The OpenGL® Shading Language

Language Version: 1.20
Document Revision: 8
07-Sept-2006

John Kessenich

Version 1.1 Authors: John Kessenich, Dave Baldwin, Randi Rost

Table of Contents

1 Introduction

This document specifies version 1.20 of the OpenGL Shading Language. It requires __VERSION__ to be 120, and **#version** to accept 110 or 120.

1.1 Acknowledgments

This specification is based on the work of those who contributed to version 1.10 of the OpenGL Language Specification, the OpenGL ES 2.0 Language Specification, version 1.10, and the following contributors to this version:

Nick Burns
Chris Dodd
Michael Gold
Jeff Juliano
Jon Leech
Bill Licea-Kane
Barthold Lichtenbelt
Benjamin Lipchak
Ian Romanick
John Rosasco
Jeremy Sandmel
Robert Simpson
Eskil Steenberg

1.2 Changes

Changes from revision 7 of version 1.20

- Issue 13's resolution is brought up to date with the change agreed on in revision 7 regarding number of attribute slots being the number of columns in a matrix.
- Restated the arithmetic operator behavior in section 5.9 with sub-bullets instead of too long of a paragraph.

Changes from revision 6 of version 1.20

- The grammar is brought up to date:
    - method support for ".length()"
    - constructors simplified and recognized through type_specifier
    - array type syntax is supported "float[5]", and array initializers are added
- Fix statement about number of attribute slots for matrices. It erroneously said it used the maximum of the rows and columns, but it only uses the number of columns.

- Deleted the last paragraph of section 5.10, which redundantly or inconsistently re-stated section 5.9, and made sure all its valid contents were incorporated into the arithmetic-binary-operator bullet in section 5.9.
- Clarify that despite invariant only being for vertex outputs, the **invariant** declarations have to match between the vertex and fragment sides, for matching names.

Changes from revision 5 of version 1.20

- Removed notation showing additions and deletions to the specification, added table of contents, and cleaned up some cross-references and some resulting text-flow issues. No functional changes.

Changes from revision 4 of version 1.20

- Updated the grammar. (It still has to be validated.)
- Removed embedded structures to match ES, waiting for scoping operator to access embedded type.
- Constant expressions are computed in an invariant way.
- Use of **invariant** and **centroid** must match between vertex and fragment shaders.
- Made the distinction between a shader as a compilation unit and a shader as an executable.
- Clarified there is no line continuation character, and that comments don't delete new lines.
- Many changes to reduce differences when compared to the ES specification.

Changes from revision 3 of version 1.20

- Add the **invariant** keyword and its support.
- Allow unsized array constructors. (This still makes an explicitly sized array.)
- Require explicitly sized arrays for assignment and comparison.
- Allow sizing unsized array declaration through initializer.
- Different compilation units can be different language versions
- Add C++ style name hiding rules
- Reserve **lowp, mediump, highp,** and **precision.**

Changes from revision 1 of version 1.20

- Disallow other signatures/return-values of **main**.
- Clarify that ?: can have the same type 2nd and 3rd operands (e.g. conversion is not required).
- Say that point sprites have to be enabled to get defined values from gl_PointCoord
- Separate out and distinguish between storage and parameter qualifiers, making it easier to add the invariant qualifier.
- **#version 120** is required to use version 1.20
- mat2x3 means 2 columns, 3 rows
- matrix construction from matrix allowed

- added **transpose**()
- signature matching takes type conversions into account, ambiguity is an error

Changes from revision 60 of version 1.10

- Accept "f" as part of a floating-point constant. E.g. "float g = 3.5f".
- Automatically convert integer types to float types, as needed by context.
- Allow initializers on uniform declarations. The value is set at link time.
- Allow built-in function calls in const initializers.
- Support non-square matrices.
- Add **matrixProduct**() [now **outerProduct()**] for multiplying vectors to yield a matrix.
- Arrays become first class objects, with constructors, comparison, length(), etc.
- Add gl_PointCoord for fragment shaders to query a fragment's position within a point sprite.
- Support centroid interpolation on multi-sample varyings.

1.3 Overview

This document describes *The OpenGL Shading Language, version 1.20.*

Independent compilation units written in this language are called *shaders*. A *program* is a complete 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 used to manipulate and communicate with programs and shaders are defined in a separate specification.

1.4 Error Handling

Compilers, in general, accept programs that are ill-formed, due to the impossibility of detecting all ill-formed programs. 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.

1.5 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.

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.

Any OpenGL state used by the shader is automatically tracked and made available to shaders. 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 a shader.

2.1 Vertex Processor

The *vertex processor* is a programmable unit that operates on incoming vertices and their associated data. Compilation units written in the OpenGL Shading Language to run on this processor are called *vertex shaders*. When a complete set of vertex shaders are compiled and linked, they result in a *vertex shader executable* that runs on the vertex processor.

The vertex processor operates on one vertex at a time. It does not replace graphics operations that require knowledge of several vertices at a time. The vertex shaders running on the vertex processor must compute the homogeneous position of the incoming vertex.

2.2 Fragment Processor

The *fragment processor* is a programmable unit that operates on fragment values and their associated data. Compilation units written in the OpenGL Shading Language to run on this processor are called *fragment* shaders. When a complete set of fragment shaders are compiled and linked, they result in a *fragment shader executable* that runs on the fragment processor.

A fragment shader cannot change a fragment's x/y position. Access to neighboring fragments is not allowed. 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.

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. There is no line continuation character.

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 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 as part of the compilation process.

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 120.

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 as follows.

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 (lowest)	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 object's information log (see the API in external documentation for how to access a shader object'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 implementation 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 a version of the language, following the same convention as *__VERSION__* above. The directive "**#version** 120" is required in any shader that uses version 1.20 of the language. Any *number* representing a version of the language a compiler does not support will cause an error to be generated. Version 1.10 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 1.10. The behavior of shaders targeting version 1.10 will not be effected by any changes introduced in version 1.20. Different shaders (compilation units) that are linked together in the same program do not have to have the same version; they can be a mix of version 1.10 and version 1.20 shaders.

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 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, unless such use is 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. New-lines are not eliminated by comments.

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

centroid

break continue do for while

if else

in out inout

float int void bool true false

invariant

discard return

mat2 mat3 mat4

mat2x2 mat2x3 mat2x4

mat3x2 mat3x3 mat3x4

mat4x2 mat4x3 mat4x4

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

lowp mediump highp precision

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, with the exception that an integer value may appear where a floating-point type is expected, and be converted to a floating-point value. Exactly how and when this can occur is described in Section 4.1.10 "Implicit Conversions" and as referenced by other sections in this specification.

4.1 Basic Types

The OpenGL Shading Language supports the following basic data types.

Type	Meaning
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

Type	Meaning
mat4	a 4×4 floating-point matrix
mat2x2	same as a **mat2**
mat2x3	a floating-point matrix with 2 columns and 3 rows
mat2x4	a floating-point matrix with 2 columns and 4 rows
mat3x2	a floating-point matrix with 3 columns and 2 rows
mat3x3	same as a **mat3**
mat3x4	a floating-point matrix with 3 columns and 4 rows
mat4x2	a floating-point matrix with 4 columns and 2 rows
mat4x3	a floating-point matrix with 4 columns and 3 rows
mat4x4	same as a **mat4**
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. The keyword **void** cannot be used in any other declarations (except for empty formal parameter lists).

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 literal 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$_{opt}$ floating-suffix$_{opt}$*
> *digit-sequence exponent-part floating-suffix$_{opt}$*

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*

floating-suffix: one of
> **f F**

A decimal point (**.**) is not needed if the exponent part is present. No white space may appear anywhere within a floating-point constant. A leading unary minus sign (**-**) is interpreted as a unary operator and is not part of the floating-point constant

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 less;
```

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

4.1.6 Matrices

The OpenGL Shading Language has built-in types for 2×2, 2×3, 2×4, 3×2, 3×3, 3×4, 4×2, 4×3, and 4×4 matrices of floating-point numbers. The first number in the type is the number of columns, the second is the number of rows. Example matrix declarations:

```
mat2 mat2D;
mat3 optMatrix;
mat4 view, projection;
mat4x4 view;  // an alternate way of declaring a mat4
mat3x2 m;     // a matrix with 3 columns and 2 rows
```

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

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 **uniform** variables (see Section 4.3.5 “Uniform”). Except for array indexing, structure field selection, and parentheses, samplers are not allowed to be operands in expressions. Samplers cannot be treated as l-values; hence cannot be used as **out** or **inout** function parameters, nor can they be assigned into. As uniforms, they are initialized only with the OpenGL API; they cannot be declared with an initializer in a shader. 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;

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, types, and functions, 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 already defined (there are no forward references). 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 “Constant Expressions”). Each level of structure has its own name space for names given in member declarators; such names need only be unique within that name space.

Anonymous structures are not supported. Embedded structures are not supported.

```
struct S { float f; };

struct T {
      S;              // Error: anonymous structures disallowed
      struct { ... }; // Error: embedded structures disallowed
      S s;            // Okay: nested structures with name are allowed
};
```

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 “Constant Expressions”) greater than zero. If an array is indexed with an expression that is not an integral constant expression, or if an array is 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];
```

An array type can be formed by specifying a type followed by square brackets ([ ]) and including a size:

```
float[5]
```

This type can be used anywhere any other type can be used, including as the return value from a function

```
float[5] foo() { }
```

as a constructor of an array

```
float[5](3.4, 4.2, 5.0, 5.2, 1.1)
```

as an unnamed parameter

```
void foo(float[5])
```

and as an alternate way of declaring a variable or function parameter.

```
float[5] a;
```

It is an error to declare arrays of arrays:

```
float a[5][3];  // illegal
float[5] a[3];  // illegal
```

Arrays can have initializers formed from array constructors:

```
float a[5] = float[5](3.4, 4.2, 5.0, 5.2, 1.1);
float a[5] = float[](3.4, 4.2, 5.0, 5.2, 1.1);  // same thing
```

Unsized arrays can be explicitly sized by an initializer at declaration time:

```
float a[5];
...
float b[] = a;  // b is explicitly size 5
float b[5] = a; // means the same thing
```

However, implicitly sized arrays cannot be assigned to. Note, this is a rare case that initializers and assignments appear to have different semantics.

Arrays know the number of elements they contain. This can be obtained by using the length method:

```
a.length();  // returns 5 for the above declarations
```

The length method cannot be called on an array that has not been explicitly sized.

4.1.10 Implicit Conversions

In some situations, an expression and its type will be implicitly converted to a different type. The following table shows all allowed implicit conversions:

Type of expression	Can be implicitly converted to
int	**float**
ivec2	**vec2**
ivec3	**vec3**
ivec4	**vec4**

There are no implicit array or structure conversions. For example, an array of **int** cannot be implicitly converted to an array of **float**.

When an implicit conversion is done, it is not just a re-interpretation of the expression's value, but a conversion of that value to an equivalent value in the new type. For example, the integer value **5** will be converted to the floating-point value **5.0**.

The conversions in the table above are done only as indicated by other sections of this specification.

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.

Within a declaration, the scope of a name starts immediately after the initializer if present or immediately after the name being declared if not. Several examples:

```
int x = 1;
{
       int x = 2, y = x; // y is initialized to 2
}

struct S
{
       int x;
};

{
       S S = S(0,0); // 'S' is only visible as a struct and constructor
       S;            // 'S' is now visible as a variable
}

int x = x;           // Error if x has not been previously defined.
```

All variable names, structure type names, and function names in a given scope share the same name space. Function names can be redeclared in the same scope, with the same or different parameters, without error. An implicitly sized array can be re-declared in the same scope as an array of the same base type. Otherwise, within one compilation unit, a declared name cannot be redeclared in the same scope; doing so results in a redeclaration error. If a nested scope redeclares a name used in an outer scope, it hides all existing uses of that name. There is no way to access the hidden name or make it unhidden, without exiting the scope that hid it.

The built-in functions are scoped in a scope outside the global scope users declare global variables in. That is, a shader's global scope, available for user-defined functions and global variables, is nested inside the scope containing the built-in functions. When a function name is redeclared in a nested scope, it hides all functions declared with that name in the outer scope. Function declarations (prototypes) cannot occur inside of functions; they must be at global scope, or for the built-in functions, outside the global scope.

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 name space, 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 explicit size. An array implicitly sized in one shader can be explicitly sized by another shader. If no shader has an explicit size for the array, the largest implicit size is used. 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 Storage Qualifiers

Variable declarations may have a storage qualifier specified in front of the type. These are summarized as

Qualifier	Meaning
< 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 **centroid varying**	linkage between a vertex shader and a fragment shader for interpolated data

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

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

Function parameters can only use the **const** storage qualifier. Parameter qualifiers are discussed in more detail in Section 6.1.1 “Function Calling Conventions”.

Function return types and structure fields do not use storage qualifiers.

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

Initializers may only be used in declarations of globals with no storage qualifier, a **const** qualifier, or a **uniform** qualifier. Global variables without storage 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 Storage Qualifier

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 constant expressions, as defined in Section 4.3.3 "Constant Expressions."

4.3.3 Constant Expressions

A *constant expression* is one of

- a literal value (e.g., **5** or **true**)
- a global or local variable qualified as **const** (i.e. not including function parameters)
- an expression formed by an operator on operands that are all constant expressions, including getting an element or length of a constant array, or a field of a constant structure, or components of a constant vector.
- a constructor whose arguments are all constant expressions
- a built-in function call whose arguments are all constant expressions, with the exception of the texture lookup functions, the noise functions, and **ftransform**. The built-in functions **dFdx**, **dFdy**, and **fwidth** must return 0 when evaluated inside an initializer with an argument that is a constant expression.

Function calls to user-defined functions (non-built-in functions) cannot be used to form constant expressions.

An *integral constant expression* is a constant expression that evaluates to a scalar integer.

Constant expressions will be evaluated in an invariant way so as to create the same value in multiple shaders when the same constant expressions appear in those shaders. See section 4.6.1 "The Invariant Qualifier" for more details on how to create invariant expressions.

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 **float,** floating-point vectors, and matrices. 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 matrix attribute will use up multiple attribute locations. The number of locations used will equal the number of columns in the matrix.

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 externally either at link time or through the API. The link time initial value is either the value of the variable's initializer, if present, or 0 if no initializer is present. Sampler types cannot have initializers.

Example declarations are:

```
uniform vec4 lightPosition;
uniform vec3 color = vec3(0.7, 0.7, 0.2);  // value assigned at link time
```

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, the types and initializers of uniform variables with the same name must match across all shaders that are linked into a single executable. It is legal for some shaders to provide an initializer for a particular uniform variable, while another shader does not, but all provided initializers must be equal.

4.3.6 Varying

Varying variables provide the interface between the vertex shaders, the fragment shaders, and the fixed functionality between them. Vertex shaders 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 value is interpolated to the pixel's center, and the **centroid** qualifier, if present, is ignored. If multi-sampling, and **varying** is not qualified with **centroid**, then the value must be interpolated to the pixel's center, or anywhere within the pixel, or to one of the pixel's samples. If multi-sampling and **varying** is qualified with **centroid**, then the value must be interpolated to a point that lies in both the pixel and in the primitive being rendered, or to one of the pixel's samples that falls within the primitive. Due to the less regular location of centroids, their derivatives may be less accurate than non-centroid varying variables.

When using the **centroid** keyword, it must immediately precede the **varying** keyword.

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 and presence of the **centroid** and **invariant** qualifiers of varying variables with the same name declared in linked vertex and fragments shaders must match, otherwise the link command will fail. Only those varying variables used (i.e. read) in the fragment shader executable must be written to by the vertex shader executable; declaring superfluous varying variables in a vertex shader is permissible.

Varying variables are declared as in the following examples:

```
varying vec3 normal;
centroid varying vec2 TexCoord;
invariant centroid varying vec4 Color;
```

The **varying** qualifier can be used only with **float**, floating-point vectors, matrices, or arrays of these. Structures cannot be **varying**.

If no vertex shader executable 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 executable. Similarly, if no fragment shader executable is active, the vertex shader executable 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.

4.4 Parameter Qualifiers

Parameters can have these qualifiers.

Qualifier	**Meaning**
< none: default >	same is **in**
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

Parameter qualifiers are discussed in more detail in Section 6.1.1 "Function Calling Conventions" .

4.5 Precision and Precision Qualifiers

Section number reserved for precision qualifiers. (Reserved for future use.)

4.6 Variance and the Invariant Qualifier

In this section, *variance* refers to the possibility of getting different values from the same expression in different programs. For example, say two vertex shaders, in different programs, each set **gl_Position** with the same expression in both shaders, and the input values into that expression are the same when both shaders run. It is possible, due to independent compilation of the two shaders, that the values assigned to **gl_Position** are not exactly the same when the two shaders run. In this example, this can cause problems with alignment of geometry in a multi-pass algorithm.

In general, such variance between shaders is allowed. When such variance does not exist for a particular output variable, that variable is said to be *invariant.*

4.6.1 The Invariant Qualifier

To ensure that a particular output variable is invariant, it is necessary to use the **invariant** qualifier. It can either be used to qualify a previously declared variable as being invariant

```
invariant gl_Position;   // make existing gl_Position be invariant

varying vec3 Color;
invariant Color;         // make existing Color be invariant
```

or as part of a declaration when a variable is declared

```
invariant varying vec3 Color;
```

The invariant qualifier must appear before any storage qualifiers (**varying**) when combined with a declaration. Only variables output from a vertex shader can be candidates for invariance. This includes user-defined varying variables, the built-in vertex-side varying variables, and the special vertex variables *gl_Position* and *gl_PointSize*. For varying variables leaving a vertex shader and coming into a fragment shader with the same name, the **invariant** keyword has to be used in both the vertex and fragment shaders. The **invariant** keyword can be followed by a comma separated list of previously declared identifiers. All uses of **invariant** must be at the global scope, and before any use of the variables being declared as invariant.

To guarantee invariance of a particular output variable across two programs, the following must also be true:

- The output variable is declared as invariant in both programs.
- The same values must be input to all shader input variables consumed by expressions and flow control contributing to the value assigned to the output variable.
- The texture formats, texel values, and texture filtering are set the same way for any texture function calls contributing to the value of the output variable.
- All input values are all operated on in the same way. All operations in the consuming expressions and any intermediate expressions must be the same, with the same order of operands and same associativity, to give the same order of evaluation. Intermediate variables and functions must be declared as the same type with the same explicit or implicit precision qualifiers. Any control flow affecting the output value must be the same, and any expressions consumed to determine this control flow must also follow these invariance rules.
- All the data flow and control flow leading to setting the invariant output variable reside in a single compilation unit.

Essentially, all the data flow and control flow leading to an invariant output must match.

Initially, by default, all output variables are allowed to be variant. To force all output variables to be invariant, use the pragma

```
#pragma STDGL invariant(all)
```

before all declarations in a shader. If this pragma is used after the declaration of any variables or functions, then the set of outputs that behave as invariant is undefined. It is an error to use this pragma in a fragment shader.

Generally, invariance is ensured at the cost of flexibility in optimization, so performance can be degraded by use of invariance. Hence, use of this pragma is intended as a debug aid, to avoid individually declaring all output variables as invariant.

4.6.2 Invariance of Constant Expressions

Invariance must be guaranteed for constant expressions. A particular constant expression must evaluate to the same result if it appears again in the same shader or a different shader. This includes the same expression appearing in both a vertex and fragment shader or the same expression appearing in different vertex or fragment shaders.

Constant expressions must evaluate to the same result when operated on as already described above for invariant variables.

4.7 Order of Qualification

When multiple qualifications are present, they must follow a strict order. This order is as follows.

invariant-qualifier storage-qualifier

storage-qualifier parameter-qualifier

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 or method 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 Operations

These are now described in Section 5.7 “Structure and Array Operations”.

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 type, and the call makes an object of that type. Constructors are used the same way in both initializers and expressions. (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.

In general, constructors are not built-in functions with predetermined prototypes. For arrays and structures, there must be exactly one argument in the constructor for each element or field. For the other types, the arguments must provide a sufficient number of components to perform the initialization, and it is an error to include so many arguments 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 a vector is constructed from multiple scalars, vectors, or matrices, or a mixture of these, the vectors' components will be constructed in order from the components of the arguments. The arguments will be consumed left to right, and each argument will have all it's components consumed, in order, before any components from the next argument are consumed. Similarly for constructing a matrix from multiple scalars or vectors, or a mixture of these. Matrix components will be constructed and consumed in column major order. In these cases, there must be enough components provided in the arguments to provide an initializer for every component in the constructed value. It is an error to provide extra arguments beyond this last used argument.

If a matrix is constructed from a matrix, then each component (column *i,* row *j*) in the result that has a corresponding component (column *i,* row *j*) in the argument will be initialized from there. All other components will be initialized to the identity matrix. If a matrix argument is given to a matrix constructor, it is an error to have any other arguments.

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 with the float
vec4(ivec4)  // makes a vec4 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)   // uses 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:

```
vec4 color = 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 scalars, the components are assigned to the matrix elements in column-major order.

```
mat2(vec2, vec2);             // one column per argument
mat3(vec3, vec3, vec3);       // one column per argument
mat4(vec4, vec4, vec4, vec4); // one column per argument
mat3x2(vec2, vec2, vec2);     // one column per argument

mat2(float, float,      // first column
     float, float);     // second column

mat3(float, float, float,     // first column
     float, float, float,     // second column
     float, float, float);    // third column

mat4(float, float, float, float,  // first column
     float, float, float, float,  // second column
     float, float, float, float,  // third column
     float, float, float, float); // fourth column

mat2x3(vec2, float,      // first column
       vec2, float);     // second column
```

A wide range of other possibilities exist, to construct a matrix from vectors and scalars, as long as enough components are present to initialize the matrix. To construct a matrix from a matrix:

```
mat3x3(mat4x4);  // takes the upper-left 3x3 of the mat4x4
mat2x3(mat4x2);  // takes the upper-left 2x2 of the mat4x4, last row is 0,0
mat4x4(mat3x3);  // puts the mat3x3 in the upper-left, sets the lower right
                 //    component to 1, and the rest to 0
```

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 will be used to set the structure's fields, in order, using one argument per field. Each argument must be the same type as the field it sets, or be a type that can be converted to the field's type according to Section 4.1.10 “Implicit Conversions.”

Structure constructors can be used as initializers or in expressions.

5.4.4 Array Constructors

Array types can also be used as constructor names, which can then be used in expressions or initializers. For example,

```
const float c[3] = float[3](5.0, 7.2, 1.1);
const float d[3] = float[](5.0, 7.2, 1.1);

float g;
...
float a[5] = float[5](g, 1, g, 2.3, g);
float b[3];

b = float[3](g, g + 1.0, g + 2.0);
```

There must be exactly the same number of arguments as the size of the array being constructed. If no size is present in the constructor, then the array is explicitly sized to the number of arguments provided. The arguments are assigned in order, starting at element 0, to the elements of the constructed array. Each argument must be the same type as the element type of the array, or be a type that can be converted to the element type of the array according to Section 4.1.10 "Implicit Conversions."

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 it 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. Reading from or writing to a vector using a constant integral expression with a value that is negative or greater than or equal to the size of the vector is illegal. When indexing with non-constant expressions, behavior is undefined if the index is negative, or 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 with a non-constant expression. It is an error to access a matrix with a constant expression that is outside the bounds of the matrix.

5.7 Structure and Array Operations

The fields of a structure and the **length** method of an array are selected using the period (**.**).

In total, only the following operators are allowed to operate on arrays and structures as whole entities:

field or method selector	**.**
equality	== **!=**
assignment	=
indexing (arrays only)	[ ]

The equality operators and assignment operator are only allowed if the two operands are same size and type. Structure types must be of the same declared structure. Both array operands must be explicitly sized. When using the equality operators, two structures are equal if and only if all the fields are component-wise equal, and two arrays are equal if and only if all the elements are element-wise equal.

Array elements are accessed using the array subscript operator (**[ ]**). An example of accessing an array element is

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

Array indices start at zero. Array 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.

Arrays can also accessed with the method operator (**.**) and the **length** method to query the size of the array:

```
lightIntensity.length()    // return the size of the array
```

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*. The *expression* and *lvalue* must have the same type, or the expression must have a type in the table in Section 4.1.10 “Implicit Conversions” that converts to the type of *lvalue*, in which case an implicit conversion will be done on the expression before the assignment is done. Any other desired type-conversions must be specified explicitly via a constructor. L-values must be writable. Variables that are built-in types, entire structures or arrays, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[ ]**) are all l-values. Other binary or unary expressions, 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 (^=) 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.
- An array name with the length method applied.
- Subscripted array names.
- Function calls that return values.
- Component field selectors and array subscript results.
- Parenthesized expression. Any expression can be parenthesized. 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 (/) operate on integer and floating-point scalars, vectors, and matrices. If one operand is floating-point based and the other is not, then the conversions from Section 4.1.10 “Implicit Conversions” are applied to the non-floating-point-based operand. All arithmetic binary operators result in the same fundamental type (integer or floating-point) as the operands they operate on, after operand type conversion. After conversion, the following cases are valid
  - The two operands are scalars. In this case the operation is applied, resulting in a scalar.
  - One operand is a scalar, and the other is a vector or matrix. In this case, the scalar operation is applied independently to each component of the vector or matrix, resulting in the same size vector or matrix.
  - The two operands are vectors of the same size. In this case, the operation is done component-wise resulting in the same size vector.
  - The operator is add (+), subtract (**-**), or divide (/), and the operands are matrices with the same number of rows and the same number of columns. In this case, the operation is done component-wise resulting in the same size matrix.
  - The operator is multiply (*****), where both operands are matrices or one operand is a vector and the other a matrix. A right vector operand is treated as a column vector and a left vector operand as a row vector. In all these cases, it is required that the number of columns of the left operand is equal to the number of rows of the right operand. Then, the multiply (*****) operation does a linear algebraic multiply, yielding an object that has the same number of rows as the left operand and the same number of columns as the right operand. Section 5.10 “Vector and Matrix Operations” explains in more detail how vectors and matrices are operated on.

  All other cases are illegal.

  Dividing by zero does not cause an exception but does result in an unspecified value. Use the built-in functions **dot, cross, matrixCompMult,** and **outerProduct**, to get, respectively, vector dot product, vector cross product, matrix component-wise multiplication, and the matrix product of a column vector times a row vector.
- The operator modulus (**%**) is reserved for future use.
- The arithmetic unary operators negate (-), post- and pre-increment and decrement (**--** and ++) operate on integer or floating-point values (including vectors and matrices). All unary operators work component-wise on their operands. 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. Either the operands' types must match, or the conversions from Section 4.1.10 "Implicit Conversions" will be applied to the integer operand, after which the types must match. To do component-wise relational comparisons on vectors, use the built-in functions **lessThan, lessThanEqual, greaterThan,** and **greaterThanEqual.**
- The equality operators **equal (**==), and not equal (**!**=) operate on all types. They result in a scalar Boolean. If the operand types do not match, then there must be a conversion from Section 4.1.10 "Implicit Conversions" applied to one operand that can make them match, in which case this conversion is done. For vectors, matrices, structures, and arrays, all components, fields, or elements of one operand must equal the corresponding components, fields, or elements in the other operand for the operands to be considered equal. To get a vector of component-wise equality results for vectors, use the built-in functions **equal** and **notEqual**.
- The logical binary operators and (**&&**), or (| |), and exclusive or (^^) 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 can be any type, as long their types match, or there is a conversion in Section 4.1.10 "Implicit Conversions" that can be applied to one of the expressions to make their types match. This resulting matching type is the type of the entire expression.
- 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."

5.10 Vector and Matrix Operations

With a few exceptions, operations are component-wise. Usually, 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.

```
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

```
mat3 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 other sizes of vectors and matrices.

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 a parameter qualifier and/or **const**.

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

Arrays are allowed as arguments and as the return type. In both cases, the array must be explicitly sized. An array is passed or returned by using just its name, without brackets, and the size of the array must match the size specified in the function's 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 (or definitions) 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. No qualifiers are included when checking if types match, function signature matching is based on parameter type only. Overloading is used heavily in the built-in functions. When function calls are resolved, an exact type match for a function's formal parameter types (signature) is sought. This includes exact match of array size as well. If an exact match is found, the other signatures are ignored, and the exact match is used. Otherwise, if no exact match is found, then the implicit conversions in Section 4.1.10 "Implicit Conversions" will be applied to the calling arguments if this can make their types match a signature. In this case, it is a semantic error if there are multiple ways to apply these conversions to the actual arguments of a call such that the call can be made to match multiple signatures.

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 executable. A shader need not contain a function named *main*, but one shader in a set of shaders linked together to form a single shader executable 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.

It is an error to declare or define a function **main** with any other signature or return type.

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, array 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.

Recursion is not allowed, not even statically. Static recursion is present if the static function call graph of the program contains cycles.

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 executable must write a value into this variable. It can be written at any time during shader execution. It may also be read back by a vertex 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. Its value is undefined if the vertex shader executable 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 executable 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 to more than once. 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 to 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 the fragment shader executable 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 no shader writes *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 the set of linked fragment shaders statically contain 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 executable 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. Multiple shaders linked together must also consistently write just one of these variables.

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. If multi-sampling, this value can be for any location within the pixel, or one of the fragment samples. The use of **centroid** varyings does not further restrict this value to be inside the current primitive. 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 no 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.

Fragment shaders have 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 a 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 storage 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_MaxTextureUnits];
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 executable 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 language. 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;
varying vec2 gl_PointCoord;
```

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.

The values in *gl_PointCoord* are two-dimensional coordinates indicating where within a point primitive the current fragment is located, when point sprites are enabled. They range from 0.0 to 1.0 across the point. If the current primitive is not a point, or if point sprites are not enabled, then the values read from gl_PointCoord are undefined.

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. Because built-in functions are in a more outer scope than user built-in functions, doing this will hide all built-in functions with the same name as the re-declared function.

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 any matrix basic type..

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, i.e. $\frac{\pi}{180} \cdot degrees$
genType **degrees** (genType *radians*)	Converts *radians* to degrees, i.e. $\frac{180}{\pi} \cdot radians$
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 $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ Results are undefined if $\lvert x\rvert > 1$.
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 $\lvert x\rvert > 1$.
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 $[-\pi, \pi]$. 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 $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

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 $\sqrt{x}$. Results are undefined if $x < 0$.
genType **inversesqrt** (genType *x*)	Returns $\frac{1}{\sqrt{x}}$. 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 $x < 0$

Syntax	**Description**
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*) Results are undefined if *minVal* > *maxVal.*
genType **mix** (genType *x*, genType *y*, genType *a*) genType **mix** (genType *x*, genType *y*, float *a*)	Returns the linear blend of *x* and *y,* i.e. $x \cdot (1 - a) + y \cdot a$
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);
	
	Results are undefined if *edge0* >= *edge1.*

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., $\sqrt{x[0]^2+x[1]^2+...}$
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., $x[0]\cdot y[0]+x[1]\cdot y[1]+...$
vec3 **cross** (vec3 *x*, vec3 *y*)	Returns the cross product of x and y, i.e. $\begin{bmatrix} x[1]\cdot y[2]-y[1]\cdot x[2] \\ x[2]\cdot y[0]-y[2]\cdot x[0] \\ x[0]\cdot y[1]-y[0]\cdot x[1] \end{bmatrix}$
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.*

Syntax	Description
genType **reflect** (genType *I*, genType *N*)	For the incident vector *I* and surface orientation *N*, returns the reflection direction: *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 result is computed by
	
	k = 1.0 - *eta* * *eta* * (1.0 - **dot**(*N*, *I*) * **dot**(*N*, *I*))
	if (k < 0.0)
	    return genType(0.0)
	else
	    return *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 (*****).
mat2 **outerProduct**(vec2 *c*, vec2 *r*) mat3 **outerProduct**(vec3 *c*, vec3 *r*) mat4 **outerProduct**(vec4 *c*, vec4 *r*) mat2x3 **outerProduct**(vec3 *c*, vec2 *r*) mat3x2 **outerProduct**(vec2 *c*, vec3 *r*) mat2x4 **outerProduct**(vec4 *c*, vec2 *r*) mat4x2 **outerProduct**(vec2 *c*, vec4 *r*) mat3x4 **outerProduct**(vec4 *c*, vec3 *r*) mat4x3 **outerProduct**(vec3 *c*, vec4 *r*)	Treats the first parameter *c* as a column vector (matrix with one column) and the second parameter *r* as a row vector (matrix with one row) and does a linear algebraic matrix multiply *c* * *r*, yielding a matrix whose number of rows is the number of components in *c* and whose number of columns is the number of components in *r*.
mat2 **transpose**(mat2 *m*) mat3 **transpose**(mat3 *m*) mat4 **transpose**(mat4 *m*) mat2x3 **transpose**(mat3x2 *m*) mat3x2 **transpose**(mat2x3 *m*) mat2x4 **transpose**(mat4x2 *m*) mat4x2 **transpose**(mat2x4 *m*) mat3x4 **transpose**(mat4x3 *m*) mat4x3 **transpose**(mat3x4 *m*)	Returns a matrix that is the transpose of *m*. The input matrix *m* is not modified.

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*.

Syntax	Description
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.

Syntax	**Description**
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 fragment shaders.

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:

$$F(x+dx)-F(x) \sim dFdx(x)\cdot dx \qquad \text{1a}$$

$$dFdx(x) \sim \frac{F(x+dx)-F(x)}{dx} \qquad \text{1b}$$

Backward differencing:

$$F(x-dx)-F(x) \sim -dFdx(x)\cdot dx \qquad \text{2a}$$

$$dFdx(x) \sim \frac{F(x)-F(x-dx)}{dx} \qquad \text{2b}$$

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.
6. Derivatives of constant arguments should be 0.

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 trade off.

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.: **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
CENTROID
MAT2X2 MAT2X3 MAT2X4
MAT3X2 MAT3X3 MAT3X4
MAT4X2 MAT4X3 MAT4X4
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

INVARIANT
```

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_or_method

function_call_or_method:
    function_call_generic
    postfix_expression DOT 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

// Grammar Note: Constructors look like functions, but lexical analysis recognized most of them as
// keywords. They are now recognized through "type_specifier".

function_identifier:
    type_specifier
    IDENTIFIER
    FIELD_SELECTION

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 assignment_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

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 LEFT_BRACKET
RIGHT_BRACKET EQUAL initializer
init_declarator_list COMMA IDENTIFIER LEFT_BRACKET constant_expression
RIGHT_BRACKET EQUAL initializer
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 LEFT_BRACKET RIGHT_BRACKET EQUAL initializer
fully_specified_type IDENTIFIER LEFT_BRACKET constant_expression
RIGHT_BRACKET EQUAL initializer
fully_specified_type IDENTIFIER EQUAL initializer
INVARIANT IDENTIFIER   // Vertex only.

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

fully_specified_type:

type_specifier
type_qualifier type_specifier

type_qualifier:
CONST
ATTRIBUTE   // Vertex only.
VARYING
CENTROID VARYING
INVARIANT VARYING
INVARIANT CENTROID VARYING
UNIFORM

type_specifier:
type_specifier_nonarray
type_specifier_nonarray LEFT_BRACKET constant_expression RIGHT_BRACKET

type_specifier_nonarray:
VOID
FLOAT
INT
BOOL
VEC2
VEC3
VEC4
BVEC2
BVEC3
BVEC4
IVEC2
IVEC3
IVEC4
MAT2
MAT3
MAT4
MAT2X2
MAT2X3
MAT2X4
MAT3X2
MAT3X3

MAT3X4
MAT4X2
MAT4X3
MAT4X4
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. Is it an error to have ambiguous function parameter matching like the following?

```
void foo(int);
void foo(float);
foo(3);
```

RESOLUTION: No. If an exact match is available, without conversion, there is no error for ambiguity.

2. Is it an error to have ambiguous function parameter matching like the following?

```
void foo(float, int)
void foo(int, float)
foo(2, 3);
```

RESOLUTION: Yes. This should raise an ambiguous call error. Anytime there are two ways to match through conversion, and there is no exact match without conversion, an error should be raised.

3. What should happen if the types of the second and third operands of the ternary (?:) operator don't match? Should they be converted to match each other, or should the conversion be delayed to match how the whole ternary expression is used (harder)?

RESOLUTION: They must be converted to match each other, so that the expression's type is determined by the expression.

4. We want **dFdx**, **dFdy**, and **fwidth** to return 0 if their argument is a constant expression for the purpose of using them in constant initializers. The spec currently theoretically allows non-zero for these derivatives.

ALTERNATIVE RESOLUTION: Say they must return 0, on the theory that no one would ever have implemented non-0, or if they did, the shaders should still work under the new rule.

ALTERNATIVE RESOLUTION: Disallow these in initializers.

RESOLUTION: Say they only have to return 0 when used in initializers.

5. Should we add the obvious missing matrix constructors?

ALTERNATIVE RESOLUTION: For remaining Cm, leave undefined.

ALTERNATIVE RESOLUTION: Require P >= N and Q >= M. That is, you can only go to smaller matrices.

RESOLUTION: For

matPxQ m;
matNxM Cm = matNxM(m)

for each 2D index [i][j] that exists in both m and Cm, set Cm[i][j] to m[i][j]. All remaining Cm[i][j] are set to the identity matrix.

5A. Should we add matrix constructors that allow row-wise construction?

RESOLUTION: Defer.

6. Should we require array operands to be explicitly sized before being used with an == or != operators?

RESOLUTION: Yes.

6A. Should we require array operands to be explicitly sized before being used with assignment?

RESOLUTION: Yes.

7. Can an unsized constructor be used? It is clear what is meant by the number of arguments listed. There are two paths to take here, that can also effect issue 8. This is, does "float[ ]" mean an implicitly sized array, or do we want to reserve that to mean a reference for a future release? If implicitly size array, it fits in neatly that "float[ ](1, 4)" is an array explicitly sized to 2, just like "float a[ ] = float[2](1, 4)" already is. We could even allow "float a[ ] = float[ ](1, 4);".

RESOLUTION: Yes, unsized constructors can be used, as can unsized declarations that are sized through initializer. We should come up with a different syntax for references in the future that is orthogonal to type (square brackets would have been an array specific reference syntax). Note that this is not another way of creating implicitly sized arrays, as the array immediately ends up with an explicit size that cannot be changed.

Make clear the new difference between initializer and assignment for unsized arrays, because we have now said the initializer can initialize what syntactically looks like an unsized array, whereas assignment cannot.

8. Can a function parameter be an unsized array? Functions can be cloned internally, and/or arrays passed by reference internally (still satisfying non-aliasing and pass by copy semantics externally). This internal function cloning will come up again if we discuss pass by reference semantics, and discuss needing different function bodies to pass in, say, uniform arrays versus of global arrays, because object has to be different.

RESOLUTION: Defer. Requires compile at link-time behavior. Possibly we are on the path of a single function body accepting arrays of different types and size, and that empty square brackets (unsized array) means accepting multiple sizes, versus meaning accepting a reference to an array.

9. Should we have matrix aliases for square matrices, e.g. **__mat2x2** is the same as **mat2**?

RESOLUTION: Yes.

10. Should we have matrix aliases for vectors, e.g. **__mat1x2** is the same as **vec2**? We either need this, or we need a built-in function for multiplying a row vector times a column vector, and returns a matrix.

RESOLUTION: No, don't alias vectors. It's not a true alias, because vectors are ambiguous regarding being row or column vectors. Add the built-in function "matN **outerProduct**(vecN, vecN)" instead (see issue 12).

11. Is the first number in a non-square matrix type the number of rows or number of columns. Number of rows is most intuitive. Number of columns is more consistent with existing order of indexing.

ALTERNATE RESOLUTION: First number is number of rows. We want to move this direction anyway... that the major-ness is independent of the abstract layout. We may be adding a construct to request row-major order, which says what the indexing means, but we can keep the type declaration the same either way by saying first number is number of rows.

RESOLUTION: Never add a mode to request row-based operations, and never move toward the language being more row-based. The first number means the number of columns.

11A. Should we add a method for selecting a row out of a matrix, e.g. m.row(i)?

RESOLUTION: Defer this.

12. What should the real spelling be for the **matrixProduct** built-in? (See issue 10.)

RESOLUTION: **outerProduct**

13. Is the number of attribute locations needed to store a matrix the number of rows, the number of columns, or the max of these?

ALTERNATE RESOLUTION: The max, to allow implementations the choice to move data most efficiently.

RESOLUTION: The number of columns; because of the current state of vertex arrays, the mapping of columns to slots is still exposed.

14. How strongly do we want to define where the centroid is located? “Near” seems a bit vague.

RESOLUTION: It at least has to be in the intersection of the primitive and the pixel, or one of the fragment samples that falls within the primitive. Preferably, it is the centroid of the intersection. The centroid of the samples lying within this intersection is also a good choice. The location of samples within the intersection that are near these centroids are also good choices.

15. Can we put in a static recursion restriction, like we did in ES? In the desktop spec, we just disallowed dynamic recursion, but I think it is better to disallow static recursion too. If no one has support static recursion so far, we could do this.

RESOLUTION: Disallow static recursion, like ES did.

16. Do we need to add **transpose**? Should it be an in-place transpose method, or a method that returns a functional transpose, or a built-in function that returns a functional transpose?

RESOLUTION: Add built-in functions

```
mat2 transpose(in mat2)
mat3 transpose(in mat3)
mat4 transpose(in mat4)

mat2x3 transpose(in mat3x2)
mat3x2 transpose(in mat2x3)

mat2x4 transpose(in mat4x2)
mat4x2 transpose(in mat2x4)

mat3x4 transpose(in mat4x3)
mat4x3 transpose(in mat3x4)
```

and reserve a .**transpose**() method for possible future use.

17. Can the application query the initializer value a uniform is set to at link time?

RESOLUTION: Yes, using the existing entry points for uniform query, so no action need be taken.

18. Do centroid varyings have derivatives?

ALTERNATE RESOLUTION: Derivatives of centroid varyings are undefined.

RESOLUTION: Say they are defined but less accurate. Bill to send out an update of condition 5 in the list.

ALTERNATIVE RESOLUTION: Derivatives are just as accurate as before, likely implemented by passing a separate non-centroid varying for use in derivatives. Could get difficult for complex expressions in the argument.

19. What ES clarifications to we want to bring across?

19A. No other mains are allowed besides the 'void main()' standard entry point.

RESOLUTION: Yes, clarify this.

19B. How undefined is reading and writing bad indexes into arrays? ES said it could cause system instability. We just say behavior is undefined.

RESOLUTION: Leave spec. as is, saying it is undefined.

19C. Better organize kinds of qualifiers.

– uniform, varying, attribute, and const became 'storage qualifiers'
– in, out, inout became 'parameter qualifiers'
– highp, mediump, lowp are 'precision qualifiers'
– invariant is an 'invariance qualifier'

Where, we would leave out the last two, but make the specs match for the first two, and make it easy to put in the others later.

RESOLUTION: Yes.

20. Should we require shaders to contain #version 120 to use 1.20 specific features? This potentially reduces portability, because codes may use it on shaders that didn't really need 120 features. On the other hand, it opens up the door for small non-backward-compatible changes that would be generally beneficial, like not having "__" in front of new keywords.

RESOLUTION: Yes. Require #version 120 to enable issue 20A.

20A. If we require #version 120, then can we drop "__" on new keywords?

RESOLUTION: Yes. New keywords for version 120 don't need the "__".

20B. Does #version need to match across all compilation units within a program object? Seems like an obvious choice, but it would be nice if libraries (or just existing code that doesn't need to be updated) don't have to be upgraded just because one compilation unit wanted a new feature.

RESOLUTION: No. Looking at the features being added, it appears there is no real need to require the same version in each compilation unit. Most are syntax, conversions, or otherwise shader local. The remaining ones are

- array objects: these are still compile time size known, so cross-unit sharing works as before
- keywords without "__": the 120 shader can't use them as names, so sharing them is not possible
- uniform initializers: works, we allow some modules to not have the initializer
- centroid: if we require the definitions (vert vs. frag) to match, then user has to go to 120 for both, otherwise, they could use 110 frag and 120 vert.

21. Do we need a centroid version of gl_FragCoord?

RESOLUTION: No. At least be clear that the existing gl_FragCoord is not centroid. Adding a new centroid frag-coord makes sense, but so far seems of marginal use, so we don't yet have to do this.

22. Does declaring a variable hide functions of the same name? The 1.10 spec. specifically says this does not happen.

ALTERNATE RESOLUTION: No. That would be a backward incompatible change that might break some shaders. E.g., code like

```
float sin = sin(expr);
float r = sin * sin;
```

is useful, and might have been written.

RESOLUTION: Yes. Using #version 120 allows us to break compatibility in this subtle way. Code can still declare a variable locally and use it, even though the name matches a function name. It just cannot then proceed to call the function of the same name in the same scope. It would be a redeclaration error if both function and variable are declared in the same scope.

22A. Does declaring a variable hide a struct of the same name in the same scope, or is it an error? That is, what should the behavior of the following be?

```
{
    struct S { ... };
    S S;
    S ...  // S is not the type 'S'.  The type is hidden by the variable 'S'
}
```

RESOLUTION: This is an error. C++ allows it, but I'm not sure there is value in us doing this. Types and variables share the name space in a scope, and this is a redefinition of a name in that name space. The reference implementation from 3Dlabs disallowed it.

23. Does declaring a structure type name hide both variables and all functions of the same name? (Redeclaring the same variable name in the same scope is of course a redeclaration error.) This was the intent of the 1.10 specification.

RESOLUTION: Yes. This is consistent with the existing spec., avoids confusion between the new constructor and existing functions of the same name, and reserves a name space for alternate constructor signatures in the future.

24. Does declaring a function hide a structure type name and it's constructor (with the same name), both in nested scopes and in the same scope? This was the intent of the 1.10 specification.

RESOLUTION: Disallow local function declarations. At the global scope, it is a redefinition error to have both a function and structure of the same name.

ALTERNATE RESOLUTION: Yes. Hiding the constructor has to be done. It would not make sense to hide the constructor and not hide the type, so both have to be hidden.

25. Does declaring a function, in a nested scope, un-hide a function definition that had been hidden by a structure type declaration in an intermediate scope? This was not addressed by the 1.10 specification, but is consistent with it's nested scoping rules.

RESOLUTION: Disallow local function declarations.

ALTERNATE RESOLUTION: Yes. There is almost no other valid interpretation of doing this, and what it is useful for doing. It's just a case of getting to do what you want in a nested scope, combined with the single global space for all function definitions.

26. Does declaring a function un-hide a function definition that had been hidden by a structure type declaration in the same scope? This was arguably not addressed by the 1.10 specification. C++ says this is an error.

RESOLUTION: Disallow local function declarations.

ALTERNATE RESOLUTION: No. Make declaring a function with the same name as a structure type in the same scope a redeclaration error. This is an obscure thing to be doing. It has a small chance of breaking an existing shader, e.g.

```
void foo(int) { ... }
void main()
{
    struct foo { float f; };  // legal
    void foo(int);  // suggested to be illegal; this scope has a foo()
}
```

27. Does declaring a function hide a variable with the same name? This would be inconsistent with the 1.10 specification.

RESOLUTION: At global scope only, this is a redeclaration error. For local scope, disallow local function declarations. There are no built-in variables that can be redeclared as a function name by the user, so this becomes a non-issue.

ALTERNATE RESOLUTION: No. It is really asymmetrical to have variables not hide functions, but have functions hide variables. It is also backward incompatible, though probably rarely done in real shaders.

ALTERNATE RESOLUTION: Yes. If we still allow local declarations, and if we change things so that variables hide functions, to keep everything symmetric. It would be a redeclaration error in the same scope.

28. Does a variable declaration un-hide a function that had been hidden by a structure type that the variable declaration hides? E.g.,

```
void foo(int) { ... }
void main()
{
    struct foo { float f; };  // legal, hides 'void foo(int)'
    {
        bool foo;  // legal, hides 'struct foo'
        foo(2);    // ?? is 'void foo(int)' visible again?
    }
}
```

RESOLUTION: No.

22-28. Summary of suggested resolutions for issues 22 through 28:

– local function declarations are disallowed
– a declaration of a variable or a structure type is a re-declaration error if there is already a function, a variable, or structure type of the same name in the same scope
– a declaration of a function is a re-declaration error if there is already a variable or a structure type of the same name in the same scope
– functions can be declared more than once in the same scope
– a declaration of a variable or a structure type hides all variables, functions, or structure types (and their constructors) of the same name from all outer scopes
– hidden functions stay hidden

Summary of what is backward incompatible from that list:

- cannot have local function declarations
- cannot use a function and a variable of the same name in the same scope

The compiler will give errors for these cases, so that nothing silently changes behavior.

29. Should we add entry points to encapsulate loading a matrix attribute, so the system can truly hide internal row-major vs. column-major memory ordering?

RESOLUTION: No. We would like to, but it would require vertex attribute array support, which are so far only four components wide, so do not yet have the ability to represent a whole matrix. Note this would go into the API extension accompanying this language specification.

30. Is there an API mechanism for setting centroid hints? Does "fastest" allow the centroid to be located at pixel center (outside the primitive)?

RESOLUTION: No specific new mechanism needs to be added. Just use the existing derivative hint mechanism.

31. Need to clarify that gl_FragCoord is also one of the samples for multi-sampling. Same sample as picked for varyings.

RESOLUTION: Yes.

32. Are the built-in functions and names behaving as if they are in the global scope, or in a scope more 'outer' than the global scope?

POSSIBLE RESOLUTION: No. Put the built-ins in the global scope. This means one cannot define a variable or structure in the global scope with the same name as a built-in function. This means shader global variables pollute the name space for built-in functions (addressable in the future by using "gl" prefix, or a name space for the "gl" case, have to decide if users can provide novel signatures of a gl name). As in 1.10, redeclaring a built-in function would not hide other functions of the same name. We would still allow for providing a new body of a built-in function, and still expect to find such a body for redeclared built-in functions.

RESOLUTION: Yes. The built-ins are in a more outer scope. This means you can define a global variable with the same name as a built-in, and that will hide all built-ins with that name. Redeclaring a built-in function would hide built-ins of that name, which breaks backward compatibility; all such signatures used will have to have a body provided. This would give an error for most cases where the coder overrode one signature, but not others, and still called one of the others, because for most built-ins, our conversion rules can't match arguments for one prototype to another. The exceptions being **lessThan**(integer arguments) vs. **lessThan**(float arguments), and similarly **greaterThan**, **lessThanEqual, greaterThanEqual, equal,** and **notEqual**.

33. What new keywords do we want to reserve.

RESOLUTION: We can reserve new keywords, because they are protected by #version. These include **lowp, mediump, highp, precision.**

34. We need to support basic invariance.

RESOLUTION: Duplicate the OpenGL ES **invariant** keyword, but only allow its use on user varyings, built-in varyings output from the vertex shader, and the special variables gl_Position and gl_PointSize.

35. Should we allow **invariant** on function return values, so that a function in a different compilation unit can be used in an invariant computation?

RESOLUTION: Defer. This is a good plan. For version 1.2, the compiler can only figure out the control flow and data flow if they lie entirely in a single compilation unit.

36. Do fragment shader varyings need to be declared as invariant if they were on the vertex side? Can they optionally be?

RESOLUTION: Invariant qualification has to match across sides. The interpolation computation can be split across sides, and both sides need to compute in an invariant way. It is an error if they do not match. Not reporting this as an error means either silently unnecessarily doing invariant computation on the vertex side or silently not being invariant when that was desired.

37. What happens if different compilation units have different implied sizes for the same implicitly sized array?

RESOLUTION: Use the maximum, don't give an error.

38. Should we remove embedded structures to match ES?

RESOLUTION: Yes. This eliminates a possible future name space problem where with a real name space operator (e.g. "::") nested structure names can be scoped to within the structure's name space, but today they have to be scoped an the same scope as the containing structure.