后续固定功能会使用片段颜色，而片段着色器执行时未向*gl_FragColor*写入值，则该片段颜色将呈现未定义状态。

如果帧缓冲区配置为颜色索引缓冲区，则使用片段着色器时行为未定义。

向*gl_FragDepth*写入将为正在处理的片段建立深度值。若启用深度缓冲，且着色器未写入*gl_FragDepth*，则将使用固定函数深度值作为片段的深度值。若着色器通过静态赋值设置*gl_FragDepth*，且存在未设置*该值*的执行路径，则走该路径的着色器执行中，片段深度值可能未定义。即：着色器若静态包含对*gl_FragDepth*的写入操作，则必须始终执行该写入操作。

(若着色器在预处理后包含会写入*变量x*的语句，则该着色器包含对*变量x*的静态赋值，无论运行时控制流是否会导致该语句被执行。)

变量*gl_FragData*是一个数组。对*gl_FragData[n]*的写入操作将指定后续固定功能管道中数据*n*所使用的片段数据。若后续固定功能消耗片段数据，而片段着色器执行时未向其写入值，则被消耗的片段数据行为未定义。

若着色器静态赋值给*gl_FragColor*，则不得为*gl_FragData*的任何元素赋值；若着色器静态写入*gl_FragData*的任意元素，则不得为*gl_FragColor*赋值。即着色器可选择性地为*gl_FragColor*或*gl_FragData*赋值，但不可同时赋值两者。

如果着色器执行**discard**关键字，则该片段将被丢弃，*gl_FragDepth*、*gl_FragColor*和*gl_FragData*的值将不再相关。

变量*gl_FragCoord*在片段着色器中作为只读变量可用，其存储片段在窗口中的相对坐标x、y、z以及1/w值。该值是顶点处理后通过固定功能对基元进行插值生成片段的结果。*z*分量即为片段深度值——当着色器未对*gl_FragDepth*进行写入时，该值将被用作深度值。此特性对保持不变性尤为重要：当着色器需条件计算*gl_FragDepth*值，但同时希望采用固定功能片段深度时可发挥作用。

片段着色器可访问只读内置变量`gl_FrontFacing`，当片段属于正面基元时该变量值为`true`。此特性可用于模拟双面光照效果——通过选择顶点着色器计算的两种颜色之一来实现。

片段着色器可访问的内置变量具有以下固有类型：

```
vec4 gl_FragCoord; bool  
gl_FrontFacing; vec4 gl_FragColor;  
vec4 gl_FragData[gl_MaxDrawBuffers]; float gl_FragDepth;
```

但它们的行为与无限定符的变量不同，其行为如上所述。这些内置变量具有全局作用域。

73

顶点着色器内置属性

下列属性名称内置于OpenGL顶点语言中，可在顶点着色器内部使用以访问OpenGL声明的属性当前值。所有页码及标记均参照OpenGL 1.4规范。

```
//  
// 顶点属性，第19页。  
//  
属性 vec4 gl_Color; 属性 vec4 gl_SecondaryColor; 属性 vec3  
gl_Normal; 属性 vec4 gl_Vertex; 属性 vec4  
gl_MultiTexCoord0; 属性 vec4 gl_MultiTexCoord1; 属性 vec4  
gl_MultiTexCoord2; 属性 vec4 gl_MultiTexCoord3; 属性 vec4  
gl_MultiTexCoord4; 属性 vec4 gl_MultiTexCoord5; 属性 vec4  
gl_MultiTexCoord6; 属性 vec4 gl_MultiTexCoord7; 属性 float  
gl_FogCoord;
```

74

内置常量

以下内置常量可用于顶点着色器和片段着色器。

```
//  
// 实现相关的常量。下面的示例值  
// 仅为这些最大值允许的最小值。  
//  
const int gl_MaxLights = 8; // GL 1.0  
const int gl_MaxClipPlanes = 6; // GL 1.0  
const int gl_MaxTextureUnits = 2; // GL 1.3  
const int gl_MaxTextureCoords = 2; // ARB_fragment_program const int  
gl_MaxVertexAttribs = 16; // ARB_vertex_shader
```

```

const int gl_MaxVertexUniformComponents = 512; // ARB_vertex_shader const int gl_MaxVaryingFloats = 32;           //
ARB_vertex_shader const int gl_MaxVertexTextureImageUnits = 0;                                // ARB_vertex_shader const int
gl_MaxCombinedTextureImageUnits = 2; // ARB_vertex_shader const int gl_MaxTextureImageUnits = 2;    //
ARB_fragment_shader const int gl_MaxFragmentUniformComponents = 64; // ARB_fragment_shader
const int gl_MaxDrawBuffers = 1;                                                               // 拟议 ARB_draw_buffers

```

75

内置统一状态

为便于访问 OpenGL 处理状态，以下统一变量已内置于 OpenGL 着色语言中。所有页码和符号均参照 1.4 版规范。

```

//  

// 矩阵状态。参见第31、32、37、39、40页。  

//  

uniform mat4 gl_ModelViewMatrix; uniform mat4  

gl_ProjectionMatrix;  

uniform mat4 gl_ModelViewProjectionMatrix;  

uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];  

  

//  

// 派生矩阵状态，提供逆矩阵和转置矩阵版本  

// 上述矩阵的。条件差的矩阵可能导致  

// 导致其逆矩阵形式出现不可预测的数值。  

//  

uniform mat3 gl_NormalMatrix; // 逆矩阵的转置形式  

                                         // gl_ModelViewMatrix 左上角 3x3 矩阵的逆矩阵转置  

  

uniform mat4 gl_ModelViewMatrixInverse; uniform mat4  

gl_ProjectionMatrixInverse;  

uniform mat4 gl_ModelViewProjectionMatrixInverse;  

uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];  

  

uniform mat4 gl_ModelViewMatrixTranspose; uniform mat4  

gl_ProjectionMatrixTranspose;  

uniform mat4 gl_ModelViewProjectionMatrixTranspose;  

uniform mat4 gl_纹理矩阵转置[gl_最大纹理坐标];  

  

uniform mat4 gl_模型视图矩阵逆转置;uniform mat4 gl_投影矩阵逆转置;  

uniform mat4 gl_模型视图投影矩阵逆转置;  

uniform mat4 gl_纹理矩阵逆转置[gl_最大纹理坐标];  

  

//  

// 法线缩放 第39页。  

//  

uniform float gl_NormalScale;  

  

//
```

```

// 窗口坐标系中的深度范围, 第33页
//
struct gl_DepthRangeParameters {float near;           // n
                                float far;            // f
                                float diff;           // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;

//
// 剪切平面 第42页。
//
uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];

//
// 点大小, 第66、67页。
//
struct gl_PointParameters {float size;
                           float minSize;float maxSize;
                           float fadeOffset;
                           float distanceConstant;float distanceLinear;float distanceQuadratic;
};
uniform gl_PointParameters gl_Point;

//
// 材质状态 第50页、55页。
//
struct gl_MaterialParameters {vec4 emission;          // Ecm vec4
                               vec4 ambient;           // Acm vec4
                               vec4 diffuse;           // Dcm vec4
                               vec4 specular;          // Scm
                               float shininess;        // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial;uniform gl_MaterialParameters
gl_BackMaterial;

//
// 光照状态 p 50, 53, 55.
//
struct gl_LightSourceParameters {
    vec4 environment;           // Acli
    vec4 diffuse;               // Dcli
    vec4 specular;              // Scli
    vec4 position;              // Pcli
    vec4 halfVector;             // 派生类: Hi
}

```

```

    vec3 焦点方向;                                // Sdli
    浮点数 spotExponent;                          // Srli
    float spotCutoff;                            // Crli
                                                // (范围: [0.0,90.0], 180.0)
    浮点型 spotCosCutoff;                        // 衍生: cos(Crli)
                                                // (范围: [1.0,0.0],-1.0)
    float constantAttenuation;                  // K0float
    linearAttenuation; // K1float quadraticAttenuation;// K2
};

uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];struct gl_LightModelParameters {
    vec4 环境光;                                // Acs
};

uniform gl_LightModelParameters gl_LightModel;
// 根据光源与材质的乘积推导状态。
//

struct gl_LightModelProducts {
    vec4 场景颜色;                                // 衍生值。Ecm + Acm * Acs
};

uniform gl_LightModelProducts gl_FrontLightModelProduct;uniform gl_LightModelProducts
gl_BackLightModelProduct;

struct gl_LightProducts {
    vec4 环境光;                                // Acm * Acli
    vec4 漫反射;                                // Dcm * Dcli
    vec4 镜面反射光;                            // Scm * Scli
};

uniform gl_LightProducts gl_FrontLightProduct[gl_MaxLights];uniform gl_LightProducts
gl_BackLightProduct[gl_MaxLights];

//
// 纹理环境与生成, 第152页, 第40-42页。
//
uniform vec4 gl_TextureEnvColor[gl_MaxTextureImageUnits];uniform vec4
gl_EyePlaneS[gl_MaxTextureCoords];
uniform vec4 视点平面T[gl_MaxTextureCoords];uniform vec4 视点平面
R[gl_MaxTextureCoords];uniform vec4 视点平面Q[gl_MaxTextureCoords];uniform vec4 物体平面
S[gl_MaxTextureCoords];uniform vec4 物体平面T[gl_MaxTextureCoords]; uniform vec4 视点平面右坐标
[gl_MaxTextureCoords]; uniform vec4 视点平面垂直坐标[gl_MaxTextureCoords];

```

```

//  

// 雾效 第161页  

//  

struct 雾参数 {vec4 颜色;  

    float 密度;float 起始点;  

    float 终点;  

    浮点数 缩放因子;           // 推导式:          1.0 / (终点 - 起点)  

};  

  

uniform gl_FogParameters gl_Fog;

```

76

变量

与用户定义的变量不同，内置变量在顶点语言和片段语言之间不存在严格的一一对应关系。系统提供了两组变量，分别对应两种语言。其关系如下所述。

以下内置变量可在顶点着色器中进行写入操作。若对应的片段着色器或固定管线中的任何功能使用了该变量或其派生状态，则必须对其进行写入。否则，行为将未定义。

```

varying vec4 gl_FrontColor; varying vec4  

gl_BackColor;  

varying vec4 gl_FrontSecondaryColor; varying vec4  

gl_BackSecondaryColor;  

varying vec4 gl_TexCoord[]; // 最大长度不超过 gl_MaxTextureCoords varying float gl_FragCoord;

```

对于 *gl_FogFragCoord*，写入的值将被固定功能管道作为 OpenGL 1.4 规范第 160 页所述的“c”值使用。例如，若需将视点空间中片段的 z 坐标作为“c”值，则顶点着色器应将该值写入 *gl_FogFragCoord*。

与所有数组相同，用于下标 *gl_TexCoord* 的索引必须是整数常量表达式，或者该数组必须由着色器重新声明并指定大小。其大小最多可达 *gl_MaxTextureCoords*。使用接近 0 的索引可能有助于实现中保存可变资源。

以下变量可在片段着色器中读取：*gl_Color* 与 *gl_SecondaryColor* 的命名与传递给顶点着色器的属性相同。但因属性仅在顶点着色器可见，而以下变量仅在片段着色器可见，故不存在命名冲突。

```

varying vec4 gl_Color;  

varying vec4 二级颜色;  

可变 vec4 gl_TexCoord[]; // 最多为 gl_MaxTextureCoords 个元素可变 float gl_FogFragCoord;

```

gl_Color 和 *gl_SecondaryColor* 的值将由系统根据可见面自动从 *gl_FrontColor*、*gl_BackColor*、*gl_FrontSecondaryColor* 和 *gl_BackSecondaryColor* 推导得出。若顶点处理采用固定功能，则 *gl_FogFragCoord* 将取值于：

视点空间中片段的 z 坐标，或雾坐标插值结果（详见 OpenGL 1.4 规范第 3.10 节）。`gl_TexCoord[]` 的值可为顶点着色器插值的 `gl_TexCoord[]` 值，或基于固定管道顶点功能的纹理坐标。

对片段着色器 `gl_TexCoord` 数组的索引方式如上文顶点着色器部分所述。

8 内置函数

OpenGL着色语言定义了一系列用于标量和向量运算的内置便利函数。其中许多内置函数可用于多种类型的着色器，但部分函数旨在提供与硬件的直接映射，因此仅适用于特定类型的着色器。

着色器类型。

内置函数基本分为三类：

- 它们以便捷方式暴露某些必要的硬件功能，例如访问纹理贴图。这些功能无法通过着色器在语言层面实现模拟。
- 它们代表一种简单的操作（如钳位、混合等），用户编写起来非常容易，但这类操作非常常见，可能获得直接的硬件支持。对于编译器而言，将表达式映射到复杂的汇编指令是一项非常困难的任务。
- 它们代表图形硬件未来可能加速的操作。三角函数即属于此类。

许多函数与常见C库中同名函数相似，但除支持传统标量输入外，还支持向量输入。

应鼓励应用程序使用内置函数，而非在着色器代码中自行实现等效计算，因为内置函数被认为是最佳方案（例如可能直接由硬件支持）。

用户代码可选择用自定义函数替换内置函数，只需重新声明并定义相同名称和参数列表即可。

当下方指定内置函数时，其输入参数（及对应输出）可为浮点型、`vec2`、`vec3` 或 `vec4` 类型，此时使用 `genType` 作为参数类型。对于函数的任何具体应用，所有参数与返回值的实际类型必须一致。`矩阵 mat` 亦同理，其类型可为 `mat2`、`mat3` 或 `mat4`。

81

角度与三角函数

以**弧度**形式指定的函数参数默认采用弧度制。这些函数在任何情况下均不会引发除以零错误。若比值的除数为零，则结果将未定义。

这些函数均按分量操作，描述内容对应各分量。

语法	描述
genType radians (genType <i>degrees</i>)	将 度数 转换为弧度并返回结果，即 $\text{result} = \pi/180 \cdot \text{degrees}$ 。
genType degrees (genType <i>radians</i>)	将 弧度 转换为度并返回结果，即 $\text{result} = 180/\pi \cdot \text{radians}$ 。
genType sin (genType <i>angle</i>)	标准三角函数正弦函数。
genType cos (genType <i>angle</i>)	标准三角函数余弦函数。
genType tan (genType <i>angle</i>)	标准三角函数正切。
genType asin (genType <i>x</i>)	反正弦函数。返回正弦值为 <i>x</i> 的角度。该函数返回值范围为 $[-\pi/2, \pi/2]$ 。若 $ x > 1$ ，则结果未定义。
genType acos (genType <i>x</i>)	反余弦。返回余弦值为 <i>x</i> 的角度。该函数返回值范围为 $[0, \pi]$ 。若 $ x > 1$ ，则结果未定义。
genType atan (genType <i>y</i> , genType <i>x</i>)	反正切。返回一个其正切值为 <i>y/x</i> 的角度。 <i>x</i> 和 <i>y</i> 的符号用于确定该角度所在的象限。该函数返回的值范围为 $[-\pi, \pi]$ 。若 <i>x</i> 和 <i>y</i> 均为 0，则结果未定义。
genType atan (genType <i>y_over_x</i>)	反正切。返回一个正切值为 <i>y_over_x</i> 的角度。该函数返回值的范围为 $[-\pi/2, \pi/2]$ 。

82 指数函数

这些函数均按分量操作。描述按分量进行。

语法	描述
genType pow (genType x , genType y)	返回 x 的 y 次方, 即 x^y 。 当 $x < 0$ 时, 结果未定义。 当 $x = 0$ 且 $y \leq 0$ 时, 结果未定义。
genType exp (genType x)	返回 x 的自然幂运算, 即 e^x 。
genType log (genType x)	返回 x 的自然对数, 即返回满足方程 $x = e^y$ 的值 y 。 当 $x \leq 0$ 时, 结果未定义。
genType exp2 (genType x)	返回 2 的 x 次方, 即 2^x 。
genType log2 (genType x)	返回 x 的 2 进制对数, 即返回满足方程 $x = 2^y$ 的值 y 。 当 $x \leq 0$ 时, 结果未定义。
genType sqrt (genType x)	返回 x 的正平方根。若 $x < 0$, 则结果未定义。
genType inversesqrt (genType x)	返回 x 正平方根的倒数。 当 $x \leq 0$ 时, 结果未定义。

83 常用函数

这些函数均按分量操作。描述内容按分量分别说明。

语法	说明
genType abs (genType x)	若 $x \geq 0$ 则返回 x , 否则返回 $-x$
genType sign (genType x)	当 $x > 0$ 时返回 1.0 , 当 $x = 0$ 时返回 0.0 , 当 $x < 0$ 时返回 -1.0
genType floor (genType x)	返回小于或等于 x 的最近整数

genType ceil (genType <i>x</i>)	返回一个值，该值等于大于或等于 <i>x</i> 的最近整数
genType fract (genType <i>x</i>)	返回 $x - \text{floor}(x)$
genType mod (genType <i>x</i> , float <i>y</i>)	模运算。返回 $x - y \times \text{floor}(x/y)$
genType mod (genType <i>x</i> , genType <i>y</i>)	模运算。返回 $x - y \times \text{floor}(x/y)$
genType min (genType <i>x</i> , genType <i>y</i>) genType min (genType <i>x</i> , float <i>y</i>)	若 $y < x$ 则返回 <i>y</i> , 否则返回 <i>x</i>
genType max (genType <i>x</i> , genType <i>y</i>) genType max (genType <i>x</i> , float <i>y</i>)	若 $x < y$ 则返回 <i>y</i> , 否则返回 <i>x</i>
genType clamp (genType <i>x</i> , genType <i>minVal</i> , genType <i>maxVal</i>) genType clamp (genType <i>x</i> , 浮点数 <i>minVal</i> , 浮点数 <i>maxVal</i>)	返回 $\min(\max(x, \minVal), \maxVal)$ 请注意，片段着色器写入的颜色和深度值将在片段着色器运行后由实现进行 钳位处理。
genType mix (genType <i>x</i> , genType <i>y</i> , genType <i>a</i>) genType mix (genType <i>x</i> , genType <i>y</i> , float <i>a</i>)	返回 $x \times (1 - a) + y \times a$, 即 <i>x</i> 和 <i>y</i> 线性混
genType step (genType <i>edge</i> , genType <i>x</i>) genType step (float <i>edge</i> , genType <i>x</i>)	若 $x < \text{边缘值}$ 则返回 0.0, 否则返回 1.0
genType 平滑步 (genType <i>edge0</i> , genType <i>edge1</i> , genType <i>x</i>) genType 平滑步长 (float <i>边缘0</i> , float <i>edge1</i> , genType <i>x</i>)	当 $x \leq \text{edge0}$ 时返回 0.0, 当 $x \geq \text{edge1}$ 时返回 1.0, 并在 $\text{edge0} < x < \text{edge1}$ 时 执行 0 到 1 之间的平滑赫尔姆霍兹插值。这在需要具有平滑过渡的阈值函数 时非常有用。这等同于： genType <i>t</i> ; <i>t</i> = clamp((<i>x</i> - <i>edge0</i>) / (<i>edge1</i> - <i>edge0</i>), 0, 1); return <i>t</i> * <i>t</i> * (3 - 2 * <i>t</i>);

84

几何函数

这些函数对向量进行整体操作，而非逐分量处理。

语法	描述
float length (genType <i>x</i>)	返回向量 <i>x</i> 的长度，即 $\sqrt{x[0]^2 + x[1]^2 + \dots}$
float distance (genType <i>p0</i> , genType <i>p1</i>)	返回 <i>p0</i> 与 <i>p1</i> 之间的距离，即 $\text{length}(p0 - p1)$
float dot (genType <i>x</i> , genType <i>y</i>)	返回 <i>x</i> 与 <i>y</i> 的点积，即 $\text{result} = x[0] * y[0] + x[1] * y[1] + \dots$
vec3 叉积 (vec3 <i>x</i> , vec3 <i>y</i>)	返回 <i>x</i> 与 <i>y</i> 的叉积，即 $\text{result.0} = x[1] * y[2] - y[1] * x[2]$ $\text{result.1} = x[2] * y[0] - y[2] * x[0]$ $\text{result.2} = x[0] * y[1] - y[0] * x[1]$
genType normalize (genType <i>x</i>)	返回与 <i>x</i> 同向但长度为 1 的向量。
vec4 ftransform()	仅适用于顶点着色器。该函数确保传入的顶点值经过变换后，其结果与 OpenGL 固定功能变换完全一致。主要用于计算 gl_Position，例如： $\text{gl_Position} = \text{ftransform()}$ 此函数适用于以下场景：当应用程序在独立渲染通道中呈现相同几何体时，其中一个通道使用固定功能路径进行渲染，另一个通道则使用可编程着色器。
genType faceforward (genType <i>N</i> , genType <i>I</i> , genType <i>Nref</i>)	若 $\text{dot}(Nref, I) < 0$ 则返回 <i>N</i> ，否则返回 <i>-N</i>

<pre>genType reflect(genType I, genType N)</pre>	<p>对于入射向量 I 和表面法向量 N，返回反射方向： N。返回反射方向： $result = I - 2 * \dot(N, I) * N$ N 必须已归一化才能获得预期结果。</p>
<pre>genType refract(genType I, genType N, float eta)</pre>	<p>对于入射向量 I 和表面法向量 N 以及折射率比 η，返回折射向量。返回结果通过以下公式计算：</p> $k = 1.0 - \eta * \eta * (1.0 - \dot(N, I) * \dot(N, I)) \text{ if } (k < 0.0)$ $\text{result} = \text{genType}(0.0) \text{ else}$ $\text{result} = \eta * I - (\eta * \dot(N, I) + \sqrt{k)) * N$ <p>入射向量 I 和表面法向量 N 的输入参数必须预先归一化，才能获得预期结果。</p>

85

矩阵函数

语法	描述
<pre>mat matrixCompMult(mat x, mat y)</pre>	<p>对矩阵 x 与矩阵 y 进行分量相乘，即 $result[i][j]$ 为 $x[i][j]$ 与 $y[i][j]$ 的点积。 注意：若需执行线性代数矩阵乘法，请使用乘法运算符 (*)。</p>

86

Vector Related Functions

关系运算符和等值运算符 ($<$, $<=$, $>$, $>=$, $==$, $!=$) 被定义 (或保留) 为产生标量布尔值结果。若需向量结果，请使用下列内置函数。下文中的“bvec”代表 `bvec2`、`bvec3` 或 `bvec4` 之一，“ivec”代表 `ivec2`、`ivec3` 或 `ivec4` 之一，“vec”代表 `vec2`、`vec3` 或 `vec4` 之一。所有情况下，特定调用中输入向量与返回向量的尺寸必须匹配。

语法	描述
<pre>bvec lessThan(vec x, vec y)bvec lessThan(ivec x, ivec y)</pre>	返回 $x < y$ 的分量级比较结果。

bvec lessThanEqual (vec x, vec y) bvec lessThanEqual (ivec x, ivec y)	返回 $x \leq y$ 的分量比较结果。
bvec greaterThan (vec x, vec y) bvec greaterThan (ivec x, ivec y)	返回 $x > y$ 的分量比较结果。
bvec greaterThanEqual (vec x, vec y) bvec greaterThanEqual (ivec x, ivec y)	返回 $x \geq y$ 的分量比较。
bvec equal (vec x, vec y) bvec equal (ivec x, ivec y) bvec equal (bvec x, bvec y)	返回 $x == y$ 的分量比较。
bvec notEqual (vec x, vec y) bvec notEqual (ivec x, ivec y) bvec notEqual (bvec x, bvec y)	返回 $x != y$ 的分量比较。
bool any (bvec x)	如果 x 的任何分量为真，则返回 true。
bool all (bvec x)	仅当 x 的所有分量均为真时返回真。
bvec not (bvec x)	返回 x 的逐项逻辑补集。

87

纹理查找 函数

纹理查找函数可同时用于顶点着色器和片段着色器。然而，顶点着色器的细节级别并非由固定功能计算，因此顶点纹理查找与片段纹理查找在操作上存在差异。下表中的函数通过采样器访问纹理，这些采样器需通过OpenGL API进行配置。纹理属性（如尺寸、像素格式、维数、过滤方法、MIP贴图层数、深度比较等）同样通过OpenGL API调用定义。当通过下文定义的内置函数访问纹理时，这些属性均会被纳入考量。

若对代表深度纹理的采样器执行非阴影纹理调用（且深度比较功能开启），则结果未定义。若对代表深度纹理的采样器执行阴影纹理调用（且深度比较功能关闭），则结果未定义。若对不代表深度纹理的采样器执行阴影纹理调用，则结果未定义。

在以下所有函数中，偏移量参数对片段着色器而言是可选的。顶点着色器不接受偏移量参数。对于片段着色器，若存在偏移量参数，则在执行纹理访问操作前将其加到计算出的细节级别上。若未提供偏移量参数，则实现会自动选择细节级别：对于未进行MIP映射的纹理，该纹理

直接使用。若采用MIP贴图且在片段着色器中运行，则使用实现计算的LOD值进行纹理查找。若采用MIP贴图且在顶点着色器中运行，则使用基础纹理。

后缀为“Lod”的内置函数仅允许在顶点着色器中使用。对于“Lod”函数，*lod*参数将直接作为细节级别参数使用。

语法	描述
<pre>vec4 texture1D (sampler1D sampler, float 坐标[, float 偏移量]) vec4 texture1DProj (sampler1D 采样器, vec2 坐标[, float 偏移量]) vec4 纹理1D投 影 (sampler1D 采样器, vec4 坐标[, float 偏移量]) vec4 纹理1D投 影 (sampler1D 采样器, float 坐标, float 低深度) vec4 纹理1D投影 低深度 (sampler1D 采样器, vec2 坐标, float 细节等级) vec4 纹 理1D投影细节等级 (采样器1D 采样器, vec4 坐标, float 细节等级)</pre>	使用纹理坐标 <i>coord</i> 对当前绑定到采样器的1D纹理进行纹理查找。对于投影 (“Proj”) 版本，纹理坐标 <i>coord.s</i> 将除以 <i>coord</i> 的最后一个分量。
<pre>vec4 纹理2D (采样器2D sampler, vec2 坐标[, float 偏移量]) vec4 投影纹理2D (sampler2D 采样器, vec3 坐标[, float 偏移量]) vec4 纹理2D投 影 (sampler2D 采样器, vec4 坐标[, 浮点偏移量]) vec4 纹理 2DLod (sampler2D 采样器, vec2 坐标, float 低深度) vec4 纹理2D投影 低深度 (sampler2D 采样器, vec3 坐标, float 低细节级别) vec4 纹理2D投影低细节级别 (sampler2D 采样器, vec4 坐标, float 低细节级别)</pre>	使用纹理坐标 <i>coord</i> 对当前绑定到采样器的2D纹理进行纹理查找。对于射影 (“Proj”) 版本，纹理坐标 (<i>coord.s, coord.i</i>) 将除以 <i>coord</i> 的最后一个分量。对于 vec4 坐标变体， <i>coord</i> 的第三个分量将被忽略。
<pre>vec4 texture3D (sampler3D sampler, vec3 坐标[, float 偏移量]) vec4 纹理3D投影 (sampler3D 采样器, vec4 坐标[, float 偏移量]) vec4 纹理 3DLod (sampler3D 采样器, vec3 坐标, float 低深度) vec4 纹理3D投影 低深度 (sampler3D 采样器, vec4 坐标, float 低深度)</pre>	使用纹理坐标 <i>coord</i> 在当前绑定到采样器的3D纹理中进行纹理查找。对于投影 (“Proj”) 版本，纹理坐标将除以 <i>coord.q</i> 。

<pre>vec4 纹理立方体 (samplerCube 采样器, vec3 坐标[, float 偏移量]) vec4 立方体贴图纹理 Lod (samplerCube 采样器, vec3 坐标, float 低细节级别)</pre>	使用纹理坐标 <i>coord</i> 对当前绑定到采样器的立方体贴图纹理进行纹理查找。坐标方向用于选择在哪个面进行二维纹理查找，具体说明详见 OpenGL 规范 1.4 版第 3.8.6 节。
<pre>vec4 shadow1D (sampler1DShadow sampler, vec3 坐标[, float 偏移量]) vec4 阴影2D (sampler2DShadow 采样器, vec3 坐标[, float 偏移量]) vec4 shadow1DProj (sampler1DShadow 采样器, vec4 坐标[, 浮点偏移量]) vec4 阴影2D投影 (sampler2DShadow 采样器, vec4 坐标[, 浮点偏移量]) vec4 阴影1D距离限制 (sampler1DShadow 采样器, vec3 坐标, float 低深度) vec4 阴影2D低深度 (sampler2DShadow 采样器, vec3 坐标, float 低深度) vec4 阴影1D投影低深度 (sampler1DShadow 采样器, vec4 坐标, 浮点数 细节级别) vec4 阴影2D投影细节级别 (sampler2DShadow 采样器, vec4 坐标, float 细节等级)</pre>	使用纹理坐标 <i>coord</i> 对绑定到采样器的深度纹理进行深度比较查找，具体操作详见章节 OpenGL 1.4 规范第 3.8.14 节。坐标 (<i>coord</i>) 的第三个分量 (<i>coord.p</i>) 用作 R 值。绑定到采样器的纹理必须是深度纹理，否则结果未定义。对于每个内置函数的投影 ("Proj") 版本，纹理坐标将除以 <i>coord.q</i> ，从而得到深度值 R 为 <i>coord.p/coord.q</i> 。对于 "ID" 变体，坐标的第二个分量被忽略。

88

片段处理函数

片段处理函数仅在用于片段处理器的着色器中可用。

导数计算可能耗时且/或数值不稳定。因此，OpenGL 实现可采用快速但非完全精确的导数计算方法来近似真实导数。

导数的预期行为通过前向/后向差分来规定。前向差分：

$$F(x+dx) - F(x) \approx dFdx(x) * dx \quad 1a$$

$$dFdx(x) \approx (F(x+dx) - F(x)) / dx \quad 1b$$

后向差分：

$$F(x-dx) - F(x) \approx -dFdx(x) * dx \quad 2a$$

$$dFdx(x) \approx (F(x) - F(x-dx)) / dx \quad 2b$$

单采样光栅化时，方程1b和2b中 $dx \leq 1.0$ 。多采样光栅化时，方程1b和2b中 $dx <$

2.0。

dFdy 的近似计算方式类似，其中 y 替代 x 。

GL实现可采用上述或其他方法进行计算，但须满足以下条件：

- 1) 该方法可采用分段线性逼近。此类线性逼近意味着高阶导数（如 $dFdx(dFdx(x))$ 及更高阶导数）未定义。
- 2) 该方法可假设被评估函数具有连续性。因此在非均匀条件语句主体内的导数均未定义。
- 3) 该方法可在不同光栅单元间存在差异，但须满足以下约束：方法变化仅可基于窗口坐标而非屏幕坐标。OpenGL 1.4规范第3.1节所述的不变性要求在导数计算中予以放宽，因该方法可作为光栅单元位置的函数。

其他可取但非必需的特性包括：

- 4) 函数应在原始函数的内部区域内进行求值（进行插值而非外推）。
- 5) 计算 $dFdx$ 时应保持 y 恒定，计算 $dFdy$ 时应保持 x 恒定。但混合高阶导数（如 $dFdx(dFdy(y))$ 和 $dFdy(dFdx(x))$ ）未定义。

在某些实现中，通过提供GL提示（参见OpenGL 1.4规范第5.6节），可获得不同程度的导数精度，从而使用户能在图像质量与运行速度之间进行权衡。

语法	描述
genType dFdx (genType p)	使用局部差分法返回输入参数 p 的 x 方向导数。
genType dFdy (genType p)	<p>返回输入参数 p 在 y 方向上的局部差分导数。</p> <p>这两个函数常用于估算用于消除过程纹理混叠的滤波器宽度。我们假设该表达式在 SIMD 数组上并行求解，因此在任意时刻，函数值在 SIMD 数组所代表的网格点上均已确定。</p> <p>因此可通过 SIMD 数组元素间的局部差分计算 $dFdx$、$dFdy$ 等导数。</p>
genType fwidth (genType p)	<p>返回输入参数 p 的 x 方向和 y 方向绝对导数之和，采用局部差分计算，即：</p> <pre>return = abs(dFdx(p)) + abs(dFdy(p));</pre>

89**噪声函数 ons**

噪声函数可同时应用于片段着色器和顶点着色器。这些随机函数能增强视觉复杂度。下列噪声函数返回的值呈现随机外观，但并非真正随机。这些噪声函数具有以下特性：

- 返回值始终位于 [-1.0, 1.0] 区间，且至少覆盖 [-0.6, 0.6] 范围，呈现高斯分布特征。
- 返回值的整体平均值为 0.0
- 具有可重复性，即特定输入值将始终产生相同返回值
- 在旋转下保持统计不变性（即无论域如何旋转，其统计特性保持恒定）
- 它们具有平移不变性（即无论域如何平移，其统计特征保持不变）
- 它们在平移下通常会产生不同的结果。
- 空间频率集中于 0.5 至 1.0 之间的狭窄区间。

- 它们在 C^1 层面上处处连续（即一阶导数连续）

语法	描述
float noise1 (genType <i>x</i>)	根据输入值 <i>x</i> 返回 1D 噪声值。
vec2 noise2 (genType <i>x</i>)	返回基于输入值 <i>x</i> 的二维噪声值。
vec3 noise3 (genType <i>x</i>)	根据输入值 <i>x</i> 返回一个 3D 噪声值。
vec4 noise4 (genType <i>x</i>)	返回基于输入值 <i>x</i> 的 4D 噪声值。

9 着色语言 语法

该语法由词法分析的输出提供。词法分析返回的标记为

```
ATTRIBUTE CONST BOOL FLOAT INT  
BREAK CONTINUE DO ELSE FOR IF DISCARD RETURN  
BVEC2 BVEC3 BVEC4 IVEC2 IVEC3 IVEC4 VEC2 VEC3 VEC4 MAT2 MAT3 MAT4 IN OUT INOUT UNIFORM  
VARYING  
采样器1D 采样器2D 采样器3D 采样器立方体 采样器1D阴影 采样器2D阴影 结构 虚量 WHILE
```

标识符 类型名称 浮点常量 整常量 布尔常量 字段选择

左操作 右操作

递增运算符 递减运算符 取左运算符 大于等于运算符 等于运算符 不等于运算符

与运算或运算异或运算乘赋值除赋值加赋值

取模赋值 左赋值 右赋值 与赋值 异或赋值 或赋值 减法赋值

左括号 右括号 左方括号 右方括号 左大括号 右大括号 点 逗号 冒号 等号 分号 感叹号 破折号 波浪号 加号 星号 斜杠 百分号 左角右角竖线 插入符号 问号

以下描述了基于上述标记的OpenGL着色语言语法。

变量标识符 标识符

基本表达式 变量标识符 整常量 浮点常量 布尔常量

左括号表达式右括号

后置表达式：主表达式

后缀表达式 左方括号 整数表达式 右方括号 函数调用

后缀表达式 DOT 字段选择后缀表达式 INC_OP 后缀表达式 DEC_OP

整数表达式 表达式

函数调用:

函数调用通用

函数调用通用: 带参数函数调用头 RIGHT_PAREN 无参数函数调用头 RIGHT_PAREN

无参数函数调用头: 函数调用头 VOID 函数调用头

带参数的函数调用头: 函数调用头 赋值表达式

带参数的函数调用头 COMMA 赋值表达式

函数调用头: 函数标识符 左括号

函数标识符: 构造函数标识符 标识符

// 语法注释: 构造函数看似函数, 但词法分析会将其大部分识别为关键字。

构造函数标识符 FLOAT

INT BOOL

```
VEC2 VEC3  
VEC4 BVEC2  
BVEC3 BVEC4  
ivec2 ivec3  
ivec4 mat2  
mat3 mat4  
类型名
```

// 语义注释：后缀表达式 INC_OP 一元表达式
DEC_OP 单项表达式 单项运算符 单项表达式

// 语义注释：不支持传统风格的类型转换。

```
一元运算符：加号 减号 感  
叹号  
波浪号 // 保留字符
```

// 语义注释：不支持 '*' 或 '&' 一元运算符。不支持指针。乘法表达式：
一元表达式
乘法表达式 星号 一元表达式 乘法表达式 斜杠 一元表达式 乘法表达式 百分号 一元表达式 // 保留字符

加法表达式：乘法表达式

加法表达式 *PLUS* 乘法表达式 加法表达式 *DASH* 乘法表达式

移位表达式：加法表达式

shift_expression LEFT_OP additive_expression // 保留 *shift_expression RIGHT_OP additive_expression // 保留*

关系表达式：移位表达式

关系表达式 *LEFT_ANGLE* 移位表达式 关系表达式 *RIGHT_ANGLE* 移位表达式 关系表达式

LE_OP 移位表达式 关系表达式 *GE_OP* 移位表达式

等式表达式：关系表达式

等值表达式 *EQ_OP* 关系表达式 等值表达式 *NE_OP* 关系表达式

and_expression:

等值表达式

与表达式 及号 等值表达式 // 保留字符

异或表达式：与表达式

异或表达式 *CARET* 且表达式 // 保留

包含或表达式 异或表达式

包含或表达式 *VERTICAL_BAR* 排他或表达式 // 保留

逻辑与表达式 包含或表达式

逻辑与表达式 AND_OP 包含或表达式

逻辑异或表达式 逻辑与表达式

逻辑异或表达式 XOR_OP 逻辑与表达式

逻辑或表达式 逻辑异或表达式

逻辑或表达式 OR_OP 逻辑异或表达式

条件表达式：逻辑或表达式

逻辑或表达式 $QUESTION$ 表达式冒号条件表达式

赋值表达式：条件表达式

一元表达式 赋值运算符 赋值表达式

赋值运算符：等于 乘法赋值 除法

赋值

取模赋值 // 保留 累加赋值 取模赋值

$LEFT_ASSIGN$ // 保留 $RIGHT_ASSIGN$ // 保留

AND_ASSIGN // 保留 XOR_ASSIGN // 保留 OR_ASSIGN //

保留

表达式：

赋值表达式

表达式逗号赋值表达式常量表达式：

条件表达式**声明**`函数原型 分号 初始化声明符列表 分号`**函数原型:**`函数声明符 右括号`**函数声明符: 函数头**`带参数的函数头`**带参数的函数头: 函数头 参数声明**`带参数的函数头 逗号 参数声明`**函数头:**`完全指定类型 标识符 左括号`**参数声明符: 类型限定符 标识符**`类型限定符 标识符 左方括号 常量表达式 右方括号`**参数声明:**`类型限定符 参数限定符 参数声明符 参数限定符 参数声明符``类型限定符 参数限定符 参数类型限定符 参数限定符 参数类型限定符`**参数限定符:**`/* 空 */ IN``OUT INOUT`

参数类型限定符：类型限定符

 类型限定符 左方括号 常量表达式 右方括号

 初始化声明符列表：单个声明

 初始化声明符列表逗号标识符

 init_declarator_list COMMA IDENTIFIER LEFT_BRACKET RIGHT_BRACKET

 初始化声明符列表逗号标识符左括号常量表达式右括号

 初始化声明符列表逗号标识符等号初始化器

单一声明：完全指定类型 完全指定类型 标识符

 完全指定类型 标识符 左括号 右括号完全指定类型 标识符 左括号 常量表达式 右括号完全指定类型 标识符 等号 初始化器

// 语法注释：不支持enum'或'typedef'。

完全指定类型：类型限定符

 类型限定符 类型指定符

类型限定符：

CONST

属性 // 仅顶点。可变

统一

类型标识符：VOID

FLOAT INT

BOOL

*VEC2 VEC3
VEC4 BVEC2
BVEC3 BVEC4
IVEC2 IVEC3
IVEC4 MAT2
MAT3 MAT4*
*采样器1D 采样器2D 采样器3D 采样器立方
体采样器1D阴影 采样器2D阴影
结构体指定符 类型名称*

结构体标识符:

结构标识符 左大括号 结构声明列表 右大括号 结构 左大括号 结构声明列表 右大括号

结构声明列表: 结构声明

结构声明列表 结构声明

结构声明

类型标识符 结构声明列表 分号

结构声明列表 结构声明项

结构声明列表 逗号 结构声明项

结构声明项: 标识符

 标识符 左方括号 常量表达式 右方括号

初始化器:

 赋值表达式

声明语句: 声明

语句:

 复合语句 简单语句

// 语法注释: 无标记语句; 不支持'goto'。

简单语句: 声明语句 表达式语句 选择语句 迭代语句

跳转语句

复合语句: 左大括号 右大括号

 LEFT_BRACE 语句列表 RIGHT_BRACE

statement_no_new_scope: compound_statement_no_new_scope simple_statement

compound_statement_no_new_scope: LEFT_BRACE RIGHT_BRACE

 LEFT_BRACE 语句列表 RIGHT_BRACE 语句列表:

语句

语句列表 语句

表达式语句; 分号

表达式 分号

选择语句;

IF 左括号 表达式 右括号 选择残余语句

selection_rest_statement: 语句 ELSE 语句 语句

// 语法注释：不支持'switch' 语句。

条件;

表达式

完全指定类型 标识符 等于 初始化器

迭代语句;

WHILE 左括号 条件 右括号 语句 (无新作用域) DO 语句 WHILE 左括号 表达式 右括号 分号

FOR 左括号 *for_init_statement* *for_rest_statement* 右括号 *statement_no_new_scope*

for_init_statement: 表达式语句 声明语句

conditionopt:

条件

/* 空 */

for_rest_statement: *conditionopt SEMICOLON*

条件选项 分号 表达式

跳转语句:

CONTINUE 分号 BREAK 分号 RETURN 分

号

RETURN 表达式 分号

DISCARD; // 仅限片段着色器。

// 语法注释：无 goto 指令。不支持跳转语句。翻译单元。

外部声明 翻译单元 外部声明

外部声明 函数定义 声明

函数定义:

函数原型复合语句无新作用域

10 问题

- 1) 在这些可编程处理器上运行的程序，应该称为着色器还是程序？

讨论：着色器（shader）符合RenderMan和DX8中的常用术语。但有人认为“着色”（shading）带有色彩运算的含义，与顶点运算（vertex operation）不符。RenderMan与DX8均未作此区分。采用“着色器”作为图形处理管道中某环节运行程序的通用术语，似乎更为明智。

2001年10月12日决议：采用术语“着色器”。

注：*Shader*指单个独立编译单元，*Program*指一组关联的着色器集合。

于2002年9月10日关闭。

- 2) 是否应设置独立的可编程单元来执行像素传输操作？

讨论：我们最初设想过独立像素着色器来执行像素与成像操作。但深入思考后发现，极少有人会将其实现为独立功能单元，而更倾向于在片段着色器中隐式处理。OpenGL将像素操作与片段操作视为互斥关系，因此共享处理单元是自然的实现方式。强行采用与实际情况相悖的抽象概念，除了增加工作量外，似乎反而会成为阻碍。

决议于2001年10月12日通过：否，该片段处理器将同时用于处理几何数据和像素数据。

2002年9月10日关闭。

- 3) 着色器是否应被允许对它们所替代的固定功能进行子集化？

讨论：定义允许子集化的接口将带来大量复杂性。编写实现完整图形处理管道的着色器并不困难。

决议于2001年10月12日通过：否，着色器不能仅实现其替代的固定功能管道的部分功能。若着色器需以某种方式改变光照效果，则必须同时处理其他项目。提供完整实现OpenGL固定功能管道的示例着色器将有所帮助。

2002年9月10日关闭。

- 4) 是否应在OpenGL之上构建更高层次的着色语言，而非将其设计为OpenGL的嵌套组件？

讨论：当前设计中，着色语言已集成于OpenGL，仅为前述状态控制管道提供替代方案。斯坦福方案则是将着色语言叠加在OpenGL之上。这种做法存在若干优缺点，通过对比差异将得以显现。

斯坦福方法采用更高的抽象层次。这有助于编写某些类型的程序，其中抽象概念与问题领域相匹配。例如，将光源和表面视为抽象实体能简化某些3D图形操作，然而OpenGL如今正被用于视频和图像处理领域，而这种抽象在此类场景中基本无关紧要。同样地，许多游戏已摒弃传统照明方式，转而采用纹理（光照贴图）实现效果。

在可编程OpenGL之上叠加更高层次的抽象层，语言本身及绑定机制均不存在任何限制。我们同样希望保持OpenGL整体抽象层级的现有水平。

斯坦福方案还支持不同的计算频率。通过采用更高层次的抽象——由单个程序完整定义当前图形操作——编译器得以将需要分别在基元组级、基元级、顶点级和片段级运行的部分分离出来。编译器因此能生成适用于CPU、顶点处理器和片段处理器的代码。这种实现方式显然比让程序员指定管道各部分的运行程序更为复杂（尽管斯坦福语言仍需某些提示），但由于编译器具备整体视图，这确实使硬件虚拟化更为简便。

这种方法的主要缺点在于，它迫使OpenGL进行更侵入性的改动，以支持对基元、顶点和片段操作的清晰划分。许多核心OpenGL功能已被替换或不可用，无法在自定义片段着色器中使用标准OpenGL变换和光照操作（反之亦然），也无法让单个顶点着色器驱动多个片段着色器。当前方案的优势在于保留了OpenGL 1.4的操作体验，并在从固定功能向全可编程性过渡期间实现了优雅的混合匹配机制。

这并非对斯坦福大学工作的批评，因为他们别无选择，只能基于OpenGL进行分层开发。

于2001年10月12日决议：OpenGL着色语言应内置于OpenGL中，而非作为附加层存在。同时指出，若未采纳此方案，OpenGL仍应具备标准着色语言，故本文件仍具效力。因此，此议题并非针对本文件，而是针对OpenGL API本身。

2002年9月20日关闭，因已移至API问题列表。

5) 着色模型是否应作为由片段处理器替代的固定功能片段处理组件？

讨论：着色模型在Gouraud着色与平面着色间进行选择，将其纳入由片段着色器替代的功能范畴似乎合乎逻辑。平面着色涉及对原始类型（针对触发顶点）的认知，而这不应属于片段着色器的职责范围。片段着色器可始终假设颜色经过插值处理且采用平面着色模型，此时颜色渐变的设置计算可将渐变值设为零。

决议于2001年10月12日通过：否，着色模型不会被片段处理器的可编程功能所取代。

2002年9月10日关闭。

6) Alpha测试是否可编程？

讨论：片段着色器具备清除片段的功能，因此可实现类似透明度测试的效果。然而OpenGL管道规范要求透明度测试应在覆盖率修改透明度值之后进行。我们不希望在片段着色器中执行覆盖率计算，因此透明度测试仍需在外部完成。若用户愿意在自身程序中进行覆盖率测试后再执行透明度测试，则可自行实现。

保留在外部。若用户愿意在自身程序中于覆盖率计算前执行Alpha测试，则可自行实现。

2001年10月12日决议：允许应用程序在片段着色器中执行Alpha测试，但前提是该操作必须发生在覆盖率计算之前。

于2002年9月10日关闭。

7) *Alpha混合是否可编程？*

片段着色器可通过内置变量`gl_FBCOLOR`、`gl_FBDDEPTH`、`gl_FB_STENCIL`和`gl_FBDATUM`读取当前帧缓冲区内容。借助这些功能，应用程序可实现自定义混合算法、模板测试等操作。但需注意，帧缓冲区读取操作可能导致性能显著下降，因此强烈建议应用程序尽可能使用OpenGL的固定功能实现这些操作。若能实现每个片段在空间和时间维度上的独立处理，则片段着色器（及顶点着色器）的硬件实现将大幅简化并加速。通过允许在片段处理过程中执行读取-修改-写入操作（如Alpha混合所需），我们引入了时空关联性。这种关联性因性能需求而必须采用深度流水线、缓存机制及内存仲裁，从而增加了设计复杂度。诸如渲染到纹理、帧缓冲区复制到纹理、辅助数据缓冲区及累加缓冲区等方法，可实现可编程Alpha混合的大部分（若非全部）功能。同时，高级着色语言与自动资源管理机制已显著减少（或至少抽象化）了多通道渲染的需求。

已解决于2001年10月12日：是的，应用程序可以执行Alpha混合，尽管相较于使用固定功能混合操作可能存在性能损失。

2002年7月9日重新开启：本议题与尚未解决的议题(23)存在关联，故本议题亦应保持开放状态。

另一种方案是创建扩展功能，使其比当前的Alpha混合更具灵活性，但仍被视为固定功能。

决议：问题23已决议允许帧缓冲区读取，故本议题再次决议允许Alpha混合，但须遵循上述限制条款。

2002年12月10日重新开启：问题23重新决议为禁止帧缓冲区读取。决议：否，应用程序无法执行Alpha混合，因其无法读取Alpha通道。2002年12月10日关闭。

8) *是否应将该语言定义为可在现有硬件上实现的形式？*

讨论：当前一代硬件确实具备一定的可编程性。定义一种既适用于当今硬件又适用于未来硬件的语言似乎是可取的。

2001年10月12日决议：我们力求使着色语言具有前瞻性，并将其定位在我们认为硬件在一两代内可达成的水平。我们避免为支持现有硬件而添加特性（如小精度浮点数据类型或隐式裁剪）或降低语言复杂度（移除循环和函数），因为这属于倒退。虽然现有硬件可运行着色器的有限子集，但应用程序难以通过可移植方式判断着色器能否运行或产生可接受的结果。总体而言，此处的决策是为未来几年硬件发展设定一个奋斗目标。

于2002年9月20日关闭。

9) 是否应放弃该语言的预处理器概念?

讨论：我们可通过使用`if(false)`替代`#ifdef`，并依赖编译器移除无法执行的代码。C++规范似乎在淡化`#ifdef`的使用，但我们仍保留该特性，因其作为通用表达方式更易于代码可读性，且可在语法不支持`if(false)`的场景中使用。

是否需要供应商专属的预定义`#define`来规避编译器问题？理想状态下不应采用，因其为扩展功能留有后门；但现实中差异在所难免。我们已观察到着色器编写者使用`#define`提升代码可读性的案例（例如`#define MVP gl_ModelViewProjectionMatrix`）。

2001年10月12日决议：否，应保留预处理器。支持的预处理器指令包括：`#ifdef`、`#ifndef`、`#undef`、`#else`、`#endif`、`#pragma`、`#define` 标记（无参数）以及`#error`。

新增议题(55)以处理额外预处理指令。于2002年9月10日关闭。

10) 纹理分量字段是否应命名为s、t、p和q?

讨论：曾考虑过其他替代方案来重命名纹理r，但因可能造成更多混淆或增加错误风险而被否决。A) 使用红、绿、蓝、透明度来选择颜色分量。这会使后文所述的分量组机制过于繁琐。

B) 将颜色或纹理组件名称的首字母大写。C) 省略颜色或纹理名称。我们不愿放弃向量重要应用场景中某种记号法带来的便利性。D) 将颜色组件顺序改为bgra，使两个r组件对齐。这种颜色顺序与OpenGL规范相悖，将导致现有API值与着色器使用值映射时产生诸多混淆。

2001年10月12日决议：采用s、t、p、q作为纹理组件字段名称确为最佳方案。

2002年9月10日关闭。

11) 是否应分别使用两个独立的活动片段着色器来处理背面和正面情况?

讨论：若用户指定两个片段着色器（分别处理朝前片段和朝后片段），系统可自动运行对应的着色器。此方案虽能提升着色效率，却迫使用户维护两个程序（其中大部分代码可能高度重复）。若实现方案希望优化此场景，编译器可透明地完成此转换。

2001年10月12日决议：否，应使用单一着色器处理所有几何体的正向与背向面。

2002年9月10日关闭。

12) 内置函数是否需要在返回类型之外存在更多差异?

讨论：仅考虑过返回类型不同的函数重载。然而，编译器的初步工作表明，当表达式中的返回类型明确但被嵌套时，该机制会严重增加语义分析的复杂度。这种复杂性可能得不偿失（这或许正是C++不允许此类重载的原因）。

若再考虑需要为程序员提供新语法以消除歧义，则直接为不同返回类型的函数赋予不同名称显然更为简便。

2002年2月25日决议：内置函数必须通过返回类型以外的特征区分。2002年9月10日结案。

13) 噪声函数如何定义才能确保不同实现之间行为一致？

讨论：噪声函数在RenderMan中极为实用，广泛应用于多种着色技术。其规范制定（及兼容性测试）面临的难题在于：完全有效的噪声函数可能产生截然不同的结果。OpenGL避免了对运算进行过于严格的规范——不同实现不必产生像素级精确的结果，这为实现者在精度/性能/成本之间提供了权衡空间，同时也避免了对细节的过度规定。噪声函数的发明者Perlin已意识到标准化噪声函数的必要性（正如人们期望sin函数具有统一行为），并记录了相关构想。或许这应作为强烈推荐的实现方案。

此议题与议题(36)基本相同。

决议于2002年9月19日：无需对噪声进行具体实现，但规范将尝试定义噪声函数，以确保不同实现之间能获得相似的结果。

2002年9月19日关闭：

14) 是否允许字段使用数字选择符（例如`foo.2`）？

讨论：此举违反标识符以字母开头的常规约定。它降低语言纯粹性，增加词法分析难度，并限制数字表达方式。

2002年9月10日决议：否，应按议题(16)建议修改语言规范，禁止使用数字选择器。

2002年9月10日关闭。

15) 是否应允许对向量元素进行元素交换的字段？

讨论：这似乎是语言中过于复杂的部分，其额外功能完全可通过语言其他特性轻松实现。另一方面，置换操作在底层汇编语言中本就存在。第三点考虑：这或许只是硬件的特性，本不该/无需在高级语言中体现。第四点补充：已有实用示例证明置换的价值，且编译器实现支持并不困难。

2002年9月10日决议：元件交换被认定为有用的语言特性，将予以保留。

2002年9月10日关闭。

16) 是否应提供间接引用向量或矩阵的方法？

讨论：问题(14)与本问题可通过添加`[]`作为数值索引向量的方式同时解决。此时应使用`foo[2]`而非`foo.2`来解决问题(14)，并采用`foo[x]`作为间接引用。数值表达方式将与C语言一致。

2002年9月10日决议：同意，应按上述讨论建议允许对向量和矩阵进行间接引用。

2002年9月10日关闭。

17) *gl_Position*等当前“只写”变量是否应支持读取？

讨论：这纯粹是编译器特性，不涉及硬件支持。缺乏此特性会导致编码繁琐。编译器可在必要时使用临时变量存储中间值，此举亦能使程序代码更简洁。

另一方面，对于着色器编写者而言，只读输入和只写输出的API模型可能更为简洁。

已解决于2002年9月19日。是的，允许“只写”变量具备可读性。已关闭于2002年9月19日。

18) 性能/空间/精度提示应如何提供？

讨论：目前基本达成共识的是针对变量：`varying`表示视差校正，而`fast varying`表示若能节省时间则可取捷径。或许我们可以为此定义一个#pragma指令。该机制或许也可应用于其他领域。

决议：性能/空间/精度提示及类型不会作为语言标准组成部分提供，但将保留相关操作的保留字。

结束日期：2002年11月26日。

19) 是否应添加内置函数“*lookup*”？

讨论：纹理不仅可作为纹理使用，还可充当查找表。主要区别在于查找表会为返回值关联类型。

决议：应将查找函数作为内置函数添加，以便着色器能明确返回值的类型。将新增如i8texture3之类的函数，表示查找三个8位整数。此类结构仍称为纹理，因其预期共享纹理资源。独立于纹理资源的通用查找表功能将推迟至1.1版本实现。

已关闭 2002年10月22日。

注：这些功能后来在适配OpenGL 1.4时被移除，因该版本不支持此类函数操作的纹理类型。

20) 是否应添加大于16位的整数？

讨论：着色语言的设计理念在于避免因冗余功能而给硬件设计者增加负担。整数在循环计数器和数组索引中具有实用性且可能更高效。为满足此类场景的效率需求，16位整数作为折中方案被纳入语言规范。浮点数的尾数部分可用于整数运算，因此硬件设计者无需在浮点运算单元之外额外配置完整的整数运算单元。若此为决定该问题的关键因素，则可将整数定义为23位（至少在处理器内部运算时如此），因这恰是IEEE FP32浮点数的尾数位宽。另据参考，Renderman完全不支持整数类型。

决议：当前存在硬件限制要求将整数限制为16位，故将遵循此规格。已关闭：2002年11月19日。

21) 是否应添加整型向量或（局部变量）整型数组？

讨论：例如，在议题(19)中提出的查找函数可返回3个整数值。着色器不仅包含浮点算法，还涉及表查找、间接寻址及其他通用算法计算。该语言无需直接映射硬件。但另一方面，为支持此功能需添加`ivec2`、`ivec3`和`ivec4`类型，或允许使用整型局部变量数组。

决议：同意。将添加整型向量支持。关闭日期：2002年11月5日。

22) 是否应支持递归？

讨论：可能没有必要，但这是又一个基于直接映射硬件来限制语言功能的例子。一种想法是递归将使光线追踪着色器受益。另一方面，许多递归操作也可以通过用户使用数组管理递归来实现。RenderMan不支持递归。如果证明有必要，可以在以后添加。

已解决于2002年9月10日：实现方案无需支持递归。已关闭于2002年9月10日。

23) 是否应允许片段着色器读取帧缓冲区中的当前位置？

讨论：在考虑多采样技术时，可能难以对此进行合理规范。硬件实现者要实现此功能也可能相当困难，至少在保持合理性能的前提下如此。但这是着色语言白皮书初版发布后用户最迫切要求的两项功能之一。独立软件供应商持续向我们反馈，他们需要此功能且必须具备高性能。

决议：允许。但需对性能影响提出强烈警告。

于2002年12月10日重新开放。人们过度担忧其对绩效的影响及实施的不切实际性。

2002年12月10日关闭。

24) 是否需要在语言中添加功能，使程序能在编译时完成编译，从而无需为OpenGL状态变更保存多个编译版本？

讨论：强烈期望实现能在编译时生成正确代码，避免因OpenGL状态后续变更（例如纹理贴图属性需在执行时才能确定）而产生多版本编译或后期重新编译的情况。这或许是语言规范需要更多提示的领域，也可能通过硬件演进解决相关问题。

决议：此议题确立为OpenGL着色语言的通用设计目标...其他问题应以确保着色器生成的目标代码独立于其他OpenGL状态为解决方向。

关闭日期：2002年11月5日。

25) 是否应添加接受生成器类型和标量参数的最小值和最大值函数，以匹配钳位语义？

决议：应添加此类函数。已关闭：2002年9月22日。

- 26) 是否应按功能（如光照着色器、表面着色器、变换着色器、纹理生成着色器等）而非当前提案中基于硬件的方法（顶点着色器和片段着色器）来划分可编程性？

2001年12月7日决议：否。顶点着色器与片段着色器的概念更契合OpenGL作为硬件中心化API的特性，且在评审中获得积极反馈。

2002年9月10日关闭。

- 27) 纹理单元应采用关键字还是数字进行标识？

讨论：对于内置的纹理访问函数，纹理单元目前以数字形式指定。是否应改为使用关键字定义？当前认为在某些情况下，以程序化值而非关键字指定纹理更为便捷。

此议题实为议题(51)的组成部分。

2001年12月7日决议：纹理单元仍以数字形式指定。在特定情况下，使用程序化值而非关键词指定纹理更为便捷。

2002年7月12日重新开启：需进一步讨论实际便利性。决议：作为问题51解决。

2002年10月22日关闭。

- 28) 全局变量是否自动初始化？

决议于2001年12月7日：否，全局值不会自动初始化。不过，实现方案可考虑将自动初始化作为调试模式选项予以支持。

2002年9月10日关闭。

- 29) 该语言是否应支持位运算？

2001年12月7日决议：语言本身支持位运算。在特定可编程单元（打包/拆包处理器）中，这些操作至关重要。但为限制各可编程单元的复杂度，顶点处理器和片段处理器仅支持布尔运算，而不支持通用位运算。此举旨在避免在现有浮点运算能力基础上，额外要求这些处理器具备完整的整数运算功能。

2002年7月12日重新开放：某些位运算非常实用，且难以通过浮点运算轻松模拟。例如，应用程序可能将数据的多个字段映射到纹理组件中，并使用位运算符提取这些值（例如，使用16位亮度纹理中的12位存储强度，剩余4位存储不透明度）。赋予着色器执行此类提取的能力，比定义新纹理格式更为可取。

另一种实现方式是提供内置函数从整数中提取位域，这正是位运算符的预期用途之一。

此议题与第90号议题相关联。在缺乏整数纹理支持的情况下，解决此议题的必要性降低。决议：位运算支持功能推迟至未来版本实现。

已关闭：2002年12月10日。

- 30) 内部计算是否必须采用32位浮点精度？还是允许实现根据需求使用更高或更低的精度进行计算？

讨论：此问题与问题(33)和问题(68)相关。

决议：浮点数要求必须遵循OpenGL规范1.4版第2.1.1节的规定，此要求已隐含其中，无需额外说明。

关闭日期：2002年11月26日。

31) 能否在片段着色器中覆盖计算得到的LOD值或偏移量？

2001年10月12日解决：片段着色器可通过提供偏移值来调整计算出的LOD。为此提供了带LOD参数的内置纹理访问函数。

已关闭：2002年9月19日。

32) 插值后的值是否符合透视校正？

已解决于2002年6月3日：是的，定义为可变的变量在视角上是正确的。已关闭于2002年9月10日。

33) 是否应支持精度提示（例如使用16位浮点数或32位浮点数）？

讨论：计算采用单一数据类型可极大简化语言规范。即使允许实现方案对精度降低值进行静默提升，若着色器编写者无意中依赖了精度降低运算符的截断或循环语义，仍可能导致行为差异。定义精度降低类型集只会迫使硬件为兼容性而强制实现这些类型。

编写通用程序时，程序员早已不再纠结使用字节、短整型或长整型计算的效率差异，我们也不希望着色器编写者陷入类似的顾虑。支持缩减精度数据类型的唯一短期效益，是可能使现有硬件更高效地运行部分着色器子集。

此议题与议题(30)及议题(68)相关。

决议：性能/空间/精度提示和类型不会作为语言的标准组成部分提供，但为此保留的保留字将予以保留。

已关闭：2002年11月26日。

34) OpenGL着色语言的设计是否应支持非实时性质的着色器？

决议于2002年9月19日通过：是，语言设计应考虑非实时性质的应用场景。

已关闭于2002年9月19日。

35) 是否应向着色语言添加点、法线、颜色等额外类型？

决议于2001年10月12日通过：不，不应添加此类类型。现有的泛型向量类型已能支持所有需求，无需为语言新增类型。

2002年9月10日关闭。

36) 是否应允许各方对平滑步长和噪声实现不同方案，还是应统一规范并强制执行共同实现？

讨论：此议题与议题(13)基本相同。平滑步进的定义已足够完善。OpenGL并非像素级精确，其平滑步进定义已达到规范所需的精度要求。

2002年9月19日决议：OpenGL着色语言存在丰富的实现空间。即便在当前OpenGL环境下，不同硬件生成的图像也不必完全一致。规范应确保各实现能产生高度相似（而非完全相同）的结果。

OpenGL着色语言的兼容性测试是OpenGL ARB未来需要解决的问题。

会议进一步决定在规范中添加noise()函数的源代码说明。但该项工作尚未完成。2002年9月19日结案。

37) 片段着色器中用于“销毁”片段的功能应采用关键字还是内置函数实现？

讨论：kill 类似于 break 和 continue 的流程控制指令。它并非应被用户函数覆盖的函数。布尔表达式可在 kill 关键字之前的 if 语句中进行评估。一旦执行 kill 就无需继续处理，因此没有必要将其伪装成函数调用。`kill(boolExpr);`作为`f(boolExpr) kill;`的快捷写法，节省的代码量微乎其微。

2002年4月15日已解决：片段着色器中“终止”片段的功能应采用关键字实现。

2002年9月19日关闭：

38) 顶点着色器内部是否应提供内置纹理与噪声函数？

讨论：众多用户请求支持位移贴图功能。该需求可通过允许顶点着色器与片段着色器均调用内置噪声和纹理函数来实现。硬件层面的实现方案是让编译器将顶点着色器拆分为序言、纹理/噪声访问及尾言三个部分。前言部分由顶点处理硬件执行，纹理/噪声访问则由片段处理硬件完成。中间结果将通过顶点处理硬件反馈执行后记，随后传递至片段着色器进行片段处理。另一方面，应用程序也可选择在主机CPU上完成全部操作。

决议：是的，纹理和噪声函数也应在顶点着色器内部提供。

已关闭 2002年10月22日。

39) 是否应规定：变量插值值应通过在片段中心采样确定？

讨论：此问题涉及多重采样机制，需进一步研究。

决议：本条款遵循GL规范1.4版第3.2.1节所述规则。

已关闭：2002年11月26日。

40) 是否应在语言中支持无符号整型用于顶点和片段处理？

讨论：此议题与议题(29)相关。若允许位运算符，则需提供指定有符号或无符号整数的方式。当前无符号整数仅定义于封装/解封装着色器语言，未涵盖顶点与片段语言。

决议：鉴于整数已确定采用16位精度（含符号位），无需引入无符号整数类型。

关闭日期：2002年11月26日。

41) *gl_FrontMaterial* 和 *gl_BackMaterial* 是属性还是统一变量？

讨论：当前规范将这些定义为属性数组，但规范同时规定属性不允许使用数组。若要将其视为统一变量，则可保留数组形式。
否则应修改名称与定义以避免使用数组（即为每个属性赋予唯一名称）。

2002年9月19日决议：将作为统一变量处理。今后更鼓励应用程序使用用户自定义属性，若需在每个顶点修改这些属性。

2002年9月19日关闭。

42) 是否应提供机制，指定顶点着色器生成的变换位置在固定功能管道中保持不变？

讨论：此功能由独立软件供应商提出。若缺少该功能，在许多图形架构中，可能无法精确匹配使用顶点着色器渲染的几何体与使用固定功能路径渲染的几何体。

一种可能的解决方案是修改规范中关于所有顶点着色器必须写入内置变量 *gl_Position* 的要求。取而代之的是：若顶点着色器未写入 *gl_Position*，则顶点位置将按固定功能管道不变的方式进行变换；若顶点着色器写入了 *gl_Position*，则最终位置可能保持不变也可能不保持不变。

但此方案增加了着色器编写者可能无意中遗漏 *gl_Position* 写入的风险——该操作现在不会报错，而是会生成不变变换 *gl_Vertex*。因此提出以下替代方案集。鉴于我们拥有内置函数，采用内置函数可能是实现不变变换需求的简洁解决方案。以下三种内置函数均可实现 *gl_Vertex* 的不变变换。方案(A)对 RenderMan 着色器编写者最为熟悉（除命名空间外）。方案(B)和(C)仅提供不变变换功能，其中(B)允许显式指定输入参数，而(C)默认输入 *gl_Vertex*。

原始建议的解决方案以及这些替代方案均能解决 ISV 请求，但方式各异。

(A) 内置函数：

`genType transform([mat xform,] genType coord)` 若指定矩阵，则返回 *xform*coord*。否则，将 *coord* 转换为固定函数方法不变量。

示例：

```
// 通过MVP变换，不保证与固定函数不变量一致。gl_Position = transform(gl_ModelViewProjectionMatrix, gl_Vertex);

// 转换保持不变，采用固定函数模式。
```

```
gl_Position = transform( gl_Vertex );
```

(B) 与(A)相关，但无可选矩阵参数：

genType transform(genType coord) 将坐标转换为固定函数不变量示例：

```
// 固定函数下的不变变换。gl_Position = transform(gl_Vertex);
```

(C) 与(B)相关，但无参数，重命名函数：

vec4 fixedtransform() 输出不变量，采用固定函数模式，隐式输入为 gl_Vertex。示例：

```
gl_Position = fixedTransform();
```

解决方案：采用上述方案C。已关闭 2002年10月22日。

43) gl_FB*内置导数函数的定义是什么？

讨论：一个简短的片段着色器示例最能说明问题。

```
void main(void)
{
    gl_FragColor = dfDy(abs(gl_FBColor));
}
```

早期白皮书允许在片段处理器中进行全局帧缓冲区读取。OpenGL通常仅规定光栅化生成的片段数量，而不规定片段生成的顺序。因此在片段处理器中进行全局帧缓冲区读取可能导致未定义行为。

后续白皮书仅允许在片段处理器内进行受限的帧缓冲区读取（即片段窗口坐标为xw, yw处的像素）。因此问题在于：内置导数函数在概念上是否隐含要求对帧缓冲区进行普遍读取（至少在片段窗口坐标为xw, yw的像素周边区域）？在此语境下“在任意给定时刻”的含义是什么？

可能的解决方案：

- a) 禁止在片段处理器中执行 gl_FB* 读取操作。（此方案与问题(23)存在关联。）
- b) 当表达式包含 gl_FB* 父元素时，内置导数函数定义为未定义（某些情况下，条件语句或循环体内的内置导数函数亦定义为未定义）。

被否决的解决方案：

- c) 明确规定OpenGL对片段进行光栅化的顺序。

决议：gl_FB* 已被移除。问题 23 已重新开启，并以禁止帧缓冲区读取为由关闭。

于2002年12月11日关闭。

44) 代表当前OpenGL状态的统一变量应仅限特定处理器访问，还是对所有处理器开放？

讨论：当前规范偏向顶点光照与片段着色器。现行规范中，内置统一变量表示的OpenGL状态仅对特定处理器可见（例如光照状态仅对顶点处理器可见）。这导致片段着色器使用OpenGL状态进行光照计算时面临不必要的困难。规范应保持中立，不预设哪些着色器需要访问哪些内置统一OpenGL状态。

决议：封装为统一变量的OpenGL状态应可供任何处理器访问。

已关闭 2002年10月22日。

45) OpenGL状态的命名规范是否应与ARB_vertex_program扩展采用的规范保持一致？

讨论：当OpenGL着色语言定义`gl_ModelViewMatrix`指向特定OpenGL状态时，ARB_vertex_program扩展使用`state.matrix.modelview`。为保持一致性，是否应采用相同命名规范？

OpenGL着色语言引用GL状态的约定制定于ARB顶点程序扩展因知识产权问题及缺乏共识而尚未明确可行的时期。ARB_vertex_program（及ARB_fragment_program）的状态绑定可能导致部分人误解其语法为C类语言的结构体（在汇编类语言中此类混淆风险较低）。且ARB_vertex_program与ARB_fragment_program均将状态封装为`vec4`类型。

在C类语言中，此类封装的需求较低。

2002年8月13日决议：否，规范不必统一。目前尚无足够动力推动此命名变更。

2002年9月10日关闭。

46) 在对象轮廓处，一般导数的预期行为是什么？

讨论：若用于滤波宽度或LOD计算，则必须采用局部瞬时导数。这意味着任何实现都不能依赖邻近片元计算导数——因为始终存在仅触及单个片元的对象（故无有效邻域）。

决议：详见论文中关于导数的章节。于2002年12月4日结案。

47) 导数函数的命名是否应更接近Renderman中的惯例？

讨论：导数函数的命名与RenderMan确立的先例存在一定冲突——其中`Du(f)`和`Dv(f)`分别计算 df/du 与 df/dv 。`dPdu`和`dPdv`虽可能更准确，但功能上等同于`Du(P)`和`Dv(p)`，其中`P`是表示采样点三维位置的内置变量。我们至少应考虑将OGL2导数函数命名为`Dx()`和`Dy()`。

决议：名称改为`dFdx`、`dFdy`。2002年12月4日结案。

48) 是否应添加`klPdz`（或`Dz`）函数？

讨论：若需求导 df/du 和 df/dv ，则需 $dPdz()$ 或 $Dz()$ 以避免物体轮廓处的奇异点。

决议：此项未予添加。于2002年12月4日结案。

49) 着色语言是否应包含结构体？

讨论：着色语言应支持结构体。结构体提供了一种将数据分组以创建抽象数据类型的简洁方式，既方便开发者使用，又被C语言及其他泛型编程语言所支持。

另一方面，目前尚无充分理由证明必须将结构体纳入语言规范。若将此功能推迟至后续版本再行添加，有助于语言规范更早定稿。但若最终纳入结构体，则可通过结构体定义照明状态。Vital Images (ISV) 公司明确表示希望语言支持结构体。

已解决于2002年9月19日：着色语言将包含结构体。已关闭于2002年9月19日。

50) 顶点处理器是否应以支持曲面细分的方式定义？

讨论：几何LOD与曲面处理问题极其复杂且持续演进，任何比通用可编程CPU更简单的硬件都难以胜任。任何基元选择都将引发激烈争议。而主流曲面基元（如折痕细分曲面或修剪NURBS）在硬件中难以实现。

若从性能角度审视问题，LOD的生成通常并非症结所在——新LOD无需频繁生成，仅当几何体显著靠近或远离摄像机时才需更新。真正的挑战在于表面交互操作的动画场景。此时拓扑结构保持不变，因此可通过高效方式解决。我们只需为LOD中的每个顶点存储一个参考列表，其中包含控制值（CVs）和权重因子。

这段简洁代码可适配所有曲面类型：

```
for(i = 0; i < vertex_count; i++)
{
    x = 0;
    y = 0;
    z = 0;
    for(j = 0; j < *影响列表长度; j++)
    {
        索引 = *索引数组++; 值 = *值数组++;
        x += value * control_vertex_array[index].x;
        y += value * 控制顶点数组[index].y;
        z += 数值 * 控制顶点数组[索引].z;
    }
    表面顶点数组[i].x = x; 表面顶点数组[i].y = y; 表面顶点数
    组[i].z = z;
```

```
影响列表长度++;
}
```

此功能可集成至顶点着色器以实现最大灵活性，但这意味着顶点着色器必须具备随机访问数据数组的能力。

决议：推迟至本规范的未来版本。已关闭：2002年10月22日。

51) 语言是否应提供机制区分位置无关变量与位置相关变量？

讨论：有意见提出“是否应在语言中暴露SIMD语义？”回应：

- 真正引入的是位置无关性。
- 现有的‘uniform’和‘varying’指令已引入该概念，本提案仅是完善其实现。
- 大多数硬件将从中受益。

可行性：编译器能够通过充分的数据流和控制流分析，找出所有可能导致位置无关变量被赋值的路径。它可能会发现额外路径，但绝不能遗漏任何实际路径。

由此，编译器可验证位置无关变量是否仅取位置无关值。极少数情况下可能错误判定变量为位置无关，但此类错误通常仅出现在退化代码中。

全局统一变量。要求编译器进行跨函数数据流分析（尤其跨不同编译单元）过于苛求。因此，全局读写位置无关变量的概念既不可行也不被提议。由此，这些‘uniform’用法之间不存在冲突。

输出统一参数。与全局统一变量存在相同问题。全局统一变量和参数必须为只读。

过去曾有替代方案：使用‘int’类型向编译器提示值的位置无关性，例如循环索引、纹理ID、数组下标等。但这会带来麻烦——基于浮点数的控制流可能使提示失效，导致编译器负担与无提示时无异。同时这削弱了‘int’类型的实用性，迫使其遵循拟议的‘uniform’语义。这种做法过度捆绑了本应独立的概念。

此问题与Issue (27)相关。

决议：使用‘uniform’限定符标识局部作用域变量、函数返回值及函数参数为位置无关。局部‘uniform’变量不可写入由位置衍生的值。声明返回‘uniform’的函数只能返回非位置衍生的值。

若编译器通过静态可识别路径发现存在将位置依赖派生值赋予位置无关变量的操作，则可能发出警告。即当变量声明为uniform或传递给uniform参数时，若编译器无法证明该变量始终为位置无关，则会报错。

已关闭 2002年10月22日。

52) 着色语言的资源限制应如何定义？

讨论：已探讨多种方案。关键考量在于制定能保障应用程序可移植性的规范（例如独立软件供应商无需支持多种渲染后端即可运行于不同硬件平台）。供应商显然更倾向于规范明确规定：着色语言实现方应确保所有有效着色器均可运行。

决议：易于计数的资源（顶点处理器统一变量数量、片段处理器统一变量数量、属性数量、变量数量、纹理单元数量）将具有可查询的限制。应用程序需在这些外部可见的限制范围内运行。着色语言实现负责对难以计数的资源进行虚拟化处理（如最终可执行文件中的机器指令数量、临时寄存器使用数量等）。

已于2002年10月29日作为API问题清单的一部分关闭。

53) 若坐标空间通过非线性变换分离，用户裁剪平面如何处理？

讨论：着色语言规范依赖于GL裁剪的标准定义。只要坐标空间仅通过线性变换分离，这种方式即可正常工作，但着色语言也取消了这些限制。

建议解决方案：采用NVIDIA某些提案中针对ARB_vertex_program提出的“裁剪坐标输出”方案（该方案在规范最终确定前已被移除）。此方案提供完全可编程的用户裁剪功能，既不依赖程序语义也不依赖任何分析结果，且不会导致多数情况未定义。

决议：明确规定用户裁剪平面仅在线性变换下有效。非线性变换下的行为目前未定义。

已关闭 2002年10月22日。

54) 全局像素操作（如直方图、最小\最大值处理）如何实现？

新增于2002年9月10日。

讨论：当前规范仅允许访问当前位置的像素片段（尽管根据问题(23)所述，此点尚待讨论）。明确禁止任何需要访问帧缓冲区中其他像素片段位置或输入数据流的操作。着色语言将如何提供支持全局像素操作（如直方图和最小\最大值计算）的功能？

期望实现能对多个片段进行操作的程序（其理念类似于运行生成新几何体的顶点程序，参见Issue (50)）。

决议：推迟至本规范未来版本解决。已关闭：2002年10月22日。

55) 除已定义指令外，预处理器是否应新增指令？

2002年9月10日新增。

讨论：可在语言规范中添加若干预处理指令，例如：#if、#elif、#include、#define token(..)（带参数）、##（令牌拼接）、#line、#error、#（单独使用）、#（将令牌转为字符串）、defined(token）（及所有其他运算符，如&&、|、+等运算符），以及预定义宏如__DATE__、FILE__。

特别是添加#if指令将有助于处理版本号、日期等信息，但这需要引入||、&&、>、<、!等运算符。

2002年9月24日决议：shading语言预处理器将基本具备C预处理器的全部功能，但不支持#include指令且不包含基于字符串的指令。

2002年9月24日关闭。

56) 如果规范规定不支持递归，那么实现支持递归是否属于错误？

2002年9月10日新增。

讨论：此问题与问题(22)相关。若声明递归（或其他功能）不被支持，实现支持该功能是否构成错误？或许规范对这类事项保持沉默更为妥当，以便日后可作为扩展或标准组成部分优雅地添加。

决议：语言中普遍存在编译器无法检测的非规范程序。可移植性仅适用于规范程序。递归检测即为此例。语言规范可规定规范程序不得递归，但编译器无需强制检测递归可能性。

已关闭：2002年11月29日。

57) 应用程序是否应存在标准化的调试模式调用方式？

2002年9月10日新增。由问题(67)涵盖。

于2002年9月24日关闭。

58) 该语言是否应包含保留字列表？

2002年9月10日新增。

讨论：当前规范未包含保留字列表。若缺乏此类列表，未来语言扩展时可能导致有效着色器失效。

建议决议：是，语言应包含保留字列表，包括以下内容：struct、union、enum、typedef、template、goto、switch、default、inline、noinline、long、short、double、sizeof、volatile、public、static、namespace、using、asm、cast、half、fixed，以及所有包含两个连续下划线的标记。

决议于2002年9月24日通过：是，着色语言应包含保留字列表。

2002年9月24日关闭。

59) 函数参数应如何传递？

2002年9月10日新增。

讨论：当前规范规定函数参数采用按引用传递，禁止别名，输出参数仅用于输出。这存在若干隐性问题：

(A) 统一变量及其他全局变量无法作为参数传递，因其会形成别名。(B) 变量及其他只写变量难以通过引用传递，因没有任何机制

声明参数为只写属性。(C)若着色器向“按引用传递”参数写入数据，则应更新调用方的参数值，否则着色器应报错（因非输出参数）。但预期用法似乎允许向非输出参数写入数据，其效果仅限于局部作用域。

规范可修改为：函数参数采用按值调用-按值返回机制，具体含义如下：(A)未加修饰符的参数表示在调用时从调用方复制参数值。(B)带输出修饰符(*或out*)的参数表示在返回时将参数值复制回调用方，但调用时不复制参数值。(C)输入输出(*或inout*)修饰符表示参数在调用时被复制，并在返回时被复制回调用方。

该语义可彻底解决所有参数别名问题。编译器无需检查别名，可直接按无别名场景编译，同时着色器编写者能明确理解：当参数传递方式看似存在别名时，实际行为已明确定义。此语义还允许向函数传递只写变量。最后，该方案允许向非输出参数写入数据，同时明确仅修改局部副本。

2002年9月24日已解决：修改规范说明函数参数采用如上定义的按值调用-按值返回机制。

2002年9月24日关闭。

60) 如何定义照明状态的内置名称？

2002年9月11日新增。

讨论：当前的照明状态命名（*gl_Light0..n[8]*及相关预定义数组索引值）使得编写处理光源的循环变得笨拙。此问题与问题(49)相关。

2002年9月24日解决：应在着色语言中添加结构体，并将光照状态重新定义为光照结构体的数组。

2002年9月24日关闭。

61) 是否允许用户自定义函数重写内置函数？

2002年9月13日新增。

讨论：允许这种情况似乎并无益处。若用户无意中使用了与内置函数相同的名称，将导致行为异常、性能下降或两者兼有。

决议：是。这是语言和库的正常行为。2002年9月10日关闭。

62) 语言是否应包含眼平面/物体平面的纹理生成系数？

2002年9月13日新增。讨论：由英特尔的Kent Lin提出。决议：是，应添加此

状态。2002年10月22日关闭。

63) 语言是否应包含投影矩阵的内置变量？

2002年9月13日新增。

讨论：模型视图矩阵和模型视图投影矩阵的内置变量已定义。是否也应为投影矩阵添加内置变量？

决议：应添加此状态。已关闭 2002年10月22日。

64) 是否应为OpenGL 1.4引入的状态添加内置变量名？

2002年9月13日新增。

讨论：当前规范基于OpenGL 1.3编写，故未包含点参数状态和雾坐标状态。是否应添加这些状态？

决议于2002年9月24日通过：应为OpenGL 1.4引入的状态添加内置变量名。

2002年9月24日关闭。

65) 矩阵乘法应执行矩阵乘法还是分量乘法？

2002年9月16日新增。

讨论：当前规范规定，当指定两个矩阵时，“*”运算符将执行分量乘法。乘法运算符(*)对标量*标量、标量*向量及矩阵*向量执行预期线性代数运算，但对矩阵*矩阵则执行分量乘法——这在实际中较为罕见。此设计旨在与其他采用分量运算的运算符保持一致。（例如，矩阵除法操作应保持分量级运算而非真实矩阵除法。是否应修改矩阵乘法运算符以明确表示矩阵乘法？）

决议：规范应修改为明确指出当两个操作数均为矩阵时，“*”运算符将执行矩阵乘法。

已关闭 2002年10月22日。

66) 着色器允许运行的时长应在规范中如何规定？

2002年9月16日新增。

讨论：早期版本的白皮书曾提及看门狗定时器。此类机制是否需要作为语言规范的一部分？

决议：语言规范不应规定着色器允许运行的时长。超时、交互性及恶意着色器检测属于实现细节和/或运行环境细节。若触发着色器执行的应用程序终止，则该着色器应被终止——此规则应在GL2扩展规范中予以明确。

已关闭 2002年10月22日。

67) 是否应制定标准化方式来指定调试和优化级别？

新增于2002年9月16日。

讨论：存在四种可能的方案。(A) 不进行调试，始终启用优化，因此不存在问题。(B) 在编译和链接的入口点添加调试与优化参数。(C) 使用#pragma指定调试和优化级别，并概述基本可移植含义。(D) 声明此完全取决于平台，不作任何规定。

(A) 方案(A)显得目光短浅，因为优化开关本身是追踪特定缺陷的技术手段与变通方案。未来我们必然需要调试着色器，且编译时与运行时存在权衡关系（例如当应用程序动态生成寿命极短的着色器时，关闭低效优化反而可能提升速度）。

(B) 此方案略显生硬，因相关操作存在平台依赖性。方案(D)对原则上应适用于所有平台的功能而言似乎过于激进。

决议：使用#pragma指令指定调试与优化级别，并概述基本可移植含义。

已关闭 2002年10月22日。

68) 语言是否应支持显式数据类型如'half' (16位浮点) 和'fixed' (固定精度带位数据类型) ?

2002年9月17日新增。

高级语言通常支持多种数值数据类型，以便程序员在性能与精度之间选择合适的平衡点。例如C语言支持float和double数据类型，以及多种整数数据类型。这一通用原则同样适用于着色语言。

对于着色计算而言，远低于32位浮点数的精度通常已足够。直至近年，多数图形硬件仍采用9位或10位定点运算执行所有着色计算。低精度数据类型可实现更高性能，尤其当数据需跨芯片传输时（如纹理数据）。因此，在硬件着色语言中提供低于32位的数据类型访问机制具有重要意义。

问题(33)讨论了精度提示。精度提示不如额外数据类型实用，因为精度提示不支持按精度进行函数重载。开发者发现能够拥有名称相同、参数列表相同但精度数据类型不同的函数非常方便且实用。

能够为每个变量（而非每个着色器）指定数据类型也至关重要，因为某些计算（例如纹理坐标计算）通常需要比其他计算更高的精度。

另一方面，人们希望确保着色器在不同实现之间具有可移植性。为了实现可移植性，那些不原生支持半精度浮点数的实现将受到惩罚，因为它们必须将中间计算结果限制在适当的精度范围内。若这些额外数据类型仅作为编译器可选的精度降低计算提示，则会使ISV面临意外截断或溢出语义的风险，导致不同架构产生截然不同的结果。该提示还暗示底层存在明确规范的类型转换机制，这将使函数重载解析更为复杂，需要额外规则来消除歧义——除非必须提供所有合法函数组合。指定所有合法组合需新增大量函数类型（点积运算需支持{float, half, fixed} × {float, half, fixed} × 数个分量，即36种版本，而仅用float时仅需4种）。

若新增数据类型为实数类型，其适用范围如何界定？若应用于统一变量和属性，不同尺寸将体现在API中，但半精度和固定精度类型在

C. 若半精度类型后接浮点类型，是否意味着浮点类型必须起始于16位边界？那么固定长度包装的半精度类型呢——其真实大小未定义。若半精度和固定长度仅限于临时变量，则情况会更简单，但此时存储效率优势便不复存在。

固定精度数值的打包——其真实大小未定义。若半精度与固定精度仅限于临时变量，虽简化了处理，但存储效率优势将丧失。

OpenGL规范当前规定：“用于表示位置或法线坐标的浮点数最大可表示量必须至少为 2^{16} 。”我们是否应引入与此相悖的方案？即使对于 $1k \times 1k$ 纹理， $s10e5$ 精度也无法满足纹理坐标需求。半精度浮点似乎为精度问题在着色器中扩散打开了大门。

决议：性能/空间/精度提示和类型不会作为语言的标准组成部分提供，但为此保留的保留字将予以保留。

已关闭：2002年11月26日。

69) 片段着色器是否能访问提供位置信息的变量？

新增于2002年9月17日。

讨论：窗口位置在特定片段着色器中具有重要价值。例如可用于实现基于片段着色器的点状纹理效果。

决议：窗口位置作为内置变量`gl_FragCoord`的一部分，可在片段处理器中使用。规范应明确说明该变量的x和y值定义了片段的窗口位置。

已关闭 2002年10月22日。

70) 该语言是否应支持布尔向量（例如`bool2`、`bool3`、`bool4`）及相应的向量运算符？

2002年9月17日新增。

讨论：该语言已支持短浮点向量（如`vec2`、`vec3`、`vec4`），因此支持布尔向量将保持一致性。若语言包含布尔向量及运算支持，则可轻松实现元素级向量计算。例如，**最小值**和**最大值**向量运算可通过布尔向量运算在语言内部优雅实现。

当将`<`等比较运算符应用于向量操作数时，结果应为包含逐元素比较结果的布尔向量。若第一个操作数为布尔向量，“?:”结构将执行逐元素运算。`if`、`while`和`for`语句仍需接受标量布尔值。

决议：将添加布尔向量。但将保留类似C语言的`&&`和`||`短路评估特性。本次版本中，`if`和`?:`语句仅支持基于标量布尔值的条件选择。由于`==`和`!=`运算符也应返回标量布尔值（如同对所有类型包括结构体所做的那样），因此比较向量时它们将返回标量布尔值而非布尔向量。由此，`<`、`>`、`<=`和`>=`运算符同样不会生成布尔向量。

这些运算符将无法合法作用于向量。取而代之的是，系统将新增用于向量关系运算的内置函数。

已关闭：2002年11月5日。

71) 阴影语言是否应支持复合数据结构，例如数组的数组、结构体的数组、数组的结构体等？

新增于2002年9月17日。

讨论：创建复合用户自定义数据结构的能力几乎是所有高级编程语言的基础组成部分。若不支持这些功能，将

与OpenGL着色语言设计工作普遍具有前瞻性的本质相悖。

另一方面，如同添加结构体的问题（议题(49)），任何语言都可能存在“总有人、总在某地、总在某时需要”的论点。值得投入时间精力确保该功能纳入规范初版吗？推迟添加是否会引发问题？

决议于2002年9月24日通过：是的，阴影语言应支持数组的数组、数组的结构体、结构体的数组等。

2002年9月24日关闭。

72) 着色语言是否应包含switch语句？

2002年9月19日新增。

讨论：C语言的switch语句是编写简洁代码的有效结构。替代方案是使用可读性较差的if语句集合。

决议：初始版本不会添加switch语句。虽然存在管理浮点数范围的需求，但该功能的实现将耗费大量时间。

2002年10月1日关闭。

73) 在main()函数中，关键字return的语义是什么？

2002年9月19日新增。

讨论：当前规范未说明当return关键字出现在main()函数末尾以外的位置时会发生什么。

决议：return表示退出main函数，与到达函数末尾效果相同。它不表示终止程序。2002年10月1日关闭。

74) 顶点着色器或片段着色器计算的数据如何传递回应用程序？

2002年9月24日新增。

讨论：此问题与问题(50)和问题(54)相关。若允许顶点处理器或片段处理器计算不继续流向处理管道的结果（例如直方图、最小/最大值计算，或从控制点计算顶点数组数据），这些结果将如何反馈给应用程序？

决议：推迟至规范的未来版本处理。已关闭：2002年10月22日。

75) 应用程序初始化的统一变量和属性是否应允许采用结构体形式？

新增于2002年9月25日。

讨论：若禁止此行为，应用程序需通过单独的全局变量传递所有数据，且着色器代码必须将数据打包为结构体才能使用结构体，这将导致冗余且丑陋的着色器代码。另一方面，初始化结构体数据不应增加API复杂度。

此外，语言中存在结构体却无法从应用程序初始化它们，这似乎是个缺陷。另一方面，为结构体提供完整的初始化方案似乎超出了本次版本的范围。

另需注意属性可为矩阵类型，但目前缺乏初始化API。解决方案：

- * 添加属性矩阵初始化入口点。绑定时，4x4矩阵占用4个连续位置，3x3矩阵占用3个连续位置，2x2矩阵占用2个连续位置。布局细节被隐藏。若实现仅需为2x2矩阵分配一个槽位，则仅需占用该槽位，但系统预留空间供两个槽位的实现方案使用。
- * 暂不支持属性数组及属性结构体。
- * 允许统一结构体和结构体数组。
- * 支持通过在GetUniformLocation时指定字符串来初始化结构体成员，该字符串用于选择待初始化的成员。例如："struct.member"、"struct[4].member"、"struct[2].member[2]"等。
- * API 暂不支持结构体级别的初始化，需等待未来完整解决方案（该方案需理解步长、对齐、填充等概念）。

已关闭 2002年10月22日。

76) *vec2、vec3、vec4、mat2、mat3 和 mat4 是否应定义为结构体？*

新增于2002年9月26日

讨论：将这些类型视为结构体将使语言更清晰易于规范，同时简化编译器开发。

目前，这些类型涉及到特殊语言特性，例如“v.xyz”语法和“v.yzx”这类交换操作。然而，在HLSL中并不需要此类操作。例如，交换操作是汇编语言技巧的典型应用，例如用两个汇编指令表示一个交叉乘积。但在HLSL中无需此类技巧，只需使用内置函数，或在真正需要时定义用户自定义函数。此外，在极少数确实需要此类操作的情况下，可通过标准语言特性轻松实现，例如vec3(v.y, v.z, v.x)。

此外，v[i][j]这类语法计划作为特殊语言特性加入。但需要动态索引组件的情况相当罕见，通常可通过内置访问函数轻松实现。

另一个方面是相同组件拥有不同名称，例如xyzw与rgba。失去这一特性可能是将vec3和vec4视为普通结构体唯一真正的缺点。另一方面，使用xyzw这样的独特名称也可能使着色器程序更清晰。

另一方面，向量与矩阵类型确实具有特殊性：(A) 大多数运算符可直接作用于它们，结构体则不然；(B) 向量或矩阵的每个元素都属于同一类型，结构体则不然。(C) 矩阵本质上是二维概念，结构体则是单维概念。当前语言在保持矩阵二维性方面存在缺陷——有时需明确其列优先顺序，但仍可视作二维处理。(D) 硬件可能为向量和矩阵类型配备专用单元，这类单元映射到通用结构体比映射到内置类型更为困难。

决议：向量与矩阵不属于结构体范畴，但将进行部分调整。总结：

- * 右值交换成为对表达式的操作。这与最初规范中将其定义为向量变量的成员选择器的说法有所不同。
- * 我们移除了空括号语法。

- * 我们保留现有的向量置换语法。
- * 我们保留向量数组访问语法。
- * 针对矩阵m，新增m[i]表示第i列且为向量类型，同时支持左值和右值操作。
- * 由于 m[i] 是向量，而向量具有数组语法，因此 m[i][j] 自然表示 m 的第 i 列第 j 行。

已于2002年10月8日。

77) 是否应将gl_FragStencil和gl_FBStencil的类型改为int?

当前采用浮点型。鉴于语言中整数类型的应用已取得进展，应重新审视此项。

决议：同意。但模板写入功能已推迟至未来版本，且帧缓冲读取功能已被移除。

已于2002年12月10日关闭。

78) 模板值是否会自动限制在模板缓冲区当前可存储的最小值与最大值范围内？

决议：是。但模板写入功能已推迟至未来版本。于2002年12月10日关闭。

79) 是否需要为片段着色器提供近裁剪平面和远裁剪平面？

讨论：ARB_fragment_program已包含此功能。决议：是。

于2002年12月10日关闭。

80) 矩形（非的幕）纹理的索引并非基于0.0至1.0范围，而是基于其实际尺寸。这是否构成问题？

讨论：是的，这确实是个问题。某些硬件需要在编译时就明确访问的是何种纹理类型。我们希望避免因状态变更而重新编译着色器。可增加更多纹理内置名称，以便在编译时明确访问的纹理类型。例如使用textureRect3表示返回3个分量的矩形纹理。此外，矩形纹理并非OpenGL 1.4的组成部分，因此这些函数可作为扩展功能添加，不必纳入本次语言规范版本。

决议：预留空间以支持其他纹理类型的扩展函数，但矩形纹理的实现需待其成为OpenGL核心标准后再行添加。

于2002年12月17日关闭。

81) 我们是否应该支持从着色器访问固定功能雾效的方式，以利用固定功能硬件中可能存在的雾效？

讨论：这类似于支持固定功能变换（我们已实现该功能）。但后者是出于不变性需求而设计，而雾效并不存在此类问题。该功能虽在ARB_fragment_program中提供，但本规范的核心精神是用可编程性取代固定功能管道及其额外访问方式。

决议：不提供对固定管道雾效果的特殊访问支持。

于2002年12月17日结案。

82) 是否真的需要强制要求写入 gl_Position、gl_FragColor 或 gl_FragDepth？

讨论：当写入操作是条件性的时，处理错误情况可能很困难。有人提出在片段着色器中应调用**kill**函数或写入**gl_Output**。但这无关紧要，因为片段着色器既不调用**kill**也不写入任何输出都是允许的。对于顶点着色器，不写入位置值仍然毫无意义，因此仍应强制要求。

决议：片段着色器无强制规则，可调用**kill**或不调用，若不调用则无需写入任何值，管道中的现有值将被直接采用。顶点着色器仍需写入**gl_Position**，编译器应在可能时提供诊断提示。

于2002年12月10日关闭。

2003年1月7日重新开启，因编译器认为默认的**gl_FragColor**和**gl_FragDepth**是条件写入时，其性能影响引发关注。对于颜色，若不写入**gl_FragColor**则应直接视为未定义。深度处理则更为复杂：若深度未写入，则应使用固定功能计算的深度值。然而，若深度值采用条件写入，编译器将始终需要初始化深度值，这可能造成性能损失。后续讨论由此衍生出两种方案：要么强制着色器始终写入**gl_FragColor**，要么根据源代码结构采用更复杂的处理机制。

决议：规定若着色器条件性写入**gl_FragDepth**，则必须始终写入。参见issue 95中关于不变性的讨论。

已关闭 2003年1月17日。

83) 修改模板时应如何处理？这会影响状态的压入/弹出操作吗？

讨论：在片段着色器中写入模板值为OpenGL引入了新功能。单就其本身而言，实用性存疑。另一方面，现有操作（如模板值的递增递减）在当前规范下难以明确表达。或许模板值仅应在满足以下条件时更新：片段着色器未输出任何数据，且API已配置为渲染至模板。但此方案尚未经过充分论证。

决议：模板修改功能推迟至未来版本实现。已关闭：2002年12月17日。

84) 是否应添加投影纹理查找？阴影纹理如何处理？

讨论：投影纹理查找可暂缓实现。阴影纹理属于1.4规范范畴，**应通过**`textureShadow*`**类接口添加**。但若引入阴影纹理，同步添加投影查找功能亦属易事。若未启用阴影模式？预期模型应呈现为硬件按应用程序配置执行操作，着色器调用阴影时将获取既定设置。因目标对象的输入数据量大于非阴影查找，故将采用独立命名的函数。阴影查找可改称“compare”替代“shadow”，但RenderMan规范采用“shadow”命名。

决议：添加投影纹理和阴影纹理，并采用新名称 `proj` 和 `shadow` 以体现此特性。此外，投影纹理将支持两种尺寸输入：所有类型均接受**vec4**参数，其中2D投影纹理额外接受**vec3**参数等。同时确保规范说明：若在未配置比较器的纹理上使用内置阴影功能，或在配置了比较器的纹理上使用非内置阴影功能，结果将未定义。

已关闭 2003年1月17日。

85) 编译器如何判断访问的是1D、2D、3D还是CUBE纹理？基于参数分量数量的判断方式容易出错。

讨论：纹理的选择取决于查找坐标参数的类型。然而，可以设想支持将多个纹理绑定到相同的纹理单元编号，在编写着色器时，作者应明确自己正在访问哪个纹理，避免因类型错误而获取错误的纹理。明确指定纹理可通过向现有纹理调用传递枚举值实现，或通过添加包含纹理类型的命名新名称实现。使用枚举的弊端在于其暗示参数可通过变量传递，而编译时需明确类型要求。这支持采用命名变更方案。

决议：增加更多纹理名称，使编译时能明确识别正在进行何种纹理查找。例如**texture1D3**、**texture2D3**、**texture3D3**和**textureCube3**，其中末位数字表示返回的组件数量（可为1、2、3或4）。

已关闭：2002年12月17日。

86) 规范中鲜少提及颜色索引支持。对此提供了哪些支持？

讨论：其他扩展（ARB_vertex_program、ARB_fragment_program、纹理应用）均声明在COLOR INDEX模式下渲染时操作结果未定义。

决议：除声明COLOR INDEX操作未定义外，删除所有相关提及。

已关闭：2002年12月10日。

87) 辅助数据缓冲区曾是OGL2白皮书的一部分，但本规范需基于当前OGL核心标准。

决议：从规范中移除辅助数据缓冲区。若未来扩展为GL核心提供其他可写入缓冲区，可重新添加该功能。

已关闭：2002年12月17日。

88) 某些数组的变量数组索引可能难以实现。

讨论：数组可能仅限于统一变量。然而这种限制过于严格，其他图形语言都支持局部变量数组。此方案还缺乏与结构体的正交性——结构体既可包含数组，又可作为统一变量或局部变量使用。或许可考虑将非统一数组的索引限制为统一索引。着色器不太可能先初始化整个非统一数组，再使用非统一索引进行访问（更常见的情况是将非统一索引应用于静态数组或纹理）。另一方面，鉴于这种情况罕见，语言规范可保持简洁且功能完备，未来一两年内编译器报错称其过于复杂亦无妨（毕竟需求概率极低）。

决议：将完整支持数组功能。已关闭：2003年1月7日。

89) 为何变量gl_TexCoord0...gl_TexCoordn不采用数组形式？

讨论：在编译时知道这些信息或不允许变量索引可能带来某些性能或编译器便利性。这可能与问题88相关联——若变长数组的索引必须统一，则更易于将其作为数组支持。

决议：将这些变量改为数组。正在处理上述两个问题的具体方案。

关闭日期：2003年1月7日。

重新开启：问题88已通过全面支持数组解决。但这导致纹理坐标存在两个潜在问题：i) 无法确定着色器中实际激活的坐标数量；ii) 无法在编译时识别被访问的坐标。

讨论：替代方案：

1. Cg语言中常量循环迭代的实现类似如下：

"可在编译时确定"的定义如下：循环迭代表达式可通过过程内常量传播与折叠在编译时求值，且传播常量值的变量不得出现在任何控制语句（if、for 或 while）或 ?: 结构中的左值位置。

2. 严格限制条件：必须是编译时常量，或具有编译时常量起始值/终止值/递增值的归纳变量。

3. 实际上解决根本的资源大小问题：要求着色器编写者在违反规则#2时重新声明数组。即允许着色器声明"varying gl_TexCoord[N]"，其中N表示所需使用的坐标数量，并在规则#2未满足时强制要求声明。允许语言中完全访问变量（当情况变得棘手时，早期硬件/编译器会发出警告）。

4. 改用#pragma实现第3条规则。

5. 变体方案：着色器中所有指向 gl_TexCoord[] 的索引必须是常量表达式，或着色器必须声明带大小的 gl_TexCoord[]。内置数组应声明为空数组，以便与 C 语言中带大小的声明保持一致。多个模块可声明不同大小的数组，链接时将采用最大值。

解决方案：采用方案5。

已关闭：2003年1月24日。

- 90) 内置函数定义了返回整数的纹理函数。然而，此类纹理并非OpenGL核心功能。

决议：这些内置函数将被移除。未来若OpenGL规范新增整数纹理功能，可将其重新纳入。

已关闭：2002年12月17日。

- 91) 片段着色器中似乎缺少雾效相关信息。仅使用gl_EyeZ不足以实现，是否需要引入衍生参数？

决议：将浮点型变量gl_EyeZ扩展为vec4类型的gl_FogFragCoord。已关闭：2002年12月17日。

- 92) 我们需要一种方法来获取目标代码，就像主机处理器上C语言的实现模式那样。

讨论：邮件中讨论甚多。这涉及最低层级的机器特定代码，甚至可能在同一厂商的卡系列中都无法移植。主要目标之一是节省编译时间。普遍共识认为此举有其价值，但不影响语言定义。

决议：此为API问题而非语言问题。2002年12月19日以API问题为由结案。

- 93) 实际上，顶点着色器的输出变量接口与片段着色器的输入变量接口存在差异，但本规范采用统一接口。这导致雾效处理出现问题，且无法兼容固定功能单元与ARB_fragment_program的混合配置。

讨论：将该接口拆分为两个部分是合理的。片段输入应与ARB_fragment_program兼容，顶点输出应与ARB_vertex_program兼容。雾效可通过此方式处理，规范中已有体现。

决议：应执行此方案。2003年1月17日结案。

- 94) 根据当前语言规范，编译时无法确定哪些目标被分配到哪些单元。若认为硬件必须预先配置正确的目标到正确单元，这将构成重大问题。

讨论：可能的解决方案：

A. 使用传统的OGL启用、绑定和优先级机制，使应用程序能够在着色器运行时指定需要激活的目标。

B. 修改语言规范，要求仅通过编译时检查即可确定每个单元使用的具体目标。

C. 预期硬件能够动态访问请求的目标，而无需驱动程序预先进行设置。

D. 为API新增设置活动纹理的入口点：a) BindTextureGL2(GLenum textureUnit, GLenum GLuint texture)（可能省略textureUnit参数）但由于与标准管道不兼容，似乎无人需要此函数。b) 另一方案是新增实用函数UseTexture(GLenum textureUnit, GLenum GLuint texture)，其功能完全等同于：i) 绑定纹理，ii) 启用该纹理的目标，iii) 禁用所有其他目标。若纹理为NULL，则禁用所有目标。此方案将100%兼容标准管道；新函数仅为提升便利性，不提供新功能（可作为glu函数实现）。

E. 采用与Cg相同的采样器用法。

解决方案：使用采样器。将新增采样器类型sampler1D、sampler2D、sampler3D和samplerCube。这些类型在着色器中不可写入，仅能作为全局统一变量使用，且支持数组形式。它们属于不透明类型，着色器内仅可引用而不可操作。纹理调用的首个参数将接受此类采样器。API必须通过某种方式将纹理或纹理单元绑定至这些采样器。启用状态、活动纹理等功能将被这些API绑定机制取代。在语言层面，它们将为未来可能实现的纹理虚拟化提供基础。

另参见问题97。

已关闭于2003年1月17日。

- 95) 是否需要解决gl_FragDepth或其他着色语言方面的不变性问题？

讨论：部分人士对此提出担忧，但具体问题及解决方案尚不明确。

决议：对于 `gl_FragDepth`, 声明固定功能深度与着色器计算深度之间不保证不变性。但同一着色器或不同着色器中多次使用相同着色器深度计算时将保持不变。

其他变异性问题推迟至下个版本处理。已关闭，2003年1月17日。

96) 内置函数`lod`在各向异性过滤中不适用。

讨论：似乎需要`lod1D`、`lod2D`、`lod3D`、`lodCube`函数。但考虑到当前纹理内置函数采用偏移量而非绝对LOD值，且若添加各向异性过滤将导致LOD规范复杂化，`lod()`内置函数似乎并无实际价值。

决议：移除LOD内置函数。已关闭 2003年1月17日。

97) 是否需要阴影采样器？例如类型`sampler1DShadow`、`sampler2DShadow`？

讨论：这使得在将正确的纹理状态与正确的内置查找函数关联时，能够实现更高级别的强制执行。另一方面，它为添加大量类型关键字指明了方向。

决议：同意添加这些类型。已关闭 2003年1月24日。

**98) 为保持与其他语言一致性，应将`vec2`、`vec3`、`vec4`改为`float2`、`float3`、`float4`。
应同样废弃。**

讨论：应将`vec2`、`vec3`、`vec4`替换为`float2`、`float3`、`float4`，以符合斯坦福RSL、Cg和HLSL的既定规范。

此外，这些命名更便于支持基于其他标量类型（如 `int2`、`int3`、`int4`、`half2`、`half3`、`double4` 等）的数据类型。

另一方面，采用以下方案似乎更为合理：先用前缀类型缩写（默认为浮点数）定义向量，后接“vec”，再跟上成分数量。

类型 4维向量

浮点型	<code>vec4</code>
整型	<code>ivec4</code>
<code>bool</code>	<code>bvec4</code>
半精度	<code>hvec4</code> <code>double</code>
	<code>dvec4</code> <code>float16f</code>
<code>16vec4</code> <code>int32</code>	<code>i32vec4</code>

是否也应将“kill”改为“discard”？理由在于OpenGL规范始终使用“discard”表示片段丢弃操作，从未采用kill。虽然Kill指令同时出现在ARB_fragment_program与NV_fragment_program规范中（可能是因为三字母指令便于记忆），但这与核心OpenGL规范并不一致。另一原因是Cg/HLSL已将“discard”设为关键字，没有必要为相同功能设置冗余关键字。

解决方案：将kill改为discard，向量指令保持原样。

已关闭 2003年1月24日。

99) 内存中处理超大表格的机制尚不完善。

讨论：需要高效地将大型信息数组传递给顶点着色器，以支持高级动画技术。目前仅能处理能放入“统一变量”空间的数组，而无法处理真正庞大的数据结构。

需要在语言层面声明此类结构，并需API支持进行初始化。有人认为纹理查找功能已基本涵盖此需求，但这似乎并非内存中大型数组的恰当抽象方案。

决议：推迟至未来版本实现。将规范作为本议题的组成部分。已关闭 2003年1月24日

100) 若未来在顶点着色器中添加LOD偏移形式，或（更可能的情况）在片段着色器中添加绝对LOD形式，则会与纹理内置函数发生名称冲突。

解决方案：偏移形式采用现有命名方案。绝对LOD形式采用现有命名方案，并在名称后添加“Lod”后缀。

已关闭 2003年1月24日。

101) 添加统一变量初始化语法。例如：uniform vec3 Color = vec3(1.0, 1.0, 0.0);

讨论：此功能由ISV和NVIDIA提出。虽看似非必需，但若多数人支持可予以添加。同时需考虑为采样器（但不包括数组？）添加此功能。需注意：仅对渲染过程中不变的统一变量有意义，因常量全局变量可用且纹理不可指定，故其实用性可能低于预期。但仍具价值。

决议：不采取任何行动。将向后兼容的增强功能保留至该语言的未来版本。

已关闭：2003年12月。

102) 修复数组相关问题（多维数组、数组成员静态初始化、可变长度数组、无尺寸数组参数）。

讨论：函数参数声明“vec4[]”未明确数组大小，暗示采用按引用传递方式。当前调用约定为按值传递。该语法应保留用于未来可能添加的按引用传递数组功能。若需按值传递数组，应明确指定大小如“vec4[5]”。遗憾的是，此方式将导致无法用两个不同大小的数组调用同一函数。不过，或许应等到数组支持按引用传递机制后再考虑此问题。静态初始化可通过类型构造函数名称后跟方括号实现，例如：“vec4[5] vArray = vec4[](v1, v2, v3, v4, v5);”

一旦踏上这条道路，便可进一步将数组提升为一等对象。这将支持“float[7] fArray”这类声明形式，并允许对数组进行复制和比较操作。这也意味着需取消数组的自动调整大小功能——该功能在节省插值资源方面尚有其他用途。若更进一步，可引入多维数组，届时需考虑数组的数组、类型别名，以及真正的多维数组形式，例如“float[7,3] fMultiArray”。

决议：对变更采取保守态度，但不排除未来所有功能扩展的向后兼容性。确保未来向后兼容所需的最小变更包括：要求函数参数声明必须指定数组大小，并使相同全局数组在不同大小时触发链接错误。这将使我们能够保留现有的自动调整大小功能。

。函数调用必须根据参数类型匹配数组大小，否则将报错“无匹配重载函数”。其余变更推迟至未来版本。

已关闭：2003年12月。

103) 明确向量矩阵构造函数在初始化元素不足时的规范：是舍弃最后一行列，还是按顺序初始化元素？用户通常期望前者。同时需限制构造函数，使其以更合理的方式运行。

讨论：规范要求必须提供足够的组件。目前缺少用于构建非列优先顺序矩阵的有趣构造函数。当前有效程序中构造函数可接收的参数数量亦无上限。我们还可列举所有实用构造函数，并精确说明各自功能。此外，构造函数作为类型转换器与类型构建器的使用界限存在模糊。通过提供清晰的原型列表可消除此模糊性。此前为简化规范而未列出完整列表。

决议：限制构造函数的参数数量。最后一个有效参数可包含超出需求的组件（仅部分消耗），但超出该数量的参数将被禁止。同时禁止当前无用但未来可能定义为有用的构造方式，例如由不同尺寸矩阵构造矩阵（如 mat4(mat3) 或 mat2(mat4)）。新型构造函数的支持将推迟至未来版本。

已关闭 2003年12月。

104) 是否应该提供快速的atan() 等函数？

讨论：存在两种不同方向。其一是增加性能与精度的权衡选项，其二是添加域限定函数。两者均可通过新命名集实现支持。例如：

atanDom() 作为域限定的反正切函数。

atanFast() 为精度较低但性能更高的反正切函数。

决议：先作为扩展功能实现。2003年12月已关闭。

105) 修改规范，使代码可通过串联所有着色器对象的着色器字符串生成。

讨论：仅等待链接时进行完整编译并一步完成链接确实具有一定实用性。实现方式之一是将所有着色器（仅限顶点着色器或片段着色器内部，而非跨模块）进行拼接后解析。这些着色器仍需在编译时进行解析以返回错误等信息，但链接时将进行二次解析。

决议：不予采纳。此变更易引发规范错误。2003年12月结案。

11 致谢

若我看得更远，是因为站在巨人的肩膀上——艾萨克·牛顿

本提案语言中的绝大多数理念与元素均源自计算机科学与计算机图形学文献。3Dlabs的Dave Baldwin撰写了奠定OpenGL着色语言基础的白皮书，其原始致谢内容如下：

- AT&T公司贡献了C语言，
- 皮克斯公司贡献RenderMan技术，
- UNC团队因其像素流语言及可编程OpenGL接口获得表彰，
- 斯坦福大学在着色语言领域的贡献，
- SGI公司因其开创性的OpenGL技术（理所当然）及其着色语言研究，
- 3Dlabs的同事们通过频繁的积极讨论，帮助厘清了诸多关键点

戴夫·鲍德温在着色语言发展过程中持续参与设计工作。兰迪·罗斯特与约翰·凯塞尼奇共同修订了着色语言白皮书的多个公开版本，并创建了本规范文档的初版。3Dlabs的安东尼奥·特哈达编写了该语言的首个解析器，助力解决若干基础语言设计问题。

在初步方向确定后，由3Dlabs的兰迪·罗斯特、约翰·凯塞尼奇、巴托尔德·利希滕贝尔特和史蒂夫·科伦组成的团队接手了OpenGL着色语言的设计与实现工作。该团队负责制定公开发布的OpenGL着色语言规范及源代码，开发语言编译器的初始实现并支持OpenGL扩展功能，同时在实施过程中持续优化语言设计。

3Dlabs的戴夫·鲍德温、戴尔·柯克兰、杰里米·莫里斯、菲尔·赫胥黎和安东尼奥·特哈达参与了多轮OpenGL着色语言讨论，在推进过程中提供了大量宝贵建议与支持。英国埃格姆、阿拉巴马州亨茨维尔及德克萨斯州奥斯汀的3Dlabs驱动开发团队成员亦作出贡献。3Dlabs管理层应获嘉许，其不仅具备推动OpenGL着色语言提案的远见，更勇于投入资源支持开发工作。特别感谢奥斯曼·肯特、尼尔·特雷维特、杰里·彼得森和约翰·辛普夫。

许多其他人士也参与了OpenGL着色语言的讨论。我们谨向ATI、SGI、NVIDIA、英特尔、微软、Evans & Sutherland、IBM、Sun、苹果、Imagination Technologies、戴尔、康柏和惠普等公司的同事及ARB代表致谢，感谢他们为讨论所作的贡献以及推动进程的努力。特别是ATI研究部的Bill Licea-Kane和Evan Hart，他们通过深刻的审阅和严谨的评论，为完善规范及语言本身做出了重要贡献。

衷心感谢那些抽出时间与我们交流、发送邮件或在OpenGL.org上回答调查问卷的软件开发者们。我们的终极目标是为您提供最优质的图形应用开发API，而您花时间告诉我们的需求的过程弥足珍贵。部分独立软件供应商（ISV）为特定功能进行了长期而深入的游说，最终成功说服我们对原始OpenGL 2.0提案作出重大调整。感谢所有软件开发者，请继续告诉我们您的需求！

其他对着色语言提出宝贵意见或评审的人士包括：Alias|Wavefront的Christian Laforté与Ian Ameline；Imagination Technologies的Jonathan Putsman；LightWork Design的Darren Roberts与Slawek Kilanowski；ID Software的John Carmack；Epic Games的Tim Sweeney与Daniel Vogel；马格德堡大学的Bert Freudenberg；Spinor的Folker Schamelin；维塔尔影像公司的卡雷尔·祖德费尔德与史蒂夫·德姆洛；体积图形公司的克里斯托夫·波利沃达、克里斯托夫·莱因哈特与沃尔夫冈·罗默；巅峰公司的克里斯蒂安·肖尔曼；Whatif制作公司的杰克·科尔布五世；软影公司的米克·韦尔斯；SD解决方案公司的德尔温·霍尔罗伊德与格尔克·休斯马；斯坦福大学的库尔特·阿克利、帕特·汉拉汉与比尔·马克；以及苹果公司的约翰·斯托弗。

对着色器对象和程序对象发展方向产生重要影响的人物包括：Alias|Wavefront的克里斯蒂安·拉福特；Discreet的皮埃尔·特伦布莱；英特尔的比马尔·波达尔；Matrox的乔恩·保罗·谢尔特；Imagination Technologies的乔纳森·普茨曼；Softimage的米克·韦尔斯；Vital Images的卡雷尔·祖德费尔德和史蒂夫·德姆洛；以及Epic Games的蒂姆·斯威尼和丹尼尔·沃格尔。

本规范文档已于2002年6月正式提交至OpenGL架构审查委员会的GL2工作组。该工作组负责识别问题、解决问题，并最终确定规范以供OpenGL架构审查委员会批准。以下GL2工作组成员为本规范最终定稿作出贡献：3Dlabs公司：Dave Baldwin、John Kessenich、Steve Koren、Barthold Lichtenbelt、Randi Rost；ATI Research公司：Evan Hart、Bill Licea-Kane（ARB-GL2工作组主席）、Victor Vedovato；戴尔公司的戴夫·曾兹；英特尔公司的布兰登·菲夫莱特与肯特·林；英伟达公司的帕特·布朗、马特·克雷格海德、卡斯·埃弗里特、史蒂夫·格兰维尔、贾扬特·科尔赫与尼克·特里安托斯；SGI公司的乔恩·利奇；Spinor公司的福尔克·沙梅尔；鸽图公司的布莱恩·保罗；Quel solaar公司的埃斯基尔·斯滕堡；马里兰大学巴尔的摩县分校的马克·奥拉诺；滑铁卢大学的迈克尔·麦库尔；以及Vital Images公司的马特·克鲁克香克、史蒂夫·德姆洛和卡雷尔·祖伊德费尔德。

最后，我们衷心感谢C语言设计者、RenderMan设计者以及OpenGL设计者——这三大标准对我们的工作产生了最深远的影响。愿OpenGL着色语言能延续他们成功的传统，继续追求卓越。

