The OpenGL® Shading Language

Language Version: 1.30
Document Revision: 08
7-Aug-2008

John Kessenich

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

Table of Contents

1 Introduction

This document specifies only version 1.30 of the OpenGL Shading Language. It requires __VERSION__ to substitute 130, and requires **#version** to accept only 130. If **#version** is declared with 110 or 120, the language accepted is a previous version of the shading language, which will be supported depending on the version and type of context in the OpenGL API. See the OpenGL Graphics System Specification, Version 3.0, for details on what language versions are supported.

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:

Rob Barris
Pierre Boudier
Pat Brown
Nick Burns
Chris Dodd
Michael Gold
Nick Haemel
James Helferty
Brent Insko
Jeff Juliano
Jon Leech
Bill Licea-Kane
Barthold Lichtenbelt
Daniel Koch
Marc Olano
Ian Romanick
John Rosasco
Dave Shreiner
Jeremy Sandmel
Robert Simpson
Eskil Steenberg

1.2 Changes

1.2.1 Summary of Functionality differences from version 1.2

The following is a summary of features added in version 1.3:

- Integer support:

    - native signed and unsigned integers, integer vectors, and operations
    - bitwise shifts and masking
    - texture indices
    - texture return values
    - integer uniforms, vertex inputs, vertex outputs, fragment inputs, and fragment outputs
    - built-in function support: abs, sign, min, max, clamp, ...
- Other texture support:
    - Size queries.
    - Texture arrays.
    - Offsetting.
    - Explicit LOD and derivative controls
- **switch**/**case**/**default** statements.
- New built-ins: **trunc**(), **round**(), **roundEven**(), **isnan**(), **isinf**(), **modf**()
- hyperbolic trigonometric functions
- Preprocessor token pasting (##).
- User-defined fragment output variables.
- Shader input and output declarations via **in** and **out**.
- Improved compatibility with OpenGL ES
- non-perspective (linear) interpolation (**nosperspective**)
- new vertex input *gl_VertexID*.

The following is a summary of features deprecated in version 1.3:

- Use of the keywords **attribute** and **varying** (use **in** and **out**).
- Use of *gl_ClipVertex* (use *gl_ClipDistance*)
- Use of *gl_FragData* and *gl_FragColor* (use user-defined **out** instead).
- Built-in attributes. Use user-defined vertex inputs instead.
- Fixed function vertex or fragment stages mixed with shader programs. Provide shaders for all active programmable pipeline stages.
- All built-in texture function names. See new names.
- Use of the built-in varyings *gl_FogFragCoord* and *gl_TexCoord*. Use user-defined variable instead.
- The built in function **ftransform**. Use the **invariant** qualifier on a vertex output instead.
- Most built-in state.
- gl_MaxVaryingFloats (use gl_MaxVaryingComponents instead)

The following is a summary of features that have been removed in version 1.3:

- None, only deprecations occurred in this release.

1.2.2 Change history of this revision

Changes from revisions 6 and 7 of version 1.30 of the OpenGL Shading Language

- Fix all references to the OpenGL Graphics System specification, including matching notation for texturing parameters.

Changes from revision 5 of version 1.30 of the OpenGL Shading Language

- Reserved **superp**.
- Made it an error to specify integer literals too big for an integer variable.
- Increased
  - gl_MaxVaryingComponents to 64
  - gl_MaxDrawBuffers to 8
  - gl_MaxTextureCoords to 8
- Fixed some typos.

Changes from revision 4 of version 1.30 of the OpenGL Shading Language

- Updated acknowledgments; let me know if anyone is missing.
- Added summary lists of what's deprecated, removed, and added
- Deprecated fixed functionality control of a programmable stage
- flat is for both user and predeclared built-in in/out variables
- only statically used built-ins have to be redeclared as flat
- Made more clear that 1.1 and 1.2 shaders work, depending on state of the API
- Made clear ## does macro expansion after pasting not before
- ftransform() is deprecated instead of removed
- built-in state is deprecated instead of removed
- highp is always present in the fragment language, the default is highp
- order of qualification is either (invariant-qualifier interpolation-qualifier storage-qualifier precision-qualifier) or (storage-qualifier parameter-qualifier precision-qualifier)
- uint and int can be mixed for <<, >> but not for other operators
- combined descriptions of << and >>, and also of &, +, and ^
- switch statements can be empty, must have a statement between a label and the end of the switch, allows flow control to fall through

- updated the minimum maximums and added gl_MaxVaryingComponents and deprecated gl_MaxVaryingFloats
- added gl_ClipDistance[] to the fragment side
- Removed #include support
- Removed row_major
- Removed common blocks
- OpenGL ES synchronization
  - (a = b) is an r-value and never an l-value
- Updated the grammar with I have added these to the grammar
  - switch statement
  - case/default labels, which are mixed with other statements (needs semantic check for in switch)
  - uint, unsigned literals, unsigned vectors
  - 17 new sampler types
  - new storage qualifiers in, out, centroid in, centroid out (untangled from parameter in/out/inout)
  - interpolation qualifiers noperspective, flat, smooth
  - precision qualifiers
  - allowed bitwise and shift operators

Changes from revision 3 of version 1.30 of the OpenGL Shading Language

- Added deprecation section 1.6
- Added user-defined fragment shader outputs.
- Remove most built-in state.
- Deprecated built-in vertex inputs (attributes) and some outputs (varyings).
- Added gl_ClipDistance.
- Deprecated mixing fixed vertex/fragment stage with programmable fragment/vertex stage.
- Removed support for multiple programs tiling the pipeline (still original 1.2 model of one program for the whole pipeline).
- Removed **inout** as a way of declaring interface variables, to avoid the problem of things like interpolation qualifiers not knowing if they are modifying the copy in or the copy out. Also removes the problem of implicit pass through for a variable declared **inout** but never used.
- True native integer support
  - signed and unsigned integer semantics
  - bitwise operators and shifts

    - built-in functions operating on integers, **abs, sign, min, max, clamp,**
    - integer-based texture lookup functions, texel fetch
- texture arrays
- projective cube map texture and shadow
- explicit gradient texture lookup
- offset-texel texture lookup
- texture size functions
- add **noperspective** interpolation qualifier
- Added **trunc**, **round**, **roundEven**, **modf**
- Removed **ftransform**
- Added **isinf** and **isnan.**
- Added hyperbolic functions **sinh, cosh, tanh, asinh, acosh, atanh.**
- Some synchronization with ES (**inout** parameter evaluation order, foo(**void**), others)
- Deprecated *gl_ClipVertex*
- Added *gl_VertexID*
- It's an error to use #**if** etc. on an undefined name

Changes from revision 2 of version 1.30 of the OpenGL Shading Language

- Large rework of section 8.7 Texture Lookup Functions. Dropped dimensionality/shadow from the names, organized by type instead of dimensionality, added in Lod control.
- Use gl_Position for clipping if gl_ClipVertex is not statically written.
- Remove language about the fixed pipeline in the description of ftransform().

Changes from revision 10 of version 1.20 of the OpenGL Shading Language

- **in**, **out**, and **inout** are used at global scope as the preferred way of declaring attributes, varyings, and fragment shader outputs. This eases the usage of **centroid, flat, smooth, invariant,** etc. by reducing the number of keywords needed to declare a variable, removes the misnomer that **flat** variables vary, provides for a default interpolation, and scales to additional future programmable pipe stages.
- Common blocks are added and can be backed by buffers in the API.
- "gl_" prefixed uniforms and attributes and several of the varyings no longer reflect built-in state, but are predeclared by the language as a convenience to the user.
- The ability to index into an array of samplers with a variable index is removed.
- Token pasting (##) is added to the preprocessor.
- Add **row_major** to support row-major matrices to allow packing of a 3-row 4-column matrix into 3 uniforms or 3 attributes.

- Support **#include** via named source strings.
- Accept the precision qualifiers from OpenGL ES with no expectation that anything is done with them.
- **switch** statements are added for integer scalars only
- **mix()** is expanded to operate on a Boolean 3rd argument that does not interpolate but selects.

1.3 Overview

This document describes *The OpenGL Shading Language, version 1.30.*

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 Graphics System Specification will specify the OpenGL entry points used to manipulate and communicate with programs and shaders.

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.

1.6 Deprecation

This version of the OpenGL Shading Language deprecates some features. These are clearly called out in this specification as “deprecated”. They are still present in this version of the language, but are targeted for potential removal in a future version of the shading language. The OpenGL API has a forward compatibility mode that will disallow use of deprecated features. If compiling in a mode where use of deprecated features is disallowed, their use causes compile time errors. See the OpenGL Graphics System Specification for details on what causes deprecated language features to be accepted or to return an error.

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.

Most OpenGL state is not tracked or made available to shaders. Typically, user-defined variables will be used for communicating between different stages of the OpenGL pipeline. However, a small amount of state is still tracked and automatically made available to shaders, and there are a few built-in variables for interfaces between different stages of the OpenGL pipeline.

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 of 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.

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 130.

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 further restricted to expressions operating on literal integer constants, plus identifiers consumed by the **defined** operator. It is an error to use **#if** or **#elif** on expressions containing undefined macro names, other than as arguments to 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 )
```

Two tokens in a macro can be concatenated into one token using the token pasting (##) operator, as is standard for C++ preprocessors. The result must be a valid single token, which will then be subject to macro expansion. That is, macro expansion happens after token pasting and does not happen before token pasting. There are no other number sign based operators (e.g. no # or #@), 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 OpenGL Graphics System Specification 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** 130" is required in any shader that uses version 1.30 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. Different shaders (compilation units) that are linked together in the same program must be the same version.

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 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 flat smooth noperspective

break continue do for while switch case default

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

uint uvec2 uvec3 uvec4

lowp mediump highp precision

sampler1D sampler2D sampler3D samplerCube

sampler1DShadow sampler2DShadow samplerCubeShadow

sampler1DArray sampler2DArray

sampler1DArrayShadow sampler2DArrayShadow

isampler1D isampler2D isampler3D isamplerCube

isampler1DArray isampler2DArray

usampler1D usampler2D usampler3D usamplerCube

usampler1DArray usampler2DArray

struct

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

common partition active

asm

class union enum typedef template this packed

goto

inline noinline volatile public static extern external interface

long short double half fixed unsigned superp

input output

hvec2 hvec3 hvec4 dvec2 dvec3 dvec4 fvec2 fvec3 fvec4

sampler2DRect sampler3DRect sampler2DRectShadow

samplerBuffer

filter

image1D image2D image3D imageCube

iimage1D iimage2D iimage3D iimageCube

uimage1D uimage2D uimage3D uimageCube

image1DArray image2DArray

iimage1DArray iimage2DArray uimage1DArray uimage2DArray

image1DShadow image2DShadow

image1DArrayShadow image2DArrayShadow

imageBuffer iimageBuffer uimageBuffer

sizeof cast

namespace using

row_major

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. However, as noted in the specification, there are some cases where previously declared variables can be redeclared to change some property, and predeclared "gl_" names are allowed to be redeclared in a shader.

3.8 Static Use

Some language rules described below depend on whether something is *statically* written or used.

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

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, grouped as follows.

Transparent 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
uint	an unsigned 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 signed integer vector
ivec3	a three-component signed integer vector
ivec4	a four-component signed integer vector

Type	Meaning
uvec2	a two-component unsigned integer vector
uvec3	a three-component unsigned integer vector
uvec4	a four-component unsigned integer vector
mat2	a 2×2 floating-point matrix
mat3	a 3×3 floating-point matrix
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**

Floating Point Sampler Types (opaque)

Type	Meaning
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
sampler1DArray	a handle for accessing a 1D array texture
sampler2DArray	a handle for accessing a 2D array texture
sampler1DArrayShadow	a handle for accessing a 1D array depth texture with comparison
sampler2DArrayShadow	a handle for accessing a 2D array depth texture with comparison

Signed Integer Sampler Types (opaque)

Type	Meaning
isampler1D	a handle for accessing an integer 1D texture

Type	Meaning
isampler2D	a handle for accessing an integer 2D texture
isampler3D	a handle for accessing an integer 3D texture
isamplerCube	a handle for accessing an integer cube mapped texture
isampler1DArray	a handle for accessing an integer 1D array texture
isampler2DArray	a handle for accessing an integer 2D array texture

Unsigned Integer Sampler Types (opaque)

Type	Meaning
usampler1D	a handle for accessing an unsigned integer 1D texture
usampler2D	a handle for accessing an unsigned integer 2D texture
usampler3D	a handle for accessing an unsigned integer 3D texture
usamplerCube	a handle for accessing an unsigned integer cube mapped texture
usampler1DArray	a handle for accessing an unsigned integer 1D array texture
usampler2DArray	a handle for accessing an unsigned integer 2D array texture

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 or actual 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 (=) must be an 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

Signed and unsigned integer variables are fully supported. In this document, the term *integer* is meant to generally include both signed and unsigned integers. Unsigned integers have exactly 32 bits of precision. Signed integers use 32 bits, including a sign bit, in two's complement form. Operations resulting in overflow or underflow will not cause any exception, nor will they saturate, rather they will "wrap" to yield the low-order 32 bits of the result.

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

```
int i, j = 42;  // default integer literal type is int
uint k = 3u;    // "u" establishes the type as uint
```

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

integer-constant :
decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

integer-suffix: one of
u U

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, or before the suffix **u** or **U**. When the suffix **u** or **U** is present, the literal has type **uint**, otherwise the type is **int**. A leading unary minus sign (-) is interpreted as an arithmetic unary negation, not as part of the constant.

It is an error to provide a literal integer whose magnitude is too large to store in a variable of matching signed or unsigned type.

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, including before a suffix. 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 colors, normals, positions, texture coordinates, texture lookup results and the like. Boolean vectors can be used for component-wise comparisons of numeric vectors. 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 and their filters. They are used with the built-in texture functions (described in Section 8.7 “Texture Lookup Functions”) to specify which texture to access and how it is to be filtered. 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 aggregated into arrays within a shader (using square brackets **[ ]**) can only be indexed with integral constant expressions (see Section 4.3.3 “Constant 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 structure definitions 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 **uint**	**float**
ivec2 **uvec2**	**vec2**
ivec3 **uvec3**	**vec3**
ivec4 **uvec4**	**vec4**

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

When an implicit conversion is done, it is not a re-interpretation of the expression's bit pattern, but a conversion of its 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**. Integer values having more bits of precision than a floating point mantissa will lose precision when converted to **float**.

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 one 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
in **centroid in**	linkage into a shader from a previous stage, variable is copied in linkage with centroid based interpolation
out **centroid out**	linkage out of a shader to a subsequent stage, variable is copied out linkage with centroid based interpolation
attribute	deprecated; 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**	deprecated; linkage between a vertex shader and a fragment shader for interpolated data

Outputs from a vertex shader (**out**) and inputs to a fragment shader (**in**) can be further qualified with one or more of these interpolation qualifiers

Qualifier	Meaning
smooth	perspective correct interpolation
flat	no interpolation
noperspective	linear interpolation

These interpolation qualifiers may only precede the qualifiers **in**, **centroid in**, **out**, or **centroid out** in a declaration. They do not apply to the deprecated storage qualifiers **varying** or **centroid varying**. They also do not apply to inputs into a vertex shader or outputs from a fragment shader.

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

Function parameters can use **const**, **in**, and **out** qualifiers, but as *parameter qualifiers*. Parameter qualifiers are discussed 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, with a **const** qualifier or with 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 pipeline stages. 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 signed or unsigned 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 Inputs

Shader input variables are declared with the **in** storage qualifier or the **centroid in** storage qualifier. They form the input interface between previous stages of the OpenGL pipeline and the declaring shader. Input variables must be declared at global scope. Values from the previous pipeline stage are copied into input variables at the beginning of shader execution. Variables declared as **in** or **centroid in** may not be written to during shader execution.

Vertex shader input variables (or attributes) receive per-vertex data. They are declared in a vertex shader with the **in** qualifier or the deprecated **attribute** qualifier. It is an error to use **centroid in** in a vertex shader. The values copied in are established by the OpenGL API. It is an error to use **attribute** in a non-vertex shader. Vertex shader inputs can only be **float**, floating-point vectors, matrices, signed and unsigned integers and integer vectors. They cannot be arrays or structures.

Example declarations in a vertex shader:

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

See Section 7 “Built-in Variables” for a list of the built-in input names.

Fragment shader inputs (or varyings) get per-fragment values, typically interpolated from a previous stage's outputs. They are declared in fragment shaders with the **in** storage qualifier, the **centroid in** storage qualifier, or the deprecated **varying** and **centroid varying** storage qualifiers. Fragment inputs can only be signed and unsigned integers and integer vectors, **float**, floating-point vectors, matrices, or arrays of these. Structures cannot be input.

Fragment inputs are declared as in the following examples:

```
in vec3 normal;
centroid in vec2 TexCoord;
invariant centroid in vec4 Color;
noperspective out float temperature;
flat in vec3 myColor;
centroid noperspective in vec2 myTexCoord;
```

It is expected that graphics hardware will have a small number of fixed vector locations for passing vertex inputs. Therefore, the OpenGL Shading language defines each non-matrix input variable as taking up one such vector location . There is an implementation dependent limit on the number of locations that can be used, and if this is exceeded it will cause a link error. (Declared input variables that are not statically used do not count against this limit.) A scalar input counts the same amount against this limit as a **vec4**, so applications may want to consider packing groups of four unrelated float inputs together into a vector to better utilize the capabilities of the underlying hardware. A matrix input will use up multiple locations. The number of locations used will equal the number of columns in the matrix.

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 Outputs

Shader output variables are declared with the **out** or **centroid out** storage qualifiers. They form the output interface between the declaring shader and the subsequent stages of the OpenGL pipeline. Output variables must be declared at global scope. During shader execution they will behave as normal unqualified global variables. Their values are copied out to the subsequent pipeline stage on shader exit.

There is *not* an **inout** storage qualifier at global scope for declaring a single variable name as both input and output to a shader. Output variables must be declared with different names than input variables.

Vertex output variables output per-vertex data and are declared using the **out** storage qualifier, the **centroid out** storage qualifier, or the deprecated **varying** storage qualifier. They can only be **float**, floating-point vectors, matrices, signed or unsigned integers or integer vectors, or arrays of any these. If a vertex output is a signed or unsigned integer or integer vector, then it must be qualified with the interpolation qualifier **flat**. Structures cannot be output.

Vertex outputs are declared as in the following examples:

```
out vec3 normal;
centroid out vec2 TexCoord;
invariant centroid out vec4 Color;
noperspective out float temperature;  // varying is deprecated
flat out vec3 myColor;
noperspective centroid out vec2 myTexCoord;
```

Fragment outputs output per-fragment data and are declared using the **out** storage qualifier. It is an error to use **centroid out** in a fragment shader. Fragment outputs can only be **float**, floating-point vectors, signed or unsigned integers or integer vectors, or arrays of any these. Matrices and structures cannot be output. Fragment outputs are declared as in the following examples:

```
out vec4 FragmentColor;
out uint Luminosity;
```

4.3.7 Interpolation

The presence of and type of interpolation is controlled by the storage qualifiers **centroid in** and **centroid out**, and by the optional interpolation qualifiers **smooth**, **flat**, and **noperspective** as well as by default behaviors established through the OpenGL API when no interpolation qualifier is present. When an interpolation qualifier is used, it overrides settings established through the OpenGL API. It is a compile-time error to use more than one interpolation qualifier.

The following predeclared variables can be redeclared with an interpolation qualifier:

Vertex language:

```
gl_FrontColor
gl_BackColor
gl_FrontSecondaryColor
gl_BackSecondaryColor
```

Fragment language:

```
gl_Color
gl_SecondaryColor
```

For example,

```
in vec4 gl_Color;       // predeclared by the fragment language
flat in vec4 gl_Color;  // redeclared by user to be flat
```

If *gl_Color* is redeclared with an interpolation qualifier, then *gl_FrontColor* and *gl_BackColor* (if they are written to) must also be redeclared with the same interpolation qualifier, and vice versa. If *gl_SecondaryColor* is redeclared with an interpolation qualifier, then *gl_FrontSecondaryColor* and *gl_BackSecondaryColor* (if they are written to) must also be redeclared with the same interpolation qualifier, and vice versa. This qualifier matching on predeclared variables is only required for variables that are statically used within the shaders in a program.

A variable qualified as **flat** will not be interpolated. Instead, it will have the same value for every fragment within a triangle. This value will come from a single provoking vertex, as described by the OpenGL Graphics System Specification. User-declared variables can be qualified as **flat** and the predeclared variables listed above and can be redeclared as **flat**. It is an error to declare any other built-in variable as **flat**. A variable may be qualified as **flat centroid**, which will mean the same thing as qualifying it only as **flat**.

A variable qualified as **smooth** will be interpolated in a perspective-correct manner over the primitive being rendered. Interpolation in a perspective correct manner is specified in equations 3.6 and 3.8 in the OpenGL Graphics System Specification, Version 3.0.

A variable qualified as **noperspective** must be interpolated linearly in screen space, as described in equation 3.7 and the approximation that follows equation 3.8 in the OpenGL Graphics System Specification, Version 3.0.

This paragraph only applies if interpolation is being done: If single-sampling, the value is interpolated to the pixel's center, and the **centroid** qualifier, if present, is ignored. If multi-sampling and the variable 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 the variable 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 interpolated variables.

The type and presence of the interpolation qualifiers and storage qualifiers and **invariant** qualifiers of variables with the same name declared in linked vertex and fragments shaders must match, otherwise the link command will fail. Only those input variables read in the fragment shader executable must be written to by the vertex shader executable; declaring superfluous output variables in a vertex shader is permissible.

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

Precision qualifiers are added for code portability with OpenGL ES, not for functionality. They have the same syntax as in OpenGL ES, as described below, but they have no semantic meaning, which includes no effect on the precision used to store or operate on variables.

If an extension adds in the same semantics and functionality in the OpenGL ES 2.0 specification for precision qualifiers, then the extension is allowed to reuse the keywords below for that purpose.

4.5.1 Range and Precision

Section number reserved for future use.

4.5.2 Precision Qualifiers

Any floating point or any integer declaration can have the type preceded by one of these precision qualifiers:

Qualifier	Meaning
highp	None.
mediump	None.
lowp	None.

For example:

```
lowp float color;
out mediump vec2 P;
lowp ivec2 foo(lowp mat3);
highp mat4 m;
```

Literal constants do not have precision qualifiers. Neither do Boolean variables. Neither do floating point constructors nor integer constructors when none of the constructor arguments have precision qualifiers.

Precision qualifiers, as with other qualifiers, do not effect the basic type of the variable. In particular, there are no constructors for precision conversions; constructors only convert types. Similarly, precision qualifiers, as with other qualifiers, do not contribute to function overloading based on parameter types. As discussed in the next chapter, function input and output is done through copies, and therefore qualifiers do not have to match.

The same object declared in different shaders that are linked together must have the same precision qualification. This applies to inputs, outputs, uniforms, and globals.

4.5.3 Default Precision Qualifiers

The precision statement

```
precision precision-qualifier type;
```

can be used to establish a default precision qualifier. The **type** field can be either **int** or **float**, and the *precision-qualifier* can be **lowp**, **mediump**, or **highp**. Any other types or qualifiers will result in an error. If *type* is **float**, the directive applies to non-precision-qualified floating point type (scalar, vector, and matrix) declarations. If *type* is **int**, the directive applies to all non-precision-qualified integer type (scalar, vector, signed, and unsigned) declarations. This includes global variable declarations, function return declarations, function parameter declarations, and local variable declarations.

Non-precision qualified declarations will use the precision qualifier specified in the most recent **precision** statement that is still in scope. The **precision** statement has the same scoping rules as variable declarations. If it is declared inside a compound statement, its effect stops at the end of the innermost statement it was declared in. Precision statements in nested scopes override precision statements in outer scopes. Multiple precision statements for the same basic type can appear inside the same scope, with later statements overriding earlier statements within that scope.

The vertex language has the following predeclared globally scoped default precision statements:

```
precision highp float;
precision highp int;
```

The fragment language has the following predeclared globally scoped default precision statement:

```
precision mediump int;
```

The fragment language has no default precision qualifier for floating point types. Hence for **float**, floating point vector and matrix variable declarations, either the declaration must include a precision qualifier or the default float precision must have been previously declared.

4.5.4 Available Precision Qualifiers

The built-in macro GL_FRAGMENT_PRECISION_HIGH is defined to 1:

```
#define GL_FRAGMENT_PRECISION_HIGH 1
```

This macro is available in both the vertex and fragment languages.

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

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

or as part of a declaration when a variable is declared

```
invariant centroid out vec3 Color;
```

The invariant qualifier must appear before any interpolation qualifiers or storage qualifiers when combined with a declaration. Only variables output from a shader can be candidates for invariance. This includes user-defined output variables and the built-in output variables. For 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 interpolation-qualifier storage-qualifier precision qualifier

storage-qualifier parameter-qualifier precision qualifier

5 Operators and Expressions

5.1 Operators

The OpenGL Shading Language has the following operators.

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	**++ --** **+ - ~ !**	Right to Left
4	multiplicative	*** / %**	Left to Right
5	additive	**+ -**	Left to Right
6	bit-wise shift	**<< >>**	Left to Right
7	relational	**< > <= >=**	Left to Right
8	equality	**== !=**	Left to Right
9	bit-wise and	**&**	Left to Right
10	bit-wise exclusive or	**^**	Left to Right
11	bit-wise inclusive or	**|**	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	**=** **+= -=** ***= /=** **%= <<= >>=** **&= ^= |=**	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 a signed integer value to a float
bool(float)  // converts a float value to a Boolean
bool(int)    // converts a signed integer value to a Boolean
uint(bool)   // converts a Boolean value to an unsigned integer
uint(float)  // converts a float value to an unsigned integer
uint(int)    // converts a signed integer value to an unsigned integer
int(uint)    // converts an unsigned integer to a signed integer
bool(uint)   // converts an unsigned integer value to a Boolean value
float(uint)  // converts an unsigned integer value to a float value
```

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

When a constructor is used to convert an **int**, **uint**, or a **float** to a **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**, **uint**, or **float**, **false** is converted to 0 or 0.0, and **true** is converted to 1 or 1.0.

The constructor **int(uint)** preserves the bit pattern in the argument, which will change the argument's value if its sign bit is set. The constructor **uint(int)** preserves the bit pattern in the argument, which will change its value if it is negative.

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, one or more vectors, or one or more 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
vec4(mat2)   // the vec4 is column 0 followed by column 1

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)
```

That is, *result[i][j]* is set to the float argument for all $i = j$ and set to 0 for all $i \neq j$.

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 the texture coordinate set, *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 resulting 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 **int** or **uint**.

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 be 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 (=):

```
lvalue-expression = rvalue-expression
```

The *lvalue-expression* evaluates to an l-value. The assignment operator stores the value of *rvalue-expression* into the l-value and returns an r-value with the type and precision of *lvalue-expression*. The *lvalue-expression* and *rvalue-expression* 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-expression*, in which case an implicit conversion will be done on the *rvalue-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.

The other assignment operators are

- add into (+=)
- subtract from (**-**=)
- multiply into (*****=)
- divide into (/=)
- modulus into (**%**=)
- left shift by (<<=)
- right shift by (>>=)
- and into (**&**=)
- inclusive-or into (|=)
- exclusive-or into (^=)

where the general expression

```
lvalue op= expression
```

is equivalent to

```
lvalue = lvalue op expression
```

where *op* is as described below, and the l-value and expression must satisfy the semantic requirements of both *op* and equals (=).

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, uint, 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. If the operands are integer types, they must both be signed or both be unsigned. All arithmetic binary operators result in the same fundamental type (signed integer, unsigned 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 (**%**) operates on signed or unsigned integers or integer vectors. The operand types must both be signed or both be unsigned. The operands cannot be vectors of differing size. If one operand is a scalar and the other vector, then the scalar is applied component-wise to the vector, resulting in the same type as the vector. If both are vectors of the same size, the result is computed component-wise. The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero second operands remain defined. If both operands are non-negative, then the remainder is non-negative. Results are undefined if one or both operands are negative. The operator modulus (**%**) is not defined for any other data types (non-integer types).
- 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.
- The one's complement operator (~). The operand must be of type signed or unsigned integer or integer vector, and the result is the one's complement of its operand; each bit of each component is complemented, including any sign bits.
- The shift operators (**<<**) and (**>>**). For both operators, the operands must be signed or unsigned integers or integer vectors. One operand can be signed while the other is unsigned. In all cases, the resulting type will be the same type as the left operand. If the first operand is a scalar, the second operand has to be a scalar as well. If the first operand is a vector, the second operand must be a scalar or a vector, and the result is computed component-wise. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's base type. The value of E1 << E2 is E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1 >> E2 is E1 right-shifted by E2 bit positions. If E1 is a signed integer, the right-shift will extend the sign bit. If E1 is an unsigned integer, the right-shift will zero-extend.
- The bitwise operators and (**&**), exclusive-or (^), and inclusive-or (|). The operands must be of type signed or unsigned integers or integer vectors. The operands cannot be vectors of differing size. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The fundamental types of the operands (signed or unsigned) must match, and will be the resulting fundamental type. For and (**&**), the result is the bitwise-and function of the operands. For exclusive-or (^), the result is the bitwise exclusive-or function of the operands. For inclusive-or (|), the result is the bitwise inclusive-or function of the operands.

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** and **switch-case-default)**
- 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 argument types. 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. The same function name can be used for multiple functions, as long as the parameter types differ. If a function name is declared twice with the same parameter types, then the return types and all qualifiers must also match, and it is the same function being declared. When function calls are resolved, an exact type match for all the arguments is sought. If an exact match is found, all other functions are ignored, and the exact match is used. If no exact match is found, then the implicit conversions in Section 4.1.10 "Implicit Conversions" will be applied to find a match. Mismatched types on input parameters (**in** or **inout** or default**)** must have a conversion from the calling argument type to the formal parameter type. Mismatched types on output parameters (**out** or **inout**) must have a conversion from the formal parameter type to the calling argument type. When argument conversions are used to find a match, it is a semantic error if there are multiple ways to apply these conversions to make the call match more than one function.

For example,

```
vec4 f(in vec4 x, out  vec4 y);
vec4 f(in vec4 x, out ivec4 y); // okay, different argument type
int  f(in vec4 x, out ivec4 y); // error, only return type differs
vec4 f(in vec4 x, in  ivec4 y); // error, only qualifier differs
int  f(const in vec4 x, out ivec4 y); // error, only qualifier differs
```

Calling the first two functions above with the following argument types yields

```
f(vec4, vec4)   // exact match of vec4 f(in vec4 x, out vec4 y)
f(vec4, ivec4)  // exact match of vec4 f(in vec4 x, out ivec4 y)
f(ivec4, vec4)  // error, convertible to both
f(ivec4, ivec4) // okay, convertible only to vec4 f(in vec4 x, out ivec4 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 of 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 parameters 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. 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**.

All arguments are evaluated at call time, exactly once, in order, from left to right. Evaluation of an **in** parameter results in a value that is copied to the formal parameter. Evaluation of an **out** parameter results in an l-value that is used to copy out a value when the function returns. Evaluation of an **inout** parameter results in both a value and an l-value; the value is copied to the formal parameter at call time and the l-value is used to copy out a value when the function returns.

The order in which output parameters are copied back to the caller is undefined.

If the function matching described in the previous section required argument type conversions, these conversions are applied at copy-in and copy-out times.

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**, **if-else**, or **switch** statements:

selection-statement :
if (*bool-expression*) *statement*
if (*bool-expression*) *statement* **else** *statement*
switch (*init-expression*) { $switch\text{-}statement\text{-}list_{opt}$ }

Where *switch-statement-list* is a list of zero or more *switch-statement* and other statements defined by the language, where *switch-statement* adds some forms of labels. That is

switch-statement-list :
switch-statement
switch-statement-list switch-statement

switch-statement :
case *constant-expression* **:**
default :
statement

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

The type of *init-expression* in a switch statement must be a scalar integer. If a **case** label has a *constant-expression* of equal value, then execution will continue after that label. Otherwise, if there is a **default** label, execution will continue after that label. Otherwise, execution skips the rest of the switch statement. It is an error to have more than one **default** or a replicated *constant-expression.* A **break** statement not nested in a loop or other switch statement (either not nested or nested only in **if** or **if-else** statements) will also skip the rest of the switch statement. Fall through labels are allowed, but it is an error to have no statement between a label and the end of the **switch** statement.

No **case** or **default** labels can be nested inside other flow control nested within their corresponding **switch**.

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 and switch statements. It is simply an immediate exit of the inner-most loop or switch statements containing the **break**. No further execution of *condition-expression, loop-expression*, or *switch-statement* 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 occur in fixed functionality between the vertex processor and the fragment processor. Shaders communicate with the fixed functionality of OpenGL through the use of built-in variables.

These built-in vertex shader variables for communicating with fixed functionality are intrinsically declared as follows:

```
out vec4  gl_Position;       // must be written to
out float gl_PointSize;      // may be written to
in  int   gl_VertexID;
out float gl_ClipDistance[]; // may be written to
out vec4  gl_ClipVertex;     // may be written to, deprecated
```

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 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_VertexID* is a vertex shader an input variable that holds an integer index for the vertex, as defined by the OpenGL Graphics System Specification. While the variable gl_VertexID is always present, its value is not always defined. For details on when it is defined, see the "Shader Inputs" subsection of section 2.20.3 "Shader Execution" of the OpenGL Graphics System Specification, Version 3.0.

The variable *gl_ClipDistance* provides the forward compatible mechanism in the vertex shader for controlling user clipping. To use this, a vertex shader is responsible for maintaining a set of clip planes, computing the distance from the vertex to each clip plane, and storing distances to the plane in *gl_ClipDistance[i]* for each plane *i*. A distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the clip plane, and a negative distance means the point is outside the clip plane. The clip distances will be linearly interpolated across the primitive and the portion of the primitive with interpolated distances less than 0 will be clipped.

The *gl_ClipDistance* array is predeclared as unsized and must be sized by the shader either redeclaring it with a size or indexing it only with integral constant expressions. This needs to size the array to include all the clip planes that are enabled via the OpenGL API; if the size does not include all enabled planes, results are undefined. The size can be at most *gl_MaxClipDistances*. The number of varying components (see *gl_MaxVaryingComponents)* consumed by *gl_ClipDistance* will match the size of the array, no matter how many planes are enabled. The shader must also set all values in *gl_ClipDistance* that have been enabled via the OpenGL API, or results are undefined. Values written into *gl_ClipDistance* for planes that are not enabled have no effect.

The variable *gl_ClipVertex* is deprecated. It 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.

If a linked set of shaders forming the vertex stage contains no static write to *gl_ClipVertex* or *gl_ClipDistance,* but the application has requested clipping against user clip planes through the API, then the coordinate written to *gl_Position* is used for comparison against the user clip planes. This behavior is also deprecated. Writing to *gl_ClipDistance* is the preferred method for user clipping. It is an error for a shader to statically write both *gl_ClipVertex* and *gl_ClipDistance.*

If *gl_PointSize* is not written to, its value is undefined in subsequent pipe stages.

7.2 Fragment Shader Special Variables

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

```
in  vec4  gl_FragCoord;
in  bool  gl_FrontFacing;
in  float gl_ClipDistance[];
out vec4  gl_FragColor;                    // deprecated
out vec4  gl_FragData[gl_MaxDrawBuffers];  // deprecated
out float gl_FragDepth;
```

Except as noted below, they behave as other input and output 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** statement is executed. Both *gl_FragColor* and *gl_FragData* are deprecated; the preferred usage is to explicitly declare these outputs in the fragment shader using the **out** storage qualifier.

The fixed functionality computed depth for a fragment may be obtained by reading *gl_FragCoord.z,* described below.

Deprecated: 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.

Deprecated: 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. Similarly, if user declared output variables are in use (statically assigned to), then the built-in variables *gl_FragColor* and *gl_FragData* may not be assigned to. These incorrect usages all generate compile time errors.

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

The variable *gl_FragCoord* is available as an input 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 in** 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 any 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 input 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 input variable *gl_ClipDistance* array contains linearly interpolated values for the vertex values written by the vertex shader to the *gl_ClipDistance* vertex output variable. This array must be sized in the fragment shader either implicitly or explicitly to be the same size as it was sized in the vertex shader. Only elements in this array that have clipping enabled will have defined values.

7.3 Vertex Shader Built-In Inputs

Deprecated: The following predeclared input names can be used from within a vertex shader to access the current values of OpenGL state.

```
in vec4  gl_Color;           // deprecated
in vec4  gl_SecondaryColor;  // deprecated
in vec3  gl_Normal;          // deprecated
in vec4  gl_Vertex;          // deprecated
in vec4  gl_MultiTexCoord0;  // deprecated
in vec4  gl_MultiTexCoord1;  // deprecated
in vec4  gl_MultiTexCoord2;  // deprecated
in vec4  gl_MultiTexCoord3;  // deprecated
in vec4  gl_MultiTexCoord4;  // deprecated
in vec4  gl_MultiTexCoord5;  // deprecated
in vec4  gl_MultiTexCoord6;  // deprecated
in vec4  gl_MultiTexCoord7;  // deprecated
in float gl_FogCoord;        // deprecated
```

7.4 Built-In Constants

The following built-in constants are provided to vertex and fragment shaders. The actual values used are implementation dependent, but must be at least the value shown. Some are deprecated, as indicated in comments.

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

const int  gl_MaxTextureUnits = 16;
const int  gl_MaxVertexAttribs = 16;
const int  gl_MaxVertexUniformComponents = 1024;
const int  gl_MaxVaryingFloats = 64;            // Deprecated
const int  gl_MaxVaryingComponents = 64;
const int  gl_MaxVertexTextureImageUnits = 16;
const int  gl_MaxCombinedTextureImageUnits = 16;
const int  gl_MaxTextureImageUnits = 16;
const int  gl_MaxFragmentUniformComponents = 1024;
const int  gl_MaxDrawBuffers = 8;
const int  gl_MaxClipDistances = 8;

//
// The following are deprecated.
//
const int  gl_MaxClipPlanes = 8;                // deprecated
const int  gl_MaxTextureCoords = 8;             // deprecated
```

The constant *gl_MaxVaryingFloats* is deprecated, use *gl_MaxVaryingComponents* instead. The constant *gl_MaxClipPlanes* is deprecated along with user clip planes, use clip distances and *gl_MaxClipDistances* instead. The constant *gl_MaxTextureCoords* is deprecated, use user-defined interpolants instead.

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 section numbers and notations are references to the OpenGL Graphics System Specification, Version 3.0.

```
//
// Depth range in window coordinates, section 2.12.1
//
struct gl_DepthRangeParameters {
    float near;        // n
    float far;         // f
    float diff;        // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;
```

The following state is deprecated:

```
//
// Deprecated.
//
uniform mat4  gl_ModelViewMatrix;
uniform mat4  gl_ProjectionMatrix;
uniform mat4  gl_ModelViewProjectionMatrix;
uniform mat4  gl_TextureMatrix[gl_MaxTextureCoords];

//
// Deprecated.
//
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];
```

```
//
// Deprecated.
//
uniform float gl_NormalScale;


//
// Deprecated.
//
uniform vec4  gl_ClipPlane[gl_MaxClipPlanes];

//
// Deprecated.
//
struct gl_PointParameters {
    float size;
    float sizeMin;
    float sizeMax;
    float fadeThresholdSize;
    float distanceConstantAttenuation;
    float distanceLinearAttenuation;
    float distanceQuadraticAttenuation;
};

uniform gl_PointParameters gl_Point;

//
// Deprecated.
//
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;
```

```
//
// Deprecated.
//

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;

//
// Deprecated.
// 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];
```

```
//
// Deprecated.
//
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];

//
// Deprecated.
//
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 Built-In Vertex Output and Fragment Input Variables

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

It is deprecated to have the GL provide fixed functionality behavior for a programmable pipeline stage. For example, mixing a fixed functionality vertex stage with a programmable fragment stage is deprecated. Pipeline configurations where only a proper subset of stages are being used do not require the unused stages to have shaders.

The following built-in vertex output variables are available. 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.

```
out vec4  gl_FrontColor;
out vec4  gl_BackColor;
out vec4  gl_FrontSecondaryColor;
out vec4  gl_BackSecondaryColor;
```

The following are also present in a vertex shader, but are deprecated:

```
out vec4  gl_TexCoord[];  // at most will be gl_MaxTextureCoords
out float gl_FogFragCoord;
```

For *gl_FogFragCoord* (deprecated), the value written will be used as the "c" value in section 3.11 of the OpenGL Graphics System Specification, Version 3.0, 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* (deprecated) 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 fragment input variables are available in a fragment shader.

```
in vec4  gl_Color;
in vec4  gl_SecondaryColor;
in vec2  gl_PointCoord;
```

The following fragment inputs are also available in a fragment shader, but are deprecated:

```
in float gl_FogFragCoord;
in vec4  gl_TexCoord[];
```

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 3.11 of the OpenGL Graphics System Specification, Version 3.0. 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. Where the input arguments (and corresponding output) can be **int**, **ivec2**, **ivec3**, or **ivec4**, *genIType* is used as the argument. Where the input arguments (and corresponding output) can be **uint**, **uvec2**, **uvec3**, or **uvec4**, *genUType* is used as the argument. For any specific use of a function, the actual type substituted for *genType*, *genIType*, or *genUType* 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 $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

Syntax	Description
genType **sinh** (genType *x*)	Returns the hyperbolic sine function $\frac{e^x - e^{-x}}{2}$
genType **cosh** (genType *x*)	Returns the hyperbolic cosine function $\frac{e^x + e^{-x}}{2}$
genType **tanh** (genType *x*)	Returns the hyperbolic tangent function $\frac{\sinh(x)}{\cosh(x)}$
genType **asinh** (genType *x*)	Arc hyperbolic sine; returns the inverse of **sinh**.
genType **acosh** (genType *x*)	Arc hyperbolic cosine; returns the non-negative inverse of **cosh**. Results are undefined if $x < 1$.
genType **atanh** (genType *x*)	Arc hyperbolic tangent; returns the inverse of **tanh**. Results are undefined if $\|x\| \geq 1$.

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 $x <= 0$.

Syntax	Description
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*) genIType **abs** (genIType *x*)	Returns *x* if *x* >= 0, otherwise it returns –*x*.
genType **sign** (genType *x*) genIType **sign** (genIType *x*)	Returns 1.0 if *x* > 0, 0.0 if *x* = 0, or –1.0 if *x* < 0.
genType **floor** (genType *x*)	Returns a value equal to the nearest integer that is less than or equal to *x*.
genType **trunc** (genType *x*)	Returns a value equal to the nearest integer to *x* whose absolute value is not larger than the absolute value of *x*.
genType **round** (genType *x*)	Returns a value equal to the nearest integer to *x*. The fraction 0.5 will round in a direction chosen by the implementation, presumably the direction that is fastest. This includes the possibility that **round**(*x*) returns the same value as **roundEven**(*x*) for all values of *x*.
genType **roundEven** (genType *x*)	Returns a value equal to the nearest integer to *x*. A fractional part of 0.5 will round toward the nearest even integer. (Both 3.5 and 4.5 for x will return 4.0.)
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*).

Syntax	Description
genType **mod** (genType *x*, float *y*) genType **mod** (genType *x*, genType *y*)	Modulus. Returns $x - y * \textbf{floor}\ (x/y)$.
genType **modf** (genType *x*, out genType *i*)	Returns the fractional part of *x* and sets *i* to the integer part (as a whole number floating point value). Both the return value and the output parameter will have the same sign as *x*.
genType **min** (genType *x*, genType *y*) genType **min** (genType *x*, float *y*) genIType **min** (genIType *x*, genIType *y*) genIType **min** (genIType *x*, int *y*) genUType **min** (genUType *x*, genUType *y*) genUType **min** (genUType *x*, uint *y*)	Returns *y* if $y < x$, otherwise it returns *x*.
genType **max** (genType *x*, genType *y*) genType **max** (genType *x*, float *y*) genIType **max** (genIType *x*, genIType *y*) genIType **max** (genIType *x*, int *y*) genUType **max** (genUType *x*, genUType *y*) genUType **max** (genUType *x*, uint *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*) genIType **clamp** (genIType *x*, genIType *minVal*, genIType *maxVal*) genIType **clamp** (genIType *x*, int *minVal*, int *maxVal*) genUType **clamp** (genUType *x*, genUType *minVal*, genUType *maxVal*) genUType **clamp** (genUType *x*, uint *minVal*, uint *maxVal*)	Returns **min** (**max** (*x*, *minVal*), *maxVal*). Results are undefined if $minVal > maxVal$.

Syntax	Description
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 **mix** (genType *x*, genType y, bvec a)	Selects which vector each returned component comes from. For a component of *a* that is **false**, the corresponding component of *x* is returned. For a component of *a* that is **true**, the corresponding component of *y* is returned. Components of *x* and *y* that are not selected are allowed to be invalid floating point values and will have no effect on the results. Thus, this provides different functionality than genType **mix**(genType *x*, genType *y*, genType(*a*)) where *a* is a Boolean vector.
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.*
bvec **isnan** (genType *x*)	Returns **true** if *x* holds a NaN (not a number) representation in the underlying implementation's set of floating point representations. Returns **false** otherwise, including for implementations with no NaN representations.
bvec **isinf** (genType *x*)	Returns **true** if *x* holds a positive infinity or negative infinity representation in the underlying implementation's set of floating point representations. Returns **false** otherwise, including for implementations with no infinity representations.

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**()	Deprecated; use **invariant**. 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 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**, "uvec" is a placeholder for **uvec2**, **uvec3**, or **uvec4**, 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) bvec **lessThan**(uvec x, uvec y)	Returns the component-wise compare of *x* < *y*.
bvec **lessThanEqual**(vec x, vec y) bvec **lessThanEqual**(ivec x, ivec y) bvec **lessThanEqual**(uvec x, uvec y)	Returns the component-wise compare of *x* <= *y*.
bvec **greaterThan**(vec x, vec y) bvec **greaterThan**(ivec x, ivec y) bvec **greaterThan**(uvec x, uvec y)	Returns the component-wise compare of *x* > *y*.
bvec **greaterThanEqual**(vec x, vec y) bvec **greaterThanEqual**(ivec x, ivec y) bvec **greaterThanEqual**(uvec x, uvec y)	Returns the component-wise compare of *x* >= *y*.
bvec **equal**(vec x, vec y) bvec **equal**(ivec x, ivec y) bvec **equal**(uvec x, uvec y) bvec **equal**(bvec x, bvec y) bvec **notEqual**(vec x, vec y) bvec **notEqual**(ivec x, ivec y) bvec **notEqual**(uvec x, uvec 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 implicitly computed for vertex shaders. 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.

Texture data can be stored by the GL as floating point, unsigned normalized integer, unsigned integer or signed integer data. This is determined by the type of the internal format of the texture. Texture lookups on unsigned normalized integer and floating point data return floating point values in the range [0, 1].

Texture lookup functions are provided that can return their result as floating point, unsigned integer or signed integer, depending on the sampler type passed to the lookup function. Care must be taken to use the right sampler type for texture access. The following table lists the supported combinations of sampler types and texture internal formats. Blank entries are unsupported. Doing a texture lookup will return undefined values for unsupported combinations.

Internal Texture Format	Floating Point Sampler Types	Signed Integer Sampler Types	Unsigned Integer Sampler Types
Floating point	Supported		
Normalized Integer	Supported		
Signed Integer		Supported	
Unsigned Integer			Supported

If an integer sampler type is used, the result of a texture lookup is an **ivec4**. If an unsigned integer sampler type is used, the result of a texture lookup is a **uvec4**. If a floating point sampler type is used, the result of a texture lookup is a **vec4**, where each component is in the range [0, 1].

In the prototypes below, the "*g*" in the return type "*gvec4*" is used as a placeholder for nothing, "*i*", or "*u*" making a return type of **vec4**, **ivec4**, or **uvec4**. In these cases, the sampler argument type also starts with "*g*", indicating the same substitution done on the return type; it is either a floating point, signed integer, or unsigned integer sampler, matching the basic type of the return type, as described above.

For shadow forms (the sampler parameter is a shadow-type), a depth comparison lookup on the depth texture bound to *sampler* is done as described in section 3.9.14 of the OpenGL Graphics System Specification, Version 3.0. See the table below for which component specifies D_{ref}. The texture bound to *sampler* must be a depth texture, or results are undefined. 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 implicit level of detail prior to performing the texture access operation.

The implicit level of detail is selected as follows: 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.

Some texture functions (non-"**Lod**" and non-"**Grad**" versions) may require implicit derivatives. Implicit derivatives are undefined within non-uniform control flow and for vertex shader texture fetches.

For **Cube** forms, the direction of *P* is used to select which face to do a 2-dimensional texture lookup in, as described in section 3.9.6 in the OpenGL Graphics System Specification, Version 3.0.

For **Array** forms, the array layer used will be

$$max(0, min(d-1, floor(layer+0.5)))$$

where *d* is the depth of the texture array and *layer* comes from the component indicated in the tables below.

Syntax	Description
int **textureSize** (gsampler1D *sampler*, int *lod*) ivec2 **textureSize** (gsampler2D *sampler*, int *lod*) ivec3 **textureSize** (gsampler3D *sampler*, int *lod*) ivec2 **textureSize** (gsamplerCube *sampler*, int *lod*) int **textureSize** (sampler1DShadow *sampler*, int *lod*) ivec2 **textureSize** (sampler2DShadow *sampler*, int *lod*) ivec2 **textureSize** (samplerCubeShadow *sampler*, int *lod*) ivec2 **textureSize** (gsampler1DArray *sampler*, int *lod*) ivec3 **textureSize** (gsampler2DArray *sampler*, int *lod*) ivec2 **textureSize** (sampler1DArrayShadow *sampler*, int *lod*) ivec3 **textureSize** (sampler2DArrayShadow *sampler*, int *lod*)	Returns the dimensions of level *lod* for the texture bound to *sampler*, as described in section 2.20.4 of the OpenGL Graphics System Specification, Version 3.0, under "Texture Size Query". The components in the return value are filled in, in order, with the width, height, depth of the texture. For the array forms, the last component of the return value is the number of layers in the texture array.
gvec4 **texture** (gsampler1D *sampler,* float *P* [, float *bias*]) gvec4 **texture** (gsampler2D *sampler*, vec2 *P* [, float *bias*]) gvec4 **texture** (gsampler3D *sampler*, vec3 *P* [, float *bias*]) gvec4 **texture** (gsamplerCube *sampler*, vec3 *P* [, float *bias*]) float **texture** (sampler1DShadow *sampler,* vec3 *P* [, float *bias*]) float **texture** (sampler2DShadow *sampler,* vec3 *P* [, float *bias*]) float **texture** (samplerCubeShadow *sampler,* vec4 *P* [, float *bias*]) gvec4 **texture** (gsampler1DArray *sampler,* vec2 *P* [, float *bias*]) gvec4 **texture** (gsampler2DArray *sampler*, vec3 *P* [, float *bias*]) float **texture** (sampler1DArrayShadow *sampler,* vec3 *P* [, float *bias*]) float **texture** (sampler2DArrayShadow *sampler,* vec4 *P*)	Use the texture coordinate *P* to do a texture lookup in the texture currently bound to *sampler*. The last component of *P* is used as D_{ref} for the shadow forms. For array forms, the array layer comes from the last component of *P* in the non-shadow forms, and the second to last component of *P* in the shadow forms.
gvec4 **textureProj** (gsampler1D *sampler*, vec2 *P* [, float *bias*]) gvec4 **textureProj** (gsampler1D *sampler*, vec4 *P* [, float *bias*]) gvec4 **textureProj** (gsampler2D *sampler*, vec3 *P* [, float *bias*]) gvec4 **textureProj** (gsampler2D *sampler*, vec4 *P* [, float *bias*]) gvec4 **textureProj** (gsampler3D *sampler*, vec4 *P* [, float *bias*]) float **textureProj** (sampler1DShadow *sampler,* vec4 *P* [, float *bias*]) float **textureProj** (sampler2DShadow *sampler,* vec4 *P* [, float *bias*])	Do a texture lookup with projection. The texture coordinates consumed from *P*, not including the last component of *P*, are divided by the last component of *P*. The resulting 3rd component of *P* in the shadow forms is used as D_{ref}. After these values are computed, texture lookup proceeds as in **texture**.

Syntax	Description
gvec4 **textureLod** (gsampler1D *sampler*, float *P*, float *lod*) gvec4 **textureLod** (gsampler2D *sampler*, vec2 *P*, float *lod*) gvec4 **textureLod** (gsampler3D *sampler*, vec3 *P*, float *lod*) gvec4 **textureLod** (gsamplerCube *sampler*, vec3 *P*, float *lod*) float **textureLod** (sampler1DShadow *sampler,* vec3 *P*, float *lod*) float **textureLod** (sampler2DShadow *sampler,* vec3 *P*, float *lod*) gvec4 **textureLod** (gsampler1DArray *sampler*, vec2 *P*, float *lod*) gvec4 **textureLod** (gsampler2DArray *sampler*, vec3 *P*, float *lod*) float **textureLod** (sampler1DArrayShadow *sampler,* vec3 *P*, float *lod*)	Do a texture lookup as in **texture** but with explicit LOD; *lod* specifies λ_{base} (see equation 3.16 in OpenGL Graphics System Specification, Version 3.0) and set the partial derivatives in section 3.9.7 as follows. $\frac{\partial u}{\partial x} = 0 \quad \frac{\partial v}{\partial x} = 0 \quad \frac{\partial w}{\partial x} = 0$ $\frac{\partial u}{\partial y} = 0 \quad \frac{\partial v}{\partial y} = 0 \quad \frac{\partial w}{\partial y} = 0$
gvec4 **textureOffset** (gsampler1D *sampler,* float *P,* int *offset* [, float *bias*]) gvec4 **textureOffset** (gsampler2D *sampler*, vec2 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureOffset** (gsampler3D *sampler*, vec3 *P*, ivec3 *offset* [, float *bias*]) float **textureOffset** (sampler1DShadow *sampler,* vec3 *P*, int *offset* [, float *bias*]) float **textureOffset** (sampler2DShadow *sampler,* vec3 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureOffset** (gsampler1DArray *sampler,* vec2 *P,* int *offset* [, float *bias*]) gvec4 **textureOffset** (gsampler2DArray *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]) float **textureOffset** (sampler1DArrayShadow *sampler,* vec3 *P*, int *offset* [, float *bias*])	Do a texture lookup as in **texture** but with *offset* added to the (u,v,w) texel coordinates before looking up each texel. The offset value must be a constant expression. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by MIN_PROGRAM_TEXEL_OFFSET and MAX_PROGRAM_TEXEL_OFFSET, respectively. Note that *offset* does not apply to the layer coordinate for texture arrays. This is explained in detail in section 3.9.7 of the OpenGL Graphics System Specification, Version 3.0, where *offset* is $(\delta_u, \delta_v, \delta_w)$. Note that texel offsets are also not supported for cube maps.

Syntax	Description
gvec4 **texelFetch** (gsampler1D *sampler*, int *P*, int *lod*) gvec4 **texelFetch** (gsampler2D *sampler*, ivec2 *P*, int *lod*) gvec4 **texelFetch** (gsampler3D *sampler*, ivec3 *P*, int *lod*) gvec4 **texelFetch** (gsampler1DArray *sampler*, ivec2 *P*, int *lod*) gvec4 **texelFetch** (gsampler2DArray *sampler*, ivec3 *P*, int *lod*)	Use integer texture coordinate *P* to lookup a single texel from *sampler*. The array layer comes from the last component of *P* for the array forms. The level-of-detail *lod* as described in section 2.20.4 of the OpenGL Graphics System Specification, Version 3.0, under "Texel Fetches".
gvec4 **texelFetchOffset** (gsampler1D *sampler*, int *P*, int *lod*, int *offset*) gvec4 **texelFetchOffset** (gsampler2D *sampler*, ivec2 *P*, int *lod*, ivec2 *offset*) gvec4 **texelFetchOffset** (gsampler3D *sampler*, ivec3 *P*, int *lod*, ivec3 *offset*) gvec4 **texelFetchOffset** (gsampler1DArray *sampler*, ivec2 *P*, int *lod*, int *offset*) gvec4 **texelFetchOffset** (gsampler2DArray *sampler*, ivec3 *P*, int *lod*, ivec2 *offset*)	Fetch a single texel as in **texelFetch** offset by *offset* as described in **textureOffset**.
gvec4 **textureProjOffset** (gsampler1D *sampler,* vec2 *P,* int *offset* [, float *bias*]) gvec4 **textureProjOffset** (gsampler1D *sampler,* vec4 *P,* int *offset* [, float *bias*]) gvec4 **textureProjOffset** (gsampler2D *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureProjOffset** (gsampler2D *sampler*, vec4 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureProjOffset** (gsampler3D *sampler*, vec4 *P*, ivec3 *offset* [, float *bias*]) float **textureProjOffset** (sampler1DShadow *sampler,* vec4 *P*, int *offset* [, float *bias*]) float **textureProjOffset** (sampler2DShadow *sampler,* vec4 *P*, ivec2 *offset* [, float *bias*])	Do a projective texture lookup as described in **textureProj** offset by *offset* as described in **textureOffset**.

Syntax	Description
gvec4 **textureLodOffset** (gsampler1D *sampler,* float *P*, float *lod*, int *offset*) gvec4 **textureLodOffset** (gsampler2D *sampler*, vec2 *P*, float *lod*, ivec2 *offset*) gvec4 **textureLodOffset** (gsampler3D *sampler*, vec3 *P*, float *lod*, ivec3 *offset*) float **textureLodOffset** (sampler1DShadow *sampler,* vec3 *P*, float *lod*, int *offset*) float **textureLodOffset** (sampler2DShadow *sampler,* vec3 *P*, float *lod*, ivec2 *offset*) gvec4 **textureLodOffset** (gsampler1DArray *sampler,* vec2 *P*, float *lod*, int *offset*) gvec4 **textureLodOffset** (gsampler2DArray *sampler*, vec3 *P*, float *lod*, ivec2 *offset*) float **textureLodOffset** (sampler1DArrayShadow *sampler,* vec3 *P*, float *lod*, int *offset*)	Do an offset texture lookup with explicit LOD. See **textureLod** and **textureOffset**.
gvec4 **textureProjLod** (gsampler1D *sampler*, vec2 *P*, float *lod*) gvec4 **textureProjLod** (gsampler1D *sampler*, vec4 *P*, float *lod*) gvec4 **textureProjLod** (gsampler2D *sampler*, vec3 *P*, float *lod*) gvec4 **textureProjLod** (gsampler2D *sampler*, vec4 *P*, float *lod*) gvec4 **textureProjLod** (gsampler3D *sampler*, vec4 *P*, float *lod*) float **textureProjLod** (sampler1DShadow *sampler,* vec4 *P*, float *lod*) float **textureProjLod** (sampler2DShadow *sampler,* vec4 *P,* float *lod*)	Do a projective texture lookup with explicit LOD. See **textureProj** and **textureLod**.
gvec4 **textureProjLodOffset** (gsampler1D *sampler,* vec2 *P*, float *lod*, int *offset*) gvec4 **textureProjLodOffset** (gsampler1D *sampler,* vec4 *P*, float *lod*, int *offset*) gvec4 **textureProjLodOffset** (gsampler2D *sampler*, vec3 *P*, float *lod*, ivec2 *offset*) gvec4 **textureProjLodOffset** (gsampler2D *sampler*, vec4 *P*, float *lod*, ivec2 *offset*) gvec4 **textureProjLodOffset** (gsampler3D *sampler*, vec4 *P*, float *lod*, ivec3 *offset*) float **textureProjLodOffset** (sampler1DShadow *sampler,* vec4 *P*, float *lod*, int *offset*) float **textureProjLodOffset** (sampler2DShadow *sampler,* vec4 *P*, float *lod*, ivec2 *offset*)	Do an offset projective texture lookup with explicit LOD. See **textureProj**, **textureLod**, and **textureOffset**.

Syntax	Description
gvec4 **textureGrad** (gsampler1D *sampler,* float *P*, float *dPdx*, float *dPdy*) gvec4 **textureGrad** (gsampler2D *sampler,* vec2 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureGrad** (gsampler3D *sampler,* vec3 *P*, vec3 *dPdx*, vec3 *dPdy*) gvec4 **textureGrad** (gsamplerCube *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy*) float **textureGrad** (sampler1DShadow *sampler*, vec3 *P*, float *dPdx*, float *dPdy*) float **textureGrad** (sampler2DShadow *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) float **textureGrad** (samplerCubeShadow *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*) gvec4 **textureGrad** (gsampler1DArray *sampler,* vec2 *P*, float *dPdx*, float *dPdy*) gvec4 **textureGrad** (gsampler2DArray *sampler,* vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) float **textureGrad** (sampler1DArrayShadow *sampler*, vec3 *P*, float *dPdx*, float *dPdy*) float **textureGrad** (sampler2DArrayShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*)	Do a texture lookup as in **texture** but with explicit gradients. The partial derivatives of *P* are with respect to window x and window y. Set $\frac{\partial s}{\partial x} = \begin{cases} \frac{\partial P}{\partial x} & \text{for a 1D texture} \\ \frac{\partial P.s}{\partial x} & \text{otherwise} \end{cases}$ $\frac{\partial s}{\partial y} = \begin{cases} \frac{\partial P}{\partial y} & \text{for a 1D texture} \\ \frac{\partial P.s}{\partial y} & \text{otherwise} \end{cases}$ $\frac{\partial t}{\partial x} = \begin{cases} 0.0 & \text{for a 1D texture} \\ \frac{\partial P.t}{\partial x} & \text{otherwise} \end{cases}$ $\frac{\partial t}{\partial y} = \begin{cases} 0.0 & \text{for a 1D texture} \\ \frac{\partial P.t}{\partial y} & \text{otherwise} \end{cases}$ $\frac{\partial r}{\partial x} = \begin{cases} 0.0 & \text{for 1D or 2D} \\ \frac{\partial P.p}{\partial x} & \text{cube, other} \end{cases}$ $\frac{\partial r}{\partial y} = \begin{cases} 0.0 & \text{for 1D or 2D} \\ \frac{\partial P.p}{\partial y} & \text{cube, other} \end{cases}$ For the cube version, the partial derivatives of *P* are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face.

Syntax	Description
gvec4 **textureGradOffset** (gsampler1D *sampler,* float *P,* float *dPdx*, float *dPdy,* int *offset*) gvec4 **textureGradOffset** (gsampler2D *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy,* ivec2 *offset*) gvec4 **textureGradOffset** (gsampler3D *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy,* ivec3 *offset*) float **textureGradOffset** (sampler1DShadow *sampler,* vec3 *P*, float *dPdx*, float *dPdy,* int *offset*) float **textureGradOffset** (sampler2DShadow *sampler,* vec3 *P*, vec2 *dPdx*, vec2 *dPdy,* ivec2 *offset*) float **textureGradOffset** (samplerCubeShadow *sampler,* vec4 *P*, vec3 *dPdx*, vec3 *dPdy,* ivec2 *offset*) gvec4 **textureGradOffset** (gsampler1DArray *sampler,* vec2 *P,* float *dPdx*, float *dPdy,* int *offset*) gvec4 **textureGradOffset** (gsampler2DArray *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy,* ivec2 *offset*) float **textureGradOffset** (sampler1DArrayShadow *sampler,* vec3 *P*, float *dPdx*, float *dPdy,* int *offset*) float **textureGradOffset** (sampler2DArrayShadow *sampler,* vec4 *P*, vec2 *dPdx*, vec2 *dPdy,* ivec2 *offset*)	Do a texture lookup with both explicit gradient and offset, as described in **textureGrad** and **textureOffset**.
gvec4 **textureProjGrad** (gsampler1D *sampler,* vec2 *P*, float *dPdx*, float *dPdy*) gvec4 **textureProjGrad** (gsampler1D *sampler,* vec4 *P*, float *dPdx*, float *dPdy*) gvec4 **textureProjGrad** (gsampler2D *sampler,* vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureProjGrad** (gsampler2D *sampler,* vec4 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureProjGrad** (gsampler3D *sampler,* vec4 *P*, vec3 *dPdx*, vec3 *dPdy*) float **textureProjGrad** (sampler1DShadow *sampler*, vec4 *P*, float *dPdx*, float *dPdy*) float **textureProjGrad** (sampler2DShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*)	Do a texture lookup both projectively, as described in **textureProj**, and with explicit gradient as described in **textureGrad**. The partial derivatives *dPdx* and *dPdy* are assumed to be already projected.

Syntax	Description
gvec4 **textureProjGradOffset** (gsampler1D *sampler,* vec2 *P*, float *dPdx*, float *dPdy*, int *offset*) gvec4 **textureProjGradOffset** (gsampler1D *sampler,* vec4 *P*, float *dPdx*, float *dPdy*, int *offset*) gvec4 **textureProjGradOffset** (gsampler2D *sampler,* vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, vec2 *offset*) gvec4 **textureProjGradOffset** (gsampler2D *sampler,* vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, vec2 *offset*) gvec4 **textureProjGradOffset** (gsampler3D *sampler,* vec4 *P*, vec3 *dPdx*, vec3 *dPdy*, vec3 *offset*) float **textureProjGradOffset** (sampler1DShadow *sampler*, vec4 *P*, float *dPdx*, float *dPdy*, int *offset*) float **textureProjGradOffset** (sampler2DShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, vec2 *offset*)	Do a texture lookup projectively and with explicit gradient as described in **textureProjGrad**, as well as with offset, as described in **textureOffset**.

The following texture functions are deprecated.

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*)	Deprecated. See corresponding signature above without "1D" in the name.
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*)	Deprecated. See corresponding signature above without "2D" in the name.
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*)	Deprecated. See corresponding signature above without "3D" in the name. 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*)	Deprecated. See corresponding signature above without "Cube" in the name.

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*)	Deprecated. Same functionality as the "**texture"** based names above with the same signature.

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. Derivatives are undefined within non-uniform control flow.

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

Forward differencing:

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

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

Backward differencing:

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

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

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 3.2 of the OpenGL Graphics System Specification, Version 3.0, 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 Graphics System Specification, Version 3.0), 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 UINT
BREAK CONTINUE DO ELSE FOR IF DISCARD RETURN SWITCH CASE DEFAULT
BVEC2 BVEC3 BVEC4 IVEC2 IVEC3 IVEC4 UVEC2 UVEC3 UVEC4 VEC2 VEC3 VEC4
MAT2 MAT3 MAT4 CENTROID IN OUT INOUT UNIFORM VARYING

NOPERSPECTIVE FLAT SMOOTH
MAT2X2 MAT2X3 MAT2X4
MAT3X2 MAT3X3 MAT3X4
MAT4X2 MAT4X3 MAT4X4
SAMPLER1D SAMPLER2D SAMPLER3D SAMPLERCUBE SAMPLER1DSHADOW SAMPLER2DSHADOW
SAMPLERCUBESHADOW SAMPLER1DARRAY SAMPLER2DARRAY SAMPLER1DARRAYSHADOW
SAMPLER2DARRAYSHADOW ISAMPLER1D ISAMPLER2D ISAMPLER3D ISAMPLERCUBE
ISAMPLER1DARRAY ISAMPLER2DARRAY USAMPLER1D USAMPLER2D USAMPLER3D
USAMPLERCUBE USAMPLER1DARRAY USAMPLER2DARRAY
STRUCT VOID WHILE

IDENTIFIER TYPE_NAME FLOATCONSTANT INTCONSTANT UINTCONSTANT 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
HIGH_PRECISION MEDIUM_PRECISION LOW_PRECISION PRECISION
```

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

variable_identifier:
    IDENTIFIER

primary_expression:
    variable_identifier
    INTCONSTANT

UINTCONSTANT
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

// 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

additive_expression:
multiplicative_expression
additive_expression PLUS multiplicative_expression
additive_expression DASH multiplicative_expression

shift_expression:
additive_expression
shift_expression LEFT_OP additive_expression
shift_expression RIGHT_OP additive_expression

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

exclusive_or_expression:
and_expression
exclusive_or_expression CARET and_expression

inclusive_or_expression:
exclusive_or_expression
inclusive_or_expression VERTICAL_BAR exclusive_or_expression

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
ADD_ASSIGN
SUB_ASSIGN
LEFT_ASSIGN
RIGHT_ASSIGN
AND_ASSIGN
XOR_ASSIGN
OR_ASSIGN

expression:
assignment_expression
expression COMMA assignment_expression

constant_expression:
conditional_expression

declaration:
function_prototype SEMICOLON
init_declarator_list SEMICOLON
PRECISION precision_qualifier type_specifier_no_prec 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:
parameter_type_qualifier parameter_qualifier parameter_declarator
parameter_qualifier parameter_declarator
parameter_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

invariant_qualifier:
INVARIANT

interpolation_qualifier:
SMOOTH
FLAT
NOPERSPECTIVE

parameter_type_qualifier:
CONST

type_qualifier:
storage_qualifier
interpolation_qualifier type_qualifier
invariant_qualifier type_qualifier
invariant_qualifier interpolation_qualifier type_qualifier

storage_qualifier:
CONST
ATTRIBUTE // Vertex only.
VARYING
CENTROID VARYING
IN
OUT
CENTROID IN
CENTROID OUT

UNIFORM

type_specifier:
type_specifier_no_prec
precision_qualifier type_specifier_no_prec

type_specifier_no_prec:
type_specifier_nonarray
type_specifier_nonarray LEFT_BRACKET RIGHT_BRACKET
type_specifier_nonarray LEFT_BRACKET constant_expression RIGHT_BRACKET

type_specifier_nonarray:
VOID
FLOAT
INT
UINT
BOOL
VEC2
VEC3
VEC4
BVEC2
BVEC3
BVEC4
IVEC2
IVEC3
IVEC4
UVEC2
UVEC3
UVEC4
MAT2
MAT3
MAT4
MAT2X2
MAT2X3
MAT2X4
MAT3X2
MAT3X3
MAT3X4

MAT4X2
MAT4X3
MAT4X4
SAMPLER1D
SAMPLER2D
SAMPLER3D
SAMPLERCUBE
SAMPLER1DSHADOW
SAMPLER2DSHADOW
SAMPLERCUBESHADOW
SAMPLER1DARRAY
SAMPLER2DARRAY
SAMPLER1DARRAYSHADOW
SAMPLER2DARRAYSHADOW
ISAMPLER1D
ISAMPLER2D
ISAMPLER3D
ISAMPLERCUBE
ISAMPLER1DARRAY
ISAMPLER2DARRAY
USAMPLER1D
USAMPLER2D
USAMPLER3D
USAMPLERCUBE
USAMPLER1DARRAY
USAMPLER2DARRAY
struct_specifier
TYPE_NAME

precision_qualifier:
HIGH_PRECISION
MEDIUM_PRECISION
LOW_PRECISION

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: labeled statements for SWITCH only; 'goto' is not supported.

simple_statement:
declaration_statement
expression_statement
selection_statement
switch_statement
case_label
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

condition:
expression
fully_specified_type IDENTIFIER EQUAL initializer

switch_statement:
SWITCH LEFT_PAREN expression RIGHT_PAREN LEFT_BRACE switch_statement_list RIGHT_BRACE

switch_statement_list:
/ nothing */*
statement_list

case_label:
CASE expression COLON
DEFAULT COLON

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

2. How do we do flat shading?

   Alternate Resolution: Add **flat** qualifier for varyings.

   Alternate Resolution: Add **flat** qualifier for varyings that obeys shade mode.

   Resolution: Add **flat**, but restrict it to gl_*color varyings with the same semantics as flat in 2.x. In the API, there is immutable state in the program object that sets the default. Also add **smooth** to the language for overriding when the default is **flat**.

3. What is the complete list of fundamental data types transparently exchangeable?

   Alternate Resolution: Just floating point types.

   Resolution: float, int, and bool.

4. Are there any cache flushing issues around writing to bound partitions? -> See issue 9.

5. Complex matrix -> matrix constructors.

   Resolution: Defer.

6. How do we do two-sided coloring? This interacts with clamping, because fixed hardware dealing with this would also clamp.

   No language impact.

   Resolution: A two-sided enable goes into program object as immutable state. It is not a link error if one-sided is enabled and back-color is written. All other mis-matches are just undefined.

7. Do we do clamping? How is this determined?

   No language impact.

   Resolution: Have immutable state in the program object that says whether color input to the rasterizer is clamped. Add another for fragment output.

9. What about cache flushing common blocks? Is it bound during writes? Do we expect all shader hardware to have to write to the same resource shaders read common blocks from?

   We might expect separate resources. But, this is part of a much bigger problem that needs to be resolved in the API.

   Alternative 1: Implementations use API calls like draw to be responsible for implicitly refreshing the resource.

   Alternative 2: Add an explicit call to say you are done changing it.

   Resolution: See API spec. resolution.

10. Problem with built-in texture function names, how do we reduce?

    - we will at least drop the texture type from the function name

- consider using named parameters

- consider using programs to specify the lookup

11. Do we want application named fragment-shader outputs.

Resolution: Yes.

12. Should we switch to application-assigned slots for the varying linkage between vertex and fragment shaders?

Alternate Resolution: No. Keep doing it the way 2.x does it.

Resolution: This is an API issue. See API spec.

13. Can we query declared varyings?

Resolution: Yes. This is an API issue. See API spec.

14. Are there any sRGB impacts to the language?

Resolution: No. The shader pipe assumes linear space. sRGB is in filtering state going into the shader and in output space after leaving the shader.

15. How do you declare a **flat varying**? We said the only variables that can be flat are "gl_*color", but those are already declared.

Resolution: Just like **invariant**, allow a built-in variable to be redeclared as long as the redeclaration just adds/changes the qualifiers. Note we also allow redeclarations to give an array explicit size.

-> don't allow bad redeclarations of built-ins

16. Should allow variations in the value of a flat varying across a triangle? (For the case where real interpolators are still used and might not get exactly the same result for each pixel.)

Resolution: No. The value is invariant across the triangle.

17. Do precision qualifiers have to match? If the intention is to aid porting, not really save space/power, it is actually more helpful if they don't have to match like they do in the ES spec.

Alternate Resolution: No. Allow precision mismatches. This would imply all precisions are interchangeable, hence we could formalize that and just say the language recognizes the keywords, but they don't have any meaning. This means the specification is very simple and easy to understand.

-> get the most recent ES spec to see what it does.

Resolution: precision qualifiers have no semantics.

18. Does an application running on mixed hardware have the same layout for a common block for all shaders the common block is in? Different contexts can be for different adapters.

Resolution: Offsets could be different on different hardware. But, if they are successfully created, then the offsets will match, and common blocks can be shared. Future could consider things like “packed common” or “vec4 common” ...

19. Do we support initializers inside of common blocks? How does the API handle them? We already chose the language syntax that makes it easy to provide.

Resolution: Yes. Face-to-face strawpoll: for 9, against 3, abstain 5.

21. Does flat/smooth/default have to match across stages?

    Alternate Resolution A. No. Exclude flat/smooth from other stage's languages, and say it's really a fragment shader control. Problem is that depends on implementation and where the interpolation happens.

    Alternate Resolution B. It is interfaces whose sides have to match, not shaders. The output of one stage has to match the input of the next stage. The input to a stage does not have to match the output of that stage. Semantics have been, but can be more clearly stated as: the variable the shader is manipulating was copied in from the input interface at the beginning and copied out to the output interface at the end.

    Alternate Resolution C. Build on B, take inout and in and out into account, with it's benefit of allowing reuse of names. That means allowing both “in color” and “out color” in the same shader to mean the same thing as “inout color”. Then, they can be decorated differently “in color” and “out flat color” without contradiction because the decorations are modifying the interface, not the variable the shader is manipulating. Note that even without this issue we need to say what happens with either “inout color” or “in color; out color”. See issue 24.

    Resolution: Interfaces match. Inputs and outputs within a shader do not have to match.

22. Does invariant still have to match across stages?

    Resolution: Yes.

23. Do we want to get rid of gl_DepthRange? Getting rid of it only really makes sense if it's the only remaining state that has to be tracked.

    Resolution: No, keep it.

24. If a shader declares “inout color” and then does not use the variable color (but the previous and/or following stage did use color), what happens?

    Alternate Resolution. There is an implicit copy from the input color to the output color. Note this is a corollary anyway of the idea that input variables are copied in at start of shader and copied out to the output at the end of the shader. Note the interaction with this and basing set of varyings off of declarations vs. static or active use.

    Resolution: Remove inout. This also fixes interpolation qualifier ambiguities.

25. What API mechanism do we use to do #include.

    Alternate Resolution A: Simple database of names mapped to include strings on the server in the context. Shader has includes that use those names, and compile time does string substitutions from the simple database.

    Alternate Resolution B: Rob's proposal. Similar to A, but strings are stored as “text objects”. More differences... see Rob's email.

    Alternative Resolution C: Pass in each time a mapping of names to strings that's used by that compile.

    Alternative Resolution D:

    Alternative Resolution E: No #include.

    Alternative Resolution F: Re-use shader objects: give them names, include by name. Requires changing the semantics of shader object to not parse shaders that have a name. Shaders that are the roots of compile cannot have names.

    Resolution: This is handled by the API spec., through new objects for storing text strings.

26. Do we want to restrict **switch** to something simpler than C?

Resolution: Yes. No **case** inside flow control.

27. Can precision qualifiers be specified to do nothing instead of ES functionality?

Resolution: leave ES-like functionality to extensions.

30. Are diagnostics for included files identifying locations as a triple (name string, string number, line number) or a pair (name string, line number). The implication being that the API sets the included text either as an array of strings or a single string. What does __FILE__ return

Resolution: As a pair. __FILE__ can return a string or a number, depending on the context.

32. Are we allowing bool/int in common blocks now, just like we do for the default partition?

Resolution: Yes, put bool/int back into common blocks.

35. Do we want to say "centroid in" or "in centroid", and similarly for smooth and flat? May have implications to expressing aliasing of varying with longer chains of qualifiers.

Resolution: "centroid in"

36. What packing rules go in the language spec.?

Resolution: It depends on whether future mechanisms for influencing packing are in the API or in the language. For now, the language spec. will specify the packing rules if a common block contains nothing but vec4s.

37. Are we removing user clip planes? Need to remove gl_ClipVertex.

Resolution: No. For 1.30, keep gl_ClipVertex with it's previous semantics. If gl_ClipVertex is not statically written, then gl_Position is used instead for clipping against user planes if they've been enabled. Also, add gl_ClipDistance.

38. Do we need to add any guarantees of precision of pass through data? Like 16 bits of color?

Resolution: Deferred.

39. int(uint) and uint(int) are not conversions, they are reinterprets. This is a poor consistency in the language.

Resolution: These are constructors, not type casts, which can do what they want. In this case they reinterpret, even though in other cases they convert.

40. How should the number of clip distances be set? How does this relate to API side enables? How does it relate to sizing the gl_ClipDistance array in the language?

Resolution: Use enables.

41. Do we need to specify the precision of integer division?

42. Is the API adding support for #include? If not, remove from language.

Resolution: Remove.

43. Some *gl_VertexID* description seems like it does not belong in the language spec.

Done.

44. What needs to be said about transform feedback of built-in vertex outputs?

Resolution: Nothing.

45. Still need to finish some ES synchronization (a set of pretty small details).

46. Lacking a partition scheme that's memory efficient for the case of lots of inactive uniforms declared in a common block (partition).

Resolution: This is unnecessary. Having just the default partition (non-common globals) support this space optimization is sufficient, as it is not for sharing between programs and offsets always have to be queried.

47. Need more clarity on the effects of **uint** vs **int**. It appears to allow binary operations between an int and a uint, but doesn't really say what happens.

Resolution: No implicit conversions; some operators are defined to take mixed operands, most are not.

48. List deprecated features in one place.

Resolution. Yes.

49. Should the token-pasting language say that that results of the token pasting are then further processed by the preprocessor?

Resolution: Yes.

50. flat, noperspective, centroid, smooth axes, combinations, etc. Is there an issue here?

51. broader support for mixing signed and unsigned in an expression

dup. of 47

52. deprecating gl_FragColor deprecates broadcast. Is that okay?

Resolution: Yes.

53. What does "Not having shaders for programming all programmable stages..." mean to commands like DrawPixels that today run a fragment shader without a vertex shader. Do you have to make up a dummy vertex shader? Of course, with fixed-function vertex attributes (and thus RasterPos attributes) gone, it seems like DrawPixels (and Bitmap) as we know and love (?) it today would have to go. Not having seen a list of deprecated features for OpenGL 3.0, I have no idea if it's already on the list.

Resolution: All programmable stages that are being used must have shaders provided for those stages. (Don't mention programs.)

54. Should we remove row_major from the language.

Resolution: Yes.

55. Should we add gl_ClipDistance as input to the fragment shader.

Resolution: Yes.

56. Don't say 1.2 and 1.1 are not accepted.

Resolution: Yes, accept 1.1 and 1.2.

57. Do we have user-defined flat variables?

Resolution: Yes. Fix spec. inconsistency. Only way to have integer varyings.

58. Should we really remove functionality instead of deprecate it?

Resolution: Deprecate before remove. Read API spec. appendix.

59. Can a global out redeclare the same name as a global in?

Resolution: No.