THE OPENGL® SHADING LANGUAGE

John Kessenich

Dave Baldwin

Randi Rost

Language Version 1.10

Document Revision 59

30-April-2004

CONTENTS

1 INTRODUCTION

Note: Document revisions for the language specified by this document are being tracked separately from the language version. Changing the revision of the document does not change the version of the language. This document specifies version 1.10 of the OpenGL Shading Language, document revision number 59. It requires __VERSION__ to be 110, and #version to accept 110.

1.1 Changes since version 1.051

- Added issues 101 through 105. Specification changes made from these issues are to make array parameters sized, and add some limitations in constructors. See sections 4.2, 5.4.2, 6.1, 6.1.1.
- Added interactions with ATI_draw_buffers and ARB_color_clamp_control, particularly, the output variable *gl_FragData[n]*.
- 3.3 Added #version and #extension to declare version and extensions.
- 7.5 Added built-in state for the inverses and transposes of matrices.
- 8 Added built-in functions **refract, exp,** and **log.**
- Added the following clarifications and corrections:
    - 2.1 Remove "Clamping of colors" from the list of what the vertex processor does. This was just out of date.
    - 2.1 Change "Perspective projection" to more clearly call out projective transform and perspective division, which belong in different lists.
    - 3.3 Reserved pre-processor macros that start “GL_”.
    - 3.6 Added reserved words **packed, this, interface, sampler2DRectShadow.** Also clarified that the listed keywords and reserved words are the only ones.
    - 4.1.5 Remove "Integer vectors can be used to get multiple integers back from a texture read." This was just out of date.
    - 4.3.5 Clarified that structs can be constants, and what **const** must be initialized with.
    - 5.8 Clarify what ***=, +=**, etc. really mean. and that **?:** is not an l-value.
    - 5.9 Clarify that operating between a scalar and a vector is allowed for integers as for floats, and that the list is to list all operators and expressions.
    - 6.1 Correct the examples of dot product prototypes. They were not correct WRT to the list of prototypes, which themselves have been correct for some time.
    - 6.1 Add the clarification "If a built-in function is redeclared in a shader (i.e. a prototype is visible) before a call to it, then the linker will only attempt to resolve that call within the set shaders that are linked with it."
    - 7.2 Remove the out of date text "an implementation will provide invariant results within shaders computing depth with the same source-level expression, but invariance is not provided between shaders and fixed functionality."

  - 7.4 Correct the list of built-in constant names: removed suffixes and brought values up to date.
  - 8.2 State the domains for the exponential functions.
  - 8.3 Change **step()** to compare *x* < *edge* instead of *x* <= *edge*.
  - 8.7 Clarify the discussion about when shadowing lookups are undefined.
  - 8.9 Further specify the range and frequency constraints of noise.
  - Grammar: MOD_ASSIGN change to reserved (to match the specification text).
  - Grammar: Change to require array sizes in function parameters.
- Several typos fixed.

1.2 Overview

This document describes a programming language called *The OpenGL Shading Language*, or *glslang*. The recent trend in graphics hardware has been to replace fixed functionality with programmability in areas that have grown exceedingly complex (e.g., vertex processing and fragment processing). The OpenGL Shading Language has been designed to allow application programmers to express the processing that occurs at those programmable points of the OpenGL pipeline.

Independently compilable units that are written in this language are called *shaders*. A *program* is a set of shaders that are compiled and linked together. The aim of this document is to thoroughly specify the programming language. The OpenGL entry points that are used to manipulate and communicate with programs and shaders are defined separately from this language specification.

The OpenGL Shading Language is based on ANSI C and many of the features have been retained except when they conflict with performance or ease of implementation. C has been extended with vector and matrix types (with hardware based qualifiers) to make it more concise for the typical operations carried out in 3D graphics. Some mechanisms from C++ have also been borrowed, such as overloading functions based on argument types, and ability to declare variables where they are first needed instead of at the beginning of blocks.

1.3 Motivation

Semiconductor technology has progressed to the point where the levels of computation that can be done per vertex or per fragment have gone beyond what is feasible to describe by the traditional OpenGL mechanisms of setting state to influence the action of fixed pipeline stages.

A desire to expose the extended capability of the hardware has resulted in a vast number of extensions being written and an unfortunate consequence of this is to reduce, or even eliminate, the portability of applications, thereby undermining one of the key motivating factors for OpenGL.

A natural way of taming this complexity and the proliferation of extensions is to allow parts of the pipeline to be replaced by user programmable stages. This has been done in some recent extensions but the programming is done in assembler, which is a direct expression of today's hardware and not forward looking. Mainstream programmers have progressed from assembler to high-level languages to gain productivity, portability and ease of use. These goals are equally applicable to programming shaders.

The goal of this work is a forward looking hardware independent high-level language that is easy to use and powerful enough to stand the test of time and drastically reduce the need for extensions. These desires must be tempered by the need for fast implementations within a generation or two of hardware.

1.4 Design Considerations

The various programmable processors we are going to introduce replace parts of the OpenGL pipeline and as a starting point they need to be able to do everything they are replacing. This is just the beginning and the examples from the RenderMan community and newer games provide some hints at the exciting possibilities ahead.

To facilitate this, the shading language should be at a high enough level and with the abstractions for the problem domain we are addressing. For graphics this means vector and matrix operations form a fundamental part of the language. This extends from being able to specify scalar/vector/matrix operations directly in expressions to efficient ways to manipulate and group the components of vectors and matrices. The language includes a rich set of built-in functions that operate just as easily on vectors as on scalars.

We are fortunate in having the C language as a base to build on and RenderMan as an existing shading language to learn from. OpenGL is associated with “real-time” graphics (as opposed to off-line graphics) so any aspects of C and RenderMan that hinder efficient compilation or hardware implementation have been dropped, but, for the most part, these are not expected to be noticeable.

The OpenGL Shading Language is designed specifically for use within the OpenGL environment. It is intended to provide programmable alternatives to certain parts of the fixed functionality of OpenGL. By design, it is possible, and quite easy to refer to existing OpenGL state for these parts from within a shader. By design, it is also possible, and quite easy to use fixed functionality in one part of the OpenGL processing pipeline and programmable processing in another. It is the intent that the object code generated for a shader be independent of other OpenGL state, so that recompiles or managing multiple copies of object code are not necessary.

Graphics hardware is developing more and more parallelism at both the vertex and the fragment processing levels. Great care has been taken in the definition of the OpenGL Shading Language to allow for even higher levels of parallel processing.

Finally, it is a goal to use the same high-level programming language for all of the programmable portions of the OpenGL pipeline. Certain types and built-in functions are not permitted on certain programmable processors, but the majority of the language is the same across all programmable processors. This makes it much easier for application developers to embrace the shading language and use it to solve their OpenGL rendering problems.

1.5 Error Handling

Compilers, in general, accept programs that are ill-formed, due to the impossibility of detecting all ill-formed programs. For example, completely accurate detection of use of an uninitialized variable is not possible. Portability is only ensured for well-formed programs, which this specification describes. Compilers are encouraged to detect ill-formed programs and issue diagnostic messages, but are not required to do so for all cases. Compilers are required to return messages regarding lexically, grammatically, or semantically incorrect shaders.

1.6 Typographical Conventions

Italic, bold, and font choices have been used in this specification primarily to improve readability. Code fragments use a fixed width font. Identifiers embedded in text are italicized. Keywords embedded in text are bold. Operators are called by their name, followed by their symbol in bold in parentheses. The clarifying grammar fragments in the text use bold for literals and italics for non-terminals. The official grammar in Section 9 "Shading Language Grammar" uses all capitals for terminals and lower case for non-terminals.

2 Overview of OpenGL Shading

The OpenGL Shading Language is actually two closely related languages. These languages are used to create shaders for the programmable processors contained in the OpenGL processing pipeline. The precise definition of these programmable units is left to separate specifications. In this document, we define them only well enough to provide a context for defining these languages.

Unless otherwise noted in this paper, a language feature applies to all languages, and common usage will refer to these languages as a single language. The specific languages will be referred to by the name of the processor they target: vertex or fragment.

2.1 Vertex Processor

The *vertex processor* is a programmable unit that operates on incoming vertex values and their associated data. The vertex processor is intended to perform traditional graphics operations such as:

- Vertex transformation (modelview and projection matrices)
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

Programs written in the OpenGL Shading Language that are intended to run on this processor are called *vertex shaders*. Vertex shaders can be used to specify a completely general sequence of operations to be applied to each vertex and its associated data. Vertex shaders that perform some of the computations in the list above are responsible for writing the code for all desired functionality from the list above. For instance, it is not possible to use the existing fixed functionality to perform the vertex and normal transformation but have a vertex shader perform a specialized lighting function. The vertex shader must be written to perform all three functions.

The vertex processor does not replace graphics operations that require knowledge of several vertices at a time or that require topological knowledge, such as:

- Perspective division
- viewport mapping
- Primitive assembly
- Frustum and user clipping
- Backface culling
- Two-sided lighting selection
- Polymode processing
- Polygon offset

- Depth Range

Any OpenGL state used by the shader is automatically tracked and made available to the shader. This automatic state tracking mechanism allows the application to use existing OpenGL state commands for state management and have the current values of such state automatically available for use in the vertex shader.

The vertex processor operates on one vertex at a time. The design of the vertex processor is focused on the functionality needed to transform and light a single vertex. Vertex shaders must compute the homogeneous position of the coordinate, and they may also compute color, texture coordinates, and other arbitrary values to be passed to the fragment processor. The output of the vertex processor is sent through subsequent stages of processing that are defined exactly the same as they are for OpenGL 1.4: primitive assembly, user clipping, frustum clipping, perspective projection, viewport mapping, polygon offset, polygon mode, shade mode, and culling. This programmable unit does not have the capability of reading from the frame buffer. However, it does have texture lookup capability. Level of detail is not computed by the implementation for a vertex shader, but can be specified in the shader. The OpenGL parameters for texture maps define the behavior of the filtering operation, borders, and wrapping.

2.2 Fragment Processor

The *fragment processor* is a programmable unit that operates on fragment values and their associated data. The fragment processor is intended to perform traditional graphics operations such as:

- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color sum

Programs written in the OpenGL Shading Language that are intended to run on this processor are called *fragment shaders*. Fragment shaders can be used to specify a completely general sequence of operations to be applied to each fragment. Fragment shaders that perform some of the computations from the list above must perform all desired functionality from the list above. For instance, it is not possible to use the existing fixed functionality to compute fog but have a fragment shader perform specialized texture access and texture application. The fragment shader must be written to perform all three functions.

The fragment processor does not replace the fixed functionality graphics operations that occur at the back end of the OpenGL pixel processing pipeline such as:

- Shading model
- Coverage
- Pixel ownership test
- Scissor
- Stipple
- Alpha test
- Depth test
- Stencil test
- Alpha blending

- Logical ops
- Dithering
- Plane masking

Related OpenGL state is also automatically tracked if used by the shader. A fragment shader cannot change a fragment's x/y position. To support parallelism at the fragment processing level, fragment shaders are written in a way that expresses the computation required for a single fragment, and access to neighboring fragments is not allowed. A fragment shader is free to read multiple values from a single texture, or multiple values from multiple textures. The values computed by the fragment shader are ultimately used to update frame-buffer memory or texture memory, depending on the current OpenGL state and the OpenGL command that caused the fragments to be generated.

The OpenGL parameters for texture maps continue to define the behavior of the filtering operation, borders, and wrapping. These operations are applied when a texture is accessed. The fragment shader is free to use the resulting texel however it chooses. It is possible for a fragment shader to read multiple values from a texture and perform a custom filtering operation. It is also possible to use a texture to perform a lookup table operation. In both cases the texture should have its texture parameters set so that nearest neighbor filtering is applied on the texture access operations.

For each fragment, the fragment shader may compute color and/or depth, or completely discard the fragment.

The results of the fragment shader are then sent on for further processing. The remainder of the OpenGL pipeline remains as defined in OpenGL 1.4. Fragments are submitted to coverage application, pixel ownership testing, scissor testing, alpha testing, stencil testing, depth testing, blending, dithering, logical operations, and masking before ultimately being written into the frame buffer. The primary reason for keeping the fixed functionality at the back end of the processing pipeline is that the fixed functionality is cheap and easy to implement in hardware. Making these functions programmable is more complex, since read/modify/write operations can introduce significant instruction scheduling issues and pipeline stalls. Most of these fixed functionality operations can be disabled, and alternate operations can be performed within a fragment shader if desired.

3 BASICS

3.1 Character Set

The source character set used for the OpenGL shading languages is a subset of ASCII. It includes the following characters:

The letters **a-z**, **A-Z,** and the underscore (_)**.**

The numbers **0-9**.

The symbols period (**.**), plus (**+**), dash (**-**), slash (**/**), asterisk (*****), percent (**%**), angled brackets (**<** and **>**), square brackets (**[** and **]**), parentheses (**(** and **)**), braces (**{** and **}**), caret (**^**), vertical bar (**|**), ampersand (**&**), tilde (**~**), equals (**=**), exclamation point (**!**), colon (**:**), semicolon (**;**), comma (**,**), and question mark (**?**).

The number sign (**#**) for preprocessor use.

White space: the space character, horizontal tab, vertical tab, form feed, carriage-return, and line-feed.

Lines are relevant for compiler diagnostic messages and the preprocessor. They are terminated by carriage-return or line-feed. If both are used together, it will count as only a single line termination. For the remainder of this document, any these combinations is simply referred to as a new-line.

In general, the language's use of this character set is case sensitive.

There are no character or string data types, so no quoting characters are included.

There is no end-of-file character. The end of a source string is indicated by a length, not a character.

3.2 Source Strings

The source for a single shader is an array of strings of characters from the character set. A single shader is made from the concatenation of these strings. Each string can contain multiple lines, separated by new-lines. No new-lines need be present in a string; a single line can be formed from multiple strings. No new-lines or other characters are inserted by the implementation when it concatenates the strings to form a single shader. Multiple shaders of the same language (vertex or fragment) can be linked together to form a single program.

Diagnostic messages returned from compiling a shader must identify both the line number within a string and which source string the message applies to. Source strings are counted sequentially with the first string being string 0. Line numbers are one more than the number of new-lines that have been processed.

3.3 Preprocessor

There is a preprocessor that processes the source strings before they are compiled.

The complete list of preprocessor directives is as follows.

```
#
#define
#undef

#if
#ifdef
#ifndef
#else
#elif
#endif

#error
#pragma

#extension
#version

#line
```

The following operators are also available

```
defined
```

Each number sign (#) can be preceded in its line only by spaces or horizontal tabs. It may also be followed by spaces and horizontal tabs, preceding the directive. Each directive is terminated by a new-line. Preprocessing does not change the number or relative location of new-lines in a source string.

The number sign (#) on a line by itself is ignored. Any directive not listed above will cause a diagnostic message and make the implementation treat the shader as ill-formed.

#define and **#undef** functionality are defined as is standard for C++ preprocessors for macro definitions both with and without macro parameters.

The following predefined macros are available

```
__LINE__
__FILE__
__VERSION__
```

__LINE__ will substitute a decimal integer constant that is one more than the number of preceding new-lines in the current source string.

__FILE__ will substitute a decimal integer constant that says which source string number is currently being processed.

__*VERSION*__ will substitute a decimal integer reflecting the version number of the OpenGL shading language. The version of the shading language described in this document will have __*VERSION*__ substitute the decimal integer 110.

All macro names containing two consecutive underscores (__) are reserved for future use as predefined macro names. All macro names prefixed with "GL_" ("GL" followed by a single underscore) are also reserved.

#if, #ifdef, #ifndef, #else, #elif, and **#endif** are defined to operate as is standard for C++ preprocessors. Expressions following **#if** and **#elif** are restricted to expressions operating on literal integer constants, plus identifiers consumed by the **defined** operator. Character constants are not supported. The operators available are

Precedence	Operator class	Operators	Associativity
1 (highest)	parenthetical grouping	( )	NA
2	unary	defined + - ~ !	Right to Left
3	multiplicative	* / %	Left to Right
4	additive	+ -	Left to Right
5	bit-wise shift	<< >>	Left to Right
6	relational	< > <= >=	Left to Right
7	equality	== !=	Left to Right
8	bit-wise and	&	Left to Right
9	bit-wise exclusive or	^	Left to Right
10	bit-wise inclusive or	|	Left to Right
11	logical and	&&	Left to Right
12	logical inclusive or	||	Left to Right

The **defined** operator can be used in either of the following ways:

```
defined identifier
defined ( identifier )
```

There are no number sign based operators (no **#, #@, ##,** etc.), nor is there a **sizeof** operator.

The semantics of applying operators to integer literals in the preprocessor match those standard in the C++ preprocessor, not those in the OpenGL Shading Language.

Preprocessor expressions will be evaluated according to the behavior of the host processor, not the processor targeted by the shader.

#error will cause the implementation to put a diagnostic message into the shader's information log (see the API in external documentation for how to access a shader's information log). The message will be the tokens following the **#error** directive, up to the first new-line. The implementation must then consider the shader to be ill-formed.

#pragma allows implementation dependent compiler control. Tokens following **#pragma** are not subject to preprocessor macro expansion. If an implementation does not recognize the tokens following **#pragma**, then it will ignore that pragma. The following pragmas are defined as part of the language.

```
#pragma STDGL
```

The **STDGL** pragma is used to reserve pragmas for use by future revisions of this language. No implemention may use a pragma whose first token is **STDGL**.

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

can be used to turn off optimizations as an aid in developing and debugging shaders. It can only be used outside function definitions. By default, optimization is turned on for all shaders. The debug pragma

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

can be used to enable compiling and annotating a shader with debug information, so that it can be used with a debugger. It can only be used outside function definitions. By default, debug is turned off.

Shaders should declare the version of the language they are written to. The language version a shader is written to is specified by

```
#version number
```

where *number* must be 110 for this specification's version of the language (following the same convention as __*VERSION*__ above), in which case the directive will be accepted with no errors or warnings. Any *number* less than 110 will cause an error to be generated. Any *number* greater than the latest version of the language a compiler supports will also cause an error to be generated. Version 110 of the language does not require shaders to include this directive, and shaders that do not include a **#version** directive will be treated as targeting version 110. Compilers for subsequent versions of this language are guaranteed, on seeing the "**#version** 110" directive in a shader, to either support version 110, or to issue an error that they do not support it.

The **#version** directive must occur in a shader before anything else, except for comments and white space.

By default, compilers of this language must issue compile time syntactic, grammatical, and semantic errors for shaders that do not conform to this specification. Any extended behavior must first be enabled. Directives to control the behavior of the compiler with to respect to extensions are declared with the **#extension** directive

```
#extension extension_name : behavior
#extension all : behavior
```

where *extension_name* is the name of an extension. Extension names are not documented in this specification. The token **all** means the behavior applies to all extensions supported by the compiler. The *behavior* can be one of the following

behavior	**Effect**
`require`	Behave as specified by the extension *extension_name.* Give an error on the **#extension** if the extension *extension_name* is not supported, or if **all** is specified.
`enable`	Behave as specified by the extension *extension_name.* Warn on the **#extension** if the extension *extension_name* is not supported. Give an error on the **#extension** if **all** is specified.
`warn`	Behave as specified by the extension *extension_name*, except issue warnings on any detectable use of that extension that is not supported by other enabled or required extensions. If **all** is specified, then warn on all detectable uses of any extension used. Warn on the **#extension** if the extension *extension_name* is not supported.
`disable`	Behave (including issuing errors and warnings) as if the extension *extension_name* is not part of the language definition. If **all** is specified, then behavior must revert back to that of the non-extended core version of the language being compiled to. Warn on the **#extension** if the extension *extension_name* is not supported.

The **extension** directive is a simple, low-level mechanism to set the behavior for each extension. It does not define policies such as which combinations are appropriate, those must be defined elsewhere. Order of directives matters in setting the behavior for each extension: Directives that occur later override those seen earlier. The **all** variant sets the behavior for all extensions, overriding all previously issued **extension** directives, but only for the *behaviors* **warn** and **disable**.

The initial state of the compiler is as if the directive

```
#extension all : disable
```

was issued, telling the compiler that all error and warning reporting must be done according to this specification, ignoring any extensions.

Each extension can define its allowed granularity of scope. If nothing is said, the granularity is a shader (that is, a single compilation unit), and the extension directives must occur before any non-preprocessor tokens. If necessary, the linker can enforce granularities larger than a single compilation unit, in which case each involved shader will have to contain the necessary extension directive.

Macro expansion is not done on lines containing **#extension** and **#version** directives.

#line must have, after macro substitution, one of the following two forms:

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

where *line* and *source-string-number* are constant integer expressions. After processing this directive (including its new-line), the implementation will behave as if it is compiling at line number *line+1* and source string number *source-string-number.* Subsequent source strings will be numbered sequentially, until another **#line** directive overrides that numbering.

3.4 Comments

Comments are delimited by /* and */, or by // and a new-line. The begin comment delimiters (/* or //) are not recognized as comment delimiters inside of a comment, hence comments cannot be nested. If a comment resides entirely within a single line, it is treated syntactically as a single space.

3.5 Tokens

The language is a sequence of tokens. A token can be

token:
   keyword
   identifier
   integer-constant
   floating-constant
   operator

3.6 Keywords

The following are the keywords in the language, and cannot be used for any other purpose than that defined by this document:

attribute const uniform varying

break continue do for while

if else

in out inout

float int void bool true false

discard return

mat2 mat3 mat4

vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4

sampler1D sampler2D sampler3D samplerCube sampler1DShadow sampler2DShadow

struct

The following are the keywords reserved for future use. Using them will result in an error:

asm

class union enum typedef template this packed

goto switch default

inline noinline volatile public static extern external interface

long short double half fixed unsigned

input output

hvec2 hvec3 hvec4 dvec2 dvec3 dvec4 fvec2 fvec3 fvec4

sampler2DRect sampler3DRect sampler2DRectShadow

sizeof cast

namespace using

In addition, all identifiers containing two consecutive underscores (__) are reserved as possible future keywords.

3.7 Identifiers

Identifiers are used for variable names, function names, struct names, and field selectors (field selectors select components of vectors and matrices similar to structure fields, as discussed in Section 5.5 “Vector Components” and Section 5.6 “Matrix Components”). Identifiers have the form

identifier
  nondigit
  identifier nondigit
  identifier digit

nondigit: one of
  _ 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

digit: one of
  0 1 2 3 4 5 6 7 8 9

Identifiers starting with “gl_” are reserved for use by OpenGL, and may not be declared in a shader as either a variable or a function.

4 Variables and Types

All variables and functions must be declared before being used. Variable and function names are identifiers.

There are no default types. All variable and function declarations must have a declared type, and optionally qualifiers. A variable is declared by specifying its type followed by one or more names separated by commas. In many cases, a variable can be initialized as part of its declaration by using the assignment operator (=). The grammar near the end of this document provides a full reference for the syntax of declaring variables.

User-defined types may be defined using **struct** to aggregate a list of existing types into a single name.

The OpenGL Shading Language is type safe. There are no implicit conversions between types.

4.1 Basic Types

The OpenGL Shading Language supports the following basic data types.

void	for functions that do not return a value
bool	a conditional type, taking on values of **true** or **false**
int	a signed integer
float	a single floating-point scalar
vec2	a two component floating-point vector
vec3	a three component floating-point vector
vec4	a four component floating-point vector
bvec2	a two component Boolean vector
bvec3	a three component Boolean vector
bvec4	a four component Boolean vector
ivec2	a two component integer vector
ivec3	a three component integer vector
ivec4	a four component integer vector
mat2	a 2×2 floating-point matrix
mat3	a 3×3 floating-point matrix
mat4	a 4×4 floating-point matrix
sampler1D	a handle for accessing a 1D texture
sampler2D	a handle for accessing a 2D texture
sampler3D	a handle for accessing a 3D texture
samplerCube	a handle for accessing a cube mapped texture

sampler1DShadow	a handle for accessing a 1D depth texture with comparison
sampler2DShadow	a handle for accessing a 2D depth texture with comparison

In addition, a shader can aggregate these using arrays and structures to build more complex types.

There are no pointer types.

4.1.1 Void

Functions that do not return a value must be declared as **void**. There is no default function return type.

4.1.2 Booleans

To make conditional execution of code easier to express, the type **bool** is supported. There is no expectation that hardware directly supports variables of this type. It is a genuine Boolean type, holding only one of two values meaning either true or false. Two keywords **true** and **false** can be used as Boolean constants. Booleans are declared and optionally initialized as in the follow example:

```
bool success;      // declare “success” to be a Boolean
bool done = false; // declare and initialize “done”
```

The right side of the assignment operator (=) can be any expression whose type is **bool**.

Expressions used for conditional jumps (**if, for, ?:, while, do-while**) must evaluate to the type **bool**.

4.1.3 Integers

Integers are mainly supported as a programming aid. At the hardware level, real integers would aid efficient implementation of loops and array indices, and referencing texture units. However, there is no requirement that integers in the language map to an integer type in hardware. It is not expected that underlying hardware has full support for a wide range of integer operations. Because of their intended (limited) purpose, integers are limited to 16 bits of precision, plus a sign representation in both the vertex and fragment languages. An OpenGL Shading Language implementation may convert integers to floats to operate on them. An implementation is allowed to use more than 16 bits of precision to manipulate integers. Hence, there is no portable wrapping behavior. Shaders that overflow the 16 bits of precision may not be portable.

Integers are declared and optionally initialized with integer expressions as in the following example:

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

Literal integer constants can be expressed in decimal (base 10), octal (base 8), or hexadecimal (base 16) as follows.

integer-constant :
  decimal-constant
  octal-constant
  hexadecimal-constant

decimal-constant :
  nonzero-digit
  decimal-constant digit

octal-constant :
  0
  octal-constant octal-digit

hexadecimal-constant :
  0x *hexadecimal-digit*
  0X *hexadecimal-digit*
  hexadecimal-constant hexadecimal-digit

digit :
  0
  nonzero-digit

nonzero-digit : one of
  1 2 3 4 5 6 7 8 9

octal-digit : one of
  0 1 2 3 4 5 6 7

hexadecimal-digit : one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

No white space is allowed between the digits of an integer constant, including after the leading **0** or after the leading **0x** or **0X** of a constant. A leading unary minus sign (-) is interpreted as an arithmetic unary negation, not as part of the constant. There are no letter suffixes.

4.1.4 Floats

Floats are available for use in a variety of scalar calculations. Floating-point variables are defined as in the following example:

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

As an input value to one of the processing units, a floating-point variable is expected to match the IEEE single precision floating-point definition for precision and dynamic range. It is not required that the precision of internal processing match the IEEE floating-point specification for floating-point operations, but the guidelines for precision established by the OpenGL 1.4 specification must be met. Similarly, treatment of conditions such as divide by 0 may lead to an unspecified result, but in no case should such a condition lead to the interruption or termination of processing.

Floating-point constants are defined as follows.

floating-constant :
  fractional-constant exponent-part$_{opt}$
  digit-sequence exponent-part

fractional-constant :
  digit-sequence **.** *digit-sequence*
  digit-sequence **.**
  . *digit-sequence*

exponent-part :
  e *sign$_{opt}$ digit-sequence*
  E *sign$_{opt}$ digit-sequence*

sign : one of
  `+ –`

digit-sequence :
  digit
  digit-sequence digit

A decimal point (**.**) is not needed if the exponent part is present.

4.1.5 Vectors

The OpenGL Shading Language includes data types for generic 2-, 3-, and 4-component vectors of floating-point values, integers, or Booleans. Floating-point vector variables can be used to store a variety of things that are very useful in computer graphics: colors, normals, positions, texture coordinates, texture lookup results and the like. Boolean vectors can be used for component-wise comparisons of numeric vectors. Defining vectors as part of the shading language allows for direct mapping of vector operations on graphics hardware that is capable of doing vector processing. In general, applications will be able to take better advantage of the parallelism in graphics hardware by doing computations on vectors rather than on scalar values. Some examples of vector declaration are:

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

Initialization of vectors can be done with constructors, which are discussed shortly.

4.1.6 Matrices

Matrices are another useful data type in computer graphics, and the OpenGL Shading Language defines support for 2×2, 3×3, and 4×4 matrices of floating point numbers. Matrices are read from and written to in column major order. Example matrix declarations:

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

Initialization of matrix values is done with constructors (described in Section 5.4 “Constructors”).

4.1.7 Samplers

Sampler types (e.g. **sampler2D**) are effectively opaque handles to textures. They are used with the built-in texture functions (described in Section 8.7 “Texture Lookup Functions”) to specify which texture to access. They can only be declared as function parameters or uniforms (see Section 4.3.5 “Uniform”). Samplers are not allowed to be operands in expressions nor can they be assigned into. As uniforms, they are initialized with the OpenGL API. As function parameters, only samplers may be passed to samplers of matching type. This enables consistency checking between shader texture accesses and OpenGL texture state before a shader is run.

4.1.8 Structures

User-defined types can be created by aggregating other already defined types into a structure using the **struct** keyword. For example,

```
struct light {
    float intensity;
    vec3 position;
} lightVar;
```

In this example, *light* becomes the name of the new type, and *lightVar* becomes a variable of type *light*. To declare variables of the new type, use its name (without the keyword **struct**).

```
light lightVar2;
```

More formally, structures are declared as follows. However, the complete correct grammar is as given in Section 9 “Shading Language Grammar”.

struct-definition :
    qualifier$_{opt}$ **struct** *name*$_{opt}$ **{** *member-list* **}** *declarators*$_{opt}$ *;*

member-list :
    member-declaration;
    member-declaration member-list;

member-declaration :
    basic-type declarators;
    embedded-struct-definition

embedded-struct-definition:
    struct *name*$_{opt}$ **{** *member-list* **}** *declarator;*

where *name* becomes the user-defined type, and can be used to declare variables to be of this new type. The *name* shares the same name space as other variables and types, with the same scoping rules. The optional *qualifier* only applies to any *declarators*, and is not part of the type being defined for *name*.

Structures must have at least one member declaration. Member declarators do not contain any qualifiers. Nor do they contain any bit fields. Member types must be either already defined (there are no forward references), or defined in-place by embedding another struct definition. Member declarations cannot contain initializers. Member declarators can contain arrays. Such arrays must have a size specified, and the size must be an integral constant expression that's greater than zero (see Section 4.3.3 “Integral Constant Expressions”). Each level of structure has its own namespace for names given in member declarators; such names need only be unique within that namespace.

Anonymous structures are not supported; so embedded structures must have a declarator. A name given to an embedded struct is scoped at the same level as the struct it is embedded in.

Structures can be initialized at declaration time using constructors, as discussed in Section 5.4.3 “Structure Constructors”.

4.1.9 Arrays

Variables of the same type can be aggregated into arrays by declaring a name followed by brackets (**[ ]**) enclosing an optional size. When an array size is specified in a declaration, it must be an integral constant expression (see Section 4.3.3 “Integral Constant Expressions”) greater than zero. If an array is indexed with an expression that is not an integral constant expression or passed as an argument to a function, then its size must be declared before any such use. It is legal to declare an array without a size and then later re-declare the same name as an array of the same type and specify a size. It is illegal to declare an array with a size, and then later (in the same shader) index the same array with an integral constant expression greater than or equal to the declared size. It is also illegal to index an array with a negative constant expression. Arrays declared as formal parameters in a function declaration must specify a size. Undefined behavior results from indexing an array with a non-constant expression that’s greater than or equal to the array’s size or less than 0. Only one-dimensional arrays may be declared. All basic types and structures can be formed into arrays. Some examples are:

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

There is no mechanism for initializing arrays at declaration time from within a shader.

4.2 Scoping

The scope of a variable is determined by where it is declared. If it is declared outside all function definitions, it has global scope, which starts from where it is declared and persists to the end of the shader it is declared in. If it is declared in a **while** test or a **for** statement, then it is scoped to the end of the following sub-statement. Otherwise, if it is declared as a statement within a compound statement, it is scoped to the end of that compound statement. If it is declared as a parameter in a function definition, it is scoped until the end of that function definition. A function body has a scope nested inside the function’s definition. The **if** statement’s expression does not allow new variables to be declared, hence does not form a new scope.

A variable declared as an empty array can be re-declared as an array of the same base type. Otherwise, within one compilation unit, a variable with the same name cannot be re-declared in the same scope. However, a nested scope can override an outer scope’s declaration of a particular variable name. Declarations in a nested scope provide separate storage from the storage associated with an overridden name.

All variables in the same scope share the same name space. Functions names are always identifiable as function names based on context, and they have their own name space.

Shared globals are global variables declared with the same name in independently compiled units (shaders) of the same language (vertex or fragment) that are linked together to make a single program. Shared globals share the same namespace, and must be declared with the same type. They will share the same storage. Shared global arrays must have the same base type and the same size. Scalars must have exactly the same type name and type definition. Structures must have the same name, sequence of type names, and type definitions, and field names to be considered the same type. This rule applies recursively for nested or embedded types. All initializers for a shared global must have the same value, or a link error will result.

4.3 Type Qualifiers

Variable declarations may have one or more qualifiers, specified in front of the type. These are summarized as

< none: default >	local read/write memory, or an input parameter to a function
const	a compile-time constant, or a function parameter that is read-only
attribute	linkage between a vertex shader and OpenGL for per-vertex data
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application
varying	linkage between a vertex shader and a fragment shader for interpolated data
in	for function parameters passed into a function
out	for function parameters passed back out of a function, but not initialized for use when passed in
inout	for function parameters passed both into and out of a function

Global variables can only use the qualifiers **const**, **attribute**, **uniform**, or **varying**. Only one may be specified.

Local variables can only use the qualifier **const**.

Function parameters can only use the **in, out, inout,** or **const** qualifiers. Parameter qualifiers are discussed in more detail in Section 6.1.1 “Function Calling Conventions”.

Function return types and structure fields do not use qualifiers.

Data types for communication from one run of a shader to its next run (to communicate between fragments or between vertices) do not exist. This would prevent parallel execution of the same shader on multiple vertices or fragments.

Declarations of globals without a qualifier, or with just the **const** qualifier may include initializers, in which case they will be initialized before the first line of *main()* is executed. Such initializers must have constant type. Global variables without qualifiers that are not initialized in their declaration or by the application will not be initialized by OpenGL, but rather will enter *main()* with undefined values.

4.3.1 Default Qualifiers

If no qualifier is present on a global variable, then the variable has no linkage to the application or shaders running on other processors. For either global or local unqualified variables, the declaration will appear to allocate memory associated with the processor it targets. This variable will provide read/write access to this allocated memory.

4.3.2 Const

Named compile-time constants can be declared using the **const** qualifier. Any variables qualified as constant are read-only variables for that shader. Declaring variables as constant allows more descriptive shaders than using hard-wired numerical constants. The **const** qualifier can be used with any of the basic data types. It is an error to write to a **const** variable outside of its declaration, so they must be initialized when declared. For example,

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

Structure fields may not be qualified with **const**. Structure variables can be declared as **const**, and initialized with a structure constructor.

Initializers for **const** declarations must be formed from literal values, other **const** variables (not including function call paramaters), or expressions of these.

Constructors may be used in such expressions, but function calls may not.

4.3.3 Integral Constant Expressions

An integral constant expression can be one of

- a literal integer value
- a global or local scalar integer variable qualified as **const**, not including function parameters qualified as **const**
- an expression whose operands are integral constant expressions, including constructors, but excluding function calls.

4.3.4 Attribute

The **attribute** qualifier is used to declare variables that are passed to a vertex shader from OpenGL on a per-vertex basis. It is an error to declare an attribute variable in any type of shader other than a vertex shader. Attribute variables are read-only as far as the vertex shader is concerned. Values for attribute variables are passed to a vertex shader through the OpenGL vertex API or as part of a vertex array. They convey vertex attributes to the vertex shader and are expected to change on every vertex shader run. The attribute qualifier can be used only with the data types **float, vec2, vec3, vec4, mat2, mat3,** and **mat4**. Attribute variables cannot be declared as arrays or structures.

Example declarations:

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

All the standard OpenGL vertex attributes have built-in variable names to allow easy integration between user programs and OpenGL vertex functions. See Section 7 “Built-in Variables” for a list of the built-in attribute names.

It is expected that graphics hardware will have a small number of fixed locations for passing vertex attributes. Therefore, the OpenGL Shading language defines each non-matrix attribute variable as having space for up to four floating-point values (i.e., a vec4). There is an implementation dependent limit on the number of attribute variables that can be used and if this is exceeded it will cause a link error. (Declared attribute variables that are not used do not count against this limit.) A float attribute counts the same amount against this limit as a vec4, so applications may want to consider packing groups of four unrelated float attributes together into a vec4 to better utilize the capabilities of the underlying hardware. A mat4 attribute will use up the equivalent of 4 vec4 attribute variable locations, a mat3 will use up the equivalent of 3 attribute variable locations, and a mat2 will use up 2 attribute variable locations. How this space is utilized by the matrices is hidden by the implementation through the API and language.

Attribute variables are required to have global scope, and must be declared outside of function bodies, before their first use.

4.3.5 Uniform

The **uniform** qualifier is used to declare global variables whose values are the same across the entire primitive being processed. All **uniform** variables are read-only and are initialized either directly by an application via API commands, or indirectly by OpenGL.

An example declaration is:

```
uniform vec4 lightPosition;
```

The **uniform** qualifier can be used with any of the basic data types, or when declaring a variable whose type is a structure, or an array of any of these.

There is an implementation dependent limit on the amount of storage for uniforms that can be used for each type of shader and if this is exceeded it will cause a compile-time or link-time error. Uniform variables that are declared but not used do not count against this limit. The number of user-defined uniform variables and the number of built-in uniform variables that are used within a shader are added together to determine whether available uniform storage has been exceeded.

If multiple shaders are linked together, then they will share a single global uniform name space. Hence, types of uniforms with the same name must match across all shaders that are linked into a single executable.

4.3.6 Varying

Varying variables provide the interface between the vertex shader, the fragment shader, and the fixed functionality between them. The vertex shader will compute values per vertex (such as color, texture coordinates, etc.) and write them to variables declared with the **varying** qualifier. A vertex shader may also read **varying** variables, getting back the same values it has written. Reading a **varying** variable in a vertex shader returns undefined values if it is read before being written.

By definition, varying variables are set per vertex and are interpolated in a perspective-correct manner over the primitive being rendered. If single-sampling, the interpolated value is for the fragment center. If multi-sampling, the interpolated value can be anywhere within the pixel, including the fragment center or one of the fragment samples.

A fragment shader may read from varying variables and the value read will be the interpolated value, as a function of the fragment's position within the primitive. A fragment shader can not write to a varying variable.

The type of varying variables with the same name declared in both the vertex and fragments shaders must match, otherwise the link command will fail. Only those varying variables used (i.e. read) in the fragment shader must be written to by the vertex shader; declaring superfluous varying variables in the vertex shader is permissible.

Varying variables are declared as in the following example:

```
varying vec3 normal;
```

The **varying** qualifier can be used only with the data types **float, vec2, vec3, vec4, mat2, mat3**, and **mat4**, or arrays of these. Structures cannot be **varying**.

If no vertex shader is active, the fixed functionality pipeline of OpenGL will compute values for the built-in varying variables that will be consumed by the fragment shader. Similarly, if no fragment shader is active, the vertex shader is responsible for computing and writing to the varying variables that are needed for OpenGL's fixed functionality fragment pipeline.

Varying variables are required to have global scope, and must be declared outside of function bodies, before their first use.

5 Operators and Expressions

5.1 Operators

The OpenGL Shading Language has the following operators. Those marked reserved are illegal.

Precedence	Operator class	Operators	Associativity
1 (highest)	parenthetical grouping	( )	NA
2	array subscript function call and constructor structure field selector, swizzler post fix increment and decrement	[ ] ( ) . ++ --	Left to Right
3	prefix increment and decrement unary (tilde is reserved)	++ -- + - ~ !	Right to Left
4	multiplicative (modulus reserved)	* / %	Left to Right
5	additive	+ -	Left to Right
6	bit-wise shift (reserved)	<< >>	Left to Right
7	relational	< > <= >=	Left to Right
8	equality	== !=	Left to Right
9	bit-wise and (reserved)	&	Left to Right
10	bit-wise exclusive or (reserved)	^	Left to Right
11	bit-wise inclusive or (reserved)	|	Left to Right
12	logical and	&&	Left to Right
13	logical exclusive or	^^	Left to Right
14	logical inclusive or	| |	Left to Right
15	selection	? :	Right to Left
16	assignment arithmetic assignments (modulus, shift, and bit-wise are reserved)	= += -= *= /= %= <<= >>= &= ^= |=	Right to Left
17 (lowest)	sequence	,	Left to Right

There is no address-of operator nor a dereference operator. There is no typecast operator, constructors are used instead.

5.2 Array Subscripting

Array elements are accessed using the array subscript operator ([ ]). This is the only operator that operates on arrays. An example of accessing an array element is

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

Array indices start at zero. Arrays elements are accessed using an expression whose type is an integer.

Behavior is undefined if a shader subscripts an array with an index less than 0 or greater than or equal to the size the array was declared with.

5.3 Function Calls

If a function returns a value, then a call to that function may be used as an expression, whose type will be the type that was used to declare or define the function.

Function definitions and calling conventions are discussed in Section 6.1 “Function Definitions”.

5.4 Constructors

Constructors use the function call syntax, where the function name is a basic-type keyword or structure name, to make values of the desired type for use in an initializer or an expression. (See Section 9 “Shading Language Grammar” for details.) The parameters are used to initialize the constructed value. Constructors can be used to request a data type conversion to change from one scalar type to another scalar type, or to build larger types out of smaller types, or to reduce a larger type to a smaller type.

There is no fixed list of constructor prototypes. Constructors are not built-in functions. Syntactically, all lexically correct parameter lists are valid. Semantically, the number of parameters must be of sufficient size and correct type to perform the initialization. It is an error to include so many arguments to a constructor that they cannot all be used. Detailed rules follow. The prototypes actually listed below are merely a subset of examples.

5.4.1 Conversion and Scalar Constructors

Converting between scalar types is done as the following prototypes indicate:

```
int(bool)   // converts a Boolean value to an int
int(float)  // converts a float value to an int
float(bool) // converts a Boolean value to a float
float(int)  // converts an integer value to a float
bool(float) // converts a float value to a Boolean
bool(int)   // converts an integer value to a Boolean
```

When constructors are used to convert a **float** to an **int**, the fractional part of the floating-point value is dropped.

When a constructor is used to convert an **int** or a **float** to **bool**, 0 and 0.0 are converted to **false**, and non-zero values are converted to **true**. When a constructor is used to convert a **bool** to an **int** or **float**, **false** is converted to 0 or 0.0, and **true** is converted to 1 or 1.0.

Identity constructors, like float(float) are also legal, but of little use.

Scalar constructors with non-scalar parameters can be used to take the first element from a non-scalar. For example, the constructor float(vec3) will select the first component of the vec3 parameter.

5.4.2 Vector and Matrix Constructors

Constructors can be used to create vectors or matrices from a set of scalars, vectors, or matrices. This includes the ability to shorten vectors.

If there is a single scalar parameter to a vector constructor, it is used to initialize all components of the constructed vector to that scalar's value. If there is a single scalar parameter to a matrix constructor, it is used to initialize all the components on the matrix's diagonal, with the remaining components initialized to 0.0. If there are non-scalar parameters, and/or multiple scalar parameters, they will be assigned in order, from left to right, to the components of the constructed value. In this case, there must be enough components provided in the parameters to provide an initializer for every component in the constructed value. If more components are provided in the last used argument to a constructor than are needed to initialize the constructed value, the left most components of that argument are used, and the remaining ones are ignored. It is an error to provide extra arguments beyond this last used argument. Matrices will be constructed in column major order. It is an error to construct matrices from other matrices. This is reserved for future use.

If the basic type (**bool, int,** or **float**) of a parameter to a constructor does not match the basic type of the object being constructed, the scalar construction rules (above) are used to convert the parameters.

Some useful vector constructors are as follows:

```
vec3(float)    // initializes each component of a vec3 with the float
vec4(ivec4)    // makes a vec4 from an ivec4, with component-wise conversion

vec2(float, float)              // initializes a vec2 with 2 floats
ivec3(int, int, int)            // initializes an ivec3 with 3 ints
bvec4(int, int, float, float)   // initializes with 4 Boolean conversions

vec2(vec3) // drops the third component of a vec3
vec3(vec4) // drops the fourth component of a vec4

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

Some examples of these are:

```
vec4color = vec4(0.0, 1.0, 0.0, 1.0);
vec4 rgba  = vec4(1.0);     // sets each component to 1.0
```

```
vec3 rgb   = vec3(color);    // drop the 4th component
```

To initialize the diagonal of a matrix with all other elements set to zero:

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

To initialize a matrix by specifying vectors, or by all 4, 9, or 16 floats for mat2, mat3 and mat4 respectively. The floats are assigned to elements in column major order.

```
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, float);
```

A wide range of other possibilities exist, as long as enough components are present to initialize the matrix. However, construction of a matrix from other matrices is currently reserved for future use.

5.4.3 Structure Constructors

Once a structure is defined, and its type is given a name, a constructor is available with the same name to construct instances of that structure. For example:

```
struct light {
    float intensity;
    vec3 position;
};

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

The arguments to the constructor must be in the same order and of the same type as they were declared in the structure.

Structure constructors can be used as initializers or in expressions.

5.5 Vector Components

The names of the components of a vector are denoted by a single letter. As a notational convenience, several letters are associated with each component based on common usage of position, color or texture coordinate vectors. The individual components of a vector can be selected by following the variable name with period (**.**) and then the component name.

The component names supported are:

{*x, y, z, w*}	useful when accessing vectors that represent points or normals
{*r, g, b, a*}	useful when accessing vectors that represent colors
{*s, t, p, q*}	useful when accessing vectors that represent texture coordinates

The component names *x*, *r*, and *s* are, for example, synonyms for the same (first) component in a vector.

Note that the third component of a texture, *r* in OpenGL, has been renamed *p* so as to avoid the confusion with *r* (for red) in a color.

Accessing components beyond those declared for the vector type is an error so, for example:

```
vec2 pos;
pos.x    // is legal
pos.z    // is illegal
```

The component selection syntax allows multiple components to be selected by appending their names (from the same name set) after the period (**.**).

```
vec4 v4;
v4.rgba;  // is a vec4 and the same as just using v4,
v4.rgb;   // is a vec3,
v4.b;     // is a float,
v4.xy;    // is a vec2,
v4.xgba;  // is illegal - the component names do not come from
          //              the same set.
```

The order of the components can be different to swizzle them, or replicated:

```
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)
```

This notation is more concise than the constructor syntax. To form an r-value, it can be applied to any expression that results in a vector r-value.

The component group notation can occur on the left hand side of an expression.

```
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);     // illegal - 'x' used twice
pos.xy = vec3(1.0, 2.0, 3.0);// illegal - mismatch between vec2 and vec3
```

To form an l-value, swizzling must be applied to an l-value of vector type, contain no duplicate components, and results in an l-value of scalar or vector type, depending on number of components specified.

Array subscripting syntax can also be applied to vectors to provide numeric indexing. So in

```
vec4 pos;
```

pos[2] refers to the third element of pos and is equivalent to *pos.z*. This allows variable indexing into a vector, as well as a generic way of accessing components. Any integer expression can be used as the subscript. The first component is at index zero. Behavior is undefined if the index is greater than or equal to the size of the vector.

5.6 Matrix Components

The components of a matrix can be accessed using array subscripting syntax. Applying a single subscript to a matrix treats the matrix as an array of column vectors, and selects a single column, whose type is a vector of the same size as the matrix. The leftmost column is column 0. A second subscript would then operate on the column vector, as defined earlier for vectors. Hence, two subscripts select a column and then a row.

```
mat4 m;
m[1] = vec4(2.0);        // sets the second column to all 2.0
m[0][0] = 1.0;           // sets the upper left element to 1.0
m[2][3] = 2.0;           // sets the 4th element of the third column to 2.0
```

Behavior is undefined when accessing a component outside the bounds of a matrix (e.g., component [3][3] of a mat3).

5.7 Structures and Fields

As with vector components and swizzling, the fields of a structure are also selected using the period (**.**).

In total, the following operators are allowed to operate on a structure:

structure field selector	**.**
equality	**== !=**
assignment	**=**

The equality and assignment operators are only valid if the two operands' types are of the same declared structure. When using the equality operators, two structures are equal if and only if all the fields are component-wise equal.

5.8 Assignments

Assignments of values to variable names are done with the assignment operator (**=**), like

```
lvalue = expression
```

The assignment operator stores the value of *expression* into *lvalue*. It will compile only if *expression* and *lvalue* have the same type. All desired type-conversions must be specified explicitly via a constructor. L-values must be writable. Variables that are built-in types, entire structures, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, and l-values dereferenced with the array subscript operator (**[ ]**) are all l-values. Other binary or unary expressions, non-dereferenced arrays, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (**?:**) is also not allowed as an l-value.

Expressions on the left of an assignment are evaluated before expressions on the right of the assignment. Other assignment operators are

- The arithmetic assignments add into (**+=**), subtract from (**-=**), multiply into (***=**), and divide into (**/=**). The expression

  ```
  lvalue op= expression
  ```

  is equivalent to

  ```
  lvalue = lvalue op expression
  ```

  and the l-value and expression must satisfy the semantic requirements of both *op* and equals (**=**).
- The assignments modulus into (**%=**), left shift by (<<=), right shift by (>>=), inclusive or into (|=), and exclusive or into (^=). These operators are reserved for future use.

Reading a variable before writing (or initializing) it is legal, however the value is undefined.

5.9 Expressions

Expressions in the shading language are built from the following:

- Constants of type **bool, int, float,** all vector types, and all matrix types.
- Constructors of all types.
- Variable names of all types, except array names not followed by a subscript.
- Subscripted array names.
- Function calls that return values.
- Component field selectors and array subscript results.
- Parenthesized expression. Parentheses can be used to group operations. Operations within parentheses are done before operations across parentheses.
- The arithmetic binary operators add (**+**), subtract (**-**), multiply (*****), and divide (**/**), that operate on integer and floating-point typed expressions (including vectors and matrices). The two operands must be the same type, or one can be a scalar float and the other a float vector or matrix, or one can be a scalar integer and the other an integer vector. Additionally, for multiply (*****), one can be a vector and the other a matrix with the same dimensional size of the vector. These result in the same fundamental type (integer or float) as the expressions they operate on. If one operand is scalar and the other is a vector or matrix, the scalar is applied component-wise to the vector or matrix, resulting in the same type as the vector or matrix. Dividing by zero does not cause an exception but does result in an unspecified

value. Multiply (*****) applied to two vectors yields a component-wise multiply. Multiply (*****) applied to two matrices yields a linear algebraic matrix multiply, not a component-wise multiply. Use the built-in functions **dot, cross, and matrixCompMult** to get, respectively, vector dot product, vector cross product, and matrix component-wise multiplication.

- The operator modulus (**%**) is reserved for future use.
- The arithmetic unary operators negate (-), post- and pre-increment and decrement (**--** and **++**) that operate on integer or floating-point values (including vectors and matrices). These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract 1 or 1.0 to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 or 1.0 to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.
- The relational operators greater than (**>**), less than (**<**), greater than or equal (**>=**), and less than or equal (**<=**) operate only on scalar integer and scalar floating-point expressions. The result is scalar Boolean. The operands' types must match. To do component-wise comparisons on vectors, use the built-in functions **lessThan, lessThanEqual, greaterThan,** and **greaterThanEqual.**
- The equality operators equal (**==**), and not equal (**!=**) operate on all types except arrays. They result in a scalar Boolean. For vectors, matrices, and structures, all components of the operands must be equal for the operands to be considered equal. To get component-wise equality results for vectors, use the built-in functions **equal** and **notEqual**.
- The logical binary operators and (**&&**), or (**||**), and exclusive or (**^^**). They operate only on two Boolean expressions and result in a Boolean expression. And (**&&**) will only evaluate the right hand operand if the left hand operand evaluated to **true**. Or (**||**) will only evaluate the right hand operand if the left hand operand evaluated to **false**. Exclusive or (**^^**) will always evaluate both operands.
- The logical unary operator not (**!**). It operates only on a Boolean expression and results in a Boolean expression. To operate on a vector, use the built-in function **not**.
- The sequence (**,**) operator that operates on expressions by returning the type and value of the right-most expression in a comma separated list of expressions. All expressions are evaluated, in order, from left to right.
- The ternary selection operator (**?:**). It operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression, which must result in a scalar Boolean. If the result is true, it selects to evaluate the second expression, otherwise it selects to evaluate the third expression. Only one of the second and third expressions is evaluated. The second and third expressions must be the same type, but can be of any type other than an array. The resulting type is the same as the type of the second and third expressions.
- Operators and (**&**), or (**|**), exclusive or (**^**), not (**~**), right-shift (**>>**), left-shift (**<<**). These operators are reserved for future use.

For a complete specification of the syntax of expressions, see Section 9 "Shading Language Grammar".

When the operands are of a different type they must fit into one of the following rules:

- one of the arguments is a float (i.e. a scalar), in which case the result is as if the scalar value was replicated into a vector or matrix before being applied.
- the left argument is a floating-point vector and the right is a matrix with a compatible dimension in which case the * operator will do a row vector matrix multiplication.
- the left argument is a matrix and the right is a floating-point vector with a compatible dimension in which case the * operator will do a column vector matrix multiplication.

5.10 Vector and Matrix Operations

With a few exceptions, operations are component-wise. When an operator operates on a vector or matrix, it is operating independently on each component of the vector or matrix, in a component-wise fashion. For example,

```
vec3 v, u;
float f;

v = u + f;
```

will be equivalent to

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

And

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

will be equivalent to

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

and likewise for most operators and all integer and floating point vector and matrix types. The exceptions are matrix multiplied by vector, vector multiplied by matrix, and matrix multiplied by matrix. These do not operate component-wise, but rather perform the correct linear algebraic multiply. They require the size of the operands match.

```
vec3 v, u;
mat3 m;

u = v * m;
```

is equivalent to

```
u.x = dot(v, m[0]); // m[0] is the left column of m
u.y = dot(v, m[1]); // dot(a,b) is the inner (dot) product of a and b
u.z = dot(v, m[2]);
```

And

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

is equivalent to

```
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;
```

And

```
mat m, n, r;

r = m * n;
```

is equivalent to

```
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;
```

and similarly for vectors and matrices of size 2 and 4.

All unary operations work component-wise on their operands. For binary arithmetic operations, if the two operands are the same type, then the operation is done component-wise and produces a result that is the same type as the operands. If one operand is a scalar float and the other operand is a vector or matrix, then the operation proceeds as if the scalar value was replicated to form a matching vector or matrix operand.

6 Statements and Structure

The fundamental building blocks of the OpenGL Shading Language are:

- statements and declarations
- function definitions
- selection (**if-else)**
- iteration (**for, while,** and **do-while)**
- jumps (**discard, return, break,** and **continue**)

The overall structure of a shader is as follows

translation-unit:
    global-declaration
    translation-unit global-declaration

global-declaration:
    function-definition
    declaration

That is, a shader is a sequence of declarations and function bodies. Function bodies are defined as

function-definition:
    function-prototype { statement-list }

statement-list:
    statement
    statement-list statement

statement:
    compound-statement
    simple-statement

Curly braces are used to group sequences of statements into compound statements.

compound-statement:
    { statement-list }

simple-statement:
    declaration-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement

Simple declaration, expression, and jump statements end in a semi-colon.

This above is slightly simplified, and the complete grammar specified in Section 9 “Shading Language Grammar” should be used as the definitive specification.

Declarations and expressions have already been discussed.

6.1 Function Definitions

As indicated by the grammar above, a valid shader is a sequence of global declarations and function definitions. A function is declared as the following example shows:

```
// prototype
returnType functionName (type0 arg0, type1 arg1, ..., typen argn);
```

and a function is defined like

```
// definition
returnType functionName (type0 arg0, type1 arg1, ..., typen argn)
{
    // do some computation
    return returnValue;
}
```

Where *returnType* must be present and include a type. Each of the *typeN* must include a type and can optionally include the qualifier **in**, **out**, **inout**, and/or **const**.

A function is called by using its name followed by a list of arguments in parentheses.

Arrays are allowed as arguments, but not as the return type. When arrays are declared as formal parameters, their size must be included. An array is passed to a function by using the array name without any subscripting or brackets, and the size of the array argument passed in must match the size specified in the formal parameter declaration.

Structures are also allowed as arguments. The return type can also be structure.

See Section 9 “Shading Language Grammar” for the definitive reference on the syntax to declare and define functions.

All functions must be either declared with a prototype or defined with a body before they are called. For example:

```
float myfunc (float f,        // f is an input parameter
              out float g);   // g is an output parameter
```

Functions that return no value must be declared as **void**. Functions that accept no input arguments need not use **void** in the argument list because prototypes are required and therefore there is no ambiguity when an empty argument list "( )" is declared. The idiom “(**void**)” as a parameter list is provided for convenience.

Function names can be overloaded. This allows the same function name to be used for multiple functions, as long as the argument list types differ. If functions' names and argument types match, then their return type and parameter qualifiers must also match. Overloading is used heavily in the built-in functions. When overloaded functions (or indeed any functions) are resolved, an exact match for the function's signature is sought. This includes exact match of array size as well. No promotion or demotion of the return type or input argument types is done. All expected combination of inputs and outputs must be defined as separate functions.

For example, the built-in dot product function has the following prototypes:

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

User-defined functions can have multiple declarations, but only one definition. A shader can redefine built-in functions. If a built-in function is redeclared in a shader (i.e. a prototype is visible) before a call to it, then the linker will only attempt to resolve that call within the set shaders that are linked with it.

The function *main* is used as the entry point to a shader. A shader need not contain a function named *main*, but one shader in a set of shaders linked together to form a single program must. This function takes no arguments, returns no value, and must be declared as type **void:**

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

The function *main* can contain uses of **return**. See Section 6.4 "Jumps" for more details.

6.1.1 Function Calling Conventions

Functions are called by value-return. This means input arguments are copied into the function at call time, and output arguments are copied back to the caller before function exit. Because the function works with local copies of parameters, there are no issues regarding aliasing of variables within a function. At call time, input arguments are evaluated in order, from left to right. However, the order in which output parameters are copied back to the caller is undefined. To control what parameters are copied in and/or out through a function definition or declaration:

- The keyword **in** is used as a qualifier to denote a parameter is to be copied in, but not copied out.
- The keyword **out** is used as a qualifier to denote a parameter is to be copied out, but not copied in. This should be used whenever possible to avoid unnecessarily copying parameters in.
- The keyword **inout** is used as a qualifier to denote the parameter is to be both copied in and copied out.
- A function parameter declared with no such qualifier means the same thing as specifying **in**.

In a function, writing to an input-only parameter is allowed. Only the function's copy is modified. This can be prevented by declaring a parameter with the **const** qualifier.

When calling a function, expressions that do not evaluate to l-values cannot be passed to parameters declared as **out** or **inout**.

No qualifier is allowed on the return type of a function.

function-prototype :
  type function-name(const-qualifier parameter-qualifier type name array-specifier, ...)

type :
  any basic type, structure name, or structure definition

const-qualifier :
  empty
  const

parameter-qualifier :
  empty
  in
  out
  inout

name :
  empty
  identifier

array-specifier :
  empty
  [*integral-constant-expression* **]**

However, the **const** qualifier cannot be used with **out** or **inout**. The above is used for function declarations (i.e. prototypes) and for function definitions. Hence, function definitions can have unnamed arguments.

Behavior is undefined if recursion is used. Recursion means having any function appearing more than once at any one time in the run-time stack of function calls. That is, a function may not call itself either directly or indirectly. Compilers may give diagnostic messages when this is detectable at compile time, but not all such cases can be detected at compile time.

6.2 Selection

Conditional control flow in the shading language is done by either if, or if-else:

```
if (bool-expression)
     true-statement
```

or

```
if (bool-expression)
     true-statement
else
```

```
        false-statement
```

If the expression evaluates to **true**, then *true-statement* is executed. If it evaluates to **false** and there is an **else** part then *false-statement* is executed.

Any expression whose type evaluates to a Boolean can be used as the conditional expression *bool-expression*. Vector types are not accepted as the expression to **if**.

Conditionals can be nested.

6.3 Iteration

For, while, and do loops are allowed as follows:

```
for (init-expression; condition-expression; loop-expression)
     sub-statement

while (condition-expression)
     sub-statement

do
     statement
while (condition-expression)
```

See Section 9 “Shading Language Grammar” for the definitive specification of loops.

The **for** loop first evaluates the *init-expression*, then the *condition-expression*. If the *condition-expression* evaluates to true, then the body of the loop is executed. After the body is executed, a **for** loop will then evaluate the *loop-expression*, and then loop back to evaluate the *condition-expression*, repeating until the *condition-expression* evaluates to false. The loop is then exited, skipping its body and skipping its *loop-expression*. Variables modified by the *loop-expression* maintain their value after the loop is exited, provided they are still in scope. Variables declared in *init-expression* or *condition-expression* are only in scope until the end of the sub-statement of the **for** loop.

The **while** loop first evaluates the *condition-expression*. If true, then the body is executed. This is then repeated, until the *condition-expression* evaluates to false, exiting the loop and skipping its body. Variables declared in the *condition-expression* are only in scope until the end of the sub-statement of the while loop.

The **do-while** loop first executes the body, then executes the *condition-expression.* This is repeated until *condition-expression* evaluates to false, and then the loop is exited.

Expressions for *condition-expression* must evaluate to a Boolean.

Both the *condition-expression* and the *init-expression* can declare and initialize a variable, except in the **do-while** loop, which cannot declare a variable in its *condition-expression.* The variable’s scope lasts only until the end of the sub-statement that forms the body of the loop.

Loops can be nested.

Non-terminating loops are allowed. The consequences of very long or non-terminating loops are platform dependent.

6.4 Jumps

These are the jumps:

jump_statement:
  continue;
  break;
  return;
  return *expression***;**
  discard; // in the fragment shader language only

There is no "goto" nor other non-structured flow of control.

The **continue** jump is used only in loops. It skips the remainder of the body of the inner most loop of which it is inside. For **while** and **do-while** loops, this jump is to the next evaluation of the loop *condition-expression* from which the loop continues as previously defined. For **for** loops, the jump is to the *loop-expression*, followed by the *condition-expression.*

The **break** jump can also be used only in loops. It is simply an immediate exit of the inner-most loop containing the **break**. No further execution of *condition-expression* or *loop-expression* is done.

The **discard** keyword is only allowed within fragment shaders. It can be used within a fragment shader to abandon the operation on the current fragment. This keyword causes the fragment to be discarded and no updates to any buffers will occur. It would typically be used within a conditional statement, for example:

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

A fragment shader may test a fragment's alpha value and discard the fragment based on that test. However, it should be noted that coverage testing occurs after the fragment shader runs, and the coverage test can change the alpha value.

The **return** jump causes immediate exit of the current function. If it has *expression* then that is the return value for the function.

The function *main* can use **return**. This simply causes *main* to exit in the same way as when the end of the function had been reached. It does not imply a use of **discard** in a fragment shader. Using **return** in main before defining outputs will have the same behavior as reaching the end of main before defining outputs.

7 BUILT-IN VARIABLES

7.1 Vertex Shader Special Variables

Some OpenGL operations still continue to occur in fixed functionality in between the vertex processor and the fragment processor. Other OpenGL operations continue to occur in fixed functionality after the fragment processor. Shaders communicate with the fixed functionality of OpenGL through the use of built-in variables.

The variable *gl_Position* is available only in the vertex language and is intended for writing the homogeneous vertex position. All executions of a well-formed vertex shader must write a value into this variable. It can be written at any time during shader execution. It may also be read back by the shader after being written. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations that operate on primitives after vertex processing has occurred. Compilers may generate a diagnostic message if they detect *gl_Position* is not written, or read before being written, but not all such cases are detectable. Results are undefined if a vertex shader is executed and does not write *gl_Position.*

The variable *gl_PointSize* is available only in the vertex language and is intended for a vertex shader to write the size of the point to be rasterized. It is measured in pixels.

The variable *gl_ClipVertex* is available only in the vertex language and provides a place for vertex shaders to write the coordinate to be used with the user clipping planes. The user must ensure the clip vertex and user clipping planes are defined in the same coordinate space. User clip planes work properly only under linear transform. It is undefined what happens under non-linear transform.

These built-in vertex shader variables for communicating with fixed functionality are intrinsically declared with the following types:

```
vec4  gl_Position;    // must be written to
float gl_PointSize;   // may be written to
vec4  gl_ClipVertex;  // may be written to
```

If *gl_PointSize* or *gl_ClipVertex* are not written to, their values are undefined. Any of these variables can be read back by the shader after writing to them, to retrieve what was written. Reading them before writing them results in undefined behavior. If they are written more than once, it is the last value written that is consumed by the subsequent operations.

These built-in variables have global scope.

7.2 Fragment Shader Special Variables

The output of the fragment shader is processed by the fixed function operations at the back end of the OpenGL pipeline. Fragment shaders output values to the OpenGL pipeline using the built-in variables *gl_FragColor, gl_FragData,* and *gl_FragDepth*, unless the **discard** keyword is executed.

These variables may be written more than once within a fragment shader. If so, the last value assigned is the one used in the subsequent fixed function pipeline. The values written to these variables may be read back after writing them. Reading from these variables before writing them results in an undefined value. The fixed functionality computed depth for a fragment may be obtained by reading *gl_FragCoord.z,* described below.

Writing to *gl_FragColor* specifies the fragment color that will be used by the subsequent fixed functionality pipeline. If subsequent fixed functionality consumes fragment color and an execution of a fragment shader does not write a value to *gl_FragColor* then the fragment color consumed is undefined.

If the frame buffer is configured as a color index buffer then behavior is undefined when using a fragment shader.

Writing to *gl_FragDepth* will establish the depth value for the fragment being processed. If depth buffering is enabled, and a shader does not write *gl_FragDepth*, then the fixed function value for depth will be used as the fragment's depth value. If a shader statically assigns a value to *gl_FragDepth*, and there is an execution path through the shader that does not set *gl_FragDepth,* then the value of the fragment's depth may be undefined for executions of the shader that take that path. That is, if a shader statically contains a write to *gl_FragDepth*, then it is responsible for always writing it.

(A shader contains a *static assignment* to a variable *x* if, after pre-processing, the shader contains a statement that would write to *x*, whether or not run-time flow of control will cause that statement to be executed.)

The variable *gl_FragData* is an array. Writing to *gl_FragData[n]* specifies the fragment data that will be used by the subsequent fixed functionality pipeline for data *n*. If subsequent fixed functionality consumes fragment data and an execution of a fragment shader does not write a value to it, then the fragment data consumed is undefined.

If a shader statically assigns a value to *gl_FragColor,* it may not assign a value to any element of *gl_FragData*. If a shader statically writes a value to any element of *gl_FragData*, it may not assign a value to *gl_FragColor.* That is, a shader may assign values to either *gl_FragColor* or *gl_FragData*, but not both.

If a shader executes the **discard** keyword, the fragment is discarded, and the values of *gl_FragDepth, gl_FragColor,* and *gl_FragData* become irrelevant.

The variable *gl_FragCoord* is available as a read-only variable from within fragment shaders and it holds the window relative coordinates x, y, z, and 1/w values for the fragment. This value is the result of the fixed functionality that interpolates primitives after vertex processing to generate fragments. The *z* component is the depth value that would be used for the fragment's depth if a shader contained no writes to *gl_FragDepth*. This is useful for invariance if a shader conditionally computes *gl_FragDepth* but otherwise wants the fixed functionality fragment depth.

The fragment shader has access to the read-only built-in variable *gl_FrontFacing* whose value is **true** if the fragment belongs to a front-facing primitive. One use of this is to emulate two-sided lighting by selecting one of two colors calculated by the vertex shader.

The built-in variables that are accessible from a fragment shader are intrinsically given types as follows:

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

However, they do not behave like variables with no qualifier; their behavior is as described above. These built-in variables have global scope.

7.3 Vertex Shader Built-In Attributes

The following attribute names are built into the OpenGL vertex language and can be used from within a vertex shader to access the current values of attributes declared by OpenGL. All page numbers and notations are references to the OpenGL 1.4 specification.

```
//
// Vertex Attributes, p. 19.
//
attribute vec4  gl_Color;
attribute vec4  gl_SecondaryColor;
attribute vec3  gl_Normal;
attribute vec4  gl_Vertex;
attribute vec4  gl_MultiTexCoord0;
attribute vec4  gl_MultiTexCoord1;
attribute vec4  gl_MultiTexCoord2;
attribute vec4  gl_MultiTexCoord3;
attribute vec4  gl_MultiTexCoord4;
attribute vec4  gl_MultiTexCoord5;
attribute vec4  gl_MultiTexCoord6;
attribute vec4  gl_MultiTexCoord7;
attribute float gl_FogCoord;
```

7.4 Built-In Constants

The following built-in constants are provided to vertex and fragment shaders.

```
//
// Implementation dependent constants.  The example values below
// are the minimum values allowed for these maximums.
//
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;              // proposed ARB_draw_buffers
```

7.5 Built-In Uniform State

As an aid to accessing OpenGL processing state, the following uniform variables are built into the OpenGL Shading Language. All page numbers and notations are references to the 1.4 specification.

```
//
// Matrix state. p. 31, 32, 37, 39, 40.
//
uniform mat4  gl_ModelViewMatrix;
uniform mat4  gl_ProjectionMatrix;
uniform mat4  gl_ModelViewProjectionMatrix;
uniform mat4  gl_TextureMatrix[gl_MaxTextureCoords];

//
// Derived matrix state that provides inverse and transposed versions
// of the matrices above.  Poorly conditioned matrices may result
// in unpredictable values in their inverse forms.
//
uniform mat3  gl_NormalMatrix; // transpose of the inverse of the
                               // upper leftmost 3x3 of gl_ModelViewMatrix

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_TextureMatrixTranspose[gl_MaxTextureCoords];

uniform mat4  gl_ModelViewMatrixInverseTranspose;
uniform mat4  gl_ProjectionMatrixInverseTranspose;
uniform mat4  gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4  gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];


//
// Normal scaling p. 39.
//
uniform float gl_NormalScale;

//
```

```
// Depth range in window coordinates, p. 33
//
struct gl_DepthRangeParameters {
    float near;        // n
    float far;         // f
    float diff;        // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;

//
// Clip planes p. 42.
//
uniform vec4  gl_ClipPlane[gl_MaxClipPlanes];

//
// Point Size, p. 66, 67.
//
struct gl_PointParameters {
    float size;
    float sizeMin;
    float sizeMax;
    float fadeThresholdSize;
    float distanceConstantAttenuation;
    float distanceLinearAttenuation;
    float distanceQuadraticAttenuation;
};

uniform gl_PointParameters gl_Point;

//
// Material State p. 50, 55.
//
struct gl_MaterialParameters {
    vec4  emission;    // Ecm
    vec4  ambient;     // Acm
    vec4  diffuse;     // Dcm
    vec4  specular;    // Scm
    float shininess;   // Srm
};
uniform gl_MaterialParameters  gl_FrontMaterial;
uniform gl_MaterialParameters  gl_BackMaterial;

//
// Light State p 50, 53, 55.
//

struct gl_LightSourceParameters {
    vec4  ambient;             // Acli
    vec4  diffuse;             // Dcli
    vec4  specular;            // Scli
    vec4  position;            // Ppli
    vec4  halfVector;          // Derived: Hi
    vec3  spotDirection;       // Sdli
    float spotExponent;        // Srli
    float spotCutoff;          // Crli
                               // (range: [0.0,90.0], 180.0)
    float spotCosCutoff;       // Derived: cos(Crli)
                               // (range: [1.0,0.0],-1.0)
    float constantAttenuation; // K0
    float linearAttenuation;   // K1
    float quadraticAttenuation;// K2
};

uniform gl_LightSourceParameters  gl_LightSource[gl_MaxLights];

struct gl_LightModelParameters {
    vec4  ambient;       // Acs
};

uniform gl_LightModelParameters  gl_LightModel;

//
// Derived state from products of light and material.
//

struct gl_LightModelProducts {
    vec4  sceneColor;     // Derived. Ecm + Acm * Acs
};

uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

struct gl_LightProducts {
    vec4  ambient;        // Acm * Acli
    vec4  diffuse;        // Dcm * Dcli
    vec4  specular;       // Scm * Scli
};

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

//
// Texture Environment and Generation, p. 152, p. 40-42.
//
uniform vec4  gl_TextureEnvColor[gl_MaxTextureImageUnits];
uniform vec4  gl_EyePlaneS[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneT[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneR[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneQ[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneS[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneT[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneR[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneQ[gl_MaxTextureCoords];
```

```
//
// Fog p. 161
//
struct gl_FogParameters {
    vec4 color;
    float density;
    float start;
    float end;
    float scale;   // Derived:   1.0 / (end - start)
};

uniform gl_FogParameters gl_Fog;
```

7.6 Varying Variables

Unlike user-defined varying variables, the built-in varying variables don't have a strict one-to-one correspondence between the vertex language and the fragment language. Two sets are provided, one for each language. Their relationship is described below.

The following built-in varying variables are available to write to in a vertex shader. A particular one should be written to if any functionality in a corresponding fragment shader or fixed pipeline uses it or state derived from it. Otherwise, behavior is undefined.

```
varying vec4  gl_FrontColor;
varying vec4  gl_BackColor;
varying vec4  gl_FrontSecondaryColor;
varying vec4  gl_BackSecondaryColor;
varying vec4  gl_TexCoord[];  // at most will be gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

For *gl_FogFragCoord*, the value written will be used as the "c" value on page 160 of the OpenGL 1.4 Specification by the fixed functionality pipeline. For example, if the z-coordinate of the fragment in eye space is desired as "c", then that's what the vertex shader should write into *gl_FogFragCoord*.

As with all arrays, indices used to subscript *gl_TexCoord* must either be an integral constant expressions, or this array must be re-declared by the shader with a size. The size can be at most *gl_MaxTextureCoords*. Using indexes close to 0 may aid the implementation in preserving varying resources.

The following varying variables are available to read from in a fragment shader. The *gl_Color* and *gl_SecondaryColor* names are the same names as attributes passed to the vertex shader. However, there is no name conflict, because attributes are visible only in vertex shaders and the following are only visible in a fragment shader.

```
varying vec4  gl_Color;
varying vec4  gl_SecondaryColor;
varying vec4  gl_TexCoord[];  // at most will be gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

The values in *gl_Color* and *gl_SecondaryColor* will be derived automatically by the system from *gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor,* and *gl_BackSecondaryColor* based on which face is visible. If fixed functionality is used for vertex processing, then *gl_FogFragCoord* will either be the z-coordinate of the fragment in eye space, or the interpolation of the fog coordinate, as described in section 3.10 of the OpenGL 1.4 Specification. The *gl_TexCoord[]* values are the interpolated *gl_TexCoord[]* values from a vertex shader or the texture coordinates of any fixed pipeline based vertex functionality.

Indices to the fragment shader *gl_TexCoord* array are as described above in the vertex shader text.

8 Built-in Functions

The OpenGL Shading Language defines an assortment of built-in convenience functions for scalar and vector operations. Many of these built-in functions can be used in more than one type of shader, but some are intended to provide a direct mapping to hardware and so are available only for a specific type of shader.

The built-in functions basically fall into three categories:

- They expose some necessary hardware functionality in a convenient way such as accessing a texture map. There is no way in the language for these functions to be emulated by a shader.
- They represent a trivial operation (clamp, mix, etc.) that is very simple for the user to write, but they are very common and may have direct hardware support. It is a very hard problem for the compiler to map expressions to complex assembler instructions.
- They represent an operation graphics hardware is likely to accelerate at some point. The trigonometry functions fall into this category.

Many of the functions are similar to the same named ones in common C libraries, but they support vector input as well as the more traditional scalar input.

Applications should be encouraged to use the built-in functions rather than do the equivalent computations in their own shader code since the built-in functions are assumed to be optimal (e.g., perhaps supported directly in hardware).

User code can replace built-in functions with their own if they choose, by simply re-declaring and defining the same name and argument list.

When the built-in functions are specified below, where the input arguments (and corresponding output) can be **float**, **vec2**, **vec3**, or **vec4**, *genType* is used as the argument. For any specific use of a function, the actual type has to be the same for all arguments and for the return type. Similarly for *mat*, which can be a **mat2, mat3,** or **mat4**.

8.1 Angle and Trigonometry Functions

Function parameters specified as *angle* are assumed to be in units of radians. In no case will any of these functions result in a divide by zero error. If the divisor of a ratio is 0, then results will be undefined.

These all operate component-wise. The description is per component.

Syntax	Description
genType **radians** (genType *degrees*)	Converts *degrees* to radians and returns the result, i.e., result = π/180 · *degrees*.
genType **degrees** (genType *radians*)	Converts *radians* to degrees and returns the result, i.e., result = 180/π · *radians.*
genType **sin** (genType *angle*)	The standard trigonometric sine function.
genType **cos** (genType *angle*)	The standard trigonometric cosine function.
genType **tan** (genType *angle*)	The standard trigonometric tangent.
genType **asin** (genType *x*)	Arc sine. Returns an angle whose sine is *x*. The range of values returned by this function is [–π/2, π/2]. Results are undefined if |*x*| > 1.
genType **acos** (genType *x*)	Arc cosine. Returns an angle whose cosine is *x*. The range of values returned by this function is [0, π]. Results are undefined if |*x*| > 1.
genType **atan** (genType *y*, genType *x*)	Arc tangent. Returns an angle whose tangent is *y/x*. The signs of *x* and *y* are used to determine what quadrant the angle is in. The range of values returned by this function is [–π, π]. Results are undefined if *x* and *y* are both 0.
genType **atan** (genType *y_over_x*)	Arc tangent. Returns an angle whose tangent is *y_over_x*. The range of values returned by this function is [–π/2, π/2].

8.2 Exponential Functions

These all operate component-wise. The description is per component.

Syntax	Description
genType **pow** (genType *x*, genType *y*)	Returns *x* raised to the *y* power, i.e., x^y. Results are undefined if $x < 0$. Results are undefined if $x = 0$ and $y <= 0$.
genType **exp** (genType *x*)	Returns the natural exponentiation of *x*, i.e., e^x.
genType **log** (genType *x*)	Returns the natural logarithm of *x*, i.e., returns the value *y* which satisfies the equation $x = e^y$. Results are undefined if $x <= 0$.
genType **exp2** (genType *x*)	Returns 2 raised to the *x* power, i.e., 2^x.
genType **log2** (genType *x*)	Returns the base 2 logarithm of *x*, i.e., returns the value *y* which satisfies the equation $x = 2^y$. Results are undefined if $x <= 0$.
genType **sqrt** (genType *x*)	Returns the positive square root of *x*. Results are undefined if $x < 0$.
genType **inversesqrt** (genType *x*)	Returns the reciprocal of the positive square root of *x*. Results are undefined if $x <= 0$.

8.3 Common Functions

These all operate component-wise. The description is per component.

Syntax	Description
genType **abs** (genType *x*)	Returns *x* if *x* >= 0, otherwise it returns –*x*
genType **sign** (genType *x*)	Returns 1.0 if *x* > 0, 0.0 if *x* = 0, or –1.0 if *x* < 0
genType **floor** (genType *x*)	Returns a value equal to the nearest integer that is less than or equal to *x*

genType **ceil** (genType *x*)	Returns a value equal to the nearest integer that is greater than or equal to *x*
genType **fract** (genType *x*)	Returns *x* – **floor** (*x*)
genType **mod** (genType *x*, float *y*)	Modulus. Returns *x* – *y* * **floor** (*x*/*y*)
genType **mod** (genType *x*, genType *y*)	Modulus. Returns *x* – *y* * **floor** (*x*/*y*)
genType **min** (genType *x*, genType *y*) genType **min** (genType *x*, float *y)*	Returns *y* if *y* < *x*, otherwise it returns *x*
genType **max** (genType *x*, genType *y*) genType **max** (genType *x*, float *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*
genType **clamp** (genType *x*, genType *minVal*, genType *maxVal*) genType **clamp** (genType *x*, float *minVal*, float *maxVal*)	Returns **min** (**max** (*x*, *minVal*), *maxVal*) Note that colors and depths written by fragment shaders will be clamped by the implementation after the fragment shader runs.
genType **mix** (genType *x*, genType *y*, genType *a*) genType **mix** (genType *x*, genType *y*, float *a*)	Returns *x* * (1 – *a*) + *y* * *a*, i.e., the linear blend of *x* and *y*
genType **step** (genType *edge*, genType *x*) genType **step** (float *edge*, genType *x*)	Returns 0.0 if *x* < *edge*, otherwise it returns 1.0

genType **smoothstep** (genType *edge0*,
genType *edge1*,
genType *x*)
genType **smoothstep** (float *edge0*,
float *edge1*,
genType *x*)
	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0 and 1 when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to:
	`genType t;`
	`t = clamp ((x – edge0) / (edge1 – edge0), 0, 1);`
	`return t * t * (3 – 2 * t);`

8.4 Geometric Functions

These operate on vectors as vectors, not component-wise.

Syntax	Description
float **length** (genType *x*)	Returns the length of vector *x*, i.e., **sqrt** (*x*[0] * *x*[0] + *x*[1] * *x*[1] + ...)
float **distance** (genType *p0*, genType *p1*)	Returns the distance between *p0* and *p1*, i.e. **length** (p0 – p1)
float **dot** (genType *x*, genType *y*)	Returns the dot product of *x* and *y*, i.e., result = *x*[0] * *y*[0] + *x*[1] * *y*[1] + ...

vec3 **cross** (vec3 *x*, vec3 *y*)	Returns the cross product of x and y, i.e.
	result.0 = *x*[1] * *y*[2] - *y*[1] * *x*[2]
	result.1 = *x*[2] * *y*[0] - *y*[2] * *x*[0]
	result.2 = *x*[0] * *y*[1] - *y*[0] * *x*[1]

genType **normalize** (genType *x*)	Returns a vector in the same direction as *x* but with a length of 1.
vec4 **ftransform**()	For vertex shaders only. This function will ensure that the incoming vertex value will be transformed in a way that produces exactly the same result as would be produced by OpenGL's fixed functionality transform. It is intended to be used to compute gl_Position, e.g., gl_Position = **ftransform**() This function should be used, for example, when an application is rendering the same geometry in separate passes, and one pass uses the fixed functionality path to render and another pass uses programmable shaders.
genType **faceforward** (genType *N*, genType *I*, genType *Nref*)	If **dot** (*Nref*, *I*) < 0 return *N* otherwise return –*N*

genType **reflect** (genType *I*, genType *N*)	For the incident vector *I* and surface orientation *N*, returns the reflection direction: result = *I* – 2 * **dot**(*N*, *I*) * *N* *N* must already be normalized in order to achieve the desired result.

genType **refract**(genType *I*, genType *N*, float *eta*)	For the incident vector *I* and surface normal *N*, and the ratio of indices of refraction *eta*, return the refraction vector. The returned result is computed by
	
	k = 1.0 - *eta* * *eta* * (1.0 - **dot**(*N*, *I*) * **dot**(*N*, *I*))
	if (k < 0.0)
	    result = genType(0.0)
	else
	    result = *eta* * *I* - (*eta* * **dot**(*N*, *I*) + **sqrt**(k)) * *N*
	
	The input parameters for the incident vector *I* and the surface normal *N* must already be normalized to get the desired results.

8.5 Matrix Functions

Syntax	Description
mat **matrixCompMult** (mat *x*, mat *y*)	Multiply matrix *x* by matrix *y* component-wise, i.e., result[i][j] is the scalar product of *x*[i][j] and *y*[i][j]. Note: to get linear algebraic matrix multiplication, use the multiply operator (*****).

8.6 Vector Relational Functions

Relational and equality operators (**<, <=, >, >=, ==, !=**) are defined (or reserved) to produce scalar Boolean results. For vector results, use the following built-in functions. Below, “bvec” is a placeholder for one of **bvec2**, **bvec3**, or **bvec4**, “ivec” is a placeholder for one of **ivec2**, **ivec3**, or **ivec4**, and “vec” is a placeholder for **vec2**, **vec3**, or **vec4**. In all cases, the sizes of the input and return vectors for any particular call must match.

Syntax	Description
bvec **lessThan**(vec x, vec y) bvec **lessThan**(ivec x, ivec y)	Returns the component-wise compare of *x* < *y*.

bvec **lessThanEqual**(vec x, vec y) bvec **lessThanEqual**(ivec x, ivec y)	Returns the component-wise compare of *x* <= *y*.
bvec **greaterThan**(vec x, vec y) bvec **greaterThan**(ivec x, ivec y)	Returns the component-wise compare of *x* > *y*.
bvec **greaterThanEqual**(vec x, vec y) bvec **greaterThanEqual**(ivec x, ivec y)	Returns the component-wise compare of *x* >= *y*.
bvec **equal**(vec x, vec y) bvec **equal**(ivec x, ivec y) bvec **equal**(bvec x, bvec y) bvec **notEqual**(vec x, vec y) bvec **notEqual**(ivec x, ivec y) bvec **notEqual**(bvec x, bvec y)	Returns the component-wise compare of *x* == *y*. Returns the component-wise compare of *x* != *y*.
bool **any**(bvec x)	Returns true if any component of *x* is **true**.
bool **all**(bvec x)	Returns true only if all components of *x* are **true**.
bvec **not**(bvec x)	Returns the component-wise logical complement of *x*.

8.7 Texture Lookup Functions

Texture lookup functions are available to both vertex and fragment shaders. However, level of detail is not computed by fixed functionality for vertex shaders, so there are some differences in operation between vertex and fragment texture lookups. The functions in the table below provide access to textures through samplers, as set up through the OpenGL API. Texture properties such as size, pixel format, number of dimensions, filtering method, number of mip-map levels, depth comparison, and so on are also defined by OpenGL API calls. Such properties are taken into account as the texture is accessed via the built-in functions defined below.

If a non-shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned on, then results are undefined. If a shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned off, then results are undefined. If a shadow texture call is made to a sampler that does not represent a depth texture, then results are undefined.

In all functions below, the *bias* parameter is optional for fragment shaders. The *bias* parameter is not accepted in a vertex shader. For a fragment shader, if *bias* is present, it is added to the calculated level of detail prior to performing the texture access operation. If the *bias* parameter is not provided, then the implementation automatically selects level of detail: For a texture that is not mip-mapped, the texture is used directly. If it is mip-mapped and running in a fragment shader, the LOD computed by the implementation is used to do the texture lookup. If it is mip-mapped and running on the vertex shader, then the base texture is used.

The built-ins suffixed with "**Lod**" are allowed only in a vertex shader. For the "**Lod**" functions, *lod* is directly used as the level of detail.

Syntax	Description
vec4 **texture1D** (sampler1D *sampler*, float *coord* [, float *bias*]) vec4 **texture1DProj** (sampler1D *sampler*, vec2 *coord* [, float *bias*]) vec4 **texture1DProj** (sampler1D *sampler*, vec4 *coord* [, float *bias*]) vec4 **texture1DLod** (sampler1D *sampler*, float *coord*, float *lod*) vec4 **texture1DProjLod** (sampler1D *sampler*, vec2 *coord*, float *lod*) vec4 **texture1DProjLod** (sampler1D *sampler*, vec4 *coord*, float *lod*)	Use the texture coordinate *coord* to do a texture lookup in the 1D texture currently bound to *sampler*. For the projective ("**Proj**") versions, the texture coordinate *coord.s* is divided by the last component of *coord*.
vec4 **texture2D** (sampler2D *sampler*, vec2 *coord* [, float *bias*]) vec4 **texture2DProj** (sampler2D *sampler*, vec3 *coord* [, float *bias*]) vec4 **texture2DProj** (sampler2D *sampler*, vec4 *coord* [, float *bias*]) vec4 **texture2DLod** (sampler2D *sampler*, vec2 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*, vec3 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*, vec4 *coord*, float *lod*)	Use the texture coordinate *coord* to do a texture lookup in the 2D texture currently bound to *sampler*. For the projective ("**Proj**") versions, the texture coordinate (*coord.s, coord.t*) is divided by the last component of *coord*. The third component of coord is ignored for the vec4 coord variant.
vec4 **texture3D** (sampler3D *sampler*, vec3 *coord* [, float *bias*]) vec4 **texture3DProj** (sampler3D *sampler*, vec4 *coord* [, float *bias*]) vec4 **texture3DLod** (sampler3D *sampler*, vec3 *coord*, float *lod*) vec4 **texture3DProjLod** (sampler3D *sampler*, vec4 *coord*, float *lod*)	Use the texture coordinate *coord* to do a texture lookup in the 3D texture currently bound to *sampler*. For the projective ("**Proj**") versions, the texture coordinate is divided by *coord.q*.

vec4 **textureCube** (samplerCube *sampler*, vec3 *coord* [, float *bias*]) vec4 **textureCubeLod** (samplerCube *sampler*, vec3 *coord*, float *lod*)	Use the texture coordinate *coord* to do a texture lookup in the cube map texture currently bound to *sampler*. The direction of *coord* is used to select which face to do a 2-dimensional texture lookup in, as described in section 3.8.6 in version 1.4 of the OpenGL specification.
vec4 **shadow1D** (sampler1DShadow *sampler*, vec3 *coord* [, float *bias*]) vec4 **shadow2D** (sampler2DShadow *sampler*, vec3 *coord* [, float *bias*]) vec4 **shadow1DProj** (sampler1DShadow *sampler*, vec4 *coord* [, float *bias*]) vec4 **shadow2DProj** (sampler2DShadow *sampler*, vec4 *coord* [, float *bias*]) vec4 **shadow1DLod** (sampler1DShadow *sampler*, vec3 *coord*, float *lod*) vec4 **shadow2DLod** (sampler2DShadow *sampler*, vec3 *coord*, float *lod*) vec4 **shadow1DProjLod**(sampler1DShadow *sampler*, vec4 *coord*, float *lod*) vec4 **shadow2DProjLod**(sampler2DShadow *sampler*, vec4 coord, float lod)	Use texture coordinate *coord* to do a depth comparison lookup on the depth texture bound to *sampler*, as described in section 3.8.14 of version 1.4 of the OpenGL specification. The 3rd component of *coord* (*coord.p*) is used as the R value. The texture bound to *sampler* must be a depth texture, or results are undefined. For the projective ("**Proj**") version of each built-in, the texture coordinate is divide by *coord.q*, giving a depth value R of *coord.p/coord.q*. The second component of *coord* is ignored for the "**1D**" variants.

8.8 Fragment Processing Functions

Fragment processing functions are only available in shaders intended for use on the fragment processor.

Derivatives may be computationally expensive and/or numerically unstable. Therefore, an OpenGL implementation may approximate the true derivatives by using a fast but not entirely accurate derivative computation.

The expected behavior of a derivative is specified using forward/backward differencing.

Forward differencing:

F(x+dx) - F(x) ~ dFdx(x) * dx 1a

dFdx(x) ~ (F(x+dx) - F(x)) / dx 1b

Backward differencing:

F(x-dx) - F(x) ~ -dFdx(x) * dx 2a

dFdx(x) ~ (F(x) - F(x-dx)) / dx 2b

With single-sample rasterization, $dx <= 1.0$ in equations 1b and 2b. For multi-sample rasterization, $dx < 2.0$ in equations 1b and 2b.

dFdy is approximated similarly, with *y* replacing *x*.

A GL implementation may use the above or other methods to perform the calculation, subject to the following conditions:

1) The method may use piecewise linear approximations. Such linear approximations imply that higher order derivatives, **dFdx**(**dFdx**(*x*)) and above, are undefined.
2) The method may assume that the function evaluated is continuous. Therefore derivatives within the body of a non-uniform conditional are undefined.
3) The method may differ per fragment, subject to the constraint that the method may vary by window coordinates, not screen coordinates. The invariance requirement described in section 3.1 of the OpenGL 1.4 specification is relaxed for derivative calculations, because the method may be a function of fragment location.

Other properties that are desirable, but not required, are:

4) Functions should be evaluated within the interior of a primitive (interpolated, not extrapolated).
5) Functions for **dFdx** should be evaluated while holding y constant. Functions for **dFdy** should be evaluated while holding x constant. However, mixed higher order derivatives, like **dFdx**(**dFdy**(*y*)) and **dFdy**(**dFdx**(*x*)) are undefined.

In some implementations, varying degrees of derivative accuracy may be obtained by providing GL hints (section 5.6 of the OpenGL 1.4 specification), allowing a user to make an image quality versus speed tradeoff.

Syntax	Description
genType **dFdx** (genType *p*)	Returns the derivative in x using local differencing for the input argument *p*.
genType **dFdy** (genType *p*)	Returns the derivative in y using local differencing for the input argument *p*. These two functions are commonly used to estimate the filter width used to anti-alias procedural textures.We are assuming that the expression is being evaluated in parallel on a SIMD array so that at any given point in time the value of the function is known at the grid points represented by the SIMD array. Local differencing between SIMD array elements can therefore be used to derive dFdx, dFdy, etc.
genType **fwidth** (genType *p*)	Returns the sum of the absolute derivative in x and y using local differencing for the input argument *p*, i.e.: return = **abs** (**dFdx** (*p*)) + **abs** (**dFdy** (*p*));

8.9 Noise Functions

Noise functions are available to both fragment and vertex shaders. They are stochastic functions that can be used to increase visual complexity. Values returned by the following noise functions give the appearance of randomness, but are not truly random. The noise functions below are defined to have the following characteristics:

- The return value(s) are always in the range [-1.0,1.0], and cover at least the range [-0.6, 0.6], with a gaussian-like distribution.
- The return value(s) have an overall average of 0.0
- They are repeatable, in that a particular input value will always produce the same return value
- They are statistically invariant under rotation (i.e., no matter how the domain is rotated, it has the same statistical character)
- They have a statistical invariance under translation (i.e., no matter how the domain is translated, it has the same statistical character)
- They typically give different results under translation.
- The spatial frequency is narrowly concentrated, centered somewhere between 0.5 to 1.0.

- They are C^1 continuous everywhere (i.e., the first derivative is continuous)

Syntax	Description
float **noise1** (genType *x*)	Returns a 1D noise value based on the input value *x*.
vec2 **noise2** (genType *x*)	Returns a 2D noise value based on the input value *x*.
vec3 **noise3** (genType *x*)	Returns a 3D noise value based on the input value *x*.
vec4 **noise4** (genType *x*)	Returns a 4D noise value based on the input value *x*.

9 Shading Language Grammar

The grammar is fed from the output of lexical analysis. The tokens returned from lexical analysis are

```
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
SAMPLER1D SAMPLER2D SAMPLER3D SAMPLERCUBE SAMPLER1DSHADOW SAMPLER2DSHADOW
STRUCT VOID WHILE

IDENTIFIER TYPE_NAME FLOATCONSTANT INTCONSTANT BOOLCONSTANT
FIELD_SELECTION
LEFT_OP RIGHT_OP
INC_OP DEC_OP LE_OP GE_OP EQ_OP NE_OP
AND_OP OR_OP XOR_OP MUL_ASSIGN DIV_ASSIGN ADD_ASSIGN
MOD_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN
SUB_ASSIGN

LEFT_PAREN RIGHT_PAREN LEFT_BRACKET RIGHT_BRACKET LEFT_BRACE RIGHT_BRACE DOT
COMMA COLON EQUAL SEMICOLON BANG DASH TILDE PLUS STAR SLASH PERCENT
LEFT_ANGLE RIGHT_ANGLE VERTICAL_BAR CARET AMPERSAND QUESTION
```

The following describes the grammar for the OpenGL Shading Language in terms of the above tokens.

variable_identifier:
    IDENTIFIER

primary_expression:
    variable_identifier
    INTCONSTANT
    FLOATCONSTANT
    BOOLCONSTANT
    LEFT_PAREN expression RIGHT_PAREN

postfix_expression:
    primary_expression
    postfix_expression LEFT_BRACKET integer_expression RIGHT_BRACKET
    function_call

postfix_expression DOT FIELD_SELECTION
postfix_expression INC_OP
postfix_expression DEC_OP

integer_expression:
expression

function_call:
function_call_generic

function_call_generic:
function_call_header_with_parameters RIGHT_PAREN
function_call_header_no_parameters RIGHT_PAREN

function_call_header_no_parameters:
function_call_header VOID
function_call_header

function_call_header_with_parameters:
function_call_header assignment_expression
function_call_header_with_parameters COMMA assignment_expression

function_call_header:
function_identifier LEFT_PAREN

function_identifier:
constructor_identifier
IDENTIFIER

// Grammar Note: Constructors look like functions, but lexical anaylsis recognized most of them as keywords.

constructor_identifier:
FLOAT
INT
BOOL

```
        VEC2
        VEC3
        VEC4
        BVEC2
        BVEC3
        BVEC4
        IVEC2
        IVEC3
        IVEC4
        MAT2
        MAT3
        MAT4
        TYPE_NAME

unary_expression:
        postfix_expression
        INC_OP unary_expression
        DEC_OP unary_expression
        unary_operator unary_expression

// Grammar Note:  No traditional style type casts.

unary_operator:
        PLUS
        DASH
        BANG
        TILDE   // reserved

// Grammar Note:  No '*' or '&' unary ops.  Pointers are not supported.

multiplicative_expression:
        unary_expression
        multiplicative_expression STAR unary_expression
        multiplicative_expression SLASH unary_expression
        multiplicative_expression PERCENT unary_expression   // reserved
```

additive_expression:

multiplicative_expression

additive_expression PLUS multiplicative_expression

additive_expression DASH multiplicative_expression

shift_expression:

additive_expression

shift_expression LEFT_OP additive_expression // reserved

shift_expression RIGHT_OP additive_expression // reserved

relational_expression:

shift_expression

relational_expression LEFT_ANGLE shift_expression

relational_expression RIGHT_ANGLE shift_expression

relational_expression LE_OP shift_expression

relational_expression GE_OP shift_expression

equality_expression:

relational_expression

equality_expression EQ_OP relational_expression

equality_expression NE_OP relational_expression

and_expression:

equality_expression

and_expression AMPERSAND equality_expression // reserved

exclusive_or_expression:

and_expression

exclusive_or_expression CARET and_expression // reserved

inclusive_or_expression:

exclusive_or_expression

inclusive_or_expression VERTICAL_BAR exclusive_or_expression // reserved

logical_and_expression:

inclusive_or_expression

logical_and_expression AND_OP inclusive_or_expression

logical_xor_expression:
logical_and_expression
logical_xor_expression XOR_OP logical_and_expression

logical_or_expression:
logical_xor_expression
logical_or_expression OR_OP logical_xor_expression

conditional_expression:
logical_or_expression
logical_or_expression QUESTION expression COLON conditional_expression

assignment_expression:
conditional_expression
unary_expression assignment_operator assignment_expression

assignment_operator:
EQUAL
MUL_ASSIGN
DIV_ASSIGN
MOD_ASSIGN // reserved
ADD_ASSIGN
SUB_ASSIGN
LEFT_ASSIGN // reserved
RIGHT_ASSIGN // reserved
AND_ASSIGN // reserved
XOR_ASSIGN // reserved
OR_ASSIGN // reserved

expression:
assignment_expression
expression COMMA assignment_expression

constant_expression:

conditional_expression

declaration:
function_prototype SEMICOLON
init_declarator_list SEMICOLON

function_prototype:
function_declarator RIGHT_PAREN

function_declarator:
function_header
function_header_with_parameters

function_header_with_parameters:
function_header parameter_declaration
function_header_with_parameters COMMA parameter_declaration

function_header:
fully_specified_type IDENTIFIER LEFT_PAREN

parameter_declarator:
type_specifier IDENTIFIER
type_specifier IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

parameter_declaration:
type_qualifier parameter_qualifier parameter_declarator
parameter_qualifier parameter_declarator
type_qualifier parameter_qualifier parameter_type_specifier
parameter_qualifier parameter_type_specifier

parameter_qualifier:
/ empty */*
IN
OUT
INOUT

parameter_type_specifier:

type_specifier

type_specifier LEFT_BRACKET constant_expression RIGHT_BRACKET

init_declarator_list:

single_declaration

init_declarator_list COMMA IDENTIFIER

init_declarator_list COMMA IDENTIFIER LEFT_BRACKET RIGHT_BRACKET

init_declarator_list COMMA IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

init_declarator_list COMMA IDENTIFIER EQUAL initializer

single_declaration:

fully_specified_type

fully_specified_type IDENTIFIER

fully_specified_type IDENTIFIER LEFT_BRACKET RIGHT_BRACKET

fully_specified_type IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

fully_specified_type IDENTIFIER EQUAL initializer

// Grammar Note: No 'enum', or 'typedef'.

fully_specified_type:

type_specifier

type_qualifier type_specifier

type_qualifier:

CONST

ATTRIBUTE // Vertex only.

VARYING

UNIFORM

type_specifier:

VOID

FLOAT

INT

BOOL

VEC2
VEC3
VEC4
BVEC2
BVEC3
BVEC4
IVEC2
IVEC3
IVEC4
MAT2
MAT3
MAT4
SAMPLER1D
SAMPLER2D
SAMPLER3D
SAMPLERCUBE
SAMPLER1DSHADOW
SAMPLER2DSHADOW
struct_specifier
TYPE_NAME

struct_specifier:
STRUCT IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
STRUCT LEFT_BRACE struct_declaration_list RIGHT_BRACE

struct_declaration_list:
struct_declaration
struct_declaration_list struct_declaration

struct_declaration:
type_specifier struct_declarator_list SEMICOLON

struct_declarator_list:
struct_declarator
struct_declarator_list COMMA struct_declarator

struct_declarator:
    IDENTIFIER
    IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

initializer:
    assignment_expression

declaration_statement:
    declaration

statement:
    compound_statement
    simple_statement

// Grammar Note: No labeled statements; 'goto' is not supported.

simple_statement:
    declaration_statement
    expression_statement
    selection_statement
    iteration_statement
    jump_statement

compound_statement:
    LEFT_BRACE RIGHT_BRACE
    LEFT_BRACE statement_list RIGHT_BRACE

statement_no_new_scope:
    compound_statement_no_new_scope
    simple_statement

compound_statement_no_new_scope:
    LEFT_BRACE RIGHT_BRACE
    LEFT_BRACE statement_list RIGHT_BRACE

statement_list:

statement
statement_list statement

expression_statement:
SEMICOLON
expression SEMICOLON

selection_statement:
IF LEFT_PAREN expression RIGHT_PAREN selection_rest_statement

selection_rest_statement:
statement ELSE statement
statement

// Grammar Note: No 'switch'. Switch statements not supported.

condition:
expression
fully_specified_type IDENTIFIER EQUAL initializer

iteration_statement:
WHILE LEFT_PAREN condition RIGHT_PAREN statement_no_new_scope
DO statement WHILE LEFT_PAREN expression RIGHT_PAREN SEMICOLON
FOR LEFT_PAREN for_init_statement for_rest_statement RIGHT_PAREN statement_no_new_scope

for_init_statement:
expression_statement
declaration_statement

conditionopt:
condition
/ empty */*

for_rest_statement:
conditionopt SEMICOLON
conditionopt SEMICOLON expression

jump_statement:
    CONTINUE SEMICOLON
    BREAK SEMICOLON
    RETURN SEMICOLON
    RETURN expression SEMICOLON
    DISCARD SEMICOLON   // Fragment shader only.

// Grammar Note: No 'goto'. Gotos are not supported.

translation_unit:
    external_declaration
    translation_unit external_declaration

external_declaration:
    function_definition
    declaration

function_definition:
    function_prototype compound_statement_no_new_scope

10 Issues

.

1) *Should the programs that run on these programmable processors be called shaders or programs?*

   DISCUSSION: Shader fits in with common usage in RenderMan and DX8. There is some argument that shading has connotations of being a color operation so doesn't fit with a vertex operation. RenderMan doesn't make this distinction, nor does DX8. It seems wise to go along with the common usage of shader as a general term for a program that operates on some part of a graphics pipeline.

   RESOLVED on October 12, 2001: The term shader will be used.

   Note: *Shader* is used to denote a single independent compilation unit. *Program* is used to denote a set of shaders linked together.

   CLOSED on September 10, 2002.

2) *Should there be a separate programmable unit for doing the pixel transfer operations?*

   DISCUSSION: We originally had the concept of a separate pixel shader where the pixel and imaging operations would be done. On further consideration it seemed very unlikely that anyone would implement this as an independent functional unit but rather do them in the fragment shader behind the scenes. OpenGL treats pixel and fragment operations as mutually exclusive so sharing one processing unit is a natural implementation. Forcing an abstraction that differed from reality seemed to be a hindrance apart from increasing the amount of work.

   RESOLVED on October 12, 2001: No, the fragment processor will be used to process both geometry and pixel data.

   CLOSED on September 10, 2002.

3) *Should shaders be allowed to subset the fixed functionality that they replace?*

   DISCUSSION: There would be a lot of complexity in defining the interfaces to allow subsetting. It isn't very difficult to write shaders that implement the whole of the graphics processing pipeline.

   RESOLVED on October 12, 2001: No, shaders cannot subset the fixed functionality they are replacing. If shaders want to change the lighting in some way then they have to do the other items as well. It will be helpful to have example shaders that fully implement the OpenGL fixed functionality pipeline.

   CLOSED on September 10, 2002.

4) *Should a higher level shading language be layered on top of OpenGL instead of being designed to fit within OpenGL?*

   DISCUSSION: In the current design, the shading language is integrated into OpenGL and just provides alternative methods to the state controlled pipeline outlined earlier. The Stanford approach is to layer their shading language on top of OpenGL. This has some advantages and disadvantages that will become apparent when the differences are examined.

The Stanford approach uses a higher abstraction level. This helps with writing some kinds of programs where the abstractions match the problem domain. For example treating lights and surfaces as abstract entities makes some 3D graphics operations easier, however OpenGL is now being used for video and image processing where this abstraction is largely irrelevant. Similarly many games have shunned lighting via traditional means and use textures (light maps) instead.

There is nothing in the language or bindings that prevent higher levels of abstractions from being layered on top of a programmable OpenGL. We also wish to keep the overall abstraction level of OpenGL at its current level.

The Stanford approach also provides for different computational frequencies. By having the higher levels of abstraction where one program defines the current graphics operation in total allows the compiler to separate out the parts that need to run at the primitive group level, primitive level, vertex level and fragment level. The compiler can therefore generate the code to run on the CPU, vertex processor and fragment processor as appropriate. This is obviously more complicated to implement than having the programmer specify the programs to run on each part of the pipeline (although some hints are still required by the Stanford language), although this does make the virtualization of the hardware easier as the compiler has the overall view.

The major disadvantage of this is that it forces more intrusive changes to OpenGL to support the clear delineation of the primitives, vertices and fragment operations. Many of the core OpenGL features have been replaced or are not available and it is not possible to use the standard OpenGL transformation and lighting operations with a custom fragment shader (or vice versa), or to allow one vertex shader to drive multiple fragment shaders. An advantage of the current approach is that the look and feel of OpenGL 1.4 is maintained and it allows a graceful mix and match approach during the transition period from fixed functionality to full programmability.

This is not a criticism of the Stanford work, as they had no choice but to layer on top of OpenGL.

RESOLVED on October 12, 2001: The OpenGL Shading Language should be built into OpenGL, and not layered on top. It is also noted that if this is not the case, OpenGL should still have a standard shading language, so this document still remains. Hence, this issue is not one against this document but one against the OpenGL API.

CLOSED on September 20, 2002, as moved to the API issues list.

5) *Should the shading model be part of the fixed functionality fragment processing that is replaced by the fragment processor?*

DISCUSSION: The shading model selects between Gouraud and flat shading and this would seem natural to have this as part of the functionality replaced by the fragment shader. Flat shading involves knowledge of the primitive type (for the provoking vertex) and this doesn't really belong in the fragment shader. The fragment shader can always assume the color is interpolated and the shading model is flat then the set up calculations for the color gradients can set the gradients to zero.

RESOLVED on October 12, 2001: No, the shading model is not replaced by the programmable functionality of the fragment processor.

CLOSED on September 10, 2002.

6) *Is alpha testing programmable?*

DISCUSSION: The fragment shader has a function to kill fragments so could do alpha-like testing, however the OpenGL pipeline specifies that alpha testing should happen after coverage has modified the alpha value. We do not want to do coverage in the fragment shader so the alpha test remains outside. If the user is happy to do alpha testing before coverage in their own programs then they can do this.

RESOLVED on October 12, 2001: Yes, applications can do alpha testing in a fragment shader, with the proviso that, when done in the fragment shader, it happens before the coverage computation.

CLOSED on September 10, 2002.

7) *Is alpha blending programmable?*

Fragment shaders can read the contents of the frame buffer at the current location using the built-in variables *gl_FBColor*, *gl_FBDepth*, *gl_FBStencil*, and *gl_FBDatan*. Using these facilities, applications can implement custom algorithms for blending, stencil testing, and the like. However, these frame buffer read operations may result in a significant reduction in performance, so applications are strongly encouraged to use the fixed functionality of OpenGL for these operations if at all possible. The hardware to implement fragment shaders (and vertex shaders) is made a lot simpler and faster if each fragment can be processed independently both in space and in time. By allowing read-modify-write operations such as is needed with alpha blending to be done as part of the fragment processing we have introduced both spatial and temporal relationships. These complicate the design because of the extremely deep pipelining, caching and memory arbitration necessary for performance. Methods such as render to texture, copy frame buffer to texture, aux data buffers and accumulation buffers can do most, if not all, what programmable alpha blending can do. Also the need for multiple passes has been reduced (or at least abstracted) by the high-level shading language and the automatic resource management.

RESOLVED on October 12, 2001: Yes, applications can do alpha blending, albeit with possible performance penalties over using the fixed functionality blending operations.

REOPENED on July 9, 2002: This issue is related to Issue (23) which remains open, so this issue should also remain open.

Another possibility would be to create an extension that allows more flexibility than the current alpha blending allows, but would still be considered fixed functionality.

RESOLUTION: Issue 23) is resolved as allowing frame buffer reads, so this is once again resolved allowing alpha blending, with the caveats listed above.

REOPENED on December 10, 2002. Issue 23 is re-resolved to disallow frame buffer reads.

RESOLUTION: No, applications cannot do alpha blending, because they cannot read alpha.

CLOSED on December 10, 2002.

8) *Should the language be defined in such a way that it can be implemented on existing hardware?*

DISCUSSION: Today's generation of hardware does have some programmability. It seems desirable to define a language that would work on today's hardware as well as tomorrow's.

RESOLVED on October 12, 2001: We have tried to make the shading language forward looking and pitched at a level we believe hardware can attain within a generation or two. We have avoided adding features (such as small fixed point data types or implicit clamping) or dumbing down (removal of loops and functions) the language to better support existing hardware as this is a retrograde step. It would be possible to run a limited subset of shaders on existing hardware but it is not going to be easy for an application to determine in a portable way if the shader will run or if it will produce acceptable results. Overall, the decision here is to set a goal for hardware to strive towards for the next few years.

CLOSED on September 20, 2002.

9) *Should the concept of the preprocessor for the language be dropped?*

DISCUSSION: We could do without the #ifdef by using if (false) and we rely on compiler stripping out code which cannot be reached. The C++ spec seems to be deemphasizing the use of #ifdef but we are still retaining it because it is a common idiom, it is easier to see in the code and it can be used in places where the grammar doesn't allow if (false).

Do we want a vendor specific predefined #define to allow compiler problems to be worked around? From an idealistic view point no as it provides a back door for extensions, but pragmatically differences will occur. We have already seen instances of shader writers using #define to make shaders more readable (e.g., #define MVP gl_ModelViewProjectionMatrix).

RESOLVED on October 12, 2001: No, the preprocessor should be retained. The preprocessor directives that are supported are #ifdef, #ifndef, #undef, #else, #endif, #pragma, #define token (without arguments), and #error.

Issue (55) is added to address additional preprocessor directives.

CLOSED on September 10, 2002.

10) *Should the fields representing texture components be named s, t, p, and q?*

DISCUSSION: Other alternatives to renaming texture r were considered but rejected because they seemed to be more confusing or error prone. A) Use red, green, blue, alpha instead for color component selection. This makes the component group mechanism described later too cumbersome. B) Capitalize either the color or texture component names. C) Drop the names for color or texture. We didn't want to abandon some notational convenience of one of the important usage of vectors. D) Change the color component order to be bgra so that the two r components now lined up. This color order is quite alien to OpenGL so this would lead to many confusing situations in how the existing API values mapped to the values the shader used.

RESOLVED on October 12, 2001: Yes, using s, t, p, and q as the names for fields representing texture components is the best choice.

CLOSED on September 10, 2002.

11) *Should there be two separate active fragment shaders to handle back facing and front facing cases?*

DISCUSSION: If the user specifies two fragment shaders, one for front facing fragments and one for back facing fragments, the appropriate one could be run automatically. This probably gives a faster shading rate but it forces the user to maintain two programs (where probably most of the code is common). This could be done transparently by the compiler if an implementation wishes to optimize for this case.

RESOLVED on October 12, 2001: No, a single shader should be used to handle both back facing and front facing geometry.

CLOSED on September 10, 2002.

12) *Should built-in functions be required to differ by more than just return type?*

DISCUSSION: Overloading functions that differed in return type only was considered. However, initial work on the compiler has shown that this facility seriously complicates the semantic analysis of an expression to deduce the return type when it is unambiguous, but buried within an expression. This may be more complicated than it is worth (and may be the reason why C++ doesn't allow this).

Add on the need for a new syntax for the programmer to use to disambiguate ambiguous cases, and it's simply easier to give functions different names for different return types.

RESOLVED on February 25, 2002: Yes, built-in functions must differ by more than just return type.

CLOSED on September 10, 2002.

13) *How is the noise function defined to allow consistent behavior from one implementation to the next?*

DISCUSSION: The noise function is very useful and plays a role in many shading techniques in RenderMan. It poses a problem in specification (and conformance testing) in that perfectly valid noise functions will give very different results. OpenGL has avoided specifying operations so tightly that different implementation will give pixel exact results — this allows an implementation some latitude in accuracy/performance/cost trade-offs. It also avoids having to specify everything down to minute detail. Perlin (the originator of the noise function) has recognized the desirability of a standard noise function (much like everyone expects the sin function to behave in the same way) and has documented his ideas. Maybe this should be a strongly recommended implementation.

This issue is nearly the same as Issue (36).

RESOLVED on September 19, 2002: No specific implementation of noise is required, but the specification will attempt to define the noise function in such a way that similar results can be achieved from one implementation to the next.

CLOSED on September 19, 2002:

14) *Should fields be allowed to have numeric selectors (e.g. foo.2)?*

DISCUSSION: This breaks the usual convention of identifiers starting with a letter. It makes the language less pure, makes lexical analysis more difficult, and adds constraints on how numbers are expressed.

RESOLVED on September 10, 2002: No, the language should be changed so there are no numeric selectors as suggested in Issue (16).

CLOSED on September 10, 2002.

15) *Should we allow fields that swizzle the components of a vector?*

DISCUSSION: This seems like an overly complicated part of the language, with no additional functionality that couldn't be easily expressed with other parts of the language. On the other hand, the swizzling is exposed in lower level assembly languages. On the third hand, perhaps this is just a “feature” of hardware that shouldn’t/doesn’t need to be exposed in a high level language. On the fourth hand, some useful examples of swizzling have been demonstrated, and it isn’t very hard to support in the compiler.

RESOLVED on September 10, 2002: Swizzling of components is deemed to be a useful language feature that will be retained.

CLOSED on September 10, 2002.

16) *Should there be a way to indirectly reference into a vector or matrix?*

DISCUSSION: Issue (14) and this one could be simultaneously fixed by adding [ ] as a numeric way of indexing into a vector. Then, one would say foo[2], never foo.2, solving Issue (14), and foo[x] as an indirect reference. Numbers could then be expressed as in C.

RESOLVED on September 10, 2002: Yes, indirect references into vectors and matrices should be allowed in the manner suggested in the discussion above.

CLOSED on September 10, 2002.

17) *Should gl_Position and other currently "write-only" variables be readable?*

DISCUSSION: This is simply a compiler feature, with no implication to hardware support. It's often cumbersome to write code without this feature. The compiler can use temporaries to store intermediate values if necessary. This will make programs a little bit cleaner as well.

On the other hand, the api model of read-only inputs and write-only outputs is probably cleaner for the shader writer.

RESOLVED on September 19, 2002. Yes, "write-only" variables are allowed to be readable.

CLOSED on September 19, 2002.

18) *How should performance/space/precision hints be provided?*

DISCUSSION: One basically agreed on so far is for varying: "varying" means perspective correct, while "fast varying" means take a short cut if it saves time. Perhaps we can define a #pragma for this. Perhaps this could also be applied in other areas.

RESOLUTION: Performance/space/precision hints and types will not be provided as a standard part of the language, but reserved words for doing so will be.

CLOSED: November 26, 2002.

19) *Should the built-in function "lookup" be added?*

DISCUSSION: Textures can be used as look-up tables, not just textures. The main difference is that look-up tables would have a type associated with the return.

RESOLUTION: Yes, the lookup functions should be added as built-in functions so that shader can express the type of the returned value. Functions like i8texture3 will be added to mean three 8bit ints are being looked up. This will still be called a texture, as it is expected to share texture resources. A generic table lookup that is separate from texture resources is deferred until version 1.1.

CLOSED October 22, 2002.

NOTE: These were later removed as part of fitting this language on OpenGL 1.4, as that does not support textures these functions would operate on.

20) *Should ints greater than 16 bits be added?*

DISCUSSION: The shading language is designed in a way that it does not overburden the hardware designer by requiring a lot of unnecessary and redundant features. Integers are useful and may be more efficient for use as loop counters and array indices. 16-bit integers were added to the language as a concession to efficiency for these cases. The mantissa of a floating point value can be used to do integer operations, therefore hardware designers are not required to have a full integer math unit in addition to the floating point math unit. If this were to end up being the key factor in deciding this issue, integers could be defined to be 23 bits in size (at least for operations within the processor), since this is the size of the mantissa of an IEEE FP32 value. As another data point, Renderman does not support integers at all.

RESOLUTION: There are hardware reasons today to limit ints to 16 bits, so they will be.

CLOSED: November 19, 2002.

21) *Should vectors or (local variable) arrays of ints be added?*

DISCUSSION: The lookup function proposed in Issue (19) could return 3 integer values, for example. Shaders are not just floating point algorithms, but also do things like table lookups, indirection, and other generic algorithmic computation. The language does not have to map directly to hardware. On the other hand, to support this we would have to add *ivec2, ivec3*, and *ivec4*, or allow local variable arrays of ints.

RESOLUTION: Yes. Vectors of ints will be added.

CLOSED: November 5, 2002.

22) *Should recursion be supported?*

DISCUSSION: Probably not necessary, but another example of limiting the language based on how it would directly map to hardware. One thought is that recursion would benefit ray tracing shaders. On the other hand, many recursion operations can also be implemented with the user managing the recursion through arrays. RenderMan doesn't support recursion. This could be added at a later date, if it proved to be necessary.

RESOLVED on September 10, 2002: Implementations are not required to support recursion.

CLOSED on September 10, 2002.

23) *Should the fragment shader be allowed to read the current location in the frame buffer?*

DISCUSSION: It may be difficult to specify this properly while taking into account multisampling. It also may be quite difficult for hardware implementors to implement this capability, at least with reasonable performance. But this was one of the top two requested items after the original release of the shading language white paper. ISVs continue to tell us that they need this capability, and that it must be high performance.

RESOLUTION: Yes. This is allowed, with strong cautions as to performance impacts.

REOPENED on December 10, 2002. There is too much concern about impact to performance and impracticallity of implementation.

CLOSED on December 10, 2002.

24) *Does anything need to be added to the language to allow programs can be compiled at compile time, and not need to have multiple compiled versions saved for OpenGL state changes?*

DISCUSSION: It is strongly desired that implementations can generate proper code at compile time, and not have to have multiple compiled versions or later recompilation in case OpenGL state changes at a later time (e.g., not knowing the attributes of a texture map until execution time). Maybe another area where more hints are needed in the language, or maybe hardware evolution can take care of the issues.

RESOLUTION: This is resolved to be a general design goal of the OpenGL shading language... that other issues be resolved with the intent that the object code generated from a shader be independent of other OpenGL state.

CLOSED: November 5, 2002.

25) *Should we add min and max that take gen-type and a scalar, to match clamp semantics?*

RESOLUTION: Yes, these should be added.

CLOSED: September 22, 2002.

26) *Should the programmability be broken out by function (e.g., light shaders, surface shaders, transform shaders, texgen shaders, etc.) rather than the hardware-centric method (vertex and fragment shaders) in the current proposal?*

RESOLVED on December 7, 2001: No, the notion of vertex and fragment shaders fits in much better with OpenGL as a hardware-centric API and has received positive feedback during review.

CLOSED on September 10, 2002.

27) *Should texture units be specified as a keyword or a number?*

DISCUSSION: For the built-in texture access functions, the texture unit is specified as a number. Should it be defined as a keyword instead? The current feeling is that in certain cases it is more convenient to specify the texture as a programmatic value rather than a keyword.

This issue is actually part of Issue (51).

RESOLVED on December 7, 2001: The texture unit is specified as a number. In certain cases it is more convenient to specify the texture as a programmatic value rather than a keyword.

REOPENED on July 12, 2002: Need further discussion of the real convenience provided.

RESOLUTION: Resolved as issue 51.

CLOSED October 22, 2002.

28) *Are global values auto-initialized?*

RESOLVED on December 7, 2001: No, global values are not auto-initialized. It may be useful for an implementation to support auto-initialization as a debug mode option, however.

CLOSED on September 10, 2002.

29) *Should the language support bit-wise operations?*

RESOLVED on December 7, 2001: The language itself has support for bit-wise operations. In certain programmable units (pack and unpack processor) these are vital. However, there is a desire to cap the complexity of each programmable unit. For the vertex and fragment processors, Boolean operations are supported but general bit-wise operations are not. This is to avoid requiring full functionality integer processing on top of the already-required floating point capabilities of these processors.

REOPENED on July 12, 2002: Certain bit operations are very useful and cannot be easily emulated with floating point operations. For instance, applications could multiple fields of data into texture components and use the bit-wise operators to extract those values (for instance, using 12-bits of a 16-bit luminance texture to store intensity, and the remaining four bits to store opacity). Giving shaders the abillity to do this type of extraction would be preferable to defining new texture formats.

Another way of handling this functionality is with a built-in function to extract a bitfield out of an integer, as this is one expected use of bit-wise operators.

This interacts with issue 90. Without integer textures, there is less need to solve this issue.

RESOLUTION: Bit-wise support is deferred to a future release.

CLOSED: December 10, 2002.

30) *Should internal computations be required to be carried out with 32-bit floating point precision? Or should implementations be allowed to carry out computations with higher or lower precision if they so desire?*

DISCUSSION: This issue is related to Issue (33) and Issue (68).

RESOLUTION: It is already implicit that floating point requirements must adhere to section 2.1.1 of version 1.4 of the OpenGL Specification. This is sufficient.

CLOSED: November 26, 2002.

31) *Can you override the computed LOD or bias within a fragment shader?*

RESOLVED on October 12, 2001: The computed LOD may be biased by a value provided by a fragment shader. Built-in texture access functions with an LOD argument are provided for this purpose.

CLOSED on September 19, 2002.

32) *Are interpolated values perspective correct?*

RESOLVED on June 3, 2002: Yes, variables defined as *varying* are perspective correct.

CLOSED on September 10, 2002.

33) *Should precision hints be supported (e.g., using 16-bit floats or 32-bit floats)?*

DISCUSSION: Standardizing on a single data type for computations greatly simplifies the specification of the language. Even if an implementation is allowed to silently promote a reduced precision value, a shader may exhibit different behavior if the writer had inadvertently relied on the clamping or wrapping semantics of the reduced operator. By defining a set of reduced precision types all we would end up doing is forcing the hardware to implement them to stay compatible. When writing general programs, programmers have long given up worrying if it is more efficient to do a calculation in bytes, shorts or longs and we do not want shader writers to believe they have to concern themselves similarly. The only short term benefit of supporting reduced precision data types is that it may allow existing hardware to run a subset of shaders more effectively.

This issue is related to Issue (30) and Issue (68).

RESOLUTION: Performance/space/precision hints and types will not be provided as a standard part of the language, but reserved words for doing so will be.

CLOSED: November 26, 2002.

34) *Should the design of the OpenGL Shading Language include support for shaders that are not real-time in nature?*

RESOLVED on September 19, 2002: Yes, the design of the language should take into account applications that are not real-time in nature.

CLOSED on September 19, 2002.

35) *Should additional types such as point, normal, color, etc. be added to the shading language?*

RESOLVED on October 12, 2001: No, these type should not be added. The existing generic vector types can support them all without the need for adding additional types to the language.

CLOSED on September 10, 2002.

36) *Will everyone have different implementations for smoothstep and noise, or should we try to specify and enforce a common implementation of these?*

DISCUSSION: This issue is nearly the same as Issue (13). The definition of smoothstep should be sufficient. OpenGL is not pixel-exact, and the definition of smoothstep is as accurate as it needs to be for the specification.

RESOLVED on September 19, 2002: There is a lot of room for a variety of implementations of the OpenGL Shading language. Even with OpenGL today, it is not expected that pictures produced on two different pieces of hardware will produce identical results. The specification should be written in such a way that implementations will produce very similar (though not identical) results. Conformance testing for the OpenGL Shading Language is an issue for the OpenGL ARB to wrestle with in the future.

It has further been decided to add a source specification of noise() to the spec. But, this is not done.

CLOSED on September 19, 2002.

37) *Should the fragment shader functionality to "kill" a fragment be a keyword or a built-in function?*

DISCUSSION: Kill feels like a flow control directive similar to *break* and *continue*. It's not a function which should be overridden by a user function. The Boolean expression can be evaluated in an *if* statement that precedes the keyword kill. There's no reason to continue processing once kill is executed, so no reason to disguise it as a function call. *kill(boolExpr);* as a shortcut for *if (boolExpr) kill;* is not much savings.

RESOLVED on April 15, 2002: The fragment shader functionality to "kill" a fragment should be a keyword.

CLOSED on September 19, 2002:

38) *Should the built-in texture and noise functions be available from within the vertex shader?*

DISCUSSION: There have been numerous requests to support displacement mapping. This request could be satisfied by allowing the built-in noise and texture functions to be available to vertex shaders as well as fragment shaders. This could be implemented in hardware by having the compiler split the vertex shader into a prolog, a texture/noise access, and an epilog. The prolog would be executed by the vertex processing hardware, and the texture/noise access would be done by the fragment processing hardware. The intermediate results would be fed back through the vertex processing hardware to execute the epilog, and then passed on to the fragment shader for fragment processing. On the other hand, it is possible for applications to do this all on the host CPU.

RESOLUTION: Yes, the texture and noise functions should be made available from within the vertex shader as well.

CLOSED October 22, 2002.

39) *Should it be defined that interpolated values for varying variables are determined by sampling at the fragment center?*

DISCUSSION: This interacts with multisampling and requires further investigation.

RESOLUTION: This is to follow the same rules as outlined in section 3.2.1 of version 1.4 of the gl specification.

CLOSED: November 26, 2002.

40) *Should unsigned ints be supported in the language for vertex and fragment processing?*

DISCUSSION: This issue is related to Issue (29). If we allow bit-wise operators, there probably needs to be a way to specify either signed or unsigned integers. Currently unsigned ints are defined only for the pack and unpack shader languages, not for the vertex and fragment languages.

RESOLUTION: Because it has been resolved that integers carry 16 bits of precision, in addition to a sign bit, it is not necessary to introduce an unsigned integer type.

CLOSED: November 26, 2002.

41) *Are gl_FrontMaterial and gl_BackMaterial attributes or uniforms?*

DISCUSSION: The spec currently defines these as attribute arrays, but the spec also says that arrays are not allowed for attributes. If we want to treat them as uniforms, they can remain as arrays. Otherwise, we should change the names and definitions so as to not use arrays (i.e., give each attribute a unique name).

RESOLVED on September 19, 2002: These will be treated as uniforms. Moving forward, we would rather encourage applications to use user-defined attributes if these need to be changed at every vertex.

CLOSED on September 19, 2002.

42) *Should there be a way to specify that the transformed position generated by a vertex shader should be invariant with respect to the fixed functionality pipeline?*

DISCUSSION: This feature was requested by an ISV. Without it, on many graphics architectures, it may be impossible to precisely match geometry rendered using a vertex shader with geometry rendered using the fixed functionality path.

One possible solution is to change the spec where it says that the built-in variable gl_Position must be written by all vertex shaders. Instead, if the variable gl_Position is not written by the vertex shader, the vertex position will be transformed in a manner that is invariant with respect to the fixed functionality pipeline. If gl_Position is written by the vertex shader, the resulting position may or may not be invariant with respect to the fixed functionality pipeline.

But this solution increases the risk that a shader writer might inadvertently fail to write gl_Position, which will now not generate an error but rather invariant transform gl_Vertex. So here is a possible set of alternative resolutions. Since we have built-in functions, a built-in function might be a clean solution to the request for an invariant transformation. All three built-in functions below could provide for invariant transform of gl_Vertex. Alternative (A) will be most familiar to RenderMan shader writers (minus the named spaces). Alternatives (B) and (C) only provide for invariant transform, with (B) allowing the input to be specified while (C) implicitly inputs gl_Vertex.

The original suggested resolution, and these alternative resolutions, solve the ISV request, but in different manners.

(A) Built-in function:

genType transform([mat xform,] genType coord) If matrix is specified, return xform*coord. Else, transform coord invariant to fixed function method.

Examples:

```
    // transform by MVP, not necessarily invariant with fixed function.
    gl_Position = transform( gl_ModelViewProjectionMatrix, gl_Vertex );

    // transform invariant with fixed function.
```

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

(B) Related to (A), but no optional matrix:

genType transform(genType coord) Transform coord invariant to fixed function method

Example:

```
   // transform invariant with fixed function.
   gl_Position = transform(gl_Vertex);
```

(C) Related to (B), but no arguments, rename function:

vec4 fixedtransform() Output invariant with fixed function method, implicit input is gl_Vertex.

Example:

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

RESOLUTION: Use option C from above.

CLOSED October 22, 2002.

43) *What is definition of built-in derivative functions of gl_FB*?*

DISCUSSION: An short example fragment shader demonstrates the question best.

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

Earlier whitepapers allowed general framebuffer read within the fragment processor. OpenGL generally only specifies the fragments to be generated by rasterization, not the order the fragments are generated by rasterization. So general framebuffer reads within the fragment processor could lead to undefined behavior.

Later whitepapers permit only restricted framebuffer reads within the fragment processor. (The pixel at the xw, yw window coordinates of the fragment.). So the question becomes, do the built-in derivative functions conceptually require an implicit general frambuffer read (at least in the immediate neighborhood of the pixel at the xw, yw window coordinates of the fragment)? What does "at any given point in time" mean in this context?

Possible resolutions:

a) Don't allow gl_FB* read operations in the fragment processor. (This interacts with Issue (23).)

b) The built-in derivative functions are undefined if a gl_FB* is a parent of an expression. (The built-in derivative functions are in some cases undefined within the body of a conditional or loop.)

Rejected resolutions:

c) Explicitly define the order which fragments are rasterized by OpenGL.

RESOLUTION: gl_FB* have been removed. Issue 23 has been reopened and closed as disallowing frame buffer reads.

CLOSED on December 11, 2002.

44) *Should the uniform variables that represent current OpenGL state be available only to specific processors or available to any processor?*

DISCUSSION: The current specification is biased toward vertex lighting and fragment shading. Currently, OpenGL state represented as built-in uniform variables is available only to a specific processor (e.g., lighting state is available only to the vertex processor). This makes it unnecessarily difficult for the fragment shader to do lighting calculations with the OpenGL state. The specification should be agnostic about which shaders will need access to what built-in uniform OpenGL state.

RESOLUTION: OpenGL state that is encapsulated as a uniform variable should be accessible to any processor.

CLOSED October 22, 2002.

45) *Should naming conventions for OpenGL state be the same as those adopted for the ARB_vertex_program extension?*

DISCUSSION: Where the OpenGL Shading Language defines *gl_ModelViewMatrix* to refer to a specific piece of OpenGL state, the ARB_vertex_program extension uses *state.matrix.modelview.* Should the conventions be the same for consistency?

The OpenGL Shading Language conventions for referring to GL state were developed before it was clear that an ARB vertex program extension would even be possible due to IP issues and lack of consensus. ARB_vertex_program (and ARB_fragment_program) state binding might confuse some to think of the syntax as a structure in a C-like language. (Less risk of this confusion in an assembly-ish language.) And ARB_vertex_program and ARB_fragment_program packs state into vec4s. There is less of a need for such packing in a C-like language.

RESOLVED on August 13, 2002: No, the conventions need not be the same. There isn’t enough interest in making this name change at this point.

CLOSED on September 10, 2002.

46) *What is the expected behavior for general derivatives at object silhouettes?*

DISCUSSION: It would have to be the local instantaneous derivative if it's to be used for filter width or lod computation. That pretty much dictates that no implementation can look to neighboring fragments to compute derivatives, since it is always possible to construct an object that hits only one fragment (and so has no valid neighbors).

RESOLUTION: See derivative section of paper.

CLOSED on December 4, 2002.

47) *Should the derivative functions have names that are more similar to those used in Renderman?*

DISCUSSION: The naming of the derivative functions is somewhat at odds with the precedent established by RenderMan, where Du(f) and Dv(f) compute df/du and df/dv respectively. dPdu and dPdv are potentially more accurate, but functionally equivalent to Du(P) and Dv(p), where P is the built-in variable for 3D location of the sample. We should at least consider Dx() and Dy() for the names of the OGL2 derivative functions.

RESOLUTION: Names are changed to dFdx, dFdy.

CLOSED on December 4, 2002.

48) *Should a dPdz (or Dz) function be added?*

DISCUSSION: If someone wanted to derive df/du and df/dv We'd also need dPdz() or Dz() to avoid a singularity at object silhouettes

RESOLUTION: This is not added.

CLOSED on December 4, 2002.

49) *Should the shading language include structs?*

DISCUSSION: The shading language should support structs. Structs provide a clean way of grouping data to create abstract data types. They are convenient for developers and are supported in C and other generic programming languages.

On the other hand, no compelling case for adding structs to the language has been made. It could help us get the language finalized sooner if we left this till a later rev of the language specification. But, if structures were added, it would be nice to define the lighting state in terms of structs. Vital Images (ISV) indicates they would like to have structs in the language.

RESOLVED on September 19, 2002: The shading language will include structs.

CLOSED on September 19, 2002.

50) *Should the vertex processor be defined in a way that allows it to perform tessellation of curved surfaces?*

DISCUSSION: The issue of geometric LOD and curved suffices is so complex and is so continuously developed that no piece of hardware simpler then a general purpose programmable CPU is up for the job. Any choice of primitive, will be heavily disputed. And most of the popular curved surface primitives like creased subdivision surfaces or trimmed NURBS are not easily implemented in hardware.

If we look at the problem form a performance view, the generation of LODs are generally not the problem since new LODs don't need to be generated to often, only when the geometry in a significant way has moved closer or further away from the camera. The problem comes in when we have animation of interactive manipulation of the surface. In these cases the topology doesn't change so this can be solved on a very efficient way. We simply store a list of references to CVs and weighting factors for each vertex in the LOD.

This simple code can accommodate all types of curved surface:

```
for(i = 0; i <vertex_count; i++)
{
    x = 0;
    y = 0;
    z = 0;
    for(j = 0; j < *influence_list_length; j++)
    {
        index = *index_array++;
        value = *value_array++;
        x += value * control_vertex_array[index].x;
        y += value * control_vertex_array[index].y;
        z += value * control_vertex_array[index].z;
    }
    surface_vertex_array[i].x = x;
    surface_vertex_array[i].y = y;
    surface_vertex_array[i].z = z;
```

```
        influence_list_length++;
    }
```

This could be integrated in the vertex shader to allow for maximal flexibility, but this means that vertex shaders must have the ability to do random access arrays of data.

RESOLUTION: Postponed to a future version of this specification.

CLOSED: October 22, 2002.

51) *Should the language provide some mechanism to distinguish variables that are position independent from those that aren't?*

DISCUSSION: Comments have been made along the lines of "should we really expose SIMD semantics in the language?" Response:

- What's really being introduced is position independence.
- The existing 'uniform' and 'varying' already introduce this concept, this proposal just completes it.
- Most hardware will benefit from it.

Feasibility: It's possible for a compiler to do sufficient data-flow and control-flow analysis to find all paths that could lead to the assignment of a position independent variable. It may find extra paths, but is not allowed to miss any actual paths. From this, a compiler can prove if a position independent variable only takes on values that are position independent. It may, on rare occasion say a position independent variable is not so, and be wrong, but these will typically occur in degenerate code.

Global Uniforms. It's asking too much of a compiler to do cross-function data-flow analysis, especially across different compilation units. So, the idea of global read/write position independent variables is not supportable and not proposed. Hence, there is no conflict between these uses of 'uniform'.

Output Uniform Parameters. Same problem as uniform globals. Uniform globals and parameters must be read only.

A past alternative was to use the 'int' type as a hint to the compiler that a value was position independent, like for loop indexes, texture ids, array subscripts, etc. This was troublesome, because floating point based control flow could make the hint invalid, leaving the compiler as burdened as it would be without the hint. And/or it made the 'int' type less useful, forcing it to adhere to the proposed 'uniform' semantics. This was too much tying together of otherwise independent ideas.

This issue is related to Issue (27).

RESOLUTION: Use the 'uniform' qualifier to identify locally scoped variables, function return values, and function parameters as being position independent. Local 'uniform' variables cannot be written to with values that were derived from position. Functions declared to return a 'uniform' can only return values not derived from position.

The compiler may return a warning if there is a statically identifiable path through the code that leaves a position dependent derived value in a position independent variable. That is, if a variable is declared uniform or passed to a uniform parameter, the compiler will issue an error if it can't prove the variable is always position independent.

CLOSED October 22, 2002.

52) *How should resource limits for the shading language be defined?*

DISCUSSION: Various proposals have been discussed. One very important consideration is to end up with a specification that provides application portability (e.g., ISVs do not need to support multiple rendering back ends in order to run on all the different flavors of hardware). ISVs definitely would prefer the specification to say that the shading language implementation is responsible for ensuring that all valid shaders must run.

RESOLUTION: Resources that are easy to count (number of vertex processor uniforms, number of fragment processor uniforms, number of attributes, number of varying, number of texture units) will have queriable limits. The application is responsible for working within these externally visible limits.The shading language implementation is responsible for virtualizing resources that are not easy to count (number of machine instructions in the final executable, number of temporary registers used in the final executable, etc.).

CLOSED on October 29, 2002, as being part of the API issues list.

53) *How are user clip planes handled if the coordinate spaces are separated by a transform that is non-linear?*

DISCUSSION: The shading language specification relies on the standard definition of GL clipping. This works as long as the coordinate spaces are only separated by a linear transformation, however the shading language also lifts these restrictions.

SUGGESTED RESOLUTION: Adopt the "clip coordinate output" approach found in certain NVIDIA proposals for ARB_vertex_program (removed long before the spec was final). This approach provides fully programmable user clipping, not dependent on any semantics of the program or any analysis thereof; and it does not leave most cases undefined.

RESOLUTION: Specify that user clip planes work only under linear transform. It is currently undefined what happens under non-linear transform.

CLOSED October 22, 2002.

54) *How are global pixel operations (e.g., histogram, min/max) supported?*

ADDED on September 10, 2002.

DISCUSSION: The current specification allows access only to the fragment at the current location (though this is open for discussion as per Issue (23)). Operations that require access to other fragment locations in the frame buffer or on the incoming data stream are expressly prohibited. How will the shading language provide functionality that supports global pixel operations such as histogram and min/max?

The desire is to run a program that operates on multiple fragments (similar in spirit to how to run a vertex program that generates new geometry, Issue (50)).

RESOLUTION: Postponed to a future version of this specification.

CLOSED: October 22, 2002.

55) *Should the preprocessor have any directive in addition to those already defined?*

ADDED on September 10, 2002.

DISCUSSION: A number of preprocessor directives could be added to the language specification, for instance, #if, #elif, #include, #define token(...) (with arguments), ## (token pasting), #line, #error, # (by itself), # (to make a token into a string), defined(token) (and all the other operators, &&, |, +, etc.), and predefined macros, like __DATE__, __FILE__.

In particular, #if would be a useful addition to support processing of versions, dates, and the like. But this necessitates bringing in the ||, &&, >, <, ! operators.

RESOLVED on September 24, 2002: The shading language preprocessor will essentially have all the capability of the C preprocessor except that the #include directive is not supported and string-based directives are also not included.

CLOSED on September 24, 2002.

56) *Is it an error for an implementation to support recursion if the specification says recursion is not supported?*

ADDED on September 10, 2002.

DISCUSSION: This issues is related to Issue (22). If we say that recursion (or some other piece of functionality) is not supported, is it an error for an implementation to support it? Perhaps the specification should remain silent on these kind of things so that they could be gracefully added later as an extension or as part of the standard.

RESOLUTION: Languages, in general, have programs that are not well-formed in ways a compiler cannot detect. Portability is only ensured for well-formed programs. Detecting recursion is an example of this. The language will say a well-formed program may not recurse, but compilers are not forced to detect that recursion may happen.

CLOSED: November 29, 2002.

57) *Should there be a standard way for applications to invoke debug mode?*

ADDED on September 10, 2002.

SUBSUMED by Issue (67).

CLOSED on September 24, 2002.

58) *Should the language include a list of reserved words?*

ADDED on September 10, 2002.

DISCUSSION: Currently the specification does not contain a list of reserved words. Without such a list, valid shaders might become invalid when we make additions to the language in the future.

SUGGESTED RESOLUTION: Yes, the language should include a list of reserved words, including the following: struct, union, enum, typedef, template, goto, switch, default, inline, noinline, long, short, double, sizeof, volatile, public, static, namespace, using, asm, cast, half, fixed, and all tokens that contain two consecutive underscores.

RESOLVED on September 24, 2002: Yes, the shading language should include a list of reserved words.

CLOSED on September 24, 2002.

59) *How should function parameters be passed?*

ADDED on September 10, 2002.

DISCUSSION: Today the specification says that function parameters are call by reference, no aliasing is allowed and *output* is used for output parameters. This has some non-obvious problems: (A) Uniforms and other globals cannot be passed in as parameters, as that would create an alias. (B) Varyings and other write-only variables are very tricky to pass by reference, as there is nothing that says a parameter is write-only. (C) If a shader writes into a “pass by reference” parameter, it should either update the caller's argument, or the shader should generate an error because it was not an *output* parameter. However, expected usage seems to be that it's all right to write to a non-output, the effect is just local.

The specification could be changed to say that function parameters are call by value-return, which means the following: (A) A parameter with no qualifier means the parameter is copied in from the caller at call time. (B) The qualifer *output* (or *out*) means the parameter will be copied back to the caller at return time, but not copied in at call time. (C) The qualifier *input output* (or *inout*) means the parameter is both copied in and copied back.

These semantics solve all the parameter-related aliasing problems. The compiler doesn’t have to check for aliasing, it can compile as if there is no aliasing, and it's well-defined to the shader writer what happens if they pass parameters in a way that looks like aliasing. These semantics also allow for write-only variables to be passed to a function. Finally, this solution allows writing to a non-output parameter, while making it clear it's only a local copy that gets modified.

RESOLVED on September 24, 2002: Change the spec to say that function parameters are call by value-return as defined above.

CLOSED on September 24, 2002.

60) *How should the built-in names for lighting state be defined?*

ADDED on September 11, 2002.

DISCUSSION: The current names for lighting state (*gl_Light0..n*[8] and the associated predefined array index values) make it awkward to write a loop to process lights. This issue is related to Issue (49).

RESOLVED on September 24, 2002: Structs should be added to the shading language, and the lighting state should be redefined as an array of light structs.

CLOSED on September 24, 2002.

61) *Should user-defined functions be allowed to redefine built-in functions?*

ADDED on September 13, 2002.

DISCUSSION: It’s not clear that there is anything to be gained by allowing this. If users inadvertently use the same name as a built-in, they will get unexpected behavior or a drop in performance or both.

RESOLUTION: Yes. This is normal behavior for a language and a library.

CLOSED on September 10th, 2002.

62) *Should the language include texture gen coefficients for eye/object plane?*

ADDED on September 13, 2002.

DISCUSSION: Issue raised by Kent Lin of Intel.

RESOLUTION: Yes, this state should be added.

CLOSED October 22, 2002.

63) *Should the language include a built-in variable for the projection matrix?*

ADDED on September 13, 2002.

DISCUSSION: Built-in variables are already defined for the model-view matrix and the model-view-projection matrix. Should a built-in variable be added for the projection matrix as well?

RESOLUTION: Yes, this state should be added.

CLOSED October 22, 2002.

64) *Should built-in variable names be added for the state introduced in OpenGL 1.4?*

ADDED on September 13, 2002.

DISCUSSION: The specification is currently written against OpenGL 1.3, therefore it does not contain the point parameter states and fog coordinate state. Should these be added?

RESOLVED on September 24, 2002: Yes, we should add built-in variable names for the state introduced in OpenGL 1.4.

CLOSED on September 24, 2002.

65) *Should mat * mat perform a matrix multiply or a component-by-component multiply?*

ADDED on September 16, 2002.

DISCUSSION: Currently the specification states that the "*" operator will cause a component-by-component multiplication if two matrices are specified. The multiply operator (*) does the expected linear algebra operations for scalar * scalar, scalar * vector, and matrix * vector but not for matrix * matrix where it is component-wise instead, which is a comparitively rare operation. This was done for consistency with other operators that behave component-wise. (E.g., we probably do not want matrix / matrix to be real matrix division instead of component-wise. Should matrix * matrix be changed to indicate a matrix multiplication operation?

RESOLUTION: The specification should be modified to indicate that the "*" operator will cause a matrix multiply if the two operands are matrices.

CLOSED October 22, 2002.

66) *What should the specification say about the length of time a shader is allowed to run?*

ADDED on September 16, 2002.

DISCUSSION: Earlier versions of the white paper talked about a watchdog timer. Is such a thing necessary as part of the language specification?

RESOLUTION: The language specification should not say anything about the length of time a shader is allowed to run. Timeouts, interactivity, and detecting malicious shaders are implementation and/or operating environment details.It probably should be somewhere in the GL2 extension specification(s) that an executing shader is terminated if the application that caused execution of that shader is terminated.

CLOSED October 22, 2002.

67) *Should there be a standardized way to specify debugging and optimization levels?*

ADDED on September 16, 2002.

DISCUSSION: Four alternatives are possible. (A) We don't debug, and we always optimize, so there is no problem. (B) We add debug and optimize parameters to the entry points for compiling and linking. (C) We use #pragma to specify debug and optimization levels, and outline basic portable meanings. (D) We say this is entirely platform dependent, and don't specify anything.

(A) seems short-sighted because turning optimization on/off is a technique and work-around for tracking down some kinds of defects, we will eventually want to debug shaders, and there will be compile-time vs. run-time trade-offs (e.g. if an application is dynamically generating shaders that have really short life-times, it may be faster to turn off slower optimizations).

(B) seems a bit awkward as there will be platform dependent aspects to these activities. (D) seems to be going to far for something that's going to, in principle, exist on all platforms.

RESOLUTION: Use #pragma to specify debug and optimization levels and outline basic portable meanings.

CLOSED October 22, 2002.

68) *Should the language support explicit data types such as 'half' (16-bit floats) and 'fixed' (fixed precision clamped data type)?*

ADDED on September 17, 2002.

It is common for high level languages to support multiple numeric data types, to allow programmers to choose the appropriate balance between performance and precision. For example, the C language supports the *float* and *double* data types, as well as a variety of integer data types. This same general consideration applies for a shading language.

For shading computations, precisions much lower than 32-bit floating point are often adequate. Until recently, most graphics hardware performed all shading computations in 9 or 10 bit fixed-point arithmetic. Lower-precision data types can be implemented with higher performance, especially when the data must travel off chip (e.g. texture data). For this reason, it is desirable to provide access to data types with precision of less than 32 bits in a hardware shading language.

Issue (33) discusses precision hints. Precision hints are less useful than additional data types, because precision hints do not allow function overloading by precision. Developers find it very convenient and useful to be able to have functions with same names and argument lists with different precision data types.

It is also important to be able to specify data type per variable (as opposed to per-shader), because it is common for some computations (e.g. texture-coordinate computations) to require higher precision than others.

On the other hand, there is a desire to ensure that shaders are portable between different implementations. In order to achieve portability, implementations that don't have native support for *half* will be penalized because they will have to clamp intermediate calculations to the appropriate precision. If these additional data types are hints that the compiler can choose to do the calculations to lower precision then this leaves the ISV open to unintended clamping or overflow semantics so different architectures can give very different results. The hint also implies that there is a well specified way to convert to between types under the hood so function overload resolution gets more complicated and additional rules are needed to resolve ambiguities, unless all legal combinations of functions must be supplied. Specifying all legal combinations requires adding quite a large number of additional function types (dot product will need {*float, half, fixed*} * {*float, half, fixed*} * number components or 36 versions (vs 4 with only *float*).

If the additional data types are real types then what can they be applied to? If it is to uniforms and attributes then the different sizes now reflect in the API, but *half* and *fixed* have no native support in C. If a *half* is followed by a *float* does this mean a *float* has to start on a 16 bit boundary? What about packing of *fixed* - the true size is undefined. If *half* and *fixed* are just restricted to temporaries then this makes things easier but now the storage efficiency benefit is lost.

The OpenGL spec currently says “The maximum representable magnitude of a floating-point number used to represent positional or normal coordinates must be at least 2^{32}.” Should we introduce something that runs counter to this? s10e5 precision is inadequate for texture coordinates even for a 1k by 1k texture. It seems that half-floats open a door for precision issues to propogate throughout a shader.

RESOLUTION: Performance/space/precision hints and types will not be provided as a standard part of the language, but reserved words for doing so will be.

CLOSED: November 26, 2002.

69) *Should the fragment shader be able to access a varying variable that provides the position?*

ADDED on September 17, 2002.

DISCUSSION: Window position can be very useful in certain fragment shaders. For example, it can be used to implement stipple patterns using fragment shaders.

RESOLUTION: The window position is part of the built-in variable *gl_FragCoord* that is available within the fragment processor. The specification should be modified to make it clear that the x and y values of this variable define the window position of the fragment.

CLOSED October 22, 2002.

70) *Should the language support boolean vectors (e.g. bool2, bool3, bool4), and corresponding vector operators?*

ADDED on September 17, 2002.

DISCUSSION: The language supports short float vectors (e.g. *vec2*, *vec3*, *vec4*), so it would be consistent to support boolean vectors as well. If the language includes support for boolean vectors and operations, it is simple to express elementwise vector computations. For example, the *min* and *max* vector operations can be elegantly implemented within the language using bool-vector operations.

When a comparison operator such as '<' is applied to vector operands, the result is a boolean vector that contains the result of the elementwise comparison. The '?:' construct operates in elementwise fashion if the first operand is a boolean vector. *if*, *while*, and *for* still require a scalar boolean value

RESOLUTION: Vectors of bool will be added. But C-like short-circuit evaluation of && and || will be kept. For this release, if and ?: will select based only on scalar bools. Since == and != should also return a scalar bool (as they do for all types, including struct), they will do so, and not return a vector of bool when vectors are compared. Hence, <, >, <=, and >= also will not create vector of bool. They will not legally operate on vectors. Rather, built-in functions will be added for relational operations on vectors.

CLOSED: November 5, 2002.

71) *Should the shading language support compound data structures such as arrays of arrays, structs of arrays, arrays of structs, etc.?*

ADDED on September 17, 2002.

DISCUSSION: The ability to create compound user-defined data structures is a fundamental part of almost all high-level programming languages. Omitting support for these capabilities would be inconsistent with the generally forward-looking nature of the OpenGL shading language design effort.

On the other hand, as with the issue of adding structs (Issue (49)), a case can be made for every language that it will be useful to someone, somewhere, sometime. Is it worth taking the time and effort to ensure that this functionality is part of the first version of the specification? Will it cause any problems to defer it and add it later?

RESOLVED on September 24, 2002: Yes, the shading language should support arrays of arrays, structs of arrays, arrays of structs, etc.

CLOSED on September 24, 2002.

72) *Should the shading language include a switch statement?*

ADDED on September 19, 2002.

DISCUSSION: The C language *switch* statement is a useful construct in writing clean code. The alternative is to use a less readable collection of *if* statements.

RESOLUTION: Switch will not be added for initial release. There is desire to manage floating point ranges, which would take a long time to work out.

CLOSED on October 1, 2002.

73) *What are the semantics of the keyword return in the main() function?*

ADDED on September 19, 2002.

DISCUSSION: Currently the specification does not say what happens if the keyword *return* is included in the main() function in any place other than the very end of the function.

RESOLUTION: return means exit main, just like getting to the end. It does not mean kill.

CLOSED on October 1, 2002.

74) *How is data computed by the vertex or fragment shader communicated back to the application?*

ADDED on September 24, 2002.

DISCUSSION: This issue is related to Issue (50) and Issue (54). If either the vertex processor or the fragment processor are allowed to compute results that don't continue down the processing pipeline (e.g., histogram, min/max, or computing vertex array data from control points), how will those results be communicated back to the application?

RESOLUTION: Postponed to a future version of the specification.

CLOSED: October 22, 2002.

75) *Should uniforms and attributes which are initialized by the application allowed to be structs?*

ADDED on September 25, 2002.

DISCUSSION: If we don't allow this, the application has to pass all data via individual global variables, and shader code must pack the data together in structs in order to use structs. This causes ugly unnecessary shader code. On the other hand, initializing struct data should not involve a complexification of the API.

Also, it seems broken to have structs in the language, but not have a way to initialize them from the application. On the other hand, a complete solution for initializing structs seems beyond this release.

It’s also been noted that attributes can be matrices, but there is no API for initializing them.

RESOLUTION:

* Add entry points to initialize an attribute matrix. At bind time, a 4x4 matrix takes 4 consecutive locations, a 3x3 matrix takes 3 consecutive locations, and a 2x2 matrix takes 2 consecutive locations. Details of layout are hidden. If an implementation only needs one slot for a 2x2, it only has to use one slot, but the room is there for implementations that need two.

* Don't yet support arrays of attributes and structs of attributes.

* Allow uniform struct and array of struct.

* Support API initialization of struct members by specifying a string at GetUniformLocation time that selects the member to be initialized. E.g. "struct.member", "struct[4].member", "struct[2].member[2]" etc.

* Don't yet support struct-level initialization in the API, wait for future full solution that understands strides, alignments, padding, etc.

CLOSED October 22, 2002.

76) *Should vec2, vec3, vec4, mat2, mat3 and mat4 be defined as structures?*

ADDED on September 26, 2002

DISCUSSION: Treating these types as structs is that the language would get much clearer and would be easier to specifiy. This also will simplify compiler development.

Currently, these types are involved in special language features like"v.xyz" syntax and swizzling operations like "v.yzx". However, such operations are not neccessary in a HLSL.For example, swizzling operations are typical for assembly language tricks, for example to express a cross product with two assembly commands. But such tricks are not necessary in a HLSL, you simply use build-in functions, or define user-defined functions if really needed.Furthermore, in the rare cases where such operations are really needed they can be easily expressed by standard language features like vec3(v.y, v.z, v.x).

Furthermore, a syntax like v[i][] is planned to be added as special language feature.But situations where you need dynamic indices for componentsare quite rare and can be easily expressed by buildin access functions.

Another aspect is having different names for identical components,for example xyzw versus rgba. Loosing this feature is probably this is the only real disadvantage of treating vec3 and vec4 as normal struct.On the other hand, having unique names xyzw also may clarify shader programs.

On the other hand, the vector and matrix type are indeed special: (A) Most operators work on them. Not so for struct. (B) Every element of a vector or matrix is the same type. Not so for struct. (C) A matrix is conceptually two-dimensional, while structs are conceptually one-dimensional. The current language isn't perfect at keeping a matrix 2D, given we have to sometimes know its column major order, but it can be treated as 2D sometimes. (D) Hardware may have special hardware for vector and matrix types that are more difficult to map to generic structs than to built-in types.

RESOLUTION: vectors and matrices are not structures, but some changes will be made. Summary:

* R-value swizzling becomes an operation on an expression. This is a change from the initial spec saying it was a member selector on a vector variable.

* We remove the empty-brackets syntax.

* We keep the swizzling syntax we already have for vectors.

* We keep the array access syntax for vectors.

* We add, for matrix m, that m[i] is the ith column and is vector type. Both l-value and r-value.

* Because m[i] is a vector, and vectors have array syntax, then it just falls out that m[i][j] is the ith column, jth row of m.

CLOSED on October 8, 2002.

77) *Should the type of gl_FragStencil and gl_FBStencil be changed to int?*

These are currently floats. The integer type has progressed in the language, so this should be reconsidered.

RESOLUTION: Yes. However, stencil writing has been deferred to a future release, and frame buffer reading has been removed.

CLOSED on December 10th, 2002.

78) *Are stencils automatically clamped to the current min and max values that can be stored in the stencil buffer?*

RESOLUTION: Yes. However, stencil writing has been deferred to a future release.

CLOSED on December 10th, 2002.

79) *Do we need to have near and far clipping planes available to the fragment shader.*

DISCUSSION: ARB_fragment_program has these.

RESOLUTION: Yes.

CLOSED on December 10th, 2002.

80) *Rectangular (non-power of 2) textures aren't indexed by 0.0 to 1.0, but rather by their actual dimensions. Is this a problem?*

DISCUSSION: Yes this is a problem. Some hardware needs to know at compile time what kind of texture is being accessed. We want to avoid having to recompile shaders due to state changes. More texture built-in names could be used, so it is known at compile time what kind of texture is being accessed. Something like **textureRect3** to mean a rectangular texture returning 3 components. It is also the case that rectangular textures are not a part of OpenGL 1.4, so these functions could be added as an extension, and not be part of this release of the language specification.

RESOLUTION: Make room for adding more functions to support other texture types, but defer doing this for rectagular textures until they have become part of core OpenGL.

CLOSED on December 17th, 2002.

81) *Should we support a way of accessing fixed functionality fog from a shader, to take advantage of possible fog that may exist in fixed functionality hardware?*

DISCUSSION: This sounds similar to supporting a fixed functionality transform, which we do. However, that was done out of need for invariance, which is not an issue with fog. It is also provided in ARB_fragment_program, but the spirit of this spec is to replace fixed functionality and extra ways of accessing it with programmability.

RESOLUTION: No special support of a fixed pipeline fog access will be provided.

CLOSED on December 17th, 2002.

82) *Is it really necessary to require writing of gl_Position, gl_FragColor or gl_FragDepth?*

DISCUSSION: This can be difficult to handle error cases for when the writes are conditional. There was also discussion that either **kill** should be called, or a gl_ output be written in a fragment shader. However, that is irrelevant, as it's okay to neither kill nor write any outputs in a fragment shader. For the vertex shader, it still makes no sense to not write a position, so this should still be required.

RESOLUTION: For the fragment shader, there are no rules; either **kill** can be called or not, and if not, nothing need be written to, existing values are picked up from the pipeline. For the vertex shader, gl_Position should still be written, with the compiler giving a diagnostic when possible.

CLOSED on December 10th, 2002.

REOPENED on January 7, 2003, on concern of performance impact of writing default *gl_FragColor* and *gl_FragDepth* when a compiler thinks they are conditionally written. For color, it should just be undefined to not write *gl_FragColor.* For depth, it is more complex, as if depth is not written, then the fixed functionality computed depth should be used. However, if depth is conditionally written, the compiler will always have to initialize depth, which is a possible performance hit. Further discussion of this generated alternatives of always having to write gl_FragColor from a shader, or more complex things based on what the source code looks like.

RESOLUTION: Say that if a shader conditionally writes gl_FragDepth, then it must always write it. See issue 95 for invariance concerns.

CLOSED January 17, 2003.

83) *What should we do for modifying stencil? Does this effect push/pop state?*

DISCUSSION: Writing a stencil value in a fragment shader introduces new functionality to OpenGL. By itself, it is of questionable use. On the other hand, existing operations like increment and decrement of stencil aren't obviously expressible with the current specification. It may be that stencils should only be updated when a fragment shaders writes no outputs and the API has been set up for rendering to stencil. But, this has not been thoroughly investigated.

RESOLUTION: Stencil modifications are deferred to a future release.

CLOSED: December 17, 2002.

84) *Should we add projective texture lookup? What about SHADOW textures?*

DISCUSSION: Projective texture lookup could be postponed. Shadow textures are part of 1.4, so should be added, using something like **textureShadow***. However, if adding these, it's trivial to also add projective lookup at the same time. What if shadow modes are not enabled? Expectation is the model looks like hardware just does whatever it's set up to do by the application, and a shadow call from a shader picks up what was thus specified. Separately named functions will be used, because more data is input for the same target than for non-shadow lookups. Shadow could be called "compare" instead of "shadow", but renderman calls it "shadow".

RESOLUTION: Add projective and shadow textures, with new names **proj** and **shadow** reflecting this. Furthermore, projective textures will accept two sizes of input: all will take a vec4, plus a 2D projective will take a vec3, etc. Also ensure spec says results are undefined if using a shadow built-in on a texture not set up with a comparator, or using a non-shadow built-on on a texture set up with a comparator.

CLOSED January 17th, 2003.

85) *How does the compiler know if it's a 1D, 2D, 3D or CUBE texture that's being accessed? Basing this on the number of components of an argument is error prone.*

DISCUSSION: Which texture to use was based on the type of the lookup coordinate argument. However, it's conceivable to support having many textures bound to the same texture unit number, and when the shader is written, the author should have in mind which one they are accessing, and they shouldn't get the wrong one due to getting a type wrong. Explicitly saying which textue could be done with enums passed to the existing texture calls, or by adding new names with the texture type in the name. The problem with using enums is that implies an argument a variable could be passed into, while the requirement was to know at compile time. This argues for a name change.

RESOLUTION: Add more texture names so that it is explicit at compile time what kind of texture lookup is being done. Something like **texture1D3, texture2D3, texture3D3,** and **textureCube3,** where the last number is the number of components returned (could be 1, 2, 3, or 4).

CLOSED: December 17th, 2002.

86) *There is little mention of color index in the spec. What support is provided for it?*

DISCUSSION: Other extensions (ARB_vertex_program, ARB_fragment_program, texture application) say operations are undefined if rendering in COLOR INDEX mode.

RESOLUTION: Remove references to this other than saying COLOR INDEX operations are undefined.

CLOSED: December 10th, 2002.

87) *Aux data buffers were part of the OGL2 white papers, but this specification needs to stand on the current OGL core.*

RESOLUTION: Remove aux data buffers from the specification. They can be added back if/when a future extension provides other buffers in core GL to write to.

CLOSED: December 17th, 2002.

88) *Variable array indexes of some arrays may be difficult to implement.*

DISCUSSION: Arrays could be limited to just uniforms. However, this is restrictive, other graphics languages had local variable arrays. It also lacks orthogonality with structs that contain arrays, but may be used as a uniform or a local. It may help to limit the indexes of non-uniform arrays to be uniform indexes. It seems unlikely that a shader would initialize a whole non-uniform array and then index it with non-uniform indexes. (It seems much more likely that a non-uniform index would be applied to a static array or texture.) On the other hand, since this is unlikely, the language spec. could be left clean and full functionality, and for the next year or two it's okay for compilers to complain that something is too complex (given that it's an unlikely need).

RESOLUTION: Full array support will be specified.

CLOSED: January 7, 2003.

89) *Why aren't the varyings gl_TexCoord0...gl_TexCoordn an array?*

DISCUSSION: There may be some performance or compiler convenience to knowing these at compile time or not allowing a variable index. This could interact with issue 88, where if indices to varying arrays must be uniform then it's easier to support these as arrays.

RESOLUTION: These will be changed to arrays. Working on details for the 2 problems listed above.

CLOSED: January 7, 2003.

REOPENED: Issue 88 was resolved with full array support. However, that leaves two possible issues with texture coords. i) not being able to tell how many coords are actually active in a shader, and ii) not being able to tell at compile time which coords are being accessed.

DISCUSSION: Alternatives:

1. Cg says something like this for constant loop-iteration:

"Can be determined at compile time" is defined as follows: The loop-iteration expressions can be evaluated at compile time by use of intra-procedural constant propagation and folding, where the variables through which constant values are propagated do not appear as lvalues within any kind of control statement (if, for, or while) or ?: construct.

2. Be really restrictive: it must be a compile time constant or an induction variable with compile time constant start/end/increment values.

3. Actually solve the fundamental resource size problem: make the shader writer re-declare the array if they violate #2. That is, allow "varying gl_TexCoord[N]" to be declared by the shader, where N is how many coordinates they want to use and require it if #2 isn't satisfied. Allow full variable access in the language (with early hardware/compilers warning when it gets to tough.)

4. Do #3 with a #pragma instead.

5. Variation: Either all indices to gl_TexCoord[] in a shader must be constant expressions, or the shader must declare gl_TexCoord[] with a size. The built-in should be declared as an empty array so it’s consistent with C to declare it again with a size. Multiple modules can declare it with different sizes, the maximum will be used at link time.

RESOLUTION: Use #5.

CLOSED: January 24, 2003.

90) *The built-ins specify texture functions that return integers. However, such textures are not in core OpenGL.*

RESOLUTION: These built-in functions will be removed. They can be added back as part of a future specification that adds integer textures to OpenGL.

CLOSED: December 17th, 2002.

91) *There seems to be missing fog information in the fragment shader. gl_EyeZ is not enough. Should there be some derived information?*

RESOLUTION: Expand float gl_EyeZ to vec4 gl_FogFragCoord.

CLOSED: December 17th, 2002.

92) *We need a way to get object code back, just like the model of C on host processors.*

DISCUSSION: Lots in email. This is about lowest-level, machine specific code that may not even be portable within a family of cards from the same vendor. One main goal is to save compilation time. There seems to be general consensus that this has merit, but does not effect the language definition.

RESOLUTION: This is an API issue, not a language issue.

CLOSED on December 19, 2002 as being an API issue.

93) *In reality, the output varying interface from a vertex shader is different than the input varying interface to a fragment shader, but this spec shares a single interface. This causes trouble with fog and lacks compatibility with mix & match of fixed functionality and ARB_fragment_program*

DISCUSSION: Splitting this interface in two makes sense. The fragment input should be compatible with ARB_fragment_program and the vertex output should be compatible with ARB_vertex_program. Fog can be dealt with this way, as reflected in the specification.

RESOLUTION: This should be done.

CLOSED January 17, 2003.

94) *The way the language spec. is today, it is not possible to know at compile time which targets are used on which units. If one believes hardware must be set up in advance with the right target on the right unit, this is a big problem.*

DISCUSSION: Possible solutions:

A. Use traditional OGL enables and binds and precedence to allow the app to communicate which target is the one that needs to be active when the shader runs.

B. Change the language to require compile-time inspection to be sufficient to see which target is being used on which unit.

C. Expect hardware to be able to dynamically access the requested target without having been set up in advance by the driver to do so.

D. Add a new entry point to the API for setting an active texture: a) BindTextureGL2(GLenum textureUnit, GLenum GLuint texture) (maybe without textureUnit parameter) But it seems that nobody wants such a function because of incompatibility to the standard pipeline. b) Another option may be a new utility function UseTexture(GLenum textureUnit, GLenum GLuint texture), which is completely equivalent to i) binding the texture, ii) enabling the target of this texture and iii) disabling all other targets. If texture is NULL, all targets are disabled. This solution would be 100% compatible to the standard pipeline; the new function is only for convinience, providing no new functionality (maybe as glu function).

E. Use 'samplers' the way Cg does.

RESOLUTION: Use samplers. Types **sampler1D, sampler2D, sampler3D,** and **samplerCube** will be added. They will not be writable in a shader. They can only be global uniforms. They can be arrays. They are opaque and cannot be operated on within a shader, only referenced. The first parameter of texture calls will accept them. The API must somehow bind texture or texture/unit to them. Enables, active-texture, etc. are superceded by these API bindings. At the language level, they will provide the basis for possible future virtualization of textures.

Also see issue 97.

CLOSED January 17th, 2003.

95) *Are there needs to solve invariance problems with gl_FragDepth or other aspects of the shading language?*

DISCUSSION: Some have raised concerns here, but it's not clear what the problems/solutions are.

RESOLUTION: For *gl_FragDepth*, say that there is no guarantee of invariance between a fixed functionality depth and a shader computed depth. However, using the same shader depth computation multiple times in the same or different shaders will be invariant.

Defer other variance issues to the next release.

CLOSED January 17th, 2003.

96) *The **lod** built-in doesn't make sense for anisotropic filtering.*

DISCUSSION: Seems like we need **lod1D, lod2D, lod3D, lodCube.** But, the lod() built-in doesn't seem useful, given that the texture built-ins currently take biases, not absolute lod's. And also given that if anisotropic filtering were added, lod's specification would be troublesome.

RESOLUTION: Remove **lod** built-ins.

CLOSED January 17th, 2003.

97) *Do we need a shadow sampler? E.g. a type **sampler1DShadow**, **sampler2DShadow**?*

DISCUSSION: This gives even higher level of enforcement over hooking up the right texture state with the right built-in lookup function. On the other hand, it sets a direction for adding lots of type keywords.

RESOLUTION: Yes, add these types.

CLOSED January 24, 2003.

98) *vec2, vec3, vec4 should be changed to float2, float3, float4 to be consistent with other languages. **kill** should likewise be **discard**.*

DISCUSSION: Should replace vec2, vec3, vec4 with float2, float3, float4 to match the established conventions of the Stanford RSL, Cg, and HLSL.

Additionaly, these names allow better support for data types based on other scalars (int2, int3, int4, half2, half3, double4, etc).

On the other hand, it seems consistent to have a scheme where a vector is made with a prefix type abbreviation (default is float), followed by "vec", followed by number of components.

type	vector of 4
float	vec4
int	ivec4
bool	bvec4
half	hvec4
double	dvec4
float16	f16vec4
int32	i32vec4

Should we also change "kill" to "discard"? The rationale is that the OpenGL specification consistently uses the term "discard" when a fragment is discarded and never uses kill. The KIL instruction does appear in both the ARB_fragment_program & NV_fragment_program specifications (probably because it makes a nice mnemonic for a three-letter instruction), but that's not consistent really with a core OpenGL specification. Another reason is that Cg/HLSL has "discard" be a keyword and there's no good reason to have redundant keywords for the same functionality.

RESOLUTION: Change kill to discard, leave vectors the way they are.

CLOSED January 24, 2003.

99) *There is a mechanism lacking for dealing with a really large table in memory.*

DISCUSSION: Want to be able to efficiently communicate a large array of information to a vertex shader to support advanced animation techniques. This can be done today for arrays that fit into the "uniform" space, but not for really large data structures. Need a way to declare these in the language, and need API support to initialize them. It can be argued that this functionality is mostly present through texture lookups. However, that doesn't seem like the proper abstraction for a large array in memory.

RESOLUTION: Defer this to a future release. Include a specification as part of this issue.

CLOSED January 24, 2003

100) *We have a name conflict with texture built-ins if in the future lod-bias forms are added to the vertex shader, or (more likely) absolute-lod forms are added to the fragment shader.*

RESOLUTION: Call bias forms the names we have now. Call absolute-lod forms the names we have now, suffixed with "Lod".

CLOSED January 24, 2003.

101) *Add initialization of uniforms. For example, uniform vec3 Color = vec3(1.0, 1.0, 0.0);*

DISCUSSION: Has been requested by ISVs and NVIDIA. Doesn't seem necessary, but if majority wants this we can add it. Also need to consider doing this for samplers (but not for arrays?). Note in only makes sense for uniforms that are not changing during rendering, const globals are available, and textures cannot be specified, so this may be of less utility than some think. Still it has value.

RESOLUTION: Do nothing. Save backward compatible enhancements for a future version of the language.

CLOSED: December 2003.

102) *Fix arrays (multi-dimensional arrays, static initialization of array members, variable sized arrays, non-sized array parameters).*

DISCUSSION: The function parameter declaration "vec4[ ]" does not say how big the array is, somewhat implying pass by reference. The calling conventions are pass by value. This syntax should be reserved for a possible future addition of passing an array by reference. To pass an array by value, one should say the size "vec4[5]". Unfortunately, this would prevent calling the same function with two differently sized arrays. But, then, perhaps, doing that should wait until there is a pass by reference mechanism for arrays. Static initialization could be done by using the constructor name of type followed by square brackets. E.g. "vec4[5] vArray = vec4[ ](v1, v2, v3, v4, v5);"

Once on this path, one could go further to making arrays first class objects. It could argue for declarations like "float[7] fArray", and allowing copying and comparing of arrays. This also implies eliminating auto-sizing of arrays, which is has other utility in making it easy to conserve interpolation resources. Going even further, multidimensional arrays could be added, and we'd have to consider arrays of arrays, typedefs, and true multidimensional arrays like "float[7,3] fMultiArray".

RESOLUTION: Be conservative with changes, but don't preclude backward compatibility of future additions of all this functionality. The minimal changes that allow future backward compatibility are: make function parameter declarations require the array size and make it a link error to have different shaders with different sizes for the same global array. This allows us to keep the existing auto-sizing of gl_TexCoord. Array sizes have to match for a function call to be selected based on parameter types, otherwise there would be a "no matching overloaded function" error. The rest is deferred to a future release.

CLOSED: December 2003.

103) Clarify specification on vector matrix constructors initialized with fewer than needed elements. Do you drop last row/column or initialize elements in order? Users would expect the former. Also, limit constructors and make them operate in a more sensible manner.

DISCUSSION: The spec says enough components have to be provided. There are missing interesting constructors for building matrices in other than column major order. There is also no upper bound today on how many parameters may be passed to a constructor in a valid program. We could also enumerate all the useful constructors, and say exactly what each one does. Also, there is some blurring between the use of a constructor as a type converter versus a type builder. This would be made more clear with a clean list of prototypes. The complete list was avoided earlier in favor of a simpler specification.

RESOLUTION: Limit the number of arguments to a constructor. The last useful argument can have more components than needed, and just be partially consumed, but arguments beyond that are disallowed. Also, prevent other constructions that are today useless, but in the future could be defined to be useful. These would be matrices constructed from other different sized matrices. E.g. mat4(mat3) or mat2(mat4). Actual support of new kinds of constructors is deferred to a future release.

CLOSED December 2003.

104) Should there be a fast atan() etc?

DISCUSSION: Two different directions to go. One is adding a performance vs. accuracy trade-off. The other is to add domain limited functions. Both could be supported, with new sets of names. Eg.

atanDom() would be a domain limited arc tangent.

atanFast() would be a lower precision, higher performance arc tangent.

RESOLUTION: Do it as an extension first.

CLOSED December 2003.

105) Change spec so that code can be generated by concatenation of all shader strings across shader objects.

DISCUSSION: There is some utility in just waiting to link time to do full compilation and link in one step. One way to do this is to concatenate all the shaders together (within vertex or within fragment, not across), and parse them. They still have to be parsed at compile time, to return errors, etc., but would be parsed a second time at link time.

RESOLUTION: Not to do this. It is easy to make a specification error in making this change.

CLOSED December 2003.

11 Acknowledgements

If I have seen further it is by standing on ye shoulders of Giants — Isaac Newton

The vast majority of the ideas and elements in the proposed language can be found in the computer science and computer graphics literature. Dave Baldwin of 3Dlabs wrote the white paper that formed the basis for the OpenGL Shading Language. His original credits included the following:

- AT&T for the C language,
- Pixar for RenderMan,
- UNC for Pixel Flow language and their programmable OpenGL interface,
- Stanford for their shading language work,
- SGI for OpenGL (obviously) and their shading language research,
- Colleagues at 3Dlabs for the frequent active discussions that helped to clarify many points

Dave Baldwin has continued to be involved in the design of the shading language as it has developed. Randi Rost and John Kessenich edited several public versions of the shading language white paper and created the first version of this specification document. Antonio Tejada of 3Dlabs wrote the first parser for the language to help resolve some of the fundamental language design issues.

Randi Rost, John Kessenich, Barthold Lichtenbelt, and Steve Koren of 3Dlabs formed the team that took over the design and implementation of the OpenGL Shading Language once the initial direction had been established. This group has been responsible for producing the publicly available specifications and source code for the OpenGL Shading Language, for producing the initial implementation of the language compiler and supporting OpenGL extensions, and for modifying the design of the language along the way.

Dave Baldwin, Dale Kirkland, Jeremy Morris, Phil Huxley, and Antonio Tejada of 3Dlabs have been involved in many of the OpenGL Shading Language discussions and have provided a wealth of good ideas and encouragement as we have moved forward. Other members of the 3Dlabs driver development teams in Egham, U.K., Huntsville, AL, and Austin, TX have contributed as well. The 3Dlabs executive staff should be commended for having the vision to move forward with the OpenGL Shading Language proposal and the courage to allocate resources to its development. Thanks to Osman Kent, Neil Trevett, Jerry Peterson, and John Schimpf in particular.

Numerous other people have been involved in the OpenGL Shading Language discussions. We would like to thank our colleagues and fellow ARB representatives at ATI, SGI, NVIDIA, Intel, Microsoft, Evans & Sutherland, IBM, Sun, Apple, Imagination Technologies, Dell, Compaq, and HP for contributing to discussions and for helping to move the process along. In particular, Bill Licea-Kane and Evan Hart of ATI Research have helped to improve the specification and the language itself through insightful review and studious comments.

A big thank you goes to the software developers who have taken the time to talk with us, send us email, or answer survey questions on OpenGL.org. Our ultimate aim is to provide you with the best possible API for doing graphics application development, and the time that you have spent telling us what you need has been invaluable. A few ISVs lobbied long and hard for certain features, and they were able to convince us to make some significant changes to the original OpenGL 2.0 proposal. Thanks, all you software developers, and keep telling us what you need!

Others that have provided useful comments or review on the shading language include Christian Laforte and Ian Ameline of Alias|Wavefront; Jonathan Putsman of Imagination Technologies; Darren Roberts and Slawek Kilanowski of LightWork Design; John Carmack of ID software; Tim Sweeney and Daniel Vogel of Epic Games; Bert Freudenberg of the University of Magdeburg; Folker Schamel of Spinor; Karel Zuiderveld and Steve Demlow of Vital Images; Christoph Poliwoda, Christof Reinhart, and Wolfgang Roemer of Volume Graphics; Christian Schormann of Pinnacle; Jake Kolb V of Whatif Productions; Mik Wells of Softimage; Delwyn Holroyd and Gerk Huisma of 5D Solutions; Kurt Akeley, Pat Hanrahan, and Bill Mark from Stanford University; and John Stauffer of Apple.

People that have helped shape the direction of shader objects and program objects include Christian Laforte of Alias|Wavefront; Pierre Tremblay of Discreet; Bimal Poddar of Intel; Jon Paul Schelter of Matrox; Jonathan Putsman of Imagination Technologies; Mik Wells of Softimage; Karel Zuiderveld and Steve Demlow of Vital Images; and Tim Sweeney and Daniel Vogel of Epic Games.

This specification document was formally submitted to the OpenGL Architecture Review Board's GL2 working group in June, 2002. That group was responsible for identifying issues, resolving issues, and finalizing the specification for approval by the OpenGL Architecture Review Board. The following GL2 working group members contributed to the finalization of this specification:. Dave Baldwin, John Kessenich, Steve Koren, Barthold Lichtenbelt, and Randi Rost from 3Dlabs, Inc.; Evan Hart, Bill Licea-Kane (chairman of the ARB-GL2 working group), and Victor Vedovato from ATI Research, Inc.; Dave Zenz of Dell; Brandon Fiflet and Kent Lin of Intel; Pat Brown, Matt Craighead, Cass Everett, Steve Glanville, Jayant Kolhe, and Nick Triantos from NVIDIA; Jon Leech from SGI; Folker Schamel of Spinor; Brian Paul from Tungsten Graphics; Eskil Steenburg from Quel solaar; Marc Olano from the University of Maryland, Baltimore County; Michael McCool from the University of Waterloo; and Matt Cruikshank, Steve Demlow, and Karel Zuiderveld from Vital Images.

Finally, a debt of gratitude is owed to the designers of the C programming language, the designers of RenderMan, and the designers of OpenGL, the three standards that have provided the strongest influence on our efforts. Hopefully, the OpenGL Shading Language will continue their traditions of success and excellence.

ACKNOWLEDGEMENTS