The OpenGL® Shading Language, Version 4.60.5

John Kessenich, Google (Editor and Author) ; Dave Baldwin and Randi Rost (Version 1.1 Authors)

Version 4.60.5, Fri, 11 May 2018 16:48:35 +0000

Table of Contents

1. Introduction . . . 2
   1.1. Changes . . . 2
   1.2. Overview . . . 3
   1.3. Error Handling . . . 3
   1.4. Typographical Conventions . . . 4
   1.5. Deprecation . . . 4
2. Overview of OpenGL Shading . . . 5
   2.1. Vertex Processor . . . 5
   2.2. Tessellation Control Processor . . . 5
   2.3. Tessellation Evaluation Processor . . . 6
   2.4. Geometry Processor . . . 6
   2.5. Fragment Processor . . . 6
   2.6. Compute Processor . . . 6
3. Basics . . . 8
   3.1. Character Set and Phases of Compilation . . . 8
   3.2. Source Strings . . . 9
   3.3. Preprocessor . . . 9
   3.4. Comments . . . 15
   3.5. Tokens . . . 16
   3.6. Keywords . . . 16
   3.7. Identifiers . . . 19
   3.8. Definitions . . . 19
4. Variables and Types . . . 22
   4.1. Basic Types . . . 22
   4.2. Scoping . . . 41
   4.3. Storage Qualifiers . . . 43
   4.4. Layout Qualifiers . . . 58
   4.5. Interpolation Qualifiers . . . 89
   4.6. Parameter Qualifiers . . . 91
   4.7. Precision and Precision Qualifiers . . . 91
   4.8. Variance and the Invariant Qualifier . . . 94
   4.9. The Precise Qualifier . . . 96
   4.10. Memory Qualifiers . . . 99
   4.11. Specialization-Constant Qualifier . . . 102
   4.12. Order and Repetition of Qualification . . . 103
   4.13. Empty Declarations . . . 104
5. Operators and Expressions . . . 105
   5.1. Operators . . . 105

5.2. Array Operations . . . 106
5.3. Function Calls . . . 106
5.4. Constructors . . . 106
5.5. Vector and Scalar Components and Length . . . 110
5.6. Matrix Components . . . 112
5.7. Structure and Array Operations . . . 113
5.8. Assignments . . . 114
5.9. Expressions . . . 115
5.10. Vector and Matrix Operations . . . 117
5.11. Out-of-Bounds Accesses . . . 119
5.12. Specialization Constant Operations . . . 119
6. Statements and Structure . . . 121
6.1. Function Definitions . . . 122
6.2. Selection . . . 128
6.3. Iteration . . . 129
6.4. Jumps . . . 129
7. Built-In Variables . . . 131
7.1. Built-In Language Variables . . . 131
7.2. Compatibility Profile Vertex Shader Built-In Inputs . . . 144
7.3. Built-In Constants . . . 144
7.4. Built-In Uniform State . . . 147
7.5. Redeclaring Built-In Blocks . . . 150
8. Built-In Functions . . . 152
8.1. Angle and Trigonometry Functions . . . 152
8.2. Exponential Functions . . . 153
8.3. Common Functions . . . 154
8.4. Floating-Point Pack and Unpack Functions . . . 159
8.5. Geometric Functions . . . 160
8.6. Matrix Functions . . . 162
8.7. Vector Relational Functions . . . 163
8.8. Integer Functions . . . 164
8.9. Texture Functions . . . 165
8.10. Atomic Counter Functions . . . 182
8.11. Atomic Memory Functions . . . 185
8.12. Image Functions . . . 186
8.13. Geometry Shader Functions . . . 189
8.14. Fragment Processing Functions . . . 190
8.15. Noise Functions . . . 193
8.16. Shader Invocation Control Functions . . . 194
8.17. Shader Memory Control Functions . . . 194
8.18. Shader Invocation Group Functions . . . 196

9. Shading Language Grammar 197
10. Acknowledgments 211
11. Normative References 212

Chapter 1. Introduction

This document specifies only version 4.60 of the OpenGL Shading Language. It requires __VERSION__ to substitute 460, and requires #version to accept only . If #version is declared with a smaller number, 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 Specification for details on what language versions are supported.

Previous versions of the OpenGL Shading Language, as well as the OpenGL ES Shading Language, are not strict subsets of the version specified here, particularly with respect to precision, name-hiding rules, and treatment of interface variables. See the specification corresponding to a particular language version for details specific to that version of the language.

The OpenGL API will specify the commands used to manipulate SPIR-V shaders. Independent offline tool chains will compile GLSL down to the SPIR-V intermediate language. SPIR-V generation is not enabled with a #extension, #version, or a profile. Instead, use of GLSL for SPIR-V is determined by offline tool-chain use. See the documentation of such tools to see how to request generation of SPIR-V for OpenGL.

> ! *editing-note*
>
> (Jon) The term ÒGLSLÓ is never actually defined in the document. We could replace all uses with OpenGL Shading Language, or define and use it consistently.

GLSL ! SPIR-V compilers must be directed as to what SPIR-V Capabilities are legal at run-time and give errors for GLSL feature use outside those capabilities. This is also true for implementation-dependent limits that can be error checked by the front-end against built-in constants present in the GLSL source: the front-end can be informed of such limits, and report errors when they are exceeded.

All references in this specification to the OpenGL Specification are to the Core profile of version 4.6, unless a different profile is specified.

1.1. Changes

1.1.1. Changes from Revision 4 of GLSL 4.6

- Public OpenGL-API issue 7: Variables can be declared as both readonly and writeonly.
- Private GLSL issue 32: Remove length() method contradiction: Non runtime-sized arrays only support length() on explicitly sized arrays.

1.1.2. Changes from Revision 3 of GLSL 4.6

- Private GLSL issue 13: Fix misspelling of allInvocationsEqual(). (The one in the table was incorrectly listed as anyInvocationsEqual(), other spellings were correct.)

1.1.3. Summary of Changes from Revision 7 of GLSL Version 4.50

- Incorporated the GL_ARB_shader_atomic_counter_ops extension.
- Incorporated the GL_ARB_shader_draw_parameters extension.
- Incorporated the GL_ARB_shader_group_vote extension.
- Incorporated the GL_ARB_gl_spirv extension.
- Private Bug 16070: Allow extra semi-colons at global scope.
- Private GLSL Issue 5: Be explicit that Òfail to linkÓ is really Òcompile-time or link-time errorÓ, for some forms of error.
- Private GLSL Issue 7: Change *gl_MaxComputeUniformComponents* to 1024.
- Private OpenGL API Issue 35: Require location on transparent individual uniform variables for SPIR-V.
- Private GLSL Issue 8: Be more clear an interpolateAt() interpolant can be a structure member.
- Private GLSL Issue 9: Specify how xfb_buffer interacts with a block array: the capturing buffer increments for each block array element.

1.2. Overview

This document describes *The OpenGL Shading Language, version 4.60*.

Independent compilation units written in this language are called *shaders*. A *program* is a set of shaders that are compiled and linked together, completely creating one or more of the programmable stages of the OpenGL pipeline. All the shaders for a single programmable stage must be within the same program. A complete set of programmable stages can be put into a single program or the stages can be partitioned across multiple programs. The aim of this document is to thoroughly specify the programming language. The OpenGL Specification will specify the OpenGL entry points used to manipulate and communicate with programs and shaders.

1.3. 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. Compile-time errors must be returned for lexically or grammatically incorrect shaders. Other errors are reported at compile time or link time as indicated. Code that is Òdead Ó must still be error checked. For example:

1.4. 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 ÒShading Language GrammarÓ uses all capitals for terminals and lower case for non-terminals.

1.5. Deprecation

The OpenGL Shading Language has deprecated some features. These are clearly called out in this specification as Òdeprecated Ó. 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 or link-time errors. See the OpenGL Specification for details on what causes deprecated language features to be accepted or to return an error.

Chapter 2. Overview of OpenGL Shading

The OpenGL Shading Language is actually several closely related languages. These languages are used to create shaders for each of the programmable processors contained in the OpenGL processing pipeline. Currently, these processors are the vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute processors.

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, tessellation control, tessellation evaluation, geometry, fragment, or compute.

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 set of vertex shaders are successfully 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.

2.2. Tessellation Control Processor

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

The tessellation control shader is invoked for each vertex of the output patch. Each invocation can read the attributes of any vertex in the input or output patches, but can only write per-vertex attributes for the corresponding output patch vertex. The shader invocations collectively produce a set of per-patch attributes for the output patch.

After all tessellation control shader invocations have completed, the output vertices and per-patch attributes are assembled to form a patch to be used by subsequent pipeline stages.

Tessellation control shader invocations run mostly independently, with undefined relative execution order. However, the built-in function `barrier()` can be used to control execution order by synchronizing invocations, effectively dividing tessellation control shader execution into a set of phases. Tessellation control shaders will get undefined results if one invocation reads from a per-vertex or per-patch attribute written by another invocation at any point during the same phase, or

if two invocations attempt to write different values to the same per-patch output 32-bit component in a single phase.

2.3. Tessellation Evaluation Processor

The *tessellation evaluation processor* is a programmable unit that evaluates the position and other attributes of a vertex generated by the tessellation primitive generator, using a patch of incoming vertices and their associated data. Compilation units written in the OpenGL Shading Language to run on this processor are called tessellation evaluation shaders. When a set of tessellation evaluation shaders are successfully compiled and linked, they result in a *tessellation evaluation shader executable* that runs on the tessellation evaluation processor.

Each invocation of the tessellation evaluation executable computes the position and attributes of a single vertex generated by the tessellation primitive generator. The executable can read the attributes of any vertex in the input patch, plus the tessellation coordinate, which is the relative location of the vertex in the primitive being tessellated. The executable writes the position and other attributes of the vertex.

2.4. Geometry Processor

The *geometry processor* is a programmable unit that operates on data for incoming vertices for a primitive assembled after vertex processing and outputs a sequence of vertices forming output primitives. Compilation units written in the OpenGL Shading Language to run on this processor are called *geometry shaders*. When a set of geometry shaders are successfully compiled and linked, they result in a *geometry shader executable* that runs on the geometry processor.

A single invocation of the geometry shader executable on the geometry processor will operate on a declared input primitive with a fixed number of vertices. This single invocation can emit a variable number of vertices that are assembled into primitives of a declared output primitive type and passed to subsequent pipeline stages.

2.5. 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 set of fragment shaders are successfully 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 framebuffer memory or texture memory, depending on the current OpenGL state and the OpenGL command that caused the fragments to be generated.

2.6. Compute Processor

The *compute processor* is a programmable unit that operates independently from the other shader processors. Compilation units written in the OpenGL Shading Language to run on this processor are

called *compute shaders*. When a set of compute shaders are successfully compiled and linked, they result in a *compute shader executable* that runs on the compute processor.

A compute shader has access to many of the same resources as fragment and other shader processors, such as textures, buffers, image variables, and atomic counters. It does not have fixed-function outputs. It is not part of the graphics pipeline and its visible side effects are through changes to images, storage buffers, and atomic counters.

A compute shader operates on a group of work items called a *work group*. A work group is a collection of shader invocations that execute the same code, potentially in parallel. An invocation within a work group may share data with other members of the same work group through shared variables and issue memory and control flow barriers to synchronize with other members of the same work group.

Chapter 3. Basics

3.1. Character Set and Phases of Compilation

The source character set used for the OpenGL Shading Language is Unicode in the UTF-8 encoding scheme.

After preprocessing, only the following characters are allowed in the resulting stream of GLSL tokens:

- 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 (?).

A compile-time error will be given if any other character is used in a GLSL token.

There are no digraphs or trigraphs. There are no escape sequences or other uses of the backslash beyond use as the line-continuation character.

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. Lines may be of arbitrary length.

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.

More formally, compilation happens as if the following logical phases were executed in order:

1. Source strings are concatenated to form a single input. All provided new-lines are retained.
2. Line numbering is noted, based on all present new-lines, and does not change when new-lines are later eliminated.
3. Wherever a backslash ('\') occurs immediately before a new-line, both are eliminated. (Note no white space is substituted, allowing a single token to span a new-line.) Any newly formed backslash followed by a new-line is not eliminated; only those pairs originally occurring after phase 1 are eliminated.
4. All comments are replaced with a single space. (Note that '//' style comments end before their terminating new-lines and white space is generally relevant to preprocessing.)
5. Preprocessing is done, resulting in a sequence of GLSL tokens, formed from the character set stated above.

6. GLSL processing is done on the sequence of GLSL tokens.

Details that fully define source strings, comments, line numbering, new-line elimination, and preprocessing are all discussed in upcoming sections. Sections beyond those describe GLSL processing.

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, including counting the new-lines that will be removed by the line-continuation character (\).

Lines separated by the line-continuation character preceding a new-line are concatenated together before either comment processing or preprocessing. This means that no white space is substituted for the line-continuation character. That is, a single token could be formed by the concatenation by taking the characters at the end of one line concatenating them with the characters at the beginning of the next line.

3.3. Preprocessor

There is a preprocessor that processes the source strings as part of the compilation process. Except as noted below, it behaves as the C++ standard preprocessor (see ÒNormative ReferencesÓ).

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. Preprocessing takes places after new-lines have been removed by the line-continuation character.

The number sign (#) on a line by itself is ignored. Any directive not listed above will cause a compile-time error.

#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__

> ! *editing-note*
>
> (Jon) How to italicize the double-underscore markup?

__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 460.

By convention, all macro names containing two consecutive underscores (__) are reserved for use

by underlying software layers. Defining or undefining such a name in a shader does not itself result in an error, but may result in unintended behaviors that stem from having multiple definitions of the same name. All macro names prefixed with ÒGL_Ó (ÒGLÓ followed by a single underscore) are also reserved, and defining or undefining such a name results in a compile-time error.

Implementations must support macro-name lengths of up to 1024 characters. Implementations are allowed to generate an error for a macro name of length greater than 1024 characters, but are also allowed to support lengths greater than 1024.

#if, #ifdef, #ifndef, #else, #elif, and #endif are defined to operate as is standard for C++ preprocessors except for the following:

- Expressions following #if and #elif are further restricted to expressions operating on literal integer constants, plus identifiers consumed by the defined operator.
- Character constants are not supported.

The operators available are as follows.

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

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

```
```

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 only after 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 compile-time diagnostic message into the shader objectÕs information log (see section 7.12 ÒShader, Program and Program Pipeline QueriesÓ of the OpenGL Specification for how to access a shader objectÕs information log). The message will be the tokens following the #error directive, up to the first new-line. The implementation must treat the presence of a #error directive as a compile-time error.

#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.

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.

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

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

where *number* must be a version of the language, following the same convention as __VERSION__ above. The directive Ò#version 460Ó is required in any shader that uses version 4.60 of the language. Any *number* representing a version of the language a compiler does not support will cause a compile-time 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. Shaders that specify #version 100 will be treated as targeting version 1.00 of the OpenGL ES Shading Language. Shaders that specify #version 300 will be treated as targeting

version 3.00 of the OpenGL ES Shading Language. Shaders that specify #version 310 will be treated as targeting version 3.10 of the OpenGL ES Shading Language.

If the optional *profile* argument is provided, it must be the name of an OpenGL profile. Currently, there are three choices:

A *profile* argument can only be used with version 150 or greater. If no profile argument is provided and the version is 150 or greater, the default is core. If version 300 or 310 is specified, the profile argument is not optional and must be es, or a compile-time error results. The Language Specification for the es profile is specified in The OpenGL ES Shading Language specification.

Shaders for the core or compatibility profiles that declare different versions can be linked together. However, es profile shaders cannot be linked with non-es profile shaders or with es profile shaders of a different version, or a link-time error will result. When linking shaders of versions allowed by these rules, remaining link-time errors will be given as per the linking rules in the GLSL version corresponding to the version of the context the shaders are linked under. Shader compile-time errors must still be given strictly based on the version declared (or defaulted to) within each shader.

Unless otherwise specified, this specification is documenting the core profile, and everything specified for the core profile is also available in the compatibility profile. Features specified as belonging specifically to the compatibility profile are not available in the core profile. Compatibility-profile features are not available when generating SPIR-V.

There is a built-in macro definition for each profile the implementation supports. All implementations provide the following macro:

Implementations providing the compatibility profile provide the following macro:

Implementations providing the es profile provide the following macro:

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 lexical and grammatical 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

```
```

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 a compile-time 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 a compile-time 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

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:

where *line* and *source-string-number* are constant integer expressions. If these constant expressions are not integer literals then behavior is undefined. After processing this directive (including its new-line), the implementation will behave as if it is compiling at line number *line* and source string number *source-string-number*. Subsequent source strings will be numbered sequentially, until another #line directive overrides that numbering.

Note

Some implementations have allowed constant expressions in #line directives and some have not. Even where expressions are supported the grammar is ambiguous and so results are implementation dependent. For example, + #line +2 +2 // Line number set to 4, or file to 2 and line to 2

When shaders are compiled for OpenGL SPIR-V, the following predefined macro is available:

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. A /* comment includes its terminating delimiter (*/). However, a // comment does not include (or eliminate) its terminating new line.

Inside comments, any byte values may be used, except a byte whose value is 0. No errors will be given for the content of comments and no validation on the content of comments need be done.

Removal of new-lines by the line-continuation character (\) logically occurs before comments are

processed. That is, a single-line comment ending in the line-continuation character (\) includes the next line in the comment.

3.5. Tokens

The language, after preprocessing, 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 (after preprocessing) can only be used as described in this specification, or a compile-time error results:

const uniform buffer shared attribute varying

coherent volatile restrict readonly writeonly

atomic_uint

layout

centroid flat smooth noperspective

patch sample

invariant precise

break continue do for while switch case default

if else

subroutine

in out inout

int void bool true false float double

discard return

vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4

uint uvec2 uvec3 uvec4

dvec2 dvec3 dvec4

mat2 mat3 mat4

mat2x2 mat2x3 mat2x4

mat3x2 mat3x3 mat3x4

mat4x2 mat4x3 mat4x4

dmat2 dmat3 dmat4

dmat2x2 dmat2x3 dmat2x4

dmat3x2 dmat3x3 dmat3x4

dmat4x2 dmat4x3 dmat4x4

lowp mediump highp precision

sampler1D sampler1DShadow sampler1DArray sampler1DArrayShadow

isampler1D isampler1DArray usampler1D usampler1DArray

sampler2D sampler2DShadow sampler2DArray sampler2DArrayShadow

isampler2D isampler2DArray usampler2D usampler2DArray

sampler2DRect sampler2DRectShadow isampler2DRect usampler2DRect

sampler2DMS isampler2DMS usampler2DMS

sampler2DMSArray isampler2DMSArray usampler2DMSArray

sampler3D isampler3D usampler3D

samplerCube samplerCubeShadow isamplerCube usamplerCube

samplerCubeArray samplerCubeArrayShadow

isamplerCubeArray usamplerCubeArray

samplerBuffer isamplerBuffer usamplerBuffer

image1D iimage1D uimage1D

image1DArray iimage1DArray uimage1DArray

image2D iimage2D uimage2D

image2DArray iimage2DArray uimage2DArray

image2DRect iimage2DRect uimage2DRect

image2DMS iimage2DMS uimage2DMS

image2DMSArray iimage2DMSArray uimage2DMSArray

image3D iimage3D uimage3D

imageCube iimageCube uimageCube

imageCubeArray iimageCubeArray uimageCubeArray

imageBuffer iimageBuffer uimageBuffer

struct

The following are the keywords reserved for future use. Using them will result in a compile-time error:

common partition active

asm

class union enum typedef template this

resource

goto

inline noinline public static extern external interface

long short half fixed unsigned superp

input output

hvec2 hvec3 hvec4 fvec2 fvec3 fvec4

filter

sizeof cast

namespace using

sampler3DRect

In addition, all identifiers containing two consecutive underscores (__) are reserved for use by underlying software layers. Defining such a name in a shader does not itself result in an error, but may result in unintended behaviors that stem from having multiple definitions of the same name.

3.7. Identifiers

Identifiers are used for variable names, function names, structure names, and field selectors (field selectors select components of and , similarly to structure members). 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 ò gl_ ó are reserved for use by OpenGL, and in general, may not be declared in a shader; this results in a compile-time error. However, as noted in the specification, there are some cases where previously declared variables can be redeclared, and predeclared ò gl_ ó names are allowed to be redeclared in a shader only for these specific purposes.

Implementations must support identifier lengths of up to 1024 characters. Implementations are allowed to generate an error for an identifier of length greater than 1024 characters, but are also allowed to support lengths greater than 1024.

3.8. Definitions

Some language rules described below depend on the following definitions.

3.8.1. Static Use

A shader contains a *static use* of a variable *x* if, after preprocessing, the shader contains a statement that would access any part of *x*, whether or not flow of control will cause that statement to be executed. Such a variable is referred to as being *statically used*. If the access is a write then *x* is further said to be *statically assigned*.

3.8.2. Dynamically Uniform Expressions and Uniform Control Flow

Some operations require an expression to be *dynamically uniform*, or that it be located in *uniform control flow*. These requirements are defined by the following set of definitions.

An *invocation* is a single execution of *main()* for a particular stage, operating only on the amount of data explicitly exposed within that stage's shaders. (Any implicit operation on additional instances of data would comprise additional invocations.) For example, in compute execution models, a single invocation operates only on a single work item, or, in a vertex execution model, a single invocation operates only on a single vertex.

An *invocation group* is the complete set of invocations collectively processing a particular compute workgroup or graphical operation, where the scope of a "graphical operation" is implementation-dependent, but at least as large as a single triangle or patch, and at most as large as a single rendering command, as defined by the client API.

Within a single invocation, a single shader statement can be executed multiple times, giving multiple *dynamic instances* of that instruction. This can happen when the instruction is executed in a loop, or in a function called from multiple call sites, or combinations of multiple of these. Different loop iterations and different dynamic function-call-site chains yield different dynamic instances of such an instruction. Dynamic instances are distinguished by their control-flow path within an invocation, not by which invocation executed it. That is, different invocations of *main()* execute the same dynamic instances of an instruction when they follow the same control-flow path.

An expression is _dynamically uniform for a dynamic instance consuming it when its value is the same for all invocations (in the invocation group) that execute that dynamic instance.

Uniform control flow (or converged control flow) occurs when all invocations in the invocation group execute the same control-flow path (and hence the same sequence of dynamic instances of instructions). Uniform control flow is the initial state at the entry into *main()*, and lasts until a conditional branch takes different control paths for different invocations (non-uniform or divergent control flow). Such divergence can reconverge, with all the invocations once again executing the same control-flow path, and this re-establishes the existence of uniform control flow. If control flow is uniform upon entry into a selection or loop, and all invocations in the invocation group subsequently leave that selection or loop, then control flow reconverges to be uniform.

For example:

Note that constant expressions are trivially dynamically uniform. It follows that typical loop counters based on these are also dynamically uniform.

Chapter 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 (=).

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 some implicit conversions between types. Exactly how and when this can occur is described in section ÒImplicit ConversionsÓ and as referenced by other sections in this specification.

4.1. Basic Types

Definition

A *basic type* is a type defined by a keyword in the language.

The OpenGL Shading Language supports the following basic data types, grouped as follows.

4.1.1. 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-precision floating-point scalar
double	a double-precision floating-point scalar
vec2	a two-component single-precision floating-point vector
vec3	a three-component single-precision floating-point vector
vec4	a four-component single-precision floating-point vector
dvec2	a two-component double-precision floating-point vector
dvec3	a three-component double-precision floating-point vector

Type	Meaning
dvec4	a four-component double-precision 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
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 single-precision floating-point matrix
mat3	a 3 " 3 single-precision floating-point matrix
mat4	a 4 " 4 single-precision floating-point matrix
mat2x2	same as a mat2
mat2x3	a single-precision floating-point matrix with 2 columns and 3 rows
mat2x4	a single-precision floating-point matrix with 2 columns and 4 rows
mat3x2	a single-precision floating-point matrix with 3 columns and 2 rows
mat3x3	same as a mat3
mat3x4	a single-precision floating-point matrix with 3 columns and 4 rows
mat4x2	a single-precision floating-point matrix with 4 columns and 2 rows
mat4x3	a single-precision floating-point matrix with 4 columns and 3 rows
mat4x4	same as a mat4
dmat2	a 2 " 2 double-precision floating-point matrix
dmat3	a 3 " 3 double-precision floating-point matrix
dmat4	a 4 " 4 double-precision floating-point matrix
dmat2x2	same as a dmat2
dmat2x3	a double-precision floating-point matrix with 2 columns and 3 rows
dmat2x4	a double-precision floating-point matrix with 2 columns and 4 rows
dmat3x2	a double-precision floating-point matrix with 3 columns and 2 rows

Type	Meaning
dmat3x3	same as a dmat3
dmat3x4	a double-precision floating-point matrix with 3 columns and 4 rows
dmat4x2	a double-precision floating-point matrix with 4 columns and 2 rows
dmat4x3	a double-precision floating-point matrix with 4 columns and 3 rows
dmat4x4	same as a dmat4

Note that where the following tables say Òaccessing a textureÓ, the sampler* opaque types access textures, and the image* opaque types access images, of a specified type.

4.1.2. Floating-Point Opaque Types

Type	Meaning
sampler1D image1D	a handle for accessing a 1D texture
sampler1DShadow	a handle for accessing a 1D depth texture with comparison
sampler1DArray image1DArray	a handle for accessing a 1D array texture
sampler1DArrayShadow	a handle for accessing a 1D array depth texture with comparison
sampler2D image2D	a handle for accessing a 2D texture
sampler2DShadow	a handle for accessing a 2D depth texture with comparison
sampler2DArray image2DArray	a handle for accessing a 2D array texture
sampler2DArrayShadow	a handle for accessing a 2D array depth texture with comparison
sampler2DMS image2DMS	a handle for accessing a 2D multisample texture
sampler2DMSArray image2DMSArray	a handle for accessing a 2D multisample array texture
sampler2DRect image2DRect	a handle for accessing a rectangle texture
sampler2DRectShadow	a handle for accessing a rectangle texture with comparison
sampler3D image3D	a handle for accessing a 3D texture
samplerCube imageCube	a handle for accessing a cube mapped texture

Type	Meaning
samplerCubeShadow	a handle for accessing a cube map depth texture with comparison
samplerCubeArray imageCubeArray	a handle for accessing a cube map array texture
samplerCubeArrayShadow	a handle for accessing a cube map array depth texture with comparison
samplerBuffer imageBuffer	a handle for accessing a buffer texture

4.1.3. Signed Integer Opaque Types

Type	Meaning
isampler1D iimage1D	a handle for accessing an integer 1D texture
isampler1DArray iimage1DArray	a handle for accessing an integer 1D array texture
isampler2D iimage2D	a handle for accessing an integer 2D texture
isampler2DArray iimage2DArray	a handle for accessing an integer 2D array texture
isampler2DMS iimage2DMS	a handle for accessing an integer 2D multisample texture
isampler2DMSArray iimage2DMSArray	a handle for accessing an integer 2D multisample array texture
isampler2DRect iimage2DRect	a handle for accessing an integer 2D rectangle texture
isampler3D iimage3D	a handle for accessing an integer 3D texture
isamplerCube iimageCube	a handle for accessing an integer cube mapped texture
isamplerCubeArray iimageCubeArray	a handle for accessing an integer cube map array texture
isamplerBuffer iimageBuffer	a handle for accessing an integer buffer texture

4.1.4. Unsigned Integer Opaque Types

Type	Meaning
usampler1D uimage1D	a handle for accessing an unsigned integer 1D texture
usampler1DArray uimage1DArray	a handle for accessing an unsigned integer 1D array texture

Type	Meaning
usampler2D uimage2D	a handle for accessing an unsigned integer 2D texture
usampler2DArray uimage2DArray	a handle for accessing an unsigned integer 2D array texture
usampler2DMS uimage2DMS	a handle for accessing an unsigned integer 2D multisample texture
usampler2DMSArray uimage2DMSArray	a handle for accessing an unsigned integer 2D multisample array texture
usampler2DRect uimage2DRect	a handle for accessing an unsigned integer rectangle texture
usampler3D uimage3D	a handle for accessing an unsigned integer 3D texture
usamplerCube uimageCube	a handle for accessing an unsigned integer cube mapped texture
usamplerCubeArray uimageCubeArray	a handle for accessing an unsigned integer cube map array texture
usamplerBuffer uimageBuffer	a handle for accessing an unsigned integer buffer texture
atomic_uint	a handle for accessing an unsigned integer atomic counter

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

There are no pointer types.

In this specification, an *aggregate* will mean a structure or array. (Matrices and vectors are not by themselves aggregates.) Aggregates, matrices, and vectors will collectively be referred to as *composites*.

4.1.5. 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), or a compile-time error results.

4.1.6. Booleans

Definition

A *boolean type* is any boolean scalar or vector type (bool, bvec2, bvec3, bvec4)

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:

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.

4.1.7. Integers

Definitions

An *integral type* is any signed or unsigned, scalar or vector integer type. It excludes arrays and structures.

A *scalar integral type* is a scalar signed or unsigned integer type:

A *vector integral type* is a vector of signed or unsigned 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.

Addition, subtraction, and shift 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. Division and multiplication operations resulting in overflow or underflow will not cause any exception but will result in an undefined value.

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

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 tokenizing, the maximal token matching the above will be recognized before a new token is started. 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. Hence, literals themselves are always expressed with non-negative syntax, though they could result in a negative value.

It is a compile-time error to provide a literal integer whose bit pattern cannot fit in 32 bits. The bit pattern of the literal is always used unmodified. So a signed literal whose bit pattern includes a set sign bit creates a negative value. Note:

1. This only applies to literals; no error checking is performed on the result of a constant expression.
2. Unlike C++, hexadecimal and decimal literals behave in the same way.

For example,

4.1.8. Floats

Single-precision and double-precision floating-point variables are available for use in a variety of scalar calculations. Generally, the term *floating-point* will refer to both single- and double-precision floating-point. Floating-point variables are defined as in the following examples:

As an input value to one of the processing units, a single-precision or double-precision floating-point variable is expected to match the corresponding IEEE 754 floating-point definition for precision and dynamic range. Floating-point variables within a shader are also encoded according to the IEEE 754 specification for single-precision floating-point values

(logically, not necessarily physically). While encodings are logically IEEE 754, operations (addition, multiplication, etc.) are not necessarily performed as required by IEEE 754. See ÒRange and PrecisionÓ for more details on precision and usage of NaNs (Not a Number) and Infs (positive or negative infinities).

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 lf LF

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. When tokenizing, the maximal token matching the above will be recognized before a new token is started. When the suffix "lf" or "LF" is present, the literal has type double. Otherwise, the literal has type float. A leading unary minus sign (-) is interpreted as a unary operator and is not part of the floating-point constant.

4.1.9. Vectors

The OpenGL Shading Language includes data types for generic 2-, 3-, and 4-component vectors of floating-point values, integers, and 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 declarations are:

Initialization of vectors can be done with constructors. See ÒVector and Matrix ConstructorsÓ.

4.1.10. 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. Matrix types beginning with "mat" have single-precision components while matrix types beginning with "dmat" have double-precision components. The first number in the type is the number of columns, the second is the number of rows. If there is only one number, the matrix is square. Example matrix declarations:

Initialization of matrix values is done with constructors (described in ÒVector and Matrix ConstructorsÓ) in column-major order.

4.1.11. Opaque Types

Definition

An *opaque type* is a type where the internal structure of the type is hidden from the language.

The opaque types, as listed in the following sections, declare variables that are effectively opaque handles to other objects. These objects are accessed through built-in functions, not through direct reading or writing of the declared variable. They can only be declared as function parameters or in uniform-qualified variables (see ÒUniform VariablesÓ). The only opaque types that take memory qualifiers are the image types. Except for array indexing, structure member selection, and parentheses, opaque variables are not allowed to be operands in expressions; such use results in a compile-time error.

Opaque variables cannot be treated as l-values; hence cannot be used as out or inout function parameters, nor can they be assigned into. Any such use results in a compile-time error. However, they can be passed as in parameters with matching types and memory qualifiers. They cannot be declared with an initializer.

Because a single opaque type declaration effectively declares two objects, the opaque handle itself and the object it is a handle to, there is room for both a storage qualifier and a memory qualifier. The storage qualifier will qualify the opaque handle, while the memory qualifier will qualify the object it is a handle to.

Samplers

Sampler types (e.g. sampler2D) are opaque types, declared and behaving as described above for opaque types. When aggregated into arrays within a shader, samplers can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.

Sampler variables are handles to one-, two-, and three- dimensional textures, cube maps, depth textures (shadowing), etc., as enumerated in the basic types tables. There are distinct sampler types for each texture target, and for each of float, integer, and unsigned integer data types. Texture accesses are done through built-in texture functions (described in ÒTexture FunctionsÓ) and samplers are used to specify which texture to access and how it is to be filtered.

Images

Image types are opaque types, declared and behaving as described above for opaque types. They can be further qualified with memory qualifiers. When aggregated into arrays within a shader, images can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.

Image variables are handles to one-, two-, or three-dimensional images corresponding to all or a portion of a single level of a texture image bound to an image unit. There are distinct image variable types for each texture target, and for each of float, integer, and unsigned integer data types. Image accesses should use an image type that matches the target of the texture whose level is bound to the image unit, or for non-layered bindings of 3D or array images should use the image type that matches the dimensionality of the layer of the image (i.e., a layer of 3D, 2DArray, Cube, or CubeArray should use image2D, a layer of 1DArray should use image1D, and a layer of 2DMSArray should use image2DMS). If the image target type does not match the bound image in this manner, if the data type does not match the bound image, or if the format layout qualifier does not match the image unit format as described in section 8.25 ÒTexture Image Loads and StoresÓ of the OpenGL Specification, the results of image accesses are undefined but cannot include program termination.

Image variables are used in the image load, store, and atomic functions described in ÒImage FunctionsÓ to specify an image to access.

Atomic Counters

Atomic counter types (e.g. atomic_uint) are opaque handles to counters, declared and behaving as described above for opaque types. The variables they declare specify which counter to access when using the built-in atomic counter functions as described in ÒAtomic Counter FunctionsÓ. They are bound to buffers as described in ÒAtomic Counter Layout QualifiersÓ.

Atomic counters aggregated into arrays within a shader can only be indexed with dynamically uniform integral expressions, otherwise results are undefined.

Members of structures cannot be declared as atomic counter types.

4.1.12. Structures

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

```
```

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).

```
```

More formally, structures are declared as follows. However, the definitive grammar is as given in ÒShading Language GrammarÓ.

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. All previously visible variables, types, constructors, or functions with that name are hidden. 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 may contain precision qualifiers, but use of any other qualifier results in a compile-time error. Bit fields are not supported. Member types must be already defined (there are no forward references). A compile-time error results if a member declaration contains an initializer. Member declarators can contain arrays. Such arrays must have a size specified, and the size must be a constant integral expression thatÕs greater than zero (see ÒConstant ExpressionsÓ). 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. These result in compile-time errors.

Structures can be initialized at declaration time using constructors, as discussed in ÒStructure ConstructorsÓ.

Any restrictions on the usage of a type or qualifier also apply to any structure that contains a member of that type or qualifier. This also applies to structure members that are structures, recursively.

4.1.13. 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 ÒConstant ExpressionsÓ) greater than zero. Except for the last declared member of a shader storage block (see section ÒInterface BlocksÓ), the size of an array must be declared (*explicitly sized*) before it is indexed with anything other than a constant integral expression. The size of any array must be declared before passing it as an argument to a function. Violation of any of these rules result in compile-time errors. It is legal to declare an array without a size (*unsized*) and then later redeclare the same name as an array of the same type and specify a size, or index it only with constant integral expressions (*implicitly sized*). However, unless noted otherwise, blocks cannot be redeclared; an unsized array member in a user-declared block cannot be sized by a block redeclaration. It is a compile-time error to declare an array with a size, and then later (in the same shader) index the same array with a constant integral expression greater than or equal to the declared size. It is a compile-time error to redeclare an unsized array with a size equal to or smaller than any index used earlier in the shader to index the array. It is also a compile-time error 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 thatÕs greater than or equal to the arrayÕs size or less than 0. Arrays only have a single dimension (a single entry within "[ ]"), however, arrays of arrays can be declared. All types (basic types, structures, arrays) can be formed into an array.

All arrays are inherently homogeneous; made of elements all having the same type and size, with one exception. The exception is a shader storage block having an unsized array as its last member (*run-time sized*); an array can be formed from such a shader storage block, even if the storage blocks have differing lengths for their last member.

Some examples are:

An array type can be formed by specifying non-array type followed by an array specifier. All dimensions of such an array specifier must include a size.

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

as a constructor of an array:

as an unnamed parameter:

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

Arrays can have initializers formed from array constructors:

An array of arrays can be declared as:

which declares a one-dimensional array of size 3 of one-dimensional arrays of size 2 of vec4. The following declarations do the same thing:

When in transparent memory (like in a uniform block), the layout is that the inner-most (right-most in declaration) dimensions iterate faster than the outer dimensions. That is, for the above, the order in memory would be:

> Low address : a[0][0] : a[0][1] : a[1][0] : a[1][1] : a[2][0] : a[2][1] : High address

The type of *a* needed for both constructors and nameless parameters is ò vec4[3][2] ó:

Alternatively, the initializer-list syntax can be used to initialize an array of arrays:

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

However, it is a compile-time error to assign to an unsized array. Note, this is a rare case that initializers and assignments appear to have different semantics. For arrays of arrays, any unsized dimension is explicitly sized by the initializer:

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

This returns a type int. If an array has been explicitly sized, the value returned by the length() method is a constant expression. If an array has not been explicitly sized and is the last declared member of a shader storage block, the value returned will not be a constant expression and will be determined at runtime based on the size of the buffer object providing storage for the block. Such arrays are runtime sized. For runtime-sized arrays, the value returned by the length() method will be undefined if the array is contained in an array of shader storage blocks that is indexed with a non-constant expression less than zero or greater than or equal to the number of blocks in the array.

The length() method cannot be called on an array that is not runtime sized and also has not yet been explicitly sized; this results in a compile-time error.

When the length() method returns a compile-time constant, the expression the length() method is applied to cannot contain any side effects, such as writes to l-values within the expression, or function calls that themselves have side effects: only the compile-time constant length itself need be computed. Behavior and results, including any compile-time error reporting, are undefined if the expression contains other effects.

The length() method works equally well for arrays of arrays:

When the length() method returns a compile-time constant, the expression in brackets (*x* above) will be parsed and subjected to the rules required for array indexes, but the array will not be dereferenced. Thus, behavior is well defined even if the run-time value of the expression is out of bounds, as long as the expression contains no side effects.

When the length() method returns a run-time value (not a compile-time constant), the array will be dereferenced. E.g., if *x* is not a compile-time constant and is out of range, an undefined value results. More generally, all involved expressions are fully evaluated and executed.

For implicitly-sized or run-time-sized arrays, only the outer-most dimension can be lacking a size. A type that includes an unknown array size cannot be formed into an array until it gets an explicit size, except for shader storage blocks where the only unsized array member is the last member of the block.

In a shader storage block, the last member may be declared without an explicit size. In this case, the effective array size is inferred at run-time from the size of the data store backing the interface block. Such run-time-sized arrays may be indexed with general integer expressions. However, it is a compile-time error to pass them as an argument to a function or index them with a negative constant expression.

4.1.14. 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

Type of expression	Can be implicitly converted to
int uint	float
int uint float	double
ivec2	uvec2
ivec3	uvec3
ivec4	uvec4
ivec2 uvec2	vec2
ivec3 uvec3	vec3
ivec4 uvec4	vec4
ivec2 uvec2 vec2	dvec2
ivec3 uvec3 vec3	dvec3
ivec4 uvec4 vec4	dvec4
mat2	dmat2
mat3	dmat3
mat4	dmat4
mat2x3	dmat2x3
mat2x4	dmat2x4
mat3x2	dmat3x2
mat3x4	dmat3x4
mat4x2	dmat4x2
mat4x3	dmat4x3

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

When an implicit conversion is done, it is not 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 single-precision floating-point mantissa will lose precision when converted to float.

When performing implicit conversion for binary operators, there may be multiple data types to

which the two operands can be converted. For example, when adding an int value to a uint value, both values can be implicitly converted to uint, float, and double. In such cases, a floating-point type is chosen if either operand has a floating-point type. Otherwise, an unsigned integer type is chosen if either operand has an unsigned integer type. Otherwise, a signed integer type is chosen. If operands can be implicitly converted to multiple data types deriving from the same base data type, the type with the smallest component size is used.

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

4.1.15. Initializers

At declaration, an initial value for a variable may be provided, specified as an equals (=) followed by an initializer. The initializer is either an *assignment-expression* or a list of initializers enclosed in curly braces. The grammar for the initializer is:

initializer :

assignment-expression
{ *initializer-list* }
{ *initializer-list* , }

initializer-list :

initializer
initializer-list , *initializer*

The *assignment-expression* is a normal expression except that a comma (,) outside parentheses is interpreted as the end of the initializer, not as the sequence operator. As explained in more detail below, this allows creation of nested initializers: The variable type and its initializer must exactly match in terms of nesting, number of components/elements/members present at each level, and types of components/elements/members. An *assignment-expression* at global scope can include calls to user-defined functions.

An *assignment-expression* in an initializer must be either the same type as the object it initializes or be a type that can be converted to the object's type according to "Implicit Conversions". Since these include constructors, a composite variable can be initialized by either a constructor or an initializer list; and an element in an initializer list can be a constructor.

If an initializer is a list of initializers enclosed in curly braces, the variable being declared must be a vector, a matrix, an array, or a structure.

A list of initializers enclosed in a matching set of curly braces is applied to one composite. This may be the variable being declared or a composite contained in the variable being declared. Individual initializers from the initializer list are applied to the elements/members of the composite, in order.

If the composite has a vector type, initializers from the list are applied to the components of the vector, in order, starting with component 0. The number of initializers must match the number of components.

If the composite has a matrix type, initializers from the list must be vector initializers and are applied to the columns of the matrix, in order, starting with column 0. The number of initializers must match the number of columns.

If the composite has a structure type, initializers from the list are applied to the members of the structure, in the order declared in the structure, starting with the first member. The number of initializers must match the number of members.

Applying these rules, the following matrix declarations are equivalent:

All of the following declarations result in a compile-time error.

In all cases, the inner-most initializer (i.e., not a list of initializers enclosed in curly braces) applied to an object must have the same type as the object being initialized or be a type that can be converted to the object's type according to "Implicit Conversions". In the latter case, an implicit conversion will be done on the initializer before the assignment is done.

All of the following declarations result in a compile-time error.

If an initializer (of either form) is provided for an unsized array, the size of the array is determined by the number of top-level (non-nested) initializers within the initializer. All of the following declarations create arrays explicitly sized with five elements:

It is a compile-time error to have too few or too many initializers in an initializer list for the composite being initialized. That is, all elements of an array, all members of a structure, all columns of a matrix, and all components of a vector must have exactly one initializer expression present, with no unconsumed initializers.

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. If it is declared in an if or else statement, it is scoped to the end of that statement. (See ÒSelectionÓ and ÒIterationÓ for the location of statements and sub-statements.) 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 functionÛs parameter declarations and body together form a single scope nested in the global scope. The if statementÛs 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:

For both for and while loops, the sub-statement itself does not introduce a new scope for variable names, so the following has a redeclaration compile-time error:

The body of a do-while loop introduces a new scope lasting only between the do and while (not including the while test expression), whether or not the body is simple or compound:

The statement following a switch (É) forms a nested scope.

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 redeclared 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 compile-time 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 that users declare global variables in. That is, a shaderÕs 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, otherwise a compile-time error results.

Shared globals are global variables declared with the same name in independently compiled units (shaders) within the same language (i.e., same stage, e.g. vertex) that are linked together when making a single program. (Globals forming the interface between two different shader languages are discussed in other sections.) 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 in the same stage. If no shader in a stage has an explicit size for the array, the largest implicit size (one more than the largest index used) in that stage is used. There is no cross-stage array sizing. If there is no static access to an implicitly sized array within the stage declaring it, then the array is given a size of 1, which is relevant when the array is declared within an interface block that is shared with other stages or the application (other unused arrays might be eliminated by the optimizer).

Shared global 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 member names to be considered the same type. This rule applies recursively for nested or embedded types. If a shared global has multiple initializers, the initializers must all be constant expressions, and they must all have the same value. Otherwise, a link-time error will result. (A shared global having only one initializer does not require that initializer to be a constant expression.)

4.3. Storage Qualifiers

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

Storage Qualifier	Meaning
<none: default>	local read/write memory, or an input parameter to a function
const	a variable whose value cannot be changed

Storage Qualifier	Meaning
in	linkage into a shader from a previous stage, variable is copied in
out	linkage out of a shader to a subsequent stage, variable is copied out
attribute	compatibility profile only and vertex language only; same as in when in a vertex shader
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application
varying	compatibility profile only and vertex and fragment languages only; same as out when in a vertex shader and same as in when in a fragment shader
buffer	value is stored in a buffer object, and can be read or written both by shader invocations and the OpenGL API
shared	compute shader only; variable storage is shared across all work items in a local work group

Some input and output qualified variables can be qualified with at most one additional auxiliary storage qualifier:

Auxiliary Storage Qualifier	Meaning
centroid	centroid-based interpolation
sample	per-sample interpolation
patch	per-tessellation-patch attributes

Not all combinations of qualification are allowed. Auxiliary storage qualifiers can only be used with the in or out storage qualifiers. Additional qualifier rules are defined in upcoming sections.

Local variables can only use the const storage qualifier (or use no storage qualifier).

Note that function parameters can use const, in, and out qualifiers, but as *parameter qualifiers*. Parameter qualifiers are discussed in ÒFunction Calling ConventionsÓ.

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

Initializers in global declarations may only be used in declarations of global variables 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.

When comparing an output from one shader stage to an input of a subsequent shader stage, the input and output donÕt match if their auxiliary qualifiers (or lack thereof) are not the same.

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. Constant Qualifier

Named compile-time constants or read-only variables can be declared using the const qualifier. The const qualifier can be used with any of the non-void transparent basic data types, as well as with structures and arrays of these. It is a compile-time error to write to a const variable outside of its declaration, so they must be initialized when declared. For example,

```
```

Structure members may not be qualified with const. Structure variables can be declared as const, and initialized with a structure constructor or initializer.

Initializers for const declarations at global scope must be constant expressions, as defined in ÒConstant ExpressionsÓ.

4.3.3. Constant Expressions

SPIR-V specialization constants are expressed in GLSL as const with the layout qualifier constant_id, as described in ÒSpecialization-Constant Qualifier.Ó

A *constant expression* is one of

- A literal value (e.g. 5 or true).
- A variable declared with the const qualifier and an initializer, where the initializer is a constant expression. This includes both const declared with a specialization-constant layout qualifier, e.g. layout(constant_id = É), and those declared without a specialization-constant layout qualifier.
- An expression formed by an operator on operands that are all constant expressions, including getting an element of a constant array, or a member of a constant structure, or components of a constant vector. However, the lowest precedence operators of the sequence operator (,) and the assignment operators (=, +=, É) are not included in the operators that can create a constant expression. Also, an array access with a specialization constant as an index does not result in a constant expression.
- Valid use of the length() method on an explicitly sized object, whether or not the object itself is constant (implicitly sized or run-time sized arrays do not return a constant expression).
- A constructor whose arguments are all constant expressions.
- For non-specialization constants only: the value returned by certain built-in function calls whose arguments are all constant expressions, including at least the list below. Any other built-

in function that does not access memory (not the texture lookup functions, image access, atomic counter, etc.), that has a non-void return type, that has no out parameter, and is not a noise function might also be considered a constant. When a function is called with an argument that is a specialization constant, the result is not a constant expression.

Angle and Trigonometric Functions

$ radians

$ degrees

$ sin

$ cos

$ asin

$ acos

Exponential Functions

$ pow

$ exp

$ log

$ exp2

$ log2

$ sqrt

$ inversesqrt

Common Functions

$ abs

$ sign

$ floor

$ trunc

$ round

$ ceil

$ mod

$ min

$ max

$ clamp

Geometric Functions

$ length

$ dot

$ normalize

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

A *constant integral 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 ÒThe Invariant QualifierÓ for more details on how to create invariant expressions and ÒPrecision QualifiersÓ for detail on how expressions are evaluated.

Constant expressions respect the precise and invariant qualifiers but will be always be evaluated in an invariant way, independent of the use of such qualification, so as to create the same value in multiple shaders when the same constant expressions appear in those shaders. See ÒThe Invariant QualifierÓ and ÒThe Precise QualifierÓ for more details on how to create invariant expressions.

Non-specialization constant expressions may be evaluated by the compilerÕs host platform, and are therefore not required to compute the same value that the same expression would evaluate to on the shader execution target. However, the host must use the same or greater precision than the target would use.

Specialization-constant expressions are never evaluated by the compiler front end, but instead retain the expressionÕs operations needed to evaluate them later on the host.

4.3.4. Input Variables

Shader input variables are declared with the 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. It is a compile-time error to write to a variable declared as an input.

Only the input variables that are statically read need to be written by the previous stage; it is allowed to have superfluous declarations of input variables. This is shown in the following table.

Treatment of Mismatched Input Variables		Consuming Shader (input variables)		
		No Declaration	Declared but no Static Use	Declared and Static Use
Generating Shader (output variables)	No Declaration	Allowed	Allowed	Link-Time Error
	Declared but no Static Use	Allowed	Allowed	Allowed (values are undefined)
	Declared and Static Use	Allowed	Allowed	Allowed (values are potentially undefined)

Consumption errors are based on static use only. Compilation may generate a warning, but not an error, for any dynamic use the compiler can deduce that might cause consumption of undefined values.

See ÒBuilt-In VariablesÓ for a list of the built-in input names.

Vertex shader input variables (or attributes) receive per-vertex data. It is a compile-time error to

use auxiliary storage or interpolation qualifiers on a vertex shader input. The values copied in are established by the OpenGL API or through the use of the layout identifier location.

It is a compile-time error to declare a vertex shader input with, or that contains, any of the following types:

- A boolean type
- An opaque type
- A structure

Example declarations in a vertex shader:

```
```

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-time 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.

Tessellation control, evaluation, and geometry shader input variables get the per-vertex values written out by output variables of the same names in the previous active shader stage. For these inputs, centroid and interpolation qualifiers are allowed, but have no effect. Since tessellation control, tessellation evaluation, and geometry shaders operate on a set of vertices, each input variable (or input block, see interface blocks below) needs to be declared as an array. For example,

```
```

Each element of such an array corresponds to one vertex of the primitive being processed. Each array can optionally have a size declared. For geometry shaders, the array size will be set by, (or if provided must be consistent with) the input layout declaration(s) establishing the type of input primitive, as described later in ÒInput Layout QualifiersÓ.

Some inputs and outputs are *arrayed*, meaning that for an interface between two shader stages either the input or output declaration requires an extra level of array indexing for the declarations to match. For example, with the interface between a vertex shader and a geometry shader, vertex shader output variables and geometry shader input variables of the same name must have matching types, except that the geometry shader will have one more array dimension than the vertex shader, to allow for vertex indexing. If such an arrayed interface variable is not declared with the necessary additional input or output array dimension, a link-time error will result.

Geometry shader inputs, tessellation control shader inputs and outputs, and tessellation evaluation inputs all have an additional level of arrayness relative to other shader inputs and outputs. Component limits for arrayed interfaces (e.g. *gl_MaxTessControlInputComponents*) are limits per vertex, not limits for the entire interface.

For non-arrayed interfaces (meaning array dimensionally stays the same between stages), it is a link-time error if the input variable is not declared with the same type, including array dimensionality, as the matching output variable.

The link-time type-matching rules apply to all declared input and output variables, whether or not they are used.

Additionally, tessellation evaluation shaders support per-patch input variables declared with the patch and in qualifiers. Per-patch input variables are filled with the values of per-patch output variables written by the tessellation control shader. Per-patch inputs may be declared as one-dimensional arrays, but are not indexed by vertex number. Applying the patch qualifier to inputs can only be done in tessellation evaluation shaders. As with other input variables, per-patch inputs must be declared using the same type and qualification as per-patch outputs from the previous (tessellation control) shader stage. It is a compile-time error to use patch with inputs in any other stage.

Fragment shader inputs get per-fragment values, typically interpolated from a previous stage's outputs. The auxiliary storage qualifiers centroid and sample can also be applied, as well as the interpolation qualifiers flat, noperspective, and smooth.

It is a compile-time error to declare a fragment shader input with, or that contains, any of the following types:

- A boolean type
- An opaque type

Fragment shader inputs that are, or contain, integral or double-precision floating-point types must be qualified with the interpolation qualifier flat.

Fragment inputs are declared as in the following examples:

The fragment shader inputs form an interface with the last active shader in the vertex processing pipeline. For this interface, the last active shader stage output variables and fragment shader input variables of the same name must match in type and qualification, with a few exceptions: The storage qualifiers must, of course, differ (one is in and one is out). Also, interpolation qualification (e.g. flat) and auxiliary qualification (e.g. centroid) may differ. These mismatches are allowed between any pair of stages. When interpolation or auxiliary qualifiers do not match, those provided

in the fragment shader supersede those provided in previous stages. If any such qualifiers are completely missing in the fragment shaders, then the default is used, rather than any qualifiers that may have been declared in previous stages. That is, what matters is what is declared in the fragment shaders, not what is declared in shaders in previous stages.

When an interface between shader stages is formed using shaders from two separate program objects, it is not possible to detect mismatches between inputs and outputs when the programs are linked. When there are mismatches between inputs and outputs on such interfaces, the values passed across the interface will be partially or completely undefined.

Shaders can ensure matches across such interfaces either by using input and output layout qualifiers (sections ÒInput Layout QualifiersÓ and ÒOutput Layout QualifiersÓ) or by using identical input and output declarations of blocks or variables. Complete rules for interface matching are found in section 7.4.1 ÒShader Interface MatchingÓ of the OpenGL Specification.

Compute shaders do not permit user-defined input variables and do not form a formal interface with any other shader stage. See ÒCompute Shader Special VariablesÓ for a description of built-in compute shader input variables. All other input to a compute shader is retrieved explicitly through image loads, texture fetches, loads from uniforms or uniform buffers, or other user supplied code. Redeclaration of built-in input variables in compute shaders is not permitted.

4.3.5. Uniform Variables

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 variableÕs initializer, if present, or 0 if no initializer is present. Opaque types cannot have initializers, or a compile-time error results.

Example declarations are:

```
```

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.

Uniforms in shaders all share a single global name space when linked into a program or separable program. Hence, the types, initializers, and any location specifiers of all declared uniform variables with the same name must match across all shaders that are linked into a single program. However it is not required to repeat the location specifier in all the linked shaders. While this single uniform name space is cross stage, a uniform variable nameÕs scope is per stage: If a uniform variable name

is declared in one stage (e.g. a vertex shader) but not in another (e.g. a fragment shader), then that name is still available in the other stage for a different use.

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. Similarly, when a layout location is used, it is not required that all declarations of that name include the location; only that those that include a location use the same location.

4.3.6. Output Variables

Shader output variables are declared with the out storage qualifier. 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. Only output variables that are read by the subsequent pipeline stage need to be written; it is allowed to have superfluous declarations of output variables.

There is *not* an inout storage qualifier for declaring a single variable name as both input and output to a shader. Also, a variable cannot be declared with both the in and the out qualifiers, this will result in a compile-time or link-time error. Output variables must be declared with different names than input variables. However, nesting an input or output inside an interface block with an instance name allows the same names with one referenced through a block instance name.

Vertex, tessellation evaluation, and geometry output variables output per-vertex data and are declared using the out storage qualifier. Applying patch to an output can only be done in a tessellation control shader. It is a compile-time error to use patch on outputs in any other stage.

It is a compile-time error to declare a vertex, tessellation evaluation, tessellation control, or geometry shader output with, or that contains, any of the following types:

- A boolean type
- An opaque type

Individual outputs are declared as in the following examples:

These can also appear in interface blocks, as described in ÒInterface BlocksÓ. Interface blocks allow simpler addition of arrays to the interface from vertex to geometry shader. They also allow a fragment shader to have the same input interface as a geometry shader for a given vertex shader.

Tessellation control shader output variables are used to output per-vertex and per-patch data. Per-vertex output variables are arrayed (see *arrayed* under ÒInput VariablesÓ) and declared using the out qualifier without the patch qualifier. Per-patch output variables are declared using the patch

and out qualifiers.

Since tessellation control shaders produce an arrayed primitive comprising multiple vertices, each per-vertex output variable (or output block, see interface blocks below) needs to be declared as an array. For example,

Each element of such an array corresponds to one vertex of the primitive being produced. Each array can optionally have a size declared. The array size will be set by (or if provided must be consistent with) the output layout declaration(s) establishing the number of vertices in the output patch, as described later in ÒTessellation Control OutputsÓ.

Each tessellation control shader invocation has a corresponding output patch vertex, and may assign values to per-vertex outputs only if they belong to that corresponding vertex. If a per-vertex output variable is used as an l-value, it is a compile-time or link-time error if the expression indicating the vertex index is not the identifier *gl_InvocationID*.

The order of execution of a tessellation control shader invocation relative to the other invocations for the same input patch is undefined unless the built-in function barrier() is used. This provides some control over relative execution order. When a shader invocation calls barrier(), its execution pauses until all other invocations have reached the same point of execution. Output variable assignments performed by any invocation executed prior to calling barrier() will be visible to any other invocation after the call to barrier() returns.

Because tessellation control shader invocations execute in undefined order between barriers, the values of per-vertex or per-patch output variables will sometimes be undefined. Consider the beginning and end of shader execution and each call to barrier() as synchronization points. The value of an output variable will be undefined in any of the three following cases:

1. At the beginning of execution.
2. At each synchronization point, unless
    # the value was well-defined after the previous synchronization point and was not written by any invocation since, or
    # the value was written by exactly one shader invocation since the previous synchronization point, or
    # the value was written by multiple shader invocations since the previous synchronization point, and the last write performed by all such invocations wrote the same value.
3. When read by a shader invocation, if
    # the value was undefined at the previous synchronization point and has not been writen by the same shader invocation since, or
    # the output variable is written to by any other shader invocation between the previous and next synchronization points, even if that assignment occurs in code following the read.

Fragment outputs output per-fragment data and are declared using the out storage qualifier. It is a compile-time error to use auxiliary storage qualifiers or interpolation qualifiers in a fragment shader output declaration. It is a compile-time error to declare a fragment shader output with, or that contains, any of the following types:

- A boolean type
- A double-precision scalar or vector (double, dvec2, dvec3, dvec4)
- An opaque type
- A matrix type
- A structure

Fragment outputs are declared as in the following examples:

Compute shaders have no built-in output variables, do not support user-defined output variables and do not form a formal interface with any other shader stage. All outputs from a compute shader take the form of the side effects such as image stores and operations on atomic counters.

4.3.7. Buffer Variables

The buffer qualifier is used to declare global variables whose values are stored in the data store of a buffer object bound through the OpenGL API. Buffer variables can be read and written, with the underlying storage shared among all active shader invocations. Buffer variable memory reads and writes within a single shader invocation are processed in order. However, the order of reads and writes performed in one invocation relative to those performed by another invocation is largely undefined. Buffer variables may be qualified with memory qualifiers affecting how the underlying memory is accessed, as described in ÒMemory QualifiersÓ.

The buffer qualifier can be used to declare interface blocks (see ÒInterface BlocksÓ), which are then referred to as shader storage blocks. It is a compile-time error to declare buffer variables outside a block.

There are implementation-dependent limits on the number of shader storage blocks used for each type of shader, the combined number of shader storage blocks used for a program, and the amount of storage required by each individual shader storage block. If any of these limits are exceeded, it will cause a compile-time or link-time error.

If multiple shaders are linked together, then they will share a single global buffer variable name space. Hence, the types of all declared buffer variables with the same name must match across all shaders that are linked into a single program.

4.3.8. Shared Variables

The shared qualifier is used to declare global variables that have storage shared between all work items in a compute shader local work group. Variables declared as shared may only be used in compute shaders (see ÒCompute ProcessorÓ). Any other declaration of a shared variable is a compile-time error. Shared variables are implicitly coherent (see ÒMemory QualifiersÓ).

Variables declared as shared may not have initializers and their contents are undefined at the beginning of shader execution. Any data written to shared variables will be visible to other work items (executing the same shader) within the same local work group.

In the absence of synchronization, the order of reads and writes to the same shared variable by different invocations of a shader is not defined.

In order to achieve ordering with respect to reads and writes to shared variables, a combination of control flow and memory barriers must be employed using the barrier() and memoryBarrier() functions (see ÒShader Invocation Control FunctionsÓ).

There is a limit to the total size of all variables declared as shared in a single program. This limit, expressed in units of basic machine units may be determined by using the OpenGL API to query the value of MAX_COMPUTE_SHARED_MEMORY_SIZE.

4.3.9. Interface Blocks

Input, output, uniform, and buffer variable declarations can be grouped into named interface blocks to provide coarser granularity backing than is achievable with individual declarations. They can have an optional instance name, used in the shader to reference their members. An output block of one programmable stage is backed by a corresponding input block in the subsequent programmable stage. A *uniform block* is backed by the application with a buffer object. A block of buffer variables, called a *shader storage block*, is also backed by the application with a buffer object. It is a compile-time error to have an input block in a vertex shader or an output block in a fragment shader. These uses are reserved for future use.

An interface block is started by an in, out, uniform, or buffer keyword, followed by a block name, followed by an open curly brace ({) as follows:

interface-block :

layout-qualifier$_{opt}$ *interface-qualifier block-name* { *member-list* } *instance-name*$_{opt}$;

interface-qualifier :

in
out
patch in // Note: Qualifiers can be in any order.
patch out
uniform

buffer

member-list :

member-declaration
member-declaration member-list

member-declaration :

layout-qualifier$_{opt}$ *qualifiers*$_{opt}$ *type declarators* ;

instance-name :

identifier
identifier [ ]
identifier [*constant-integral-expression*]

Each of the above elements is discussed below, with the exception of layout qualifiers (*layout-qualifier*), which are defined in the next section.

First, an example,

The above establishes a uniform block named ÒTransformÓ with four uniforms grouped inside it.

Types and declarators are the same as for other input, output, uniform, and buffer variable declarations outside blocks, with these exceptions:

- Initializers are not allowed
- Opaque types are not allowed
- Structure definitions cannot be nested inside a block

Any of these would result in a compile-time error.

If no optional qualifier is used in a member-declaration, the qualification of the member includes all in, out, patch, uniform, or buffer as determined by *interface-qualifier*. If optional qualifiers are used, they can include interpolation qualifiers, auxiliary storage qualifiers, and storage qualifiers and they must declare an input, output, or uniform member consistent with the interface qualifier of the block: Input variables, output variables, uniform variables, and buffer members can only be in in blocks, out blocks, uniform blocks, and shader storage blocks, respectively.

Repeating the in, out, patch, uniform, or buffer interface qualifier for a memberÕs storage qualifier is optional. For example,

A *shader interface* is defined to be one of these:

- All the uniform variables and uniform blocks declared in a program. This spans all compilation units linked together within one program.
- All the buffer blocks declared in a program.
- The boundary between adjacent programmable pipeline stages: This spans all the outputs declared in all compilation units of the first stage and all the inputs declared in all compilation units of the second stage. Note that for the purposes of this definition, the fragment shader and the preceding shader are considered to have a shared boundary even though in practice, all values passed to the fragment shader first pass through the rasterizer and interpolator.

The block name (*block-name*) is used to match within shader interfaces: an output block of one pipeline stage will be matched to an input block with the same name in the subsequent pipeline stage. For uniform or shader storage blocks, the application uses the block name to identify the block. Block names have no other use within a shader beyond interface matching; it is a compile-time error to use a block name at global scope for anything other than as a block name (e.g. use of a block name for a global variable name or function name is currently reserved). It is a compile-time error to use the same block name for more than one block declaration in the same shader interface (as defined above) within one shader, even if the block contents are identical.

Matched block names within a shader interface (as defined above) must match in terms of having the same number of declarations with the same sequence of types and the same sequence of member names, as well as having matching member-wise layout qualification (see next section). Matched uniform or shader storage block names (but not input or output block names) must also either all be lacking an instance name or all having an instance name, putting their members at the same scoping level. When instance names are present on matched block names, it is allowed for the instance names to differ; they need not match for the blocks to match. Furthermore, if a matching block is declared as an array, then the array sizes must also match (or follow array matching rules for the shader interface between consecutive shader stages). Any mismatch will generate a link-time error. A block name is allowed to have different definitions in different shader interfaces within the same shader, allowing, for example, an input block and output block to have the same name.

If an instance name (*instance-name*) is not used, the names declared inside the block are scoped at the global level and accessed as if they were declared outside the block. If an instance name (*instance-name*) is used, then it puts all the members inside a scope within its own name space, accessed with the field selector (.) operator (analogously to structures). For example,

Outside the shading language (i.e., in the API), members are similarly identified except the block name is always used in place of the instance name (API accesses are to shader interfaces, not to shaders). If there is no instance name, then the API does not use the block name to access a member, just the member name.

Within a shader interface, all declarations of the same global name must be for the same object and must match in type and in whether they declare a variable or member of a block with no instance name. The API also needs this name to uniquely identify an object in the shader interface. It is a link-time error if any particular shader interface contains

- two different blocks, each having no instance name, and each having a member of the same name, or
- a variable outside a block, and a block with no instance name, where the variable has the same name as a member in the block.

For blocks declared as arrays, the array index must also be included when accessing members, as in this example

For uniform or shader storage blocks declared as an array, each individual array element corresponds to a separate buffer object bind range, backing one instance of the block. As the array size indicates the number of buffer objects needed, uniform and shader storage block array declarations must specify an array size. A uniform or shader storage block array can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.

When using OpenGL API entry points to identify the name of an individual block in an array of blocks, the name string may include an array index (e.g. *Transform[2]*). When using OpenGL API entry points to refer to offsets or other characteristics of a block member, an array index must not be specified (e.g. *Transform.ModelViewMatrix*).

Tessellation control, tessellation evaluation and geometry shader input blocks must be declared as arrays and follow the array declaration and linking rules for all shader inputs for the respective stages. All other input and output block arrays must specify an array size.

There are implementation-dependent limits on the number of uniform blocks and the number of shader storage blocks that can be used per stage. If either limit is exceeded, it will cause a link-time error.

4.4. Layout Qualifiers

Layout qualifiers can appear in several forms of declaration. They can appear as part of an interface block definition or block member, as shown in the grammar in the previous section. They can also appear with just an *interface-qualifier* to establish layouts of other declarations made with that qualifier:

layout-qualifier interface-qualifier ;

Or, they can appear with an individual variable declared with an interface qualifier:

layout-qualifier interface-qualifier declaration ;

Declarations of layouts can only be made at global scope or block members, and only where

indicated in the following subsections; their details are specific to what the interface qualifier is, and are discussed individually.

The *layout-qualifier* expands to:

layout-qualifier :

layout (*layout-qualifier-id-list*)

layout-qualifier-id-list :

layout-qualifier-id
layout-qualifier-id , *layout-qualifier-id-list*

layout-qualifier-id :

layout-qualifier-name
layout-qualifier-name = *layout-qualifier-value*
shared

layout-qualifier-value :

integer-constant-expression

The tokens used for *layout-qualifier-name* are identifiers, not keywords, however, the shared keyword is allowed as a *layout-qualifier-id*. Generally, they can be listed in any order. Order-dependent meanings exist only if explicitly called out below. Similarly, these identifiers are not case sensitive, unless explicitly noted otherwise.

More than one layout qualifier may appear in a single declaration. Additionally, the same *layout-qualifier-name* can occur multiple times within a layout qualifier or across multiple layout qualifiers in the same declaration. When the same *layout-qualifier-name* occurs multiple times, in a single declaration, the last occurrence overrides the former occurrence(s). Further, if such a *layout-qualifier-name* will affect subsequent declarations or other observable behavior, it is only the last occurrence that will have any effect, behaving as if the earlier occurrence(s) within the declaration are not present. This is also true for overriding *layout-qualifier-name*, where one overrides the other (e.g. row_major vs. column_major); only the last occurrence has any effect.

integer-constant-expression is defined in ÒConstant ExpressionsÓ as Ò*constant integral expression*Ó, with it being a compile-time error for *integer-constant-expression* to be a specialization constant.

The following table summarizes the use of layout qualifiers. It shows for each one what kinds of declarations it may be applied to. These are all discussed in detail in the following sections.

Layout Qualifier	Qualifier Only	Individual Variable	Block	Block Member	Allowed Interfaces
shared packed std140 std430	X		X		uniform / buffer
row_major column_major	X		X	X	
binding =		opaque types only	X		
offset =		atomic counters only		X	
align =			X	X	
location =		X			uniform / buffer and subroutine variables
location =		X	X	X	all in / out, except for compute
component =		X		X	
index =		X			fragment out and subroutine functions
triangles quads isolines	X				tessellation evaluation in
equal_spacing fractional_even_spacing fractional_odd_spacing	X				tessellation evaluation in
cw ccw	X				tessellation evaluation in
point_mode	X				tessellation evaluation in
points	X				geometry in /out
[points] lines lines_adjacency triangles triangles_adjacency	X				geometry in
invocations =	X				geometry in

Layout Qualifier	Qualifier Only	Individual Variable	Block	Block Member	Allowed Interfaces
origin_upper_left pixel_center_integer		*gl_FragCoord* only			fragment in
early_fragment_tests	X				
local_size_x = local_size_y = local_size_z =	X				compute in
local_size_x_id = local_size_y_id = local_size_z_id =	X				compute in (SPIR-V generation only)
xfb_buffer = xfb_stride =	X	X	X	X	vertex, tessellation, and geometry out
xfb_offset =		X	X	X	
vertices =	X				tessellation control out
[points] line_strip triangle_strip	X				geometry out
max_vertices =	X				
stream =	X	X	X	X	
depth_any depth_greater depth_less depth_unchanged		*gl_FragDepth* only			fragment out
constant_id =		scalar only			const (SPIR-V generation only)

Layout Qualifier	Qualifier Only	Individual Variable	Block	Block Member	Allowed Interfaces
rgba32f rgba16f rg32f rg16f r11f_g11f_b10f r32f r16f rgba16 rgb10_a2 rgba8 rg16 rg8 r16 r8 rgba16_snorm rgba8_snorm rg16_snorm rg8_snorm r16_snorm r8_snorm rgba32i rgba16i rgba8i rg32i rg16i rg8i r32i r16i r8i rgba32ui rgba16ui rgb10_a2ui rgba8ui rg32ui rg16ui rg8ui r32ui r16ui r8ui		image types only			uniform

4.4.1. Input Layout Qualifiers

Layout qualifiers specific to a particular shader language are discussed in separate sections below.

All shaders except compute shaders allow location layout qualifiers on input variable declarations, input block declarations, and input block member declarations. Of these, variables and block members (but not blocks) additionally allow the component layout qualifier.

layout-qualifier-id :

location = *layout-qualifier-value*
component = *layout-qualifier-value*

For example,

will establish that the shader input *normal* is assigned to vector location number 3 and *v* is assigned location number 8. For vertex shader inputs, the location specifies the number of the vertex attribute from which input values are taken. For inputs of all other shader types, the location specifies a vector number that can be used to match against outputs from a previous shader stage, even if that shader is in a different program object.

The following language describes how many locations are consumed by a given type. However, geometry shader inputs, tessellation control shader inputs and outputs, and tessellation evaluation inputs all have an additional level of arrayness relative to other shader inputs and outputs. This outer array level is removed from the type before considering how many locations the type consumes.

If a vertex shader input is any scalar or vector type, it will consume a single location. If a non-vertex shader input is a scalar or vector type other than dvec3 or dvec4, it will consume a single location, while types dvec3 or dvec4 will consume two consecutive locations. Inputs of type double and dvec2 will consume only a single location, in all stages.

If the declared input (after potentially removing an outer array level as just described above) is an array of size *n* and each of the elements takes *m* locations, it will be assigned *m* * *n* consecutive locations starting with the location specified. For example,

will establish that the shader input *colors* is assigned to vector location numbers 6, 7, and 8.

If the declared input is an *n* " *m* matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an *n* -element array of *m*-component vectors. For example,

will establish that shader input *transforms* is assigned to vector locations 9-16, with *transforms[0]* being assigned to locations 9-12, and *transforms[1]* being assigned to locations 13-16.

If the declared input is a structure or block, its members will be assigned consecutive locations in their order of declaration, with the first member assigned the location provided in the layout qualifier. For a structure, this process applies to the entire structure. It is a compile-time error to use a location qualifier on a member of a structure. For a block, this process applies to the entire

block, or until the first member is reached that has a location layout qualifier.

When a block member is declared with a location qualifier, its location comes from that qualifier; the member's location qualifier overrides the block-level declaration. Subsequent members are again assigned consecutive locations, based on the newest location, until the next member declared with a location qualifier. The values used for locations do not have to be declared in increasing order.

If a block has no block-level location layout qualifier, it is required that either all or none of its members have a location layout qualifier, or a compile-time error results. For some blocks declared as arrays, the location can only be applied at the block level: When a block is declared as an array where additional locations are needed for each member for each block array element, it is a compile-time error to specify locations on the block members. That is, when locations would be under specified by applying them on block members, they are not allowed on block members. For *arrayed* interfaces (those generally having an extra level of arrayness due to interface expansion), the outer array is stripped before applying this rule.

When generating SPIR-V, all in and out qualified user-declared (non built-in) variables and blocks (or all their members) must have a shader-specified location. Otherwise, a compile-time error is generated.

The locations consumed by block and structure members are determined by applying the rules above recursively as though the structure member were declared as an input variable of the same type. For example:

The number of input locations available to a shader is limited. For vertex shaders, the limit is the advertised number of vertex attributes. For all other shaders, the limit is implementation-dependent and must be no less than one fourth of the advertised maximum input component count.

A program will fail to link if any attached shader uses a location greater than or equal to the number of supported locations, unless device-dependent optimizations are able to make the program fit within available hardware resources.

A program will fail to link if explicit location assignments leave the linker unable to find space for other variables without explicit assignments.

For the purposes of determining if a non-vertex input matches an output from a previous shader stage, the location layout qualifier (if any) must match.

If a vertex shader input variable with no location assigned in the shader text has a location specified through the OpenGL API, the API-assigned location will be used. Otherwise, such variables will be assigned a location by the linker. See section 11.1.1 ÒVertex AttributesÓ of the OpenGL Specification for more details. A link-time error will occur if an input variable is declared in multiple shaders of the same language with conflicting locations.

The component qualifier allows the location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. It is a compile-time error to use component without also specifying the location qualifier (order does not matter). The components within a location are 0, 1, 2, and 3. A variable or block member starting at component *N* will consume components *N*, *N+1*, *N+2*, É up through its size. It is a compile-time error if this sequence of components gets larger than 3. A scalar double will consume two of these components, and a dvec2 will consume all four components available within a location. A dvec3 or dvec4 can only be declared without specifying a component. A dvec3 will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations.

For example:

If the variable is an array, each element of the array, in order, is assigned to consecutive locations, but all at the same specified component within each location. For example:

That is, location 2 component 3 will hold *d[0]*, location 3 component 3 will hold *d[1]*, É, up through location 7 component 3 holding *d[5]*.

This allows packing of two arrays into the same set of locations:

If applying this to an array of arrays, all levels of arrayness are removed to get to the elements that are assigned per location to the specified component. These non-arrayed elements will fill the locations in the order specified for arrays of arrays in ÒArraysÓ.

It is a compile-time error to apply the component qualifier to a matrix, a structure, a block, or an array containing any of these. It is a compile-time error to use component 1 or 3 as the beginning of a double or dvec2. It is a link-time error to specify different components for the same variable

within a program.

Location aliasing is causing two variables or block members to have the same location number. *Component aliasing* is assigning the same (or overlapping) component numbers for two location aliases. (Recall if component is not used, components are assigned starting with 0.) With one exception, location aliasing is allowed only if it does not cause component aliasing; it is a compile-time or link-time error to cause component aliasing. Further, when location aliasing, the aliases sharing the location must have the same underlying numerical type and bit width (floating-point or integer, 32-bit versus 64-bit, etc.) and the same auxiliary storage and interpolation qualification. The one exception where component aliasing is permitted is for two input variables (not block members) to a vertex shader, which are allowed to have component aliasing. This vertex-variable component aliasing is intended only to support vertex shaders where each execution path accesses at most one input per each aliased component. Implementations are permitted, but not required, to generate link-time errors if they detect that every path through the vertex shader executable accesses multiple inputs aliased to any single component.

Tessellation Evaluation Inputs

Additional input layout qualifier identifiers allowed for tessellation evaluation shaders are described below.

layout-qualifier-id :

primitive_mode
vertex_spacing
ordering
point_mode

The primitive-mode is used to specify a tessellation primitive mode to be used by the tessellation primitive generator.

primitive-mode:

triangles
quads
isolines

If present, the *primitive-mode* specifies that the tessellation primitive generator should subdivide a triangle into smaller triangles, a quad into triangles, or a quad into a collection of lines, respectively.

A second group of layout identifiers, *vertex spacing*, is used to specify the spacing used by the tessellation primitive generator when subdividing an edge.

vertex-spacing:

equal_spacing
fractional_even_spacing
fractional_odd_spacing

equal_spacing specifies that edges should be divided into a collection of equal-sized segments;

fractional_even_spacing specifies that edges should be divided into an even number of equal-

length segments plus two additional shorter Òfractional Ó segments; or

fractional_odd_spacing specifies that edges should be divided into an odd number of equal-length segments plus two additional shorter Òfractional Ó segments.

A third group of layout identifiers, *ordering*, specifies whether the tessellation primitive generator produces triangles in clockwise or counter-clockwise order, according to the coordinate system depicted in the OpenGL Specification.

ordering:

cw
ccw

The identifiers cw and ccw indicate clockwise and counter-clockwise triangles, respectively. If the tessellation primitive generator does not produce triangles, the order is ignored.

Finally, *point mode* indicates that the tessellation primitive generator should produce one point for each distinct vertex in the subdivided primitive, rather than generating lines or triangles.

point-mode:

point_mode

Any or all of these identifiers may be specified one or more times in a single input layout declaration. If primitive mode, vertex spacing, or ordering is declared more than once in the tessellation evaluation shaders of a program, all such declarations must use the same identifier.

At least one tessellation evaluation shader (compilation unit) in a program must declare a primitive mode in its input layout. Declaring vertex spacing, ordering, or point mode identifiers is optional. It is not required that all tessellation evaluation shaders in a program declare a primitive mode. If spacing or vertex ordering declarations are omitted, the tessellation primitive generator will use equal spacing or counter-clockwise vertex ordering, respectively. If a point mode declaration is omitted, the tessellation primitive generator will produce lines or triangles according to the primitive mode.

Geometry Shader Inputs

Additional layout qualifier identifiers for geometry shader inputs include *primitive* identifiers and an *invocation count* identifier:

layout-qualifier-id :

points
lines
lines_adjacency
triangles
triangles_adjacency
invocations = *layout-qualifier-value*

The identifiers points, lines, lines_adjacency, triangles, and triangles_adjacency are used to specify the type of input primitive accepted by the geometry shader, and only one of these is accepted. At least one geometry shader (compilation unit) in a program must declare this input

primitive layout, and all geometry shader input layout declarations in a program must declare the same layout. It is not required that all geometry shaders in a program declare an input primitive layout.

The identifier invocations is used to specify the number of times the geometry shader executable is invoked for each input primitive received. Invocation count declarations are optional. If no invocation count is declared in any geometry shader in a program, the geometry shader will be run once for each input primitive. If an invocation count is declared, all such declarations must specify the same count. If a shader specifies an invocation count greater than the implementation-dependent maximum, or less than or equal to zero, a compile-time error results.

For example,

```

```

will establish that all inputs to the geometry shader are triangles and that the geometry shader executable is run six times for each triangle processed.

All geometry shader input unsized array declarations will be sized by an earlier input primitive layout qualifier, when present, as per the following table.

Layout	Size of Input Arrays
points	1
lines	2
lines_adjacency	4
triangles	3
triangles_adjacency	6

The intrinsically declared input array *gl_in[]* will also be sized by any input primitive-layout declaration. Hence, the expression

```

```

will return the value from the table above.

For inputs declared without an array size, including intrinsically declared inputs (i.e., *gl_in*), a layout must be declared before any use of the method length() or other any array use that requires the array size to be known.

It is a compile-time error if a layout declaration's array size (from the table above) does not match all the explicit array sizes specified in declarations of an input variables in the same shader. The following includes examples of compile-time errors:

It is a link-time error if not all provided sizes (sized input arrays and layout size) match across all geometry shaders in a program.

Fragment Shader Inputs

Additional fragment layout qualifier identifiers include the following for *gl_FragCoord* :

layout-qualifier-id :

origin_upper_left
pixel_center_integer

By default, *gl_FragCoord* assumes a lower-left origin for window coordinates and assumes pixel centers are located at half-pixel coordinates. For example, the (*x*, *y*) location (0.5, 0.5) is returned for the lower-left-most pixel in a window. The origin can be changed by redeclaring *gl_FragCoord* with the origin_upper_left qualifier, moving the origin of *gl_FragCoord* to the upper left of the window, with *y* increasing in value toward the bottom of the window. The values returned can also be shifted by half a pixel in both *x* and *y* by pixel_center_integer so it appears the pixels are centered at whole number pixel offsets. This moves the (*x*, *y*) value returned by *gl_FragCoord* of (0.5, 0.5) by default, to (0.0, 0.0) with pixel_center_integer.

Redeclarations are done as follows

If *gl_FragCoord* is redeclared in any fragment shader in a program, it must be redeclared in all the fragment shaders in that program that have a static use *gl_FragCoord*. All redeclarations of *gl_FragCoord in all fragment shaders in a single program must have the same set of qualifiers. Within any shader, the first redeclarations of _glFragCoord* must appear before any use of *gl_FragCoord*. The built-in *gl_FragCoord* is only predeclared in fragment shaders, so redeclaring it in any other shader language results in a compile-time error.

Redeclaring *glFragCoord* with origin_upper_left and/or pixel_center_integer qualifiers only affects *gl_FragCoord.x* and *gl_FragCoord.y*. It has no effect on rasterization, transformation, or any other part of the OpenGL pipeline or language features.

Fragment shaders allow the following layout qualifier on in only (not with variable declarations):

layout-qualifier-id :

early_fragment_tests

to request that fragment tests be performed before fragment shader execution, as described in section 15.2.4 ÒEarly Fragment TestsÓ of the OpenGL Specification.

For example,

Specifying this will make per-fragment tests be performed before fragment shader execution. If this is not declared, per-fragment tests will be performed after fragment shader execution. Only one fragment shader (compilation unit) need declare this, though more than one can. If at least one declares this, then it is enabled.

Compute Shader Inputs

There are no layout location qualifiers for compute shader inputs.

Layout qualifier identifiers for compute shader inputs are the work-group size qualifiers:

layout-qualifier-id :

local_size_x = *layout-qualifier-value*
local_size_y = *layout-qualifier-value*
local_size_z = *layout-qualifier-value*

The local_size_x, local_size_y, and local_size_z qualifiers are used to declare a fixed local group size by the compute shader in the first, second, and third dimension, respectively. If a shader does not specify a size for one of the dimensions, that dimension will have a size of 1.

For example, the following declaration in a compute shader

is used to declare a two-dimensional compute shader with a local size of 32 X 32 elements, which is equivalent to a three-dimensional compute shader where the third dimension has size one.

As another example, the declaration

effectively specifies that a one-dimensional compute shader is being compiled, and its size is 8 elements.

If the fixed local group size of the shader in any dimension is less than or equal to zero or greater

than the maximum size supported by the implementation for that dimension, a compile-time error results. Also, if such a layout qualifier is declared more than once in the same shader, all those declarations must set the same set of local work-group sizes and set them to the same values; otherwise a compile-time error results. If multiple compute shaders attached to a single program object declare a fixed local group size, the declarations must be identical; otherwise a link-time error results.

Furthermore, if a program object contains any compute shaders, at least one must contain an input layout qualifier specifying a fixed local group size for the program, or a link-time error will occur.

4.4.2. Output Layout Qualifiers

Some output layout qualifiers apply to all shader stages and some apply only to specific stages. The latter are discussed in separate sections below.

As with input layout qualifiers, all shaders except compute shaders allow location layout qualifiers on output variable declarations, output block declarations, and output block member declarations. Of these, variables and block members (but not blocks) additionally allow the component layout qualifier.

layout-qualifier-id :

location = *layout-qualifier-value*
component = *layout-qualifier-value*

The usage and rules for applying the location qualifier and the component qualifier to blocks and structures are exactly as described in ÒInput Layout QualifiersÓ. Additionally, for fragment shader outputs, if two variables are placed within the same location, they must have the same underlying type (floating-point or integer). No component aliasing of output variables or members is allowed.

Fragment shaders allow an additional index output layout qualifier:

layout-qualifier-id :

index = *layout-qualifier-value*

Each of these qualifiers may appear at most once. If index is specified, location must also be specified. If index is not specified, the value 0 is used. For example, in a fragment shader,

```
```

will establish that the fragment shader output *color* is assigned to fragment color 3 as the first (index zero) input to the blend equation. And,

```
```

will establish that the fragment shader output *factor* is assigned to fragment color 3 as the second (index one) input to the blend equation.

For fragment shader outputs, the location and index specify the color output number and index

receiving the values of the output. For outputs of all other shader stages, the location specifies a vector number that can be used to match against inputs in a subsequent shader stage, even if that shader is in a different program object.

If a declared output is a scalar or vector type other than dvec3 or dvec4, it will consume a single location. Outputs of type dvec3 or dvec4 will consume two consecutive locations. Outputs of type double and dvec2 will consume only a single location, in all stages.

If the declared output is an array, it will be assigned consecutive locations starting with the location specified. For example,

```
```

will establish that *colors* is assigned to vector location numbers 2, 3, and 4.

If the declared output is an *n* " *m* matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned will be the same as for an *n*-element array of *m* -component vectors.

If the declared output is a structure, its members will be assigned consecutive locations in the order of declaration, with the first member assigned the location specified for the structure. The number of locations consumed by a structure member is determined by applying the rules above recursively as though the structure member were declared as an output variable of the same type.

location layout qualifiers may be used on output variables declared as structures. However, it is a compile-time error to use a location qualifier on a structure member. Location layout qualifiers may be used on output blocks and output block members.

The number of output locations available to a shader is limited. For fragment shaders, the limit is the advertised number of draw buffers.

For all other shaders, the limit is implementation-dependent and must be no less than one fourth of the advertised maximum output component count (compute shaders have no outputs). A program will fail to link if any attached shader uses a location greater than or equal to the number of supported locations, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Compile-time errors may also be given if at compile time it is known the link will fail. A negative output location will result in a compile-time error. It is also a compile-time error if a fragment shader sets a layout index to less than 0 or greater than 1.

It is a compile-time or link-time error if any of the following occur:

- ¥ any two fragment shader output variables are assigned to the same location and index.
- ¥ if any two output variables from the same vertex, tessellation or geometry shader stage are assigned to the same location.

For fragment shader outputs, locations can be assigned using either a layout qualifier or via the OpenGL API.

For all shader types, a program will fail to link if explicit location assignments leave the linker unable to find space for other variables without explicit assignments.

If an output variable with no location or index assigned in the shader text has a location specified through the OpenGL API, the API-assigned location will be used. Otherwise, such variables will be assigned a location by the linker. All such assignments will have a color index of zero. See section 15.2 ÒShader ExecutionÓ of the OpenGL Specification for more details. A link-time error will occur if an output variable is declared in multiple shaders of the same language with conflicting location or index values.

For the purposes of determining if a non-fragment output matches an input from a subsequent shader stage, the location layout qualifier (if any) must match.

Transform Feedback Layout Qualifiers

The vertex, tessellation, and geometry stages allow shaders to control transform feedback. When doing this, shaders will dictate which transform feedback buffers are in use, which output variables will be written to which buffers, and how each buffer is laid out. To accomplish this, shaders allow the following layout qualifier identifiers on output declarations:

layout-qualifier-id :

xfb_buffer = *layout-qualifier-value*
xfb_offset = *layout-qualifier-value*
xfb_stride = *layout-qualifier-value*

Any shader making any static use (after preprocessing) of any of these xfb_ qualifiers will cause the shader to be in a transform feedback capturing mode and hence responsible for describing the transform feedback setup. This mode will capture any output selected by xfb_offset, directly or indirectly, to a transform feedback buffer.

The xfb_buffer qualifier specifies which transform feedback buffer will capture outputs selected with xfb_offset. The xfb_buffer qualifier can be applied to the qualifier out, to output variables, to output blocks, and to output block members. Shaders in the transform feedback capturing mode have an initial global default of

```
```

This default can be changed by declaring a different buffer with xfb_buffer on the interface qualifier out. This is the only way the global default can be changed. When a variable or output block is declared without an xfb_buffer qualifier, it inherits the global default buffer. When a variable or output block is declared with an xfb_buffer qualifier, it has that declared buffer. All members of a block inherit the blockÕs buffer. A member is allowed to declare an xfb_buffer, but it must match the buffer inherited from its block, or a compile-time error results.

Note this means all members of a block that go to a transform feedback buffer will go to the same buffer.

When a block is declared as an array, all members of block array-element 0 are captured, as previously described, by the declared or inherited xfb_buffer. Generally, an array of size *N* of blocks is captured by *N* consecutive buffers, with all members of block array-element *E* captured by buffer *B*, where *B* equals the declared or inherited xfb_buffer plus *E*.

It is a compile-time or link-time error to specify an xfb_buffer, including any additional buffers needed to capture an arrays of blocks, that is less than zero or greater than or equal to the implementation-dependent constant *gl_MaxTransformFeedbackBuffers*.

The xfb_offset qualifier assigns a byte offset within a transform feedback buffer. Only variables, block members, or blocks can be qualified with xfb_offset. If a block is qualified with xfb_offset, all its members are assigned transform feedback buffer offsets. If a block is not qualified with xfb_offset, any members of that block not qualified with an xfb_offset will not be assigned transform feedback buffer offsets. Only variables and block members that are assigned offsets will be captured (thus, a proper subset of a block can be captured). Each time such a variable or block member is written in a shader, the written value is captured at the assigned offset. If such a block member or variable is not written during a shader invocation, the buffer contents at the assigned offset will be undefined. Even if there are no static writes to a variable or member that is assigned a transform feedback offset, the space is still allocated in the buffer and still affects the stride.

Variables and block members qualified with xfb_offset can be scalars, vectors, matrices, structures, and (sized) arrays of these. The offset must be a multiple of the size of the first component of the first qualified variable or block member, or a compile-time error results. Further, if applied to an aggregate containing a double, the offset must also be a multiple of 8, and the space taken in the buffer will be a multiple of 8. The given offset applies to the first component of the first member of the qualified entity. Then, within the qualified entity, subsequent components are each assigned, in order, to the next available offset aligned to a multiple of that componentÕs size. Aggregate types are flattened down to the component level to get this sequence of components. It is a compile-time error to apply xfb_offset to the declaration of an unsized array.

No aliasing in output buffers is allowed: It is a compile-time or link-time error to specify variables with overlapping transform feedback offsets.

The xfb_stride qualifier specifies how many bytes are consumed by each captured vertex. It applies to the transform feedback buffer for that declaration, whether it is inherited or explicitly declared. It can be applied to variables, blocks, block members, or just the qualifier out. If the buffer is capturing any outputs with double-precision components, the stride must be a multiple of 8, otherwise it must be a multiple of 4, or a compile-time or link-time error results. It is a compile-time or link-time error to have any xfb_offset that overflows xfb_stride, whether stated on declarations before or after the xfb_stride, or in different compilation units. While xfb_stride can be declared multiple times for the same buffer, it is a compile-time or link-time error to have different values specified for the stride for the same buffer.

For example:

When no xfb_stride is specified for a buffer, the stride of the buffer will be the smallest needed to hold the variable placed at the highest offset, including any required padding. For example:

The resulting stride (implicit or explicit), when divided by 4, must be less than or equal to the implementation-dependent constant *gl_MaxTransformFeedbackInterleavedComponents*.

Tessellation Control Outputs

Other than for the transform feedback layout qualifiers, tessellation control shaders allow output layout qualifiers only on the interface qualifier out, not on an output block, block member, or variable declaration. The output layout qualifier identifiers allowed for tessellation control shaders are:

layout-qualifier-id :

vertices = *layout-qualifier-value*

The identifier vertices specifies the number of vertices in the output patch produced by the tessellation control shader, which also specifies the number of times the tessellation control shader is invoked. It is a compile- or link-time error for the output vertex count to be less than or equal to zero, or greater than the implementation-dependent maximum patch size.

The intrinsically declared tessellation control output array *gl_out[]* will also be sized by any output layout declaration. Hence, the expression

```
```

will return the output patch vertex count specified in a previous output layout qualifier. For outputs declared without an array size, including intrinsically declared outputs (i.e., *gl_out*), a layout must be declared before any use of the method length() or other array use that requires its size to be known.

It is a compile-time error if the output patch vertex count specified in an output layout qualifier does not match the array size specified in any output variable declaration in the same shader.

All tessellation control shader layout declarations in a program must specify the same output patch vertex count. There must be at least one layout qualifier specifying an output patch vertex count in any program containing tessellation control shaders; however, such a declaration is not required in all tessellation control shaders.

Geometry Outputs

Geometry shaders can have three additional types of output layout identifiers: an output *primitive type*, a maximum output *vertex count*, and per-output *stream* numbers. The primitive type and vertex count identifiers are allowed only on the interface qualifier out, not on an output block, block member, or variable declaration. The stream identifier is allowed on the interface qualifier out, on output blocks, and on variable declarations.

The layout qualifier identifiers for geometry shader outputs are

layout-qualifier-id :

points
line_strip

triangle_strip
max_vertices = *layout-qualifier-value*
stream = *layout-qualifier-value*

The primitive type identifiers points, line_strip, and triangle_strip are used to specify the type of output primitive produced by the geometry shader, and only one of these is accepted. At least one geometry shader (compilation unit) in a program must declare an output primitive type, and all geometry shader output primitive type declarations in a program must declare the same primitive type. It is not required that all geometry shaders in a program declare an output primitive type.

The vertex count identifier max_vertices is used to specify the maximum number of vertices the shader will ever emit in a single invocation. At least one geometry shader (compilation unit) in a program must declare a maximum output vertex count, and all geometry shader output vertex count declarations in a program must declare the same count. It is not required that all geometry shaders in a program declare a count.

In this example,

all outputs from the geometry shader are triangles and at most 60 vertices will be emitted by the shader. It is an error for the maximum number of vertices to be greater than *gl_MaxGeometryOutputVertices.*

The identifier stream is used to specify that a geometry shader output variable or block is associated with a particular vertex stream (numbered beginning with zero). A default stream number may be declared at global scope by qualifying interface qualifier out as in this example:

The stream number specified in such a declaration replaces any previous default and applies to all subsequent block and variable declarations until a new default is established. The initial default stream number is zero.

Each output block or non-block output variable is associated with a vertex stream. If the block or variable is declared with the stream identifier, it is associated with the specified stream; otherwise, it is associated with the current default stream. A block member may be declared with a stream identifier, but the specified stream must match the stream associated with the containing block. One example:

Each vertex emitted by the geometry shader is assigned to a specific stream, and the attributes of the emitted vertex are taken from the set of output blocks and variables assigned to the targeted stream. After each vertex is emitted, the values of all output variables become undefined. Additionally, the output variables associated with each vertex stream may share storage. Writing to an output variable associated with one stream may overwrite output variables associated with any other stream. When emitting each vertex, a geometry shader should write to all outputs associated with the stream to which the vertex will be emitted and to no outputs associated with any other stream.

If a geometry shader output block or variable is declared more than once, all such declarations must associate the variable with the same vertex stream. If any stream declaration specifies a non-existent stream number, the shader will fail to compile.

Built-in geometry shader outputs are always associated with vertex stream zero.

All geometry shader output layout declarations in a program must declare the same layout and same value for max_vertices. If geometry shaders are in a program, there must be at least one geometry output layout declaration somewhere in that program, but not all geometry shaders (compilation units) are required to declare it.

Fragment Outputs

The built-in fragment shader variable *gl_FragDepth* may be redeclared using one of the following layout qualifiers.

layout-qualifier-id :

- depth_any
- depth_greater
- depth_less
- depth_unchanged

The layout qualifier for *gl_FragDepth* constrains intentions of the final value of *gl_FragDepth* written by any shader invocation. GL implementations are allowed to perform optimizations assuming that the depth test fails (or passes) for a given fragment if all values of *gl_FragDepth* consistent with the layout qualifier would fail (or pass). This potentially includes skipping shader

execution if the fragment is discarded because it is occluded and the shader has no side effects. If the final value of *gl_FragDepth* is inconsistent with its layout qualifier, the result of the depth test for the corresponding fragment is undefined. However, no error will be generated in this case. If the depth test passes and depth writes are enabled, the value written to the depth buffer is always the value of *gl_FragDepth*, whether or not it is consistent with the layout qualifier.

By default, *gl_FragDepth* is qualified as *depth_any*. When the layout qualifier for *gl_FragDepth* is *depth_any*, the shader compiler will note any assignment to *gl_FragDepth* modifying it in an unknown way, and depth testing will always be performed after the shader has executed. When the layout qualifier is *depth_greater*, the GL can assume that the final value of *gl_FragDepth* is greater than or equal to the fragment's interpolated depth value, as given by the *z* component of *gl_FragCoord*. When the layout qualifier is *depth_less*, the GL can assume that any modification of *gl_FragDepth* will only decrease its value. When the layout qualifier is *depth_unchanged*, the shader compiler will honor any modification to *gl_FragDepth*, but the rest of the GL can assume that *gl_FragDepth* is not assigned a new value.

Redeclarations of *gl_FragDepth* are performed as follows:

If *gl_FragDepth* is redeclared in any fragment shader in a program, it must be redeclared in all fragment shaders in that program that have static assignments to *gl_FragDepth*. All redeclarations of *gl_FragDepth* in all fragment shaders in a single program must have the same set of qualifiers. Within any shader, the first redeclarations of *gl_FragDepth* must appear before any use of *gl_FragDepth*. The built-in *gl_FragDepth* is only predeclared in fragment shaders, so redeclaring it in any other shader language results in a compile-time error.

4.4.3. Uniform Variable Layout Qualifiers

Layout qualifiers can be used for uniform variables and subroutine uniforms. The layout qualifier identifiers for uniform variables and subroutine uniforms are:

layout-qualifier-id :

location = *layout-qualifier-value*

The location identifier can be used with default-block uniform variables and subroutine uniforms.

The location specifies the location by which the OpenGL API can reference the uniform and update its value. Individual elements of a uniform array are assigned consecutive locations with the first element taking location location. No two default-block uniform variables in the program can have the same location, even if they are unused, otherwise a compile-time or link-time error will be generated. No two subroutine uniform variables can have the same location in the same shader stage, otherwise a compile-time or link-time error will be generated. Valid locations for default-block uniform variable locations are in the range of 0 to the implementation-defined maximum number of uniform locations minus one. Valid locations for subroutine uniforms are in the range of 0 to the implementation-defined per-stage maximum number of subroutine uniform locations minus one.

Locations can be assigned to default-block uniform arrays and structures. The first inner-most scalar, vector or matrix member or element takes the specified location and the compiler assigns the next inner-most member or element the next incremental location value. Each subsequent inner-most member or element gets incremental locations for the entire structure or array. This rule applies to nested structures and arrays and gives each inner-most scalar, vector, or matrix member a unique location. For arrays without an explicit size, the size is calculated based on its static usage. When the linker generates locations for uniforms without an explicit location, it assumes for all uniforms with an explicit location all their array elements and structure members are used and the linker will not generate a conflicting location, even if that element or member is deemed unused.

When generating SPIR-V for OpenGL, it is a compile-time error to not include a location for individual (default block) non-opaque uniform variables.

4.4.4. Subroutine Function Layout Qualifiers

Layout qualifiers can be used for subroutine functions. The layout qualifier identifiers for subroutine functions are:

layout-qualifier-id :

index = *layout-qualifier-value*

Each subroutine with an index qualifier in the shader must be given a unique index, otherwise a compile- or link-time error will be generated. The indices must be in the range of 0 to the implementation defined maximum number of subroutines minus one. It is recommended, but not required, that the shader assigns a range of tightly packed *index* values starting from zero so that the OpenGL subroutine function enumeration API returns a non-empty name for all active indices.

4.4.5. Uniform and Shader Storage Block Layout Qualifiers

Layout qualifiers can be used for uniform and shader storage blocks, but not for non-block uniform declarations. The layout qualifier identifiers (and shared keyword) for uniform and shader storage blocks are:

layout-qualifier-id :

shared
packed
std140

std430
row_major
column_major
binding = *layout-qualifier-value*
offset = *layout-qualifier-value*
align = *layout-qualifier-value*

None of these have any semantic effect at all on the usage of the variables being declared; they only describe how data is laid out in memory. For example, matrix semantics are always column-based, as described in the rest of this specification, no matter what layout qualifiers are being used.

Uniform and shader storage block layout qualifiers can be declared for global scope, on a single uniform or shader storage block, or on a single block member declaration.

Default layouts are established at global scope for uniform blocks as:

and for shader storage blocks as:

When this is done, the previous default qualification is first inherited and then overridden as per the override rules listed below for each qualifier listed in the declaration. The result becomes the new default qualification scoped to subsequent uniform or shader storage block definitions.

The initial state of compilation when generating SPIR-V is as if the following were declared:

The initial state of compilation when not generating SPIR-V is as if the following were declared:

Uniform and shader storage blocks can be declared with optional layout qualifiers, and so can their individual member declarations. Such block layout qualification is scoped only to the content of the block. As with global layout declarations, block layout qualification first inherits from the current default qualification and then overrides it. Similarly, individual member layout qualification is scoped just to the member declaration, and inherits from and overrides the block's qualification.

The shared qualifier overrides only the std140, std430, and packed qualifiers; other qualifiers are inherited. The compiler/linker will ensure that multiple programs and programmable stages containing this definition will share the same memory layout for this block, as long as all arrays are declared with explicit sizes and all matrices have matching row_major and/or column_major

qualifications (which may come from a declaration outside the block definition). This allows use of the same buffer to back the same block definition across different programs. It is a compile-time error to use the shared qualifier when generating SPIR-V.

The packed qualifier overrides only std140, std430, and shared; other qualifiers are inherited. When packed is used, no shareable layout is guaranteed. The compiler and linker can optimize memory use based on what variables actively get used and on other criteria. Offsets must be queried, as there is no other way of guaranteeing where (and which) variables reside within the block.

It is a link-time error to access the same packed uniform or shader storage block in multiple stages within a program. Attempts to access the same packed uniform or shader storage block across programs can result in conflicting member offsets and in undefined values being read. However, implementations may aid application management of packed blocks by using canonical layouts for packed blocks. It is a compile-time error to use the packed qualifier when generating SPIR-V.

The std140 and std430 qualifiers override only the packed, shared, std140, and std430 qualifiers; other qualifiers are inherited. The std430 qualifier is supported only for shader storage blocks; a shader using the std430 qualifier on a uniform block will fail to compile.

The layout is explicitly determined by this, as described in section 7.6.2.2 ÒStandard Uniform Block LayoutÓ of the OpenGL Specification. Hence, as in shared above, the resulting layout is shareable across programs.

Layout qualifiers on member declarations cannot use the shared, packed, std140, or std430 qualifiers. These can only be used at global scope (without an object) or on a block declaration, or a compile-time error results.

The row_major and column_major qualifiers only affect the layout of matrices, including all matrices contained in structures and arrays they are applied to, to all depths of nesting. These qualifiers can be applied to other types, but will have no effect.

The row_major qualifier overrides only the column_major qualifier; other qualifiers are inherited. Elements within a matrix row will be contiguous in memory.

The column_major qualifier overrides only the row_major qualifier; other qualifiers are inherited. Elements within a matrix column will be contiguous in memory.

The binding qualifier specifies the uniform buffer binding point corresponding to the uniform or shader storage block, which will be used to obtain the values of the member variables of the block. It is a compile-time error to specify the binding qualifier for the global scope or for block member declarations. Any uniform or shader storage block declared without a binding qualifier is initially assigned to block binding point zero. After a program is linked, the binding points used for uniform and shader storage blocks declared with or without a binding qualifier can be updated by the OpenGL API.

If the binding qualifier is used with a uniform block or shader storage block instanced as an array, the first element of the array takes the specified block binding and each subsequent element takes the next consecutive binding point. For an array of arrays, each element (e.g. 6 elements for a[2][3]) gets a binding point, and they are ordered per the array of array ordering described in ÒArrays.Ó

If the binding point for any uniform or shader storage block instance is less than zero, or greater than or equal to the corresponding implementation-dependent maximum number of buffer bindings, a compile-time error will occur. When the binding qualifier is used with a uniform or shader storage block instanced as an array of size *N*, all elements of the array from binding through *binding* + *N* - *1* must be within this range. It is a compile-time or link-time error to use the same binding number for more than one uniform block or for more than one buffer block.

When multiple arguments are listed in a layout declaration, the effect will be the same as if they were declared one at a time, in order from left to right, each in turn inheriting from and overriding the result from the previous qualification.

For example

results in the qualification being column_major. Other examples:

The offset qualifier can only be used on block members of blocks declared with std140 or std430 layouts. The offset qualifier forces the qualified member to start at or after the specified *layout-qualifier-value*, which will be its byte offset from the beginning of the buffer. It is a compile-time error to have any offset, explicit or assigned, that lies within another member of the block. When not generating SPIR-V, it is a compile-time error to specify an offset that is smaller than the offset of the previous member in the block. Two blocks linked together in the same program with the same block name must have the exact same set of members qualified with offset and their *layout-qualifier-value* values must be the same, or a link-time error results. The specified offset must be a multiple of the base alignment of the type of the block member it qualifies, or a compile-time error results.

The align qualifier can only be used on blocks or block members, and only for blocks declared with std140 or std430 layouts. The align qualifier makes the start of each block member have a

minimum byte alignment. It does not affect the internal layout within each member, which will still follow the std140 or std430 rules. The specified alignment must be greater than 0 and a power of 2, or a compile-time error results.

The *actual alignment* of a member will be the greater of the specified align alignment and the standard (e.g. std140) base alignment for the member's type. The *actual offset* of a member is computed as follows: If offset was declared, start with that offset, otherwise start with the next available offset. If the resulting offset is not a multiple of the *actual alignment*, increase it to the first offset that is a multiple of the *actual alignment*. This results in the *actual offset* the member will have.

When align is applied to an array, it affects only the start of the array, not the array's internal stride. Both an offset and an align qualifier can be specified on a declaration.

The align qualifier, when used on a block, has the same effect as qualifying each member with the same align value as declared on the block, and gets the same compile-time results and errors as if this had been done. As described in general earlier, an individual member can specify its own align, which overrides the block-level align, but just for that member.

Examples:

4.4.6. Opaque Uniform Layout Qualifiers

Uniform layout qualifiers can be used to bind opaque uniform variables to specific buffers or units. Samplers can be bound to texture image units, images can be bound to image units, and atomic counters can be bound to buffers.

Sampler, image and atomic counter types take the uniform layout qualifier identifier for binding:

layout-qualifier-id :

binding = *layout-qualifier-value*

The identifier binding specifies which unit will be bound. Any uniform sampler or image variable declared without a binding qualifier is initially bound to unit zero. After a program is linked, the unit referenced by a sampler or image uniform variable declared with or without a binding

qualifier can be updated by the OpenGL API.

If the binding qualifier is used with an array, the first element of the array takes the specified unit and each subsequent element takes the next consecutive unit.

If the binding is less than zero, or greater than or equal to the implementation-dependent maximum supported number of units, a compile-time error will occur. When the binding qualifier is used with an array of size *N*, all elements of the array from binding through *binding* + *N* - *1* must be within this range. It is a compile-time or link-time error to use the same binding number for more than one atomic counter, unless the *offset* for the atomic counters sharing the same binding are all different.

A link-time error will result if two shaders in a program specify different *layout-qualifier-value* bindings for the same opaque-uniform name. However, it is not an error to specify a binding on some but not all declarations for the same name, as shown in the examples below.

4.4.7. Atomic Counter Layout Qualifiers

Atomic counter layout qualifiers can be used on atomic counter declarations. The atomic counter qualifiers are:

layout-qualifier-id :

binding = *layout-qualifier-value*
offset = *layout-qualifier-value*

For example,

will establish that the opaque handle to the atomic counter *a* will be bound to atomic counter buffer binding point 2 at an offset of 4 basic machine units into that buffer. The default *offset* for binding point 2 will be post incremented by 4 (the size of an atomic counter).

A subsequent atomic counter declaration will inherit the previous (post incremented) offset. For example, a subsequent declaration of

will establish that the atomic counter *bar* has a binding to buffer binding point 2 at an offset of 8 basic machine units into that buffer. The offset for binding point 2 will again be post-incremented by 4 (the size of an atomic counter).

When multiple variables are listed in a layout declaration, the effect will be the same as if they were declared one at a time, in order from left to right.

Binding points are not inherited, only offsets. Each binding point tracks its own current default *offset* for inheritance of subsequent variables using the same binding. The initial state of compilation is that all binding points have an *offset* of 0. The *offset* can be set per binding point at global scope (without declaring a variable). For example,

Establishes that the next atomic_uint declaration for binding point 2 will inherit *offset* 4 (but does not establish a default binding):

Atomic counters may share the same binding point, but if a binding is shared, their offsets must be either explicitly or implicitly (from inheritance) unique and non overlapping.

Example valid uniform declarations, assuming top of shader:

Example of an invalid uniform declaration:

It is a compile-time error to bind an atomic counter with a binding value greater than or equal to *gl_MaxAtomicCounterBindings*. It is a compile-time error to declare an unsized array of atomic_uint.

4.4.8. Format Layout Qualifiers

Format layout qualifiers can be used on image variable declarations (those declared with a basic type having Òimage Ó in its keyword). The format layout qualifier identifiers for image variable declarations are:

layout-qualifier-id :

float-image-format-qualifier
int-image-format-qualifier
uint-image-format-qualifier
binding = *layout-qualifier-value*

float-image-format-qualifier :

rgba32f
rgba16f
rg32f
rg16f
r11f_g11f_b10f
r32f
r16f
rgba16
rgb10_a2
rgba8
rg16
rg8
r16
r8
rgba16_snorm
rgba8_snorm
rg16_snorm
rg8_snorm
r16_snorm
r8_snorm

int-image-format-qualifier :

rgba32i
rgba16i
rgba8i
rg32i
rg16i
rg8i
r32i
r16i
r8i

uint-image-format-qualifier :

rgba32ui
rgba16ui

rgb10_a2ui
rgba8ui
rg32ui
rg16ui
rg8ui
r32ui
r16ui
r8ui

A format layout qualifier specifies the image format associated with a declared image variable. Only one format qualifier may be specified for any image variable declaration. For image variables with floating-point component types (keywords starting with Òimage Ó), signed integer component types (keywords starting with Òiimage Ó), or unsigned integer component types (keywords starting with Òuimage Ó), the format qualifier used must match the *float-image-format-qualifier*, *int-image-format-qualifier*, or *uint-image-format-qualifier* grammar rules, respectively. It is a compile-time error to declare an image variable where the format qualifier does not match the image variable type.

Any image variable used for image loads or atomic operations must specify a format layout qualifier; it is a compile-time error to pass an image uniform variable or function parameter declared without a format layout qualifier to an image load or atomic function.

Uniforms not qualified with writeonly must have a format layout qualifier. Note that an image variable passed to a function for read access cannot be declared as writeonly and hence must have been declared with a format layout qualifier.

The binding qualifier was described in ÒOpaque Uniform Layout Qualifiers Ó.

4.5. Interpolation Qualifiers

Inputs and outputs that could be interpolated can be further qualified by at most one of the following interpolation qualifiers:

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

The presence of and type of interpolation is controlled by the above interpolation qualifiers as well as the auxiliary storage qualifiers centroid and sample. When no interpolation qualifier is present, smooth interpolation is used. It is a compile-time error to use more than one interpolation qualifier. The auxiliary storage qualifier patch is not used for interpolation; it is a compile-time error to use interpolation qualifiers with patch.

A variable qualified as flat will not be interpolated. Instead, it will have the same value for every fragment within a primitive. This value will come from a single provoking vertex, as described by the OpenGL Specification. A variable qualified as flat may also be qualified as centroid or sample, 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 equation 14.7 of the OpenGL Specification, section 14.5 ÒLine SegmentsÓ.

A variable qualified as noperspective must be interpolated linearly in screen space, as described in equation 3.7 of the OpenGL Specification, section 3.5 ÒLine SegmentsÓ.

When multisample rasterization is disabled, or for fragment shader input variables qualified with neither centroid nor sample, the value of the assigned variable may be interpolated anywhere within the pixel and a single value may be assigned to each sample within the pixel, to the extent permitted by the OpenGL Specification.

When multisample rasterization is enabled, centroid and sample may be used to control the location and frequency of the sampling of the qualified fragment shader input. If a fragment shader input is qualified with centroid, a single value may be assigned to that variable for all samples in the pixel, but that value must be interpolated at a location that lies in both the pixel and in the primitive being rendered, including any of the pixelÕs samples covered by the primitive. Because the location at which the variable is interpolated may be different in neighboring pixels, and derivatives may be computed by computing differences between neighboring pixels, derivatives of centroid-sampled inputs may be less accurate than those for non-centroid interpolated variables. If a fragment shader input is qualified with sample, a separate value must be assigned to that variable for each covered sample in the pixel, and that value must be sampled at the location of the individual sample.

It is a link-time error if, within the same stage, the interpolation qualifiers of variables of the same name do not match.

4.5.1. Redeclaring Built-In Interpolation Variables in the Compatibility Profile

The following predeclared variables can be redeclared with an interpolation qualifier when using the compatibility profile:

Vertex, tessellation control, tessellation evaluation, and geometry languages:

```
```

Fragment language:

```
```

For example,

Ideally, these are redeclared as part of the redeclaration of an interface block, as described in ÒCompatibility Profile Built-In Language VariablesÓ. However, for the above purpose, they can be redeclared as individual variables at global scope, outside an interface block. Such redeclarations also allow adding the transform-feedback qualifiers xfb_buffer, xfb_stride, and xfb_offset to output variables. (Using xfb_buffer on a variable does not change the global default buffer.) A compile-time error will result if a shader has both an interface block redeclaration and a separate redeclaration of a member of that interface block outside the interface block redeclaration.

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.

4.6. Parameter Qualifiers

In addition to precision qualifiers and memory qualifiers, parameters can have these parameter qualifiers.

Qualifier	Meaning
<none: default>	same as in
const	for function parameters that cannot be written to
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 ÒFunction Calling ConventionsÓ.

4.7. 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.

For the purposes of determining if an output from one shader stage matches an input of the next stage, the precision qualifier need not match.

4.7.1. Range and Precision

The precision of stored single- and double-precision floating-point variables is defined by the IEEE 754 standard for 32-bit and 64-bit floating-point numbers. This includes support for NaNs (Not a Number) and Infs (positive or negative infinities) and positive and negative zeros.

The following rules apply to both single and double-precision operations: Signed infinities and zeros are generated as dictated by IEEE, but subject to the precisions allowed in the following table. Any subnormal (denormalized) value input into a shader or potentially generated by any operation in a shader can be flushed to 0. The rounding mode cannot be set and is undefined but must not affect the result by more than 1 ULP. NaNs are not required to be generated. Support for signaling NaNs is not required and exceptions are never raised. Operations including built-in functions that operate on a NaN are not required to return a NaN as the result. However if NaNs are generated, isnan() must return the correct value.

Precisions are expressed in terms of maximum relative error in units of ULP (units in the last place), unless otherwise noted.

For single precision operations, precisions are required as follows:

Operation	Precision
a + *b*, *a* - *b*, *a* * *b*	Correctly rounded.
<, <=, ==, >, >=	Correct result.
a / *b*, 1.0 / *b*	2.5 ULP for |b| in the range [2^{-126}, 2^{126}].
a * *b* + *c*	Correctly rounded single operation or sequence of two correctly rounded operations.
fma()	Inherited from *a* * *b* + *c*.
pow(*x*, *y*)	Inherited from exp2(*y* * log2(*x*)).
exp(*x*), exp2(*x*)	(3 + 2 á |x|) ULP.
log(), log2()	3 ULP outside the range [0.5,2.0]. Absolute error < 2^{-21} inside the range [0.5,2.0].
sqrt()	Inherited from 1.0 / inversesqrt().
inversesqrt()	2 ULP.
implicit and explicit conversions between types	Correctly rounded.

Built-in functions defined in the specification with an equation built from the above operations inherit the above errors. These include, for example, the geometric functions, the common functions, and many of the matrix functions. Built-in functions not listed above and not defined as equations of the above have undefined precision. These include, for example, the trigonometric functions and determinant.

The precision of double-precision operations is at least that of single precision.

4.7.2. Precision Qualifiers

Any single-precision floating-point, integer, or opaque-type declaration can have the type preceded by one of these precision qualifiers:

Qualifier	Meaning
highp	None.
mediump	None.
lowp	None.

For example:

```
```

Literal constants do not have precision qualifiers. Neither do Boolean variables. Neither do constructors.

Precision qualifiers, as with other qualifiers, do not affect 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 ÒFunction Calling ConventionsÓ, function input and output is done through copies, and therefore qualifiers do not have to match.

4.7.3. Default Precision Qualifiers

The precision statement

```
```

can be used to establish a default precision qualifier. The *type* field can be either int, float, or any of the opaque types, and the *precision-qualifier* can be lowp, mediump, or highp.

Any other types or qualifiers will result in a compile-time error. If *type* is float, the directive applies to non-precision-qualified single-precision 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 inner-most 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.

All languages except for the fragment language have the following predeclared globally scoped default precision statements:

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

There are no errors for omission of a precision qualifier; so the above is just for reference of what may happen in OpenGL ES versions of the shading languages.

4.7.4. Available Precision Qualifiers

The built-in macro GL_FRAGMENT_PRECISION_HIGH is defined to one:

This macro is available in all languages except compute.

4.8. 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, consider the situation where two vertex shaders, in different programs, each set *gl_Position* with the same expression, 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*.

4.8.1. 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:

or as part of a declaration when a variable is declared:

Only variables output from a shader can be candidates for invariance. This includes user-defined output variables and the built-in output variables. As only outputs can be declared as invariant, an output from one shader stage will still match an input of a subsequent stage without the input being declared as invariant.

Input or output instance names on blocks are not used when redeclaring built-in variables.

The invariant keyword can be followed by a comma separated list of previously declared identifiers. All uses of invariant must be at global scope or on block members, 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 control flow 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

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.

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.

4.8.2. 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 two shaders of the same language or shaders of two different languages.

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

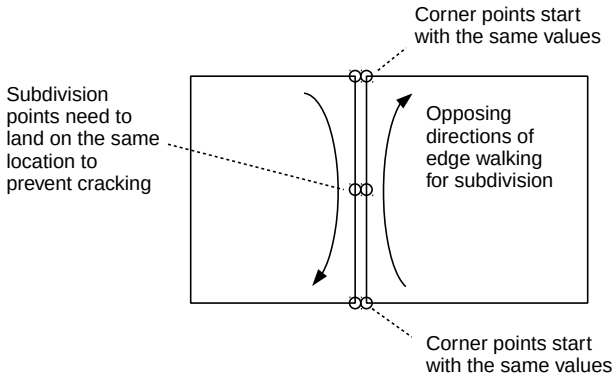
4.9. The Precise Qualifier

Some algorithms require floating-point computations to exactly follow the order of operations specified in the source code and to treat all operations consistently, even if the implementation supports optimizations that could produce nearly equivalent results with higher performance. For example, many GL implementations support a ò multiply-add ó instruction that can compute a floating-point expression such as

```
```

in two operations instead of three operations; one multiply and one multiply-add instead of two multiplies and one add. The result of a floating-point multiply-add might not always be identical to first doing a multiply yielding a floating-point result and then doing a floating-point add. Hence, in this example, the two multiply operations would not be treated consistently; the two multiplies could effectively appear to have differing precisions.

The key computation that needs to be made consistent appears when tessellating, where intermediate points for subdivision are synthesized in different directions, yet need to yield the same result, as shown in the diagram below.



Without any qualifiers, implementations are permitted to perform optimizations that effectively modify the order or number of operations used to evaluate an expression, even if those optimizations may produce slightly different results relative to unoptimized code.

The precise qualifier ensures that operations contributing to a variable's value are done in their stated order and with operator consistency. The order is determined by operator precedence and parenthesis, as described in "Operators". Operator consistency means for each particular operator,

for example the multiply operator (*), its operation is always computed with the same precision. Specifically, values computed by compiler-generated code must adhere to the following identities:

1. a + b = b + a
2. a * b = b * a
3. a * b + c * d = b * a + c* d = d * c + b * a = <any other mathematically valid combination>

While the following are prevented:

1. a + (b + c) is not allowed to become (a + b) + c
2. a * (b * c) is not allowed to become (a * b) * c
3. a * b + c is not allowed to become a single operation fma(a, b, c)

Where *a*, *b*, *c*, and *d*, are scalars or vectors, not matrices. (Matrix multiplication generally does not commute.) It is the shader writerûs responsibility to express the computation in terms of these rules and the compilerûs responsibility to follow these rules. See the description of *gl_TessCoord* for the rules the tessellation stages are responsible for following, which in conjunction with the above allow avoiding cracking when subdividing.

For example,

declares that operations used to produce the value of *position* must be performed in exactly the order specified in the source code and with all operators being treated consistently. As with the invariant qualifier (see ÒThe Invariant QualifierÓ), the precise qualifier may be used to qualify a built-in or previously declared user-defined variable as being precise:

When applied to a block, a structure type, or a variable of structure type, precise applies to each contained member, recursively.

This qualifier will affect the evaluation of an r-value in a particular function if and only if the result is eventually consumed in the same function by an l-value qualified as precise. Any other expressions within a function are not affected, including return values and output parameters not declared as precise but that are eventually consumed outside the function by a variable qualified as precise.

Some examples of the use of precise:

For the purposes of determining if an output from one shader stage matches an input of the next stage, the precise qualifier need not match between the input and the output.

All constant expressions are evaluated as if precise was present, whether or not it is present. However, as described in ÒConstant ExpressionsÓ, there is no requirement that a compile-time constant expression evaluates to the same value as a corresponding non-constant expression.

4.10. Memory Qualifiers

Shader storage blocks, variables declared within shader storage blocks and variables declared as image types (the basic opaque types with Òimage Ó in their keyword), can be further qualified with one or more of the following memory qualifiers:

Qualifier	Meaning
coherent	memory variable where reads and writes are coherent with reads and writes from other shader invocations

Qualifier	Meaning
volatile	memory variable whose underlying value may be changed at any point during shader execution by some source other than the current shader invocation
restrict	memory variable where use of that variable is the only way to read and write the underlying memory in the relevant shader stage
readonly	memory variable that can be used to read the underlying memory, but cannot be used to write the underlying memory
writeonly	memory variable that can be used to write the underlying memory, but cannot be used to read the underlying memory

Memory accesses to image variables declared using the coherent qualifier are performed coherently with accesses to the same location from other shader invocations. In particular, when reading a variable declared as coherent, the values returned will reflect the results of previously completed writes performed by other shader invocations. When writing a variable declared as coherent, the values written will be reflected in subsequent coherent reads performed by other shader invocations.

As described in section 7.12 ÒShader Memory AccessÓ of the OpenGL Specification, shader memory reads and writes complete in a largely undefined order. The built-in function memoryBarrier() can be used if needed to guarantee the completion and relative ordering of memory accesses performed by a single shader invocation.

When accessing memory using variables not declared as coherent, the memory accessed by a shader may be cached by the implementation to service future accesses to the same address. Memory stores may be cached in such a way that the values written may not be visible to other shader invocations accessing the same memory. The implementation may cache the values fetched by memory reads and return the same values to any shader invocation accessing the same memory, even if the underlying memory has been modified since the first memory read. While variables not declared as coherent may not be useful for communicating between shader invocations, using non-coherent accesses may result in higher performance.

Memory accesses to image variables declared using the volatile qualifier must treat the underlying memory as though it could be read or written at any point during shader execution by some source other than the executing shader invocation. When a volatile variable is read, its value must be re-fetched from the underlying memory, even if the shader invocation performing the read had previously fetched its value from the same memory. When a volatile variable is written, its value must be written to the underlying memory, even if the compiler can conclusively determine that its value will be overwritten by a subsequent write. Since the external source reading or writing a volatile variable may be another shader invocation, variables declared as volatile are automatically treated as coherent.

Memory accesses to image variables declared using the restrict qualifier may be compiled assuming that the variable used to perform the memory access is the only way to access the

underlying memory using the shader stage in question. This allows the compiler to coalesce or reorder loads and stores using restrict-qualified image variables in ways that wouldn't be permitted for image variables not so qualified, because the compiler can assume that the underlying image won't be read or written by other code. Applications are responsible for ensuring that image memory referenced by variables qualified with restrict will not be referenced using other variables in the same scope; otherwise, accesses to restrict-qualified variables will have undefined results.

Memory accesses to image variables declared using the readonly qualifier may only read the underlying memory, which is treated as read-only memory and cannot be written to. It is a compile-time error to pass an image variable qualified with readonly to imageStore() or other built-in functions that modify image memory.

Memory accesses to image variables declared using the writeonly qualifier may only write the underlying memory; the underlying memory cannot be read. It is a compile-time error to pass an image variable qualified with writeonly to imageLoad() or other built-in functions that read image memory.

A variable could be qualified as both readonly and writeonly, disallowing both read and write. Such variables can still be used with some queries, for example imageSize() and .length().

The memory qualifiers coherent, volatile, restrict, readonly, and writeonly may be used in the declaration of buffer variables (i.e., members of shader storage blocks). When a buffer variable is declared with a memory qualifier, the behavior specified for memory accesses involving image variables described above applies identically to memory accesses involving that buffer variable. It is a compile-time error to assign to a buffer variable qualified with readonly or to read from a buffer variable qualified with writeonly. The combination readonly writeonly is allowed.

Additionally, memory qualifiers may be used at the block-level declaration of a shader storage block, including the combination readonly writeonly. When a block declaration is qualified with a memory qualifier, it is as if all of its members were declared with the same memory qualifier. For example, the block declaration

```
```

is equivalent to

```
```

Memory qualifiers are only supported in the declarations of image variables, buffer variables, and shader storage blocks; it is an error to use such qualifiers in any other declarations.

When calling user-defined functions, variables qualified with coherent, volatile, readonly, or writeonly may not be passed to functions whose formal parameters lack such qualifiers. (See ÒFunction DefinitionsÓ for more detail on function calling.) It is legal to have any additional memory qualifiers on a formal parameter, but only restrict can be taken away from a calling argument, by a formal parameter that lacks the restrict qualifier.

When a built-in function is called, the code generated is to be based on the actual qualification of the calling argument, not on the list of memory qualifiers specified on the formal parameter in the prototype.

Layout qualifiers cannot be used on formal function parameters, and layout qualification is not included in parameter matching.

Note that the use of const in an image variable declaration is qualifying the const-ness of the variable being declared, not the image it refers to. The qualifier readonly qualifies the image memory (as accessed through that variable) while const qualifies the variable itself.

4.11. Specialization-Constant Qualifier

Specialization constants are used only for SPIR-V and declared using the constant_id layout qualifier. For example:

The above makes a specialization constant with a default value of 12. The number 17 is an example author-chosen id by which the API or other tools can later refer to this specific specialization constant. If it is never changed before final lowering, it will retain the value of 12. It is a compile-time error to use the constant_id qualifier on anything but SPIR-V generation of a scalar bool, int, uint, float, or double.

Built-in constants can be declared to be specialization constants. For example:

The declaration uses just the name of the previously declared built-in variable, with a constant_id layout-qualifier declaration. It is a compile-time error to do this after the constant has been used: Constants are strictly either non-specialization constants or specialization constants, not both.

The built-in constant vector *gl_WorkGroupSize* can be specialized using the local_size_{xyz}_id qualifiers, to individually give the components an id. For example:

This leaves *gl_WorkGroupSize.y* as a non-specialization constant, with *gl_WorkGroupSize* being a partially specialized vector. Its *x* and *z* components can be later specialized, after generating SPIR-V, using the ids 18 and 19. These ids are declared independently from declaring the work-group size:

Existing rules for declaring local_size_x, local_size_y, and local_size_z are not changed. For the local-size ids, it is a compile-time error to provide different id values for the same local-size id, or to provide them after any use. Otherwise, order, placement, number of statements, and replication do not cause errors.

Two arrays sized with specialization constants are the same type only if sized with the same symbol, and involving no operations. For example:

Types containing arrays sized with a specialization constant cannot be compared, assigned as aggregates, declared with an initializer, or used as an initializer. They can, however, be passed as arguments to functions having formal parameters of the same type. Only the outer-most dimension of a variable declared as an array of arrays can be a specialization constant, otherwise a compile-time error results.

Arrays inside a block may be sized with a specialization constant, but the block will have a static layout. Changing the specialized size will not re-layout the block. In the absence of explicit offsets, the layout will be based on the default size of the array.

4.12. Order and Repetition of Qualification

When multiple qualifiers are present in a declaration, they may appear in any order, but they must all appear before the type. The layout qualifier is the only qualifier that can appear more than once. Further, a declaration can have at most one storage qualifier, at most one auxiliary storage qualifier, and at most one interpolation qualifier. If inout is used, neither in nor out may be used. Multiple memory qualifiers can be used. Any violation of these rules will cause a compile-time

error.

4.13. Empty Declarations

Empty declarations are declarations without a variable name, meaning no object is instantiated by the declaration. Generally, empty declarations are allowed. Some are useful when declaring structures, while many others have no effect. For example:

The combinations of qualifiers that cause compile-time or link-time errors are the same whether or not the declaration is empty, for example:

Chapter 5. Operators and Expressions

5.1. Operators

The OpenGL Shading Language has the following operators.

editing-note

(Jon) Operator column needs lots of love.

Precedence	Operator Class	Operators	Associativity
1 (highest)	parenthetical grouping	( )	NA
2	array subscript function call and constructor structure field or method selector, swizzle 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 ÒStructure and Array OperationsÓ.

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 ÒFunction DefinitionsÓ.

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 ÒShading Language GrammarÓ 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 member. For the other types, the arguments must provide a sufficient number of components to perform the initialization, and it is a compile-time 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:

When constructors are used to convert a floating-point type to an integer type, 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 any integer or floating-point type 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 any integer or floating-point type, 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 vector's 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 its 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 a compile-time 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 a compile-time error to have any other arguments.

If the basic type (bool, int, float, or double) 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:

Some examples of these are:

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

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.

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:

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:

The arguments to the constructor will be used to set the structure's members, in order, using one

argument per member. Each argument must be the same type as the member it sets, or be a type that can be converted to the member's type according to section '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,

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 'Implicit Conversions'.

Arrays of arrays are similarly constructed, and the size for any dimension is optional

5.5. Vector and Scalar Components and Length

The names of the components of a vector or scalar 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 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. They are also the names of the only component in a scalar.

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 type is a compile-time error so, for example:

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

No more than 4 components can be selected.

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

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 or scalar r-value.

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

To form an l-value, swizzling must further be applied to an l-value, 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 (but not to scalars) to provide numeric indexing. So in

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 results in a compile-time error. 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.

The length() method may be applied to vectors (but not scalars). The result is the number of components in the vector. For example,

sets the constant *L* to 3. The type returned by .length() on a vector is int, and the value returned is a constant expression.

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 (column size of 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.

Behavior is undefined when accessing a component outside the bounds of a matrix with a non-

constant expression. It is a compile-time error to access a matrix with a constant expression that is outside the bounds of the matrix.

The length() method may be applied to matrices. The result is the number of columns of the matrix. For example,

```
```

sets the constant *L* to 3. The type returned by .length() on a matrix is int, and the value returned is a constant expression.

5.7. Structure and Array Operations

The members 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 selector	.
equality	== !=
assignment	=
Ternary operator	?:
Sequence operator	,
indexing (arrays only)	[ ]

The equality operators and assignment operator are only allowed if the two operands are same size and type. The operands cannot contain any opaque types. 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 members 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

```
```

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:

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 ÒImplicit ConversionsÓ 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 members, 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. Using an incorrect expression as an l-value results in a compile-time error.

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 *lvalue* is the value returned by *lvalue-expression*, *op* is as described below, and the *lvalue-expression* 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, all integral types, all floating-point types, all vector types, and all matrix types.
- Constructors of all types.
- Variable names of all types.
- An array, vector, or matrix expression with the length() method applied.
- Subscripted arrays.
- Function calls that return values. In some cases, function calls returning void are also allowed in expressions as specified below.
- Component field selectors and array subscript results.
- Parenthesized expressions. Any expression, including expressions with void type 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 the fundamental types in the operands do not match, then the conversions from Ò Implicit Conversions Ó are applied to create matching types. All arithmetic binary operators result in the same fundamental type (signed integer, unsigned integer, single-precision floating-point, or double-precision 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. Ò Vector and

Matrix Operationsó explains in more detail how vectors and matrices are operated on.

All other cases result in a compile-time error.

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. If the fundamental types in the operands do not match, then the conversions from ÒImplicit Conversionsó are applied to create matching types. The operands cannot be vectors of differing size; this is a compile-time error. 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 expressionÕs 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 ÒImplicit Conversionsó will be applied to obtain matching types. 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 except opaque types, aggregates that contain opaque types, subroutine uniforms, and aggregates that contain subroutine uniforms. They result in a scalar Boolean. If the operand types do not match, then there must be a conversion from ÒImplicit Conversionsó applied to one operand that can make them match, in which case this conversion is done. For vectors, matrices, structures, and arrays, all components, members, or elements of one operand must equal the corresponding components, members, 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 operands to the sequence operator may have void type.
- 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, including void, as long their types match, or there is a conversion in section ÒImplicit ConversionsÓ 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 oneÕs complement operator (~). The operand must be of type signed or unsigned integer or integer vector, and the result is the oneÕs 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 with the same size as the first operand, 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 expressionÕs 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; this is a compile-time error. 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. If the fundamental types in the operands do not match, then the conversions from ÒImplicit ConversionsÓ are applied to create matching types, and this 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 ÒShading Language GrammarÓ.

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,

will be equivalent to

And

will be equivalent to

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.

is equivalent to

And

is equivalent to

And

is equivalent to

and similarly for other sizes of vectors and matrices.

5.11. Out-of-Bounds Accesses

In the subsections described above for array, vector, matrix and structure accesses, any out-of-bounds access produced undefined behavior. However, if robust buffer access is enabled via the OpenGL API, such accesses will be bound within the memory extent of the active program. It will not be possible to access memory from other programs, and accesses will not result in abnormal program termination. Out-of-bounds reads return undefined values, which include values from other variables of the active program or zero. Out-of-bounds writes may be discarded or overwrite other variables of the active program, depending on the value of the computed index and how this relates to the extent of the active program's memory. Applications that require defined behavior for out-of-bounds accesses should range check all computed indices before dereferencing an array.

5.12. Specialization Constant Operations

Only some operations discussed in this section may be applied to a specialization constant and still yield a result that is a specialization constant. The operations that do so are listed below. When a specialization constant is operated on with one of these operators and with another constant or specialization constant, the result is implicitly a specialization constant.

- int(), uint(), and bool() constructors for type conversions from any of the following types to any of the following types:
  - int

- uint
- bool

¥ vector versions of the above conversion constructors

¥ allowed implicit conversions of the above

¥ swizzles (e.g.)

¥ the following when applied to integer or unsigned integer types:

- unary negative (-)
- binary operations (+, -, *, /, %)
- shift (<<, >>)
- bitwise operations (&, |, ^)

¥ the following when applied to integer or unsigned integer scalar types:

- comparison (==, !=, >, >=, <, !)

¥ The following when applied to the Boolean scalar type:

- not (!)
- logical operations (&&, | |, ^^)
- comparison (==, !=)

¥ the ternary operator (?:)

Chapter 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 ÒShading Language GrammarÓ 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:

and a function is defined like

where *returnType* must be present and cannot be void, or:

If the type of *returnValue* does not match *returnType*, there must be an implicit conversion in ÒImplicit ConversionsÓ that converts the type of *returnValue* to *returnType*, or a compile-time error will result.

Each of the *typeN* must include a type and can optionally include parameter qualifiers. The formal argument names (*args* above) in the declarations are optional for both the declaration and definition forms.

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 functionÕs declaration.

Structures are also allowed as argument types. The return type can also be a structure.

See ÒShading Language GrammarÓ 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:

Functions that return no value must be declared as void. A void function can only use return without a return argument, even if the return argument has void type. Return statements only accept values:

Only a precision qualifier is allowed on the return type of a function. Formal parameters can have parameter, precision, and memory qualifiers, but no other qualifiers.

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 Ò( )Ó is declared. The idiom Ò(void)Ó 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.

For example,

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 ÒImplicit ConversionsÓ 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.

If implicit conversions can be used to find more than one matching function, a single best-matching function is sought. To determine a best match, the conversions between calling argument and formal parameter types are compared for each function argument and pair of matching functions.

After these comparisons are performed, each pair of matching functions are compared. A function declaration *A* is considered a better match than function declaration *B* if

- for at least one function argument, the conversion for that argument in *A* is better than the corresponding conversion in *B*; and
- there is no function argument for which the conversion in *B* is better than the corresponding conversion in *A*.

If a single function declaration is considered a better match than every other matching function declaration, it will be used. Otherwise, a compile-time semantic error for an ambiguous overloaded function call occurs.

To determine whether the conversion for a single argument in one match is better than that for another match, the following rules are applied, in order:

1. An exact match is better than a match involving any implicit conversion.
2. A match involving an implicit conversion from float to double is better than a match involving any other implicit conversion.
3. A match involving an implicit conversion from either int or uint to float is better than a match involving an implicit conversion from either int or uint to double.

If none of the rules above apply to a particular pair of conversions, neither conversion is considered better than the other.

For the example function prototypes (A), (B), and © above, the following examples show how the rules apply to different sets of calling argument types:

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, or a link-time error results. This function takes no arguments, returns no value, and must be declared as type void:

The function *main* can contain uses of return. See ÒJumpsÓ for more details.

It is a compile-time or link-time 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. It means the same thing as specifying both in and 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 functionÕs 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, or a compile-time error results.

function-prototype :

precision-qualifier type function-name (*parameter-qualifiers precision-qualifier type name array-specifier* , É)

type :

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

parameter-qualifiers :

empty
list of *parameter-qualifier*

> ! *editing-note*
> (Jon) Wouldn't this make more sense as instead of 'list of' ?

parameter-qualifier :

const
in
out
inout
precise
memory-qualifier
precision-qualifier

name :

empty
identifier

array-specifier :

empty
[*integral-constant-expression*]

The const qualifier cannot be used with out or inout, or a compile-time error results. The above is used both 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 a program contains cycles. This includes all potential function calls through variables declared as subroutine uniform (described below). It is a compile-time or link-time error if a single compilation unit (shader) contains either static recursion or the potential for recursion through subroutine variables.

6.1.2. Subroutines

Subroutines provide a mechanism allowing shaders to be compiled in a manner where the target of one or more function calls can be changed at run-time without requiring any shader recompilation. For example, a single shader may be compiled with support for multiple illumination algorithms to handle different kinds of lights or surface materials. An application using such a shader may switch illumination algorithms by changing the value of its subroutine uniforms. To use subroutines, a subroutine type is declared, one or more functions are associated with that subroutine type, and a subroutine variable of that type is declared. The function currently assigned to the variable function is then called by using function calling syntax replacing a function name with the name of

the subroutine variable. Subroutine variables are uniforms, and are assigned to specific functions only through commands (UniformSubroutinesuiv) in the OpenGL API.

Subroutine functionality is not available when generating SPIR-V.

Subroutine types are declared using a statement similar to a function declaration, with the subroutine keyword, as follows:

As with function declarations, the formal argument names (*args* above) are optional. Functions are associated with subroutine types of matching declarations by defining the function with the subroutine keyword and a list of subroutine types the function matches:

It is a compile-time error if arguments and return type don't match between the function and each associated subroutine type.

Functions declared with subroutine must include a body. An overloaded function cannot be declared with subroutine; a program will fail to compile or link if any shader or stage contains two or more functions with the same name if the name is associated with a subroutine type.

A function declared with subroutine can also be called directly with a static use of *functionName*, as is done with non-subroutine function declarations and calls.

Subroutine type variables are required to be *subroutine uniforms*, and are declared with a specific subroutine type in a subroutine uniform variable declaration:

Subroutine uniform variables are called the same way functions are called. When a subroutine variable (or an element of a subroutine variable array) is associated with a particular function, all function calls through that variable will call that particular function.

Unlike other uniform variables, subroutine uniform variables are scoped to the shader execution stage the variable is declared in.

Subroutine variables may be declared as explicitly-sized arrays, which can be indexed only with dynamically uniform expressions.

It is a compile-time error to use the subroutine keyword in any places other than (as shown above) to

¥ declare a subroutine type at global scope,

¥ declare a function as a subroutine, or

¥ declare a subroutine variable at global scope.

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-statement-list*$_{opt}$ }

Where *switch-statement-list* is a nested scope containing 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*

Note the above grammarÕs purpose is to aid discussion in this section; the normative grammar is in ÒShading Language GrammarÓ.

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. The type of the *constant-expression* value in a case label also must be a scalar integer. When any pair of these values is tested for Òequal valueÓ and the types do not match, an implicit conversion will be done to convert the int to a uint (see ÒImplicit ConversionsÓ) before the compare is done. If a case label has a *constant-expression* of equal value to *init-expression*, 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 a compile-time 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 a compile-time error to have no statement between a label and the end of the switch statement. No statements are allowed in a switch statement before the first case statement.

The case and default labels can only appear within a switch statement. No case or default labels can be nested inside other statements or compound statements within their corresponding switch.

6.3. Iteration

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

See ÒShading Language GrammarÓ 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 variableÕs 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 ÒgotoÓ or 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. Any prior writes to other buffers such as shader storage buffers are unaffected. Control flow exits the shader, and subsequent implicit or explicit derivatives are undefined when this control flow is non-uniform (meaning different fragments within the primitive take different control paths). It would typically be used within a conditional statement, for example:

A fragment shader may test a fragmentÕs 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.

Chapter 7. Built-In Variables

7.1. Built-In Language Variables

Some OpenGL operations occur in fixed functionality and need to provide values to or receive values from shader executables. Shaders communicate with fixed-function OpenGL pipeline stages, and optionally with other shader executables, through the use of built-in input and output variables.

7.1.1. Vertex Shader Special Variables

The built-in vertex shader variables are intrinsically declared as follows:

The variable *gl_Position* is intended for writing the homogeneous vertex position. It can be written at any time during shader execution. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations, if present, that operate on primitives after vertex processing has occurred. Its value is undefined after the vertex processing stage if the vertex shader executable does not write *gl_Position*.

The variable *gl_PointSize* is intended for a shader to write the size of the point to be rasterized. It is measured in pixels. If *gl_PointSize* is not written to, its value is undefined in subsequent pipe stages.

The variable *gl_ClipDistance* is intended for writing clip distances, and provides the forward compatible mechanism for controlling user clipping. The element *gl_ClipDistance[i]* specifies a clip distance 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 explicitly sized by the shader either redeclaring it with a size or implicitly sized by indexing it only with constant integral 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_CullDistance* provides a mechanism for controlling user culling. The element *gl_CullDistance[i]* specifies a cull distance for plane *i*. A distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the cull volume, and a negative distance means the point is outside the cull volume. Primitives whose vertices all have a negative clip distance for plane *i* will be discarded.

The *gl_CullDistance* array is predeclared as unsized and must be sized by the shader either redeclaring it with a size or indexing it only with constant integral expressions. The size determines the number and set of enabled cull distances and can be at most *gl_MaxCullDistances*. The number of varying components (see *gl_MaxVaryingComponents*) consumed by *gl_CullDistance* will match the size of the array. Shaders writing *gl_CullDistance* must write all enabled distances, or culling results are undefined.

As an output variable, *gl_CullDistance* provides the place for the shader to write these distances. As an input in all but the fragment language, it reads the values written in the previous shader stage. In the fragment language, *gl_CullDistance* array contains linearly interpolated values for the vertex values written by a shader to the *gl_CullDistance* vertex output variable.

It is a compile-time or link-time error for the set of shaders forming a program to have the sum of the sizes of the *gl_ClipDistance* and *gl_CullDistance* arrays to be larger than *gl_MaxCombinedClipAndCullDistances*.

The variable *gl_VertexID* is a vertex shader input variable that holds an integer index for the vertex, as defined under ÒShader InputsÓ in section 11.1.3.9 ÒShader InputsÓ of the OpenGL Specification. While the variable *gl_VertexID* is always present, its value is not always defined.

The variable *gl_InstanceID* is a vertex shader input variable that holds the instance number of the current primitive in an instanced draw call (see ÒShader InputsÓ in section 11.1.3.9 ÒShader InputsÓ of the OpenGL Specification). If the current primitive does not come from an instanced draw call, the value of *gl_InstanceID* is zero.

The variable *gl_DrawID* is a vertex shader input variable that holds the integer index of the drawing command to which the current vertex belongs (see ÒShader InputsÓ in section 11.1.3.9 of the OpenGL Specification). If the vertex is not invoked by a Multi* form of a draw command, then the value of *gl_DrawID* is zero.

The variable *gl_BaseVertex* is a vertex shader input variable that holds the integer value passed to the baseVertex parameter of the command that resulted in the current shader invocation (see ÒShader InputsÓ in section 11.1.3.9 of the OpenGL Specification).

The variable *gl_BaseInstance* is a vertex shader input variable that holds the integer value passed to the baseInstance parameter of the command that resulted in the current shader invocation (see ÒShader InputsÓ in section 11.1.3.9 of the OpenGL Specification).

7.1.2. Tessellation Control Shader Special Variables

In the tessellation control shader, built-in variables are intrinsically declared as:

Tessellation Control Input Variables

gl_Position, *gl_PointSize*, *gl_ClipDistance*, and *gl_CullDistance* contain the values written in the previous shader stage to the corresponding outputs.

gl_PatchVerticesIn contains the number of vertices in the input patch being processed by the shader. A single shader can read patches of differing sizes, so the value of *gl_PatchVerticesIn* may differ between patches.

gl_PrimitiveID contains the number of primitives processed by the shader since the current set of rendering primitives was started.

gl_InvocationID contains the number of the output patch vertex assigned to the tessellation control shader invocation. It is assigned integer values in the range [0, N-1], where N is the number of output patch vertices per primitive.

Tessellation Control Output Variables

gl_Position, *gl_PointSize*, *gl_ClipDistance*, and *gl_CullDistance* are used in the same fashion as the corresponding output variables in the vertex shader.

The values written to *gl_TessLevelOuter* and *gl_TessLevelInner* are assigned to the corresponding outer and inner tessellation levels of the output patch. They are used by the tessellation primitive generator to control primitive tessellation and may be read by tessellation evaluation shaders.

7.1.3. Tessellation Evaluation Shader Special Variables

In the tessellation evaluation shader, built-in variables are intrinsically declared as:

```
```

Tessellation Evaluation Input Variables

gl_Position, *gl_PointSize*, *gl_ClipDistance*, and *gl_CullDistance* contain the values written in the previous shader stage to the corresponding outputs.

gl_PatchVerticesIn and *gl_PrimitiveID* are defined in the same fashion as the corresponding input variables in the tessellation control shader.

gl_TessCoord specifies a three-component *(u,v,w)* vector identifying the position of the vertex being processed by the shader relative to the primitive being tessellated. Its values will obey the properties

```
```

to aid in replicating subdivision computations.

If a tessellation control shader is active, the input variables *gl_TessLevelOuter* and *gl_TessLevelInner* are filled with the corresponding outputs written by the tessellation control shader. Otherwise, they are assigned with default tessellation levels specified in section 11.2.3.3 ÒTessellation Evaluation Shader InputsÓ of the OpenGL Specification.

Tessellation Evaluation Output Variables

gl_Position, *gl_PointSize*, *gl_ClipDistance*, and *gl_CullDistance* are used in the same fashion as the corresponding output variables in the vertex shader.

7.1.4. Geometry Shader Special Variables

In the geometry shader, built-in variables are intrinsically declared as:

Geometry Shader Input Variables

gl_Position, *gl_PointSize*, *gl_ClipDistance*, and *gl_CullDistance* contain the values written in the previous shader stage to the corresponding outputs.

gl_PrimitiveIDIn contains the number of primitives processed by the shader since the current set of rendering primitives was started.

gl_InvocationID contains the invocation number assigned to the geometry shader invocation. It is assigned integer values in the range [0, N-1], where N is the number of geometry shader invocations per primitive.

Geometry Shader Output Variables

gl_Position, *gl_PointSize*, *gl_ClipDistance*, and *gl_CullDistance* are used in the same fashion as the corresponding output variables in the vertex shader.

gl_PrimitiveID is filled with a single integer that serves as a primitive identifier to the fragment shader. This is then available to fragment shaders, which will select the written primitive ID from the provoking vertex of the primitive being shaded. If a fragment shader using *gl_PrimitiveID* is

active and a geometry shader is also active, the geometry shader must write to *gl_PrimitiveID* or the fragment shader input *gl_PrimitiveID* is undefined. See section 11.3.4.5 ÒGeometry Shader OutputsÓ of the OpenGL Specification for more information.

gl_Layer is used to select a specific layer (or face and layer of a cube map) of a multi-layer framebuffer attachment. The actual layer used will come from one of the vertices in the primitive being shaded. Which vertex the layer comes from is determined as discussed in section 11.3.4.6 ÒLayer and Viewport SelectionÓ of the OpenGL Specification but may be undefined, so it is best to write the same layer value for all vertices of a primitive. If a shader statically assigns a value to *gl_Layer*, layered rendering mode is enabled. See section 11.3.4.5 ÒGeometry Shader OutputsÓ and section 9.4.9 ÒLayered FramebuffersÓ of the OpenGL Specification for more information. If a shader statically assigns a value to *gl_Layer*, and there is an execution path through the shader that does not set *gl_Layer*, then the value of *gl_Layer* is undefined for executions of the shader that take that path.

The output variable *gl_Layer* takes on a special value when used with an array of cube map textures. Instead of only referring to the layer, it is used to select a cube map face and a layer. Setting *gl_Layer* to the value *layer*6+face* will render to face *face* of the cube defined in layer *layer*. The face values are defined in table 9.3 of section 9.4.9 ÒLayered FramebuffersÓ of the OpenGL Specification, but repeated below for clarity.

Face Value	Resulting Target
0	TEXTURE_CUBE_MAP_POSITIVE_X
1	TEXTURE_CUBE_MAP_NEGATIVE_X
2	TEXTURE_CUBE_MAP_POSITIVE_Y
3	TEXTURE_CUBE_MAP_NEGATIVE_Y
4	TEXTURE_CUBE_MAP_POSITIVE_Z
5	TEXTURE_CUBE_MAP_NEGATIVE_Z

For example, to render to the positive *y* cube map face located in the 5th layer of the cube map array, *gl_Layer* should be set to *5 * 6 + 2*.

The output variable *gl_ViewportIndex* provides the index of the viewport to which the next primitive emitted from the geometry shader should be drawn. Primitives generated by the geometry shader will undergo viewport transformation and scissor testing using the viewport transformation and scissor rectangle selected by the value of *gl_ViewportIndex*. The viewport index used will come from one of the vertices in the primitive being shaded. However, which vertex the viewport index comes from is implementation-dependent, so it is best to use the same viewport index for all vertices of the primitive. If a geometry shader does not assign a value to *gl_ViewportIndex*, viewport transform and scissor rectangle zero will be used. If a geometry shader statically assigns a value to *gl_ViewportIndex* and there is a path through the shader that does not assign a value to *gl_ViewportIndex*, the value of *gl_ViewportIndex* is undefined for executions of the shader that take that path. See section 11.3.4.6 ÒLayer and Viewport SelectionÓ of the OpenGL Specification for more information.

7.1.5. Fragment Shader Special Variables

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

The output of the fragment shader executable is processed by the fixed function operations at the back end of the OpenGL pipeline.

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

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.

If a shader executes the discard keyword, the fragment is discarded, and the values of any user-defined fragment outputs, *gl_FragDepth*, and *gl_SampleMask* 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*, *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 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 or geometry shader.

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.

For both the input array *gl_SampleMaskIn[]* and the output array *gl_SampleMask[]*, bit *B* of mask *M* (*gl_SampleMaskIn[M]* or *gl_SampleMask[M]*) corresponds to sample *32*M+B*. These arrays have ceil(*s*/32) elements, where *s* is the maximum number of color samples supported by the implementation.

The input variable *gl_SampleMaskIn* indicates the set of samples covered by the primitive generating the fragment during multisample rasterization. It has a sample bit set if and only if the sample is considered covered for this fragment shader invocation.

The output array *gl_SampleMask[]* sets the sample mask for the fragment being processed. Coverage for the current fragment will become the logical AND of the coverage mask and the output *gl_SampleMask*. This array must be sized in the fragment shader either implicitly or explicitly, to be no larger than the implementation-dependent maximum sample-mask (as an array of 32bit elements), determined by the maximum number of samples.. If the fragment shader statically assigns a value to *gl_SampleMask*, the sample mask will be undefined for any array elements of any fragment shader invocations that fail to assign a value. If a shader does not statically assign a value to *gl_SampleMask*, the sample mask has no effect on the processing of a fragment.

The input variable *gl_SampleID* is filled with the sample number of the sample currently being processed. This variable is in the range *0* to *gl_NumSamples-1*, where *gl_NumSamples* is the total number of samples in the framebuffer, or 1 if rendering to a non-multisample framebuffer. Any static use of this variable in a fragment shader causes the entire shader to be evaluated per-sample.

The input variable *gl_SamplePosition* contains the position of the current sample within the multisample draw buffer. The *x* and *y* components of *gl_SamplePosition* contain the sub-pixel coordinate of the current sample and will have values in the range 0.0 to 1.0. Any static use of this variable in a fragment shader causes the entire shader to be evaluated per sample.

The value *gl_HelperInvocation* is true if the fragment shader invocation is considered a *helper invocation* and is false otherwise. A helper invocation is a fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations. Such derivatives are computed implicitly in the built-in function texture() (see ÒTexture FunctionsÓ), and explicitly in the derivative functions in ÒDerivative FunctionsÓ, for example dFdx() and dFdy().

Fragment shader helper invocations execute the same shader code as non-helper invocations, but will not have side effects that modify the framebuffer or other shader-accessible memory. In particular:

- Fragments corresponding to helper invocations are discarded when shader execution is complete, without updating the framebuffer.
- Stores to image and buffer variables performed by helper invocations have no effect on the

underlying image or buffer memory.

- Atomic operations to image, buffer, or atomic counter variables performed by helper invocations have no effect on the underlying image or buffer memory. The values returned by such atomic operations are undefined.

Helper invocations may be generated for pixels not covered by a primitive being rendered. While fragment shader inputs qualified with centroid are normally required to be sampled in the intersection of the pixel and the primitive, the requirement is ignored for such pixels since there is no intersection between the pixel and primitive.

Helper invocations may also be generated for fragments that are covered by a primitive being rendered when the fragment is killed by early fragment tests (using the early_fragment_tests qualifier) or where the implementation is able to determine that executing the fragment shader would have no effect other than assisting in computing derivatives for other fragment shader invocations.

The set of helper invocations generated when processing any set of primitives is implementation-dependent.

gl_ClipDistance contains linearly interpolated values for the vertex- pipeline values written by a shader to the *gl_ClipDistance* output variable. Only elements in this array that have clipping enabled will have defined values.

The input variable *gl_PrimitiveID* is filled with the value written to the *gl_PrimitiveID* geometry shader output, if a geometry shader is present. Otherwise, it is filled with the number of primitives processed by the shader since the current set of rendering primitives was started.

The input variable *gl_Layer* is filled with the value written to the *gl_Layer* geometry shader output, if a geometry shader is present. If the geometry stage does not dynamically assign a value to *gl_Layer*, the value of *gl_Layer* in the fragment stage will be undefined. If the geometry stage makes no static assignment to *gl_Layer*, the input value in the fragment stage will be zero. Otherwise, the fragment stage will read the same value written by the geometry stage, even if that value is out of range. If a fragment shader contains a static access to *gl_Layer*, it will count against the implementation defined limit for the maximum number of inputs to the fragment stage.

The input variable *gl_ViewportIndex* is filled with the value written to the output variable *gl_ViewportIndex* in the geometry stage, if a geometry shader is present. If the geometry stage does not dynamically assign a value to *gl_ViewportIndex*, the value of *gl_ViewportIndex* in the fragment shader will be undefined. If the geometry stage makes no static assignment to *gl_ViewportIndex*, the fragment stage will read zero. Otherwise, the fragment stage will read the same value written by the geometry stage, even if that value is out of range. If a fragment shader contains a static access to *gl_ViewportIndex*, it will count against the implementation defined limit for the maximum number of inputs to the fragment stage.

7.1.6. Compute Shader Special Variables

In the compute shader, built-in variables are declared as follows:

The built-in variable *gl_NumWorkGroups* is a compute-shader input variable containing the total number of global work items in each dimension of the work group that will execute the compute shader. Its content is equal to the values specified in the *num_groups_x*, *num_groups_y*, and *num_groups_z* parameters passed to the *DispatchCompute* API entry point.

The built-in constant *gl_WorkGroupSize* is a compute-shader constant containing the local work-group size of the shader. The size of the work group in the *X*, *Y*, and *Z* dimensions is stored in the *x*, *y*, and *z* components. The constants values in *gl_WorkGroupSize* will match those specified in the required local_size_x, local_size_y, and local_size_z layout qualifiers for the current shader. This is a constant so that it can be used to size arrays of memory that can be shared within the local work group. It is a compile-time error to use *gl_WorkGroupSize* in a shader that does not declare a fixed local group size, or before that shader has declared a fixed local group size, using local_size_x, local_size_y, and local_size_z. When a size is given for some of these identifiers, but not all, the corresponding *gl_WorkGroupSize* will have a size of 1.

The built-in variable *gl_WorkGroupID* is a compute-shader input variable containing the three-dimensional index of the global work group that the current invocation is executing in. The possible values range across the parameters passed into *DispatchCompute*, i.e., from (0, 0, 0) to (*gl_NumWorkGroups.x* - 1, *gl_NumWorkGroups.y* - 1, *gl_NumWorkGroups.z* -1).

The built-in variable *gl_LocalInvocationID* is a compute-shader input variable containing the t-dimensional index of the local work group within the global work group that the current invocation is executing in. The possible values for this variable range across the local work group size, i.e., (0,0,0) to (*gl_WorkGroupSize.x* - 1, *gl_WorkGroupSize.y* - 1, *gl_WorkGroupSize.z* - 1).

The built-in variable *gl_GlobalInvocationID* is a compute shader input variable containing the global index of the current work item. This value uniquely identifies this invocation from all other invocations across all local and global work groups initiated by the current *DispatchCompute* call. This is computed as:

The built-in variable *gl_LocalInvocationIndex* is a compute shader input variable that contains the one-dimensional representation of the *gl_LocalInvocationID*. This is computed as:

7.1.7. Compatibility Profile Built-In Language Variables

When using the compatibility profile, the GL can provide fixed functionality behavior for the vertex and fragment programmable pipeline stages. For example, mixing a fixed functionality vertex stage with a programmable fragment stage.

The following built-in vertex, tessellation control, tessellation evaluation, and geometry output variables are available to specify inputs for the subsequent programmable shader stage or the fixed functionality fragment stage. 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. The following members are added to the output *gl_PerVertex* block in these languages:

The output variable *gl_ClipVertex* provides a place for vertex and geometry shaders to write the coordinate to be used with the user clipping planes. Writing to *gl_ClipDistance* is the preferred method for user clipping. It is a compile-time or link-time error for the set of shaders forming a program to statically read or write both *gl_ClipVertex* and either *gl_ClipDistance* or *gl_CullDistance*. If neither *gl_ClipVertex* nor *gl_ClipDistance* is written, their values are undefined and any clipping against user clip planes is also undefined.

Similarly to what was previously described for the core profile, the *gl_PerVertex* block can be redeclared in a shader to explicitly include these additional members. For example:

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.

The output variables *gl_FrontColor*, *glFrontSecondaryColor*, *gl_BackColor*, and *glBackSecondaryColor* assign primary and secondary colors for front and back faces of primitives containing the vertex being processed. The output variable *gl_TexCoord* assigns texture coordinates for the vertex being processed.

For *gl_FogFragCoord*, the value written will be used as the ÒcÓ value in section 16.4 ÒFogÓ of the Compatibility profile of the OpenGL Specification, by the fixed functionality pipeline. For example, if the z-coordinate of the fragment in eye space is desired as ÒcÓ, then thatÕs what the vertex shader executable should write into *gl_FogFragCoord*.

As with all arrays, indices used to subscript *gl_TexCoord* must either be a constant integral expressions, or this array must be redeclared 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 redeclaration of *gl_TexCoord* can also be done at global scope as, for example:

(This treatment is a special case for *gl_TexCoord[]*, not a general method for redeclaring members of blocks.) It is a compile-time error to redeclare *gl_TexCoord[]* at global scope if there is a redeclaration of the corresponding built-in block; only one form of redeclaration is allowed within a shader (and hence within a stage, as block redeclarations must match across all shaders using it).

In the tessellation control, evaluation, and geometry shaders, the outputs of the previous stage described above are also available in the input *gl_PerVertex* block in these languages.

These can be redeclared to establish an explicit pipeline interface, the same way as described above for the output block *gl_PerVertex*, and the input redeclaration must match the output redeclaration of the previous stage. However, when a built-in interface block with an instance name is redeclared (e.g. *gl_in*), the instance name must be included in the redeclaration. It is a compile-time error to not include the built-in instance name or to change its name. For example,

Built-in block arrays predeclared with a size can be redeclared with unsized syntax. This keeps their size equal to the original predeclared size.

Treatment of *gl_TexCoord[]* redeclaration is also identical to that described for the output block *gl_TexCoord[]* redeclaration.

The following fragment input block is also available in a fragment shader when using the compatibility profile:

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 in the primitive producing the fragment. 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 16.4 ÒFogÓ of the Compatibility profile of the OpenGL Specification. The *gl_TexCoord[]* values are the interpolated *gl_TexCoord[]* values from a vertex shader or the texture coordinates of any fixed pipeline based vertex functionality.

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

As described above for the input and output *gl_PerVertex* blocks, the *gl_PerFragment* block can be redeclared to create an explicit interface to another program. When matching these interfaces between separate programs, members in the *gl_PerVertex* output block must be declared if and only if the corresponding fragment shader members generated from them are present in the *gl_PerFragment* input block. These matches are described in detail in section 7.4.1 ÒShader Interface MatchingÓ of the OpenGL Specification. If they don't match within a program, a link-time error will result. If the mismatch is between two programs, values passed between programs are undefined. Unlike with all other block matching, the order of declaration within *gl_PerFragment* does not have to match across shaders and does not have to correspond with order of declaration in a matching *gl_PerVertex* redeclaration.

The following fragment output variables are available in a fragment shader when using the compatibility profile:

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.

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 or link-time errors.

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

7.2. Compatibility Profile Vertex Shader Built-In Inputs

The following predeclared input names can be used from within a vertex shader to access the current values of OpenGL state when using the compatibility profile.

7.3. Built-In Constants

The following built-in constants are provided to all shaders. The actual values used are

implementation-dependent, but must be at least the value shown.

The constant *gl_MaxVaryingFloats* is removed in the core profile, use *gl_MaxVaryingComponents* instead.

7.3.1. Compatibility Profile Built-In Constants

7.4. Built-In Uniform State

Built-in uniform state is not available when generating SPIR-V.

Otherwise, as an aid to accessing OpenGL processing state, the following uniform variables are built into the OpenGL Shading Language.

These variables are only guaranteed to be available in the fragment stage. In other stages, their presence and function is implementation-defined.

7.4.1. Compatibility Profile State

These variables are present only in the compatibility profile. They are not available to compute shaders, but are available to all other shaders.

7.5. Redeclaring Built-In Blocks

The *gl_PerVertex* block can be redeclared in a shader to explicitly indicate what subset of the fixed pipeline interface will be used. This is necessary to establish the interface between multiple programs. For example:

This establishes the output interface the shader will use with the subsequent pipeline stage. It must be a subset of the built-in members of *gl_PerVertex*. Such a redeclaration can also add the invariant

qualifier, interpolation qualifiers, and the layout qualifiers xfb_offset, xfb_buffer, and xfb_stride. It can also add an array size for unsized arrays. For example:

Other layout qualifiers, like location, cannot be added to such a redeclaration, unless specifically stated.

If a built-in interface block is redeclared, it must appear in the shader before any use of any member included in the built-in declaration, or a compile-time error will result. It is also a compile-time error to redeclare the block more than once or to redeclare a built-in block and then use a member from that built-in block that was not included in the redeclaration. Also, if a built-in interface block is redeclared, no member of the built-in declaration can be redeclared outside the block redeclaration. If multiple shaders using members of a built-in block belonging to the same interface are linked together in the same program, they must all redeclare the built-in block in the same way, as described in ÒInterface BlocksÓ for interface block matching, or a link-time error will result. It will also be a link-time error if some shaders in a program redeclare a specific built-in interface block while another shader in that program does not redeclare that interface block yet still uses a member of that interface block. If a built-in block interface is formed across shaders in different programs, the shaders must all redeclare the built-in block in the same way (as described for a single program), or the values passed along the interface are undefined.

Chapter 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 redeclaring 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 redeclared function.

When the built-in functions are specified below, where the input arguments (and corresponding output) can be float, vec2, vec3, or vec4, *genFType* 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. Where the input arguments (or corresponding output) can be bool, bvec2, bvec3, or bvec4, *genBType* is used as the argument. Where the input arguments (and corresponding output) can be double, dvec2, dvec3, dvec4, *genDType* is used as the argument. For any specific use of a function, the actual types substituted for *genFType*, *genIType*, *genUType*, or *genBType* have to have the same number of components for all arguments and for the return type. Similarly, *mat* is used for any matrix basic type with single-precision components and *dmat* is used for any matrix basic type with double-precision components.

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
genFType radians(genFType *degrees*)	Converts *degrees* to radians, i.e., (¹ / 180) á degrees.
genFType degrees(genFType *radians*)	Converts *radians* to degrees, i.e., (180 / ¹) á radians.
genFType sin(genFType *angle*)	The standard trigonometric sine function.
genFType cos(genFType *angle*)	The standard trigonometric cosine function.
genFType tan(genFType *angle*)	The standard trigonometric tangent.
genFType asin(genFType *x*)	Arc sine. Returns an angle whose sine is *x*. The range of values returned by this function is [-¹ / 2, ¹ / 2]. Results are undefined if |x| > 1.
genFType acos(genFType *x*)	Arc cosine. Returns an angle whose cosine is *x*. The range of values returned by this function is [0,¹]. Results are undefined if |x| > 1.
genFType atan(genFType *y*, genFType *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 [-¹, ¹. Results are undefined if *x* and *y* are both 0.
genFType atan(genFType *y_over_x*)	Arc tangent. Returns an angle whose tangent is *y_over_x*. The range of values returned by this function is [-¹ / 2, ¹ / 2].
genFType sinh(genFType *x*)	Returns the hyperbolic sine function $(e^{x} - e^{-x}) / 2$.
genFType cosh(genFType *x*)	Returns the hyperbolic cosine function $(e^{x} + e^{-x}) / 2$.
genFType tanh(genFType *x*)	Returns the hyperbolic tangent function sinh(x) / cosh(x).
genFType asinh(genFType *x*)	Arc hyperbolic sine; returns the inverse of sinh.
genFType acosh(genFType *x*)	Arc hyperbolic cosine; returns the non-negative inverse of cosh. Results are undefined if x < 1.
genFType atanh(genFType *x*)	Arc hyperbolic tangent; returns the inverse of tanh. Results are undefined if x ³ 1.

8.2. Exponential Functions

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

Syntax	Description
genFType pow(genFType *x*, genFType *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 ² 0.
genFType exp(genFType *x*)	Returns the natural exponentiation of *x*, i.e., e^{x}.

Syntax	Description
genFType log(genFType *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 ² 0.
genFType exp2(genFType *x*)	Returns 2 raised to the *x* power, i.e., 2^x.
genFType log2(genFType *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 ² 0.
genFType sqrt(genFType *x*) genDType sqrt(genDType *x*)	Returns sqrt(x). Results are undefined if x < 0.
genFType inversesqrt(genFType *x*) genDType inversesqrt(genDType *x*)	Returns 1 / sqrt(x). Results are undefined if x ² 0.

8.3. Common Functions

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

Syntax	Description
genFType abs(genFType *x*) genIType abs(genIType *x*) genDType abs(genDType *x*)	Returns *x* if x ³ 0; otherwise it returns -*x*.
genFType sign(genFType *x*) genIType sign(genIType *x*) genDType sign(genDType *x*)	Returns 1.0 if *x* > 0, 0.0 if *x* = 0, or -1.0 if *x* < 0.
genFType floor(genFType *x*) genDType floor(genDType *x*)	Returns a value equal to the nearest integer that is less than or equal to *x*.
genFType trunc(genFType *x*) genDType trunc(genDType *x*)	Returns a value equal to the nearest integer to *x* whose absolute value is not larger than the absolute value of *x*.
genFType round(genFType *x*) genDType round(genDType *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*.
genFType roundEven(genFType *x*) genDType roundEven(genDType *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.)
genFType ceil(genFType *x*) genDType ceil(genDType *x*)	Returns a value equal to the nearest integer that is greater than or equal to *x*.
genFType fract(genFType *x*) genDType fract(genDType *x*)	Returns *x* - floor(*x*).

Syntax	Description
genFType mod(genFType *x*, float *y*) genFType mod(genFType *x*, genFType *y*) genDType mod(genDType *x*, double *y*) genDType mod(genDType *x*, genDType *y*)	Modulus. Returns x - y á floor(x / y).
genFType modf(genFType *x*, out genFType *i*) genDType modf(genDType *x*, out genDType *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*.
genFType min(genFType *x*, genFType *y*) genFType min(genFType *x*, float *y*) genDType min(genDType *x*, genDType *y*) genDType min(genDType *x*, double *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*.
genFType max(genFType *x*, genFType *y*) genFType max(genFType *x*, float *y*) genDType max(genDType *x*, genDType *y*) genDType max(genDType *x*, double *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*.
genFType clamp(genFType *x*, genFType *minVal*, genFType *maxVal*) genFType clamp(genFType *x*, float *minVal*, float *maxVal*) genDType clamp(genDType *x*, genDType *minVal*, genDType *maxVal*) genDType clamp(genDType *x*, double *minVal*, double *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*.
genFType mix(genFType *x*, genFType *y*, genFType *a*) genFType mix(genFType *x*, genFType *y*, float *a*) genDType mix(genDType *x*, genDType *y*, genDType *a*) genDType mix(genDType *x*, genDType *y*, double *a*)	Returns the linear blend of *x* and *y*, i.e., x á (1 - a) + y á a.

Syntax	Description
genFType mix(genFType *x*, genFType *y*, genBType *a*) genDType mix(genDType *x*, genDType *y*, genBType *a*) genIType mix(genIType *x*, genIType *y*, genBType *a*) genUType mix(genUType *x*, genUType *y*, genBType *a*) genBType mix(genBType *x*, genBType *y*, genBType *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, for example, genFType mix(genFType *x*, genFType *y*, genFType(*a*)) where *a* is a Boolean vector.
genFType step(genFType *edge*, genFType *x*) genFType step(float *edge*, genFType *x*) genDType step(genDType *edge*, genDType *x*) genDType step(double *edge*, genDType *x*)	Returns 0.0 if *x* < *edge;* otherwise it returns 1.0.
genFType smoothstep(genFType *edge0*, genFType *edge1*, genFType *x*) genFType smoothstep(float *edge0*, float *edge1*, genFType *x*) genDType smoothstep(genDType *edge0*, genDType *edge1*, genDType *x*) genDType smoothstep(double *edge0*, double *edge1*, genDType *x*)	Returns 0.0 if x ² edge0 and 1.0 if x ³ 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: (And similarly for doubles.) Results are undefined if edge0 ³ edge1.
genBType isnan(genFType *x*) genBType isnan(genDType *x*)	Returns true if *x* holds a NaN. Returns false otherwise. Always returns false if NaNs are not implemented.
genBType isinf(genFType *x*) genBType isinf(genDType *x*)	Returns true if *x* holds a positive infinity or negative infinity. Returns false otherwise.
genIType floatBitsToInt(genFType *value*) genUType floatBitsToUint(genFType *value*)	Returns a signed or unsigned integer value representing the encoding of a floating-point value. The float value's bit-level representation is preserved.

Syntax	Description
genFType intBitsToFloat(genIType *value*) genFType uintBitsToFloat(genUType *value*)	Returns a floating-point value corresponding to a signed or unsigned integer encoding of a floating-point value. If a NaN is passed in, it will not signal, and the resulting value is unspecified. If an Inf is passed in, the resulting value is the corresponding Inf. Otherwise, the bit-level representation is preserved.
genFType fma(genFType *a*, genFType *b*, genFType *c*) genDType fma(genDType *a*, genDType *b*, genDType *c*)	Computes and returns a * b + c. In uses where the return value is eventually consumed by a variable declared as precise: ¥ fma() is considered a single operation, whereas the expression Òa * b + cÓ consumed by a variable declared precise is considered two operations. ¥ The precision of fma() can differ from the precision of the expression Òa * b + cÓ. ¥ fma() will be computed with the same precision as any other fma() consumed by a precise variable, giving invariant results for the same input values of *a*, *b*, and *c*. Otherwise, in the absence of precise consumption, there are no special constraints on the number of operations or difference in precision between fma() and the expression Ò*a* * *b* + *c*Ó.

Syntax	Description
genFType frexp(genFType *x*, out genIType *exp*) genDType frexp(genDType *x*, out genIType *exp*)	Splits *x* into a floating-point significand in the range [0.5,1.0], and an integral exponent of two, such that x = significant á $2^{exponent}$ The significand is returned by the function and the exponent is returned in the parameter *exp*. For a floating-point value of zero, the significand and exponent are both zero. If an implementation supports signed zero, an input value of minus zero should return a significand of minus zero. For a floating-point value that is an infinity or is not a number, the results are undefined. If the input *x* is a vector, this operation is performed in a component-wise manner; the value returned by the function and the value written to *exp* are vectors with the same number of components as *x*.
genFType ldexp(genFType *x*, genIType *exp*) genDType ldexp(genDType *x*, genIType *exp*)	Builds a floating-point number from *x* and the corresponding integral exponent of two in *exp*, returning: significand á $2^{exponent}$ If this product is too large to be represented in the floating-point type, the result is undefined. If *exp* is greater than +128 (single-precision) or +1024 (double-precision), the value returned is undefined. If *exp* is less than -126 (single-precision) or -1022 (double-precision), the value returned may be flushed to zero. Additionally, splitting the value into a significand and exponent using frexp() and then reconstructing a floating-point value using ldexp() should yield the original input for zero and all finite non-denormalized values. If the input *x* is a vector, this operation is performed in a component-wise manner; the value passed in *exp* and returned by the function are vectors with the same number of components as *x*.

8.4. Floating-Point Pack and Unpack Functions

These functions do not operate component-wise, rather, as described in each case.

Syntax	Description
uint packUnorm2x16(vec2 *v*) uint packSnorm2x16(vec2 *v*) uint packUnorm4x8(vec4 *v*) uint packSnorm4x8(vec4 *v*)	First, converts each component of the normalized floating-point value *v* into 16-bit (2x16) or 8-bit (4x8) integer values. Then, the results are packed into the returned 32-bit unsigned integer. The conversion for component *c* of *v* to fixed point is done as follows: packUnorm2x16: round(clamp(*c*, 0, +1) * 65535.0) packSnorm2x16: round(clamp(*c*, -1, +1) * 32767.0) packUnorm4x8: round(clamp(*c*, 0, +1) * 255.0) packSnorm4x8: round(clamp(*c*, -1, +1) * 127.0) The first component of the vector will be written to the least significant bits of the output; the last component will be written to the most significant bits.
vec2 unpackUnorm2x16(uint *p*) vec2 unpackSnorm2x16(uint *p*) vec4 unpackUnorm4x8(uint *p*) vec4 unpackSnorm4x8(uint *p*)	First, unpacks a single 32-bit unsigned integer *p* into a pair of 16-bit unsigned integers, a pair of 16-bit signed integers, four 8-bit unsigned integers, or four 8-bit signed integers, respectively. Then, each component is converted to a normalized floating-point value to generate the returned two- or four-component vector. The conversion for unpacked fixed-point value *f* to floating-point is done as follows: unpackUnorm2x16: *f* / 65535.0 unpackSnorm2x16: clamp(*f* / 32767.0, -1, +1) unpackUnorm4x8: *f* / 255.0 unpackSnorm4x8: clamp(*f* / 127.0, -1, +1) The first component of the returned vector will be extracted from the least significant bits of the input; the last component will be extracted from the most significant bits.

Syntax	Description
uint packHalf2x16(vec2 *v*)	Returns an unsigned integer obtained by converting the components of a two-component floating-point vector to the 16-bit floating-point representation found in the OpenGL Specification, and then packing these two 16-bit integers into a 32-bit unsigned integer. The first vector component specifies the 16 least-significant bits of the result; the second component specifies the 16 most-significant bits.
vec2 unpackHalf2x16(uint *v*)	Returns a two-component floating-point vector with components obtained by unpacking a 32-bit unsigned integer into a pair of 16-bit values, interpreting those values as 16-bit floating-point numbers according to the OpenGL Specification, and converting them to 32-bit floating-point values. The first component of the vector is obtained from the 16 least-significant bits of *v*; the second component is obtained from the 16 most-significant bits of *v*.
double packDouble2x32(uvec2 *v*)	Returns a double-precision value obtained by packing the components of *v* into a 64-bit value. If an IEEE 754 Inf or NaN is created, it will not signal, and the resulting floating-point value is unspecified. Otherwise, the bit-level representation of *v* is preserved. The first vector component specifies the 32 least significant bits; the second component specifies the 32 most significant bits.
uvec2 unpackDouble2x32(double *v*)	Returns a two-component unsigned integer vector representation of *v*. The bit-level representation of *v* is preserved. The first component of the vector contains the 32 least significant bits of the double; the second component consists of the 32 most significant bits.

8.5. Geometric Functions

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

Syntax	Description
float length(genFType *x*) double length(genDType *x*)	Returns the length of vector *x*, i.e., sqrt($x_0^2 + x_1^2$ + É).

Syntax	Description
float distance(genFType *p0*, genFType *p1*) double distance(genDType *p0*, genDType *p1*)	Returns the distance between *p0* and *p1*, i.e., length(*p0* - *p1*)
float dot(genFType *x*, genFType *y*) double dot(genDType *x*, genDType *y*)	Returns the dot product of *x* and *y*, i.e., $x_0 á y_0 + x_1 á y_1 +$ É
vec3 cross(vec3 *x*, vec3 *y*) dvec3 cross(dvec3 *x*, dvec3 *y*)	Returns the cross product of *x* and *y*, i.e., ($x_1 á y_2 - y_1 á x_2$, $x_2 á y_0 - y_2 á x_0$, $x_0 á y_1 - y_0 á x_1$).
genFType normalize(genFType *x*) genDType normalize(genDType *x*)	Returns a vector in the same direction as *x* but with a length of 1, i.e. *x* / length(x).
compatibility profile only vec4 ftransform()	Available only when using the compatibility profile. For core OpenGL, 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.
genFType faceforward(genFType *N*, genFType *I*, genFType *Nref*) genDType faceforward(genDType *N*, genDType *I*, genDType *Nref*)	If dot(*Nref*, *I*) < 0 return *N*, otherwise return -*N*.
genFType reflect(genFType *I*, genFType *N*) genDType reflect(genDType *I*, genDType *N*)	For the incident vector *I* and surface orientation *N*, returns the reflection direction: I - 2 á dot(N, I) á N. *N* must already be normalized in order to achieve the desired result.
genFType refract(genFType *I*, genFType *N*, float *eta*) genDType refract(genDType *I*, genDType *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 the refraction equation shown below.

8.5.1. Refraction Equation

editing-note

(Jon) Moved to new section from refract table entry above because asciidoctor-mathematical doesn't support math blocks in table cells yet.

$$k = 1.0 - eta * eta * (1.0 - \mathbf{dot}(N, I) \cdot \mathbf{dot}(N, I))$$

$$result = \begin{cases} genFType(0.0), & k < 0.0 \\ eta * I - (eta * \mathbf{dot}(N, I) + \sqrt{k}) * N, & \mathbf{otherwise} \end{cases}$$

8.6. Matrix Functions

For each of the following built-in matrix functions, there is both a single-precision floating-point version, where all arguments and return values are single precision, and a double-precision floating-point version, where all arguments and return values are double precision. Only the single-precision floating-point version is shown.

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.
float determinant(mat2 *m*) float determinant(mat3 *m*) float determinant(mat4 *m*)	Returns the determinant of *m*.

Syntax	Description
mat2 inverse(mat2 *m*) mat3 inverse(mat3 *m*) mat4 inverse(mat4 *m*)	Returns a matrix that is the inverse of *m*. The input matrix *m* is not modified. The values in the returned matrix are undefined if *m* is singular or poorly-conditioned (nearly singular).

8.7. Vector Relational Functions

Relational and equality operators (<, <=, >, >=, ==, !=) are defined to operate on scalars and produce scalar Boolean results. For vector results, use the following built-in functions. Below, the following placeholders are used for the listed specific types:

Placeholder	Specific Types Allowed
bvec	bvec2, bvec3, bvec4
ivec	ivec2, ivec3, ivec4
uvec	uvec2, uvec3, uvec4
vec	vec2, vec3, vec4, dvec2, dvec3, dvec4

In all cases, the sizes of all 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 ² 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 ³ 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)	Returns the component-wise compare of x == 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.
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.

Syntax	Description
bvec not(bvec x)	Returns the component-wise logical complement of *x*.

8.8. Integer Functions

These all operate component-wise. The description is per component. The notation [*a*, *b*] means the set of bits from bit-number *a* through bit-number *b*, inclusive. The lowest-order bit is bit 0. ÒBit numberÓ will always refer to counting up from the lowest-order bit as bit 0.

Syntax	Description
genUType uaddCarry(genUType *x*, genUType *y*, out genUType *carry*)	Adds 32-bit unsigned integers *x* and *y*, returning the sum modulo 2^{32}. The value *carry* is set to zero if the sum was less than 2^{32}, or one otherwise.
genUType usubBorrow(genUType *x*, genUType *y*, out genUType *borrow*)	Subtracts the 32-bit unsigned integer *y* from *x*, returning the difference if non-negative, or 2^{32} plus the difference otherwise. The value *borrow* is set to zero if x ³ y, or one otherwise.
void umulExtended(genUType *x*, genUType *y*, out genUType *msb*, out genUType *lsb*) void imulExtended(genIType *x*, genIType *y*, out genIType *msb*, out genIType *lsb*)	Multiplies 32-bit unsigned or signed integers *x* and *y*, producing a 64-bit result. The 32 least-significant bits are returned in *lsb*. The 32 most-significant bits are returned in *msb*.
genIType bitfieldExtract(genIType *value*, int *offset*, int *bits*) genUType bitfieldExtract(genUType *value*, int *offset*, int *bits*)	Extracts bits [offset, offset + bits - 1] from *value*, returning them in the least significant bits of the result. For unsigned data types, the most significant bits of the result will be set to zero. For signed data types, the most significant bits will be set to the value of bit offset + bits - 1. If *bits* is zero, the result will be zero. The result will be undefined if *offset* or *bits* is negative, or if the sum of *offset* and *bits* is greater than the number of bits used to store the operand. Note that for vector versions of bitfieldExtract(), a single pair of *offset* and *bits* values is shared for all components.

Syntax	Description
genIType bitfieldInsert(genIType *base*, genIType *insert*, int *offset*, int *bits*) genUType bitfieldInsert(genUType *base*, genUType *insert*, int *offset*, int *bits*)	Inserts the *bits* least significant bits of *insert* into *base*. The result will have bits [offset, offset + bits - 1] taken from bits [0, bits - 1] of *insert*, and all other bits taken directly from the corresponding bits of *base*. If *bits* is zero, the result will simply be *base*. The result will be undefined if *offset* or *bits* is negative, or if the sum of *offset* and *bits* is greater than the number of bits used to store the operand. Note that for vector versions of bitfieldInsert(), a single pair of *offset* and *bits* values is shared for all components.
genIType bitfieldReverse(genIType *value*) genUType bitfieldReverse(genUType *value*)	Reverses the bits of *value*. The bit numbered *n* of the result will be taken from bit (bits - 1) - n of *value*, where *bits* is the total number of bits used to represent *value*.
genIType bitCount(genIType *value*) genIType bitCount(genUType *value*)	Returns the number of one bits in the binary representation of *value*.
genIType findLSB(genIType *value*) genIType findLSB(genUType *value*)	Returns the bit number of the least significant one bit in the binary representation of *value*. If *value* is zero, -1 will be returned.
genIType findMSB(genIType *value*) genIType findMSB(genUType *value*)	Returns the bit number of the most significant bit in the binary representation of *value*. For positive integers, the result will be the bit number of the most significant one bit. For negative integers, the result will be the bit number of the most significant zero bit. For a *value* of zero or negative one, -1 will be returned.

8.9. Texture Functions

Texture lookup functions are available in all shading stages. However, level-of-detail is implicitly computed only for fragment shaders. Other shaders operate as though the base level-of-detail were computed as zero. 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 mipmap 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 single-precision 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 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.

For depth/stencil textures, the internal texture format is determined by the component being accessed as set through the OpenGL API. When the depth/stencil texture mode is set to DEPTH_COMPONENT, the internal format of the depth component should be used. When the depth/stencil texture mode is set to STENCIL_INDEX, the internal format of the stencil component should be used.

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.

In the prototypes below, the ò*g*ó in the return type ò*gvec4*ó is used as a placeholder for nothing, ò*i*ó, or ò*u*ó making a return type of vec4, ivec4, or uvec4. In these cases, the sampler argument type also starts with ò*g*ó, indicating the same substitution done on the return type; it is either a single-precision 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 8.23 ÒTexture Comparison ModesÓ of the OpenGL Specification. 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 any other shader stage. For a fragment shader, if *bias* is present, it is added to the implicit level-of-detail prior to performing the texture access operation. No *bias* or *lod* parameters for rectangle textures, multisample textures, or texture buffers are supported because mipmaps are not allowed for these types of textures.

The implicit level-of-detail is selected as follows: For a texture that is not mipmapped, the texture is used directly. If it is mipmapped and running in a fragment shader, the level-of-detail computed by the implementation is used to do the texture lookup. If it is mipmapped and running in a non-fragment shader, then the base texture is used.

Some texture functions (non-ÒLodÓ and non-ÒGradÓ versions) may require implicit derivatives. Implicit derivatives are undefined within non-uniform control flow and for non-fragment 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 8.13 ÒCube Map Texture SelectionÓ of the OpenGL Specification.

For Array forms, the array layer used will be

$$\max(0, \min(d - 1, \lfloor layer + 0.5 \rfloor))$$

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

8.9.1. Texture Query Functions

The textureSize functions query the dimensions of a specific texture level for a sampler.

The textureQueryLod functions are available only in a fragment shader. They take the components of *P* and compute the level-of-detail information that the texture pipe would use to access that texture through a normal texture lookup. The level-of-detail λ' (equation 3.18 of the OpenGL Specification) is obtained after any level-of-detail bias, but prior to clamping to [TEXTURE_MIN_LOD, TEXTURE_MAX_LOD]. The mipmap array(s) that would be accessed are also computed. If a single level-of-detail would be accessed, the level-of-detail number relative to the base level is returned. If multiple levels-of-detail would be accessed, a floating-point number between the two levels is returned, with the fractional part equal to the fractional part of the computed and clamped level-of-detail.

The algorithm used is given by the following pseudo-code:

The value *maxAccessibleLevel* is the level number of the smallest accessible level of the mipmap array (the value *q* in section 8.14.3 ÒMipmappingÓ of the OpenGL Specification) minus the base level.

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*) ivec3 textureSize(gsamplerCubeArray *sampler*, int *lod*) ivec3 textureSize(samplerCubeArrayShadow *sampler*, int *lod*) ivec2 textureSize(gsampler2DRect *sampler*) ivec2 textureSize(sampler2DRectShadow *sampler*) ivec2 textureSize(gsampler1DArray *sampler*, int *lod*) ivec2 textureSize(sampler1DArrayShadow *sampler*, int *lod*) ivec3 textureSize(gsampler2DArray *sampler*, int *lod*) ivec3 textureSize(sampler2DArrayShadow *sampler*, int *lod*) int textureSize(gsamplerBuffer *sampler*) ivec2 textureSize(gsampler2DMS *sampler*) ivec3 textureSize(gsampler2DMSArray *sampler*)	Returns the dimensions of level *lod* (if present) for the texture bound to *sampler*, as described in section 8.11 ÒTexture QueriesÓ of the OpenGL Specification. The components in the return value are filled in, in order, with the width, height, and depth of the texture. For the array forms, the last component of the return value is the number of layers in the texture array, or the number of cubes in the texture cube map array.

Syntax	Description
vec2 textureQueryLod(gsampler1D *sampler*, float *P*) vec2 textureQueryLod(gsampler2D *sampler*, vec2 *P*) vec2 textureQueryLod(gsampler3D *sampler*, vec3 *P*) vec2 textureQueryLod(gsamplerCube *sampler*, vec3 *P*) vec2 textureQueryLod(gsampler1DArray *sampler*, float *P*) vec2 textureQueryLod(gsampler2DArray *sampler*, vec2 *P*) vec2 textureQueryLod(gsamplerCubeArray *sampler*, vec3 *P*) vec2 textureQueryLod(sampler1DShadow *sampler*, float *P*) vec2 textureQueryLod(sampler2DShadow *sampler*, vec2 *P*) vec2 textureQueryLod(samplerCubeShadow *sampler*, vec3 *P*) vec2 textureQueryLod(sampler1DArrayShadow *sampler*, float *P*) vec2 textureQueryLod(sampler2DArrayShadow *sampler*, vec2 *P*) vec2 textureQueryLod(samplerCubeArrayShadow *sampler*, vec3 *P*)	Returns the mipmap array(s) that would be accessed in the *x* component of the return value. Returns the computed level-of-detail relative to the base level in the *y* component of the return value. If called on an incomplete texture, the results are undefined.

Syntax	Description
int textureQueryLevels(gsampler1D *sampler*) int textureQueryLevels(gsampler2D *sampler*) int textureQueryLevels(gsampler3D *sampler*) int textureQueryLevels(gsamplerCube *sampler*) int textureQueryLevels(gsampler1DArray *sampler*) int textureQueryLevels(gsampler2DArray *sampler*) int textureQueryLevels(gsamplerCubeArray *sampler*) int textureQueryLevels(sampler1DShadow *sampler*) int textureQueryLevels(sampler2DShadow *sampler*) int textureQueryLevels(samplerCubeShadow *sampler*) int textureQueryLevels(sampler1DArrayShadow *sampler*) int textureQueryLevels(sampler2DArrayShadow *sampler*) int textureQueryLevels(samplerCubeArrayShadow *sampler*)	Returns the number of mipmap levels accessible in the texture associated with *sampler*, as defined in the OpenGL Specification. The value zero will be returned if no texture or an incomplete texture is associated with *sampler*. Available in all shader stages.
int textureSamples(gsampler2DMS *sampler*) int textureSamples(gsampler2DMSArray *sampler*)	Returns the number of samples of the texture or textures bound to *sampler*.

8.9.2. Texel Lookup Functions

Syntax	Description
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(gsampler2DArray *sampler*, vec3 *P* [, float *bias*]) gvec4 texture(gsamplerCubeArray *sampler*, vec4 *P* [, float *bias*]) gvec4 texture(gsampler1DArray *sampler*, *vec2 _P* [, float *bias*]) float texture(sampler1DArrayShadow *sampler*, *P* [, float *bias*]) float texture(sampler2DArrayShadow *sampler*, vec4 *P*) gvec4 texture(gsampler2DRect *sampler*, vec2 *P*) float texture(sampler2DRectShadow *sampler*, vec3 *P*) float texture(samplerCubeArrayShadow *sampler*, vec4 *P*, float *compare*)	Use the texture coordinate *P* to do a texture lookup in the texture currently bound to *sampler*. For shadow forms: When *compare* is present, it is used as D_{ref} and the array layer comes from the last component of *P*. When *compare* is not present, the last component of *P* is used as D_{ref} and the array layer comes from the second to last component of *P*. (The second component of *P* is unused for 1D shadow lookups.) For non-shadow forms: the array layer comes from the last component of *P*.
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*]) gvec4 textureProj(gsampler2DRect *sampler*, vec3 *P*) gvec4 textureProj(gsampler2DRect *sampler*, vec4 *P*) float textureProj(sampler2DRectShadow *sampler*, vec4 *P*)	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* to form projected coordinates *P′*. The resulting third component of *P* in the shadow forms is used as D_{ref}. The third component of *P* is ignored when *sampler* has type gsampler2D and *P* has type vec4. 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(sampler2DShadow *sampler*, vec3 *P*, float *lod*) float textureLod(sampler1DShadow *sampler*, vec3 *P*, float *lod*) gvec4 textureLod(gsampler1DArray *sampler*, vec2 *P*, float *lod*) float textureLod(sampler1DArrayShadow *sampler*, *P*, float *lod*) gvec4 textureLod(gsampler2DArray *sampler*, vec3 *P*, float *lod*) gvec4 textureLod(gsamplerCubeArray *sampler*, vec4 *P*, float *lod*)	Do a texture lookup as in texture but with explicit level-of-detail; *lod* specifies $\%_{base}$] and sets the partial derivatives as follows: (See section 8.14 ÒTexture MinificationÓ and equations 8.4-8.6 of the OpenGL Specification.) ¶u / ¶x = ¶v / ¶x = ¶w / ¶x = 0 ¶u / ¶y = ¶v / ¶y = ¶w / ¶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(sampler2DShadow *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]) gvec4 textureOffset(gsampler2DRect *sampler*, vec2 *P*, ivec2 *offset*) float textureOffset(sampler2DRectShadow *sampler*, vec3 *P*, ivec2 *offset*) float textureOffset(sampler1DShadow *sampler*, vec3 *P*, int *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*]) float textureOffset(sampler2DArrayShadow *sampler*, vec4 P, ivec2 *offset*)	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 *gl_MinProgramTexelOffset* and *gl_MaxProgramTexelOffset*, respectively. Note that *offset* does not apply to the layer coordinate for texture arrays. This is explained in detail in section 8.14.2 ÒCoordinate Wrapping and Texel SelectionÓ of the OpenGL Specification, where *offset* is ($\&_u$, $\&_v$, $\&_w$). 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(gsampler2DRect *sampler*, ivec2 *P*) gvec4 texelFetch(gsampler1DArray *sampler*, ivec2 *P*, int *lod*) gvec4 texelFetch(gsampler2DArray *sampler*, ivec3 *P*, int *lod*) gvec4 texelFetch(gsamplerBuffer *sampler*, int *P*) gvec4 texelFetch(gsampler2DMS *sampler*, ivec2 *P*, int *sample*) gvec4 texelFetch(gsampler2DMSArray *sampler*, ivec3 *P*, int *sample*)	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* (if present) is as described in sections 11.1.3.2 ÒTexel Fetches Ó and 8.14.1 ÒScale Factor and Level of Detail Ó of the OpenGL Specification.
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(gsampler2DRect *sampler*, ivec2 *P*, ivec2 *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*]) gvec4 textureProjOffset(gsampler2DRect *sampler*, vec3 *P*, ivec2 *offset*) gvec4 textureProjOffset(gsampler2DRect *sampler*, vec4 *P*, ivec2 *offset*) float textureProjOffset(sampler2DRectShadow *sampler*, vec4 *P*, ivec2 *offset*) 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*, *P*, float *lod*, int *offset*) float textureLodOffset(sampler2DShadow *sampler*, *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*, *P*, float *lod*, int *offset*)	Do an offset texture lookup with explicit level-of-detail. 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 level-of-detail. 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 level-of-detail. 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*, *P*, vec3 *dPdx*, vec3 *dPdy*) gvec4 textureGrad(gsamplerCube *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy*) gvec4 textureGrad(gsampler2DRect *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy*) float textureGrad(sampler2DRectShadow *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) float textureGrad(sampler1DShadow *sampler*, vec3 *P*, float *dPdx*, float *dPdy*) gvec4 textureGrad(gsampler1DArray *sampler*, vec2 *P*, float *dPdx*, float *dPdy*) gvec4 textureGrad(gsampler2DArray *sampler*, *P*, vec2 *dPdx*, vec2 *dPdy*) float textureGrad(sampler1DArrayShadow *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*) float textureGrad(sampler2DArrayShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 textureGrad(gsamplerCubeArray *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*)	Do a texture lookup as in texture but with explicit gradients as shown below. The partial derivatives of *P* are with respect to window *x* and window *y*. 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*) gvec4 textureGradOffset(gsampler2DRect *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) float textureGradOffset(sampler2DRectShadow *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *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*) gvec4 textureGradOffset(gsampler2DArray *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) gvec4 textureGradOffset(gsampler1DArray *sampler*, vec2 *P*, float *dPdx*, float *dPdy*, int *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*) gvec4 textureProjGrad(gsampler2DRect *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 textureProjGrad(gsampler2DRect *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*) float textureProjGrad(sampler2DRectShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *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*, ivec2 *offset*) gvec4 textureProjGradOffset(gsampler2D *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) gvec4 textureProjGradOffset(gsampler3D *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*, ivec3 *offset*) gvec4 textureProjGradOffset(gsampler2DRect *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) gvec4 textureProjGradOffset(gsampler2DRect *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) float textureProjGradOffset(sampler2DRectShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) float textureProjGradOffset(sampler1DShadow *sampler*, vec4 *P*, float *dPdx*, float *dPdy*, int *offset*) float textureProjGradOffset(sampler2DShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*)	Do a texture lookup projectively and with explicit gradient as described in textureProjGrad, as well as with offset, as described in textureOffset.

8.9.3. Explicit Gradients

editing-note

(Jon) Moved to new section from textureGrad table entry above because asciidoctor-mathematical doesn't support math blocks in table cells yet.

In the textureGrad functions described above, explicit gradients control texture lookups as follows:

$$\frac{\partial s}{\partial x} = \begin{cases} \frac{\partial P}{\partial x}, & \text{for a 1D texture} \\ \frac{\partial P.s}{\partial x}, & \text{otherwise} \end{cases}$$

$$\frac{\partial s}{\partial y} = \begin{cases} \frac{\partial P}{\partial y}, & \text{for a 1D texture} \\ \frac{\partial P.s}{\partial y}, & \text{otherwise} \end{cases}$$

$$\frac{\partial t}{\partial x} = \begin{cases} 0.0, & \text{for a 1D texture} \\ \frac{\partial P.t}{\partial x}, & \text{otherwise} \end{cases}$$

$$\frac{\partial t}{\partial y} = \begin{cases} 0.0, & \text{for a 1D texture} \\ \frac{\partial P.t}{\partial y}, & \text{otherwise} \end{cases}$$

$$\frac{\partial r}{\partial x} = \begin{cases} 0.0, & \text{for 1D or 2D} \\ \frac{\partial P.p}{\partial x}, & \text{cube, other} \end{cases}$$

$$\frac{\partial r}{\partial y} = \begin{cases} 0.0, & \text{for 1D or 2D} \\ \frac{\partial P.p}{\partial y}, & \text{cube, other} \end{cases}$$

8.9.4. Texture Gather Functions

The texture gather functions take components of a single floating-point vector operand as a texture coordinate, determine a set of four texels to sample from the base level-of-detail of the specified texture image, and return one component from each texel in a four-component result vector.

When performing a texture gather operation, the minification and magnification filters are ignored, and the rules for LINEAR filtering in the OpenGL Specification are applied to the base level of the texture image to identify the four texels $i_0\, j_1$, $i_1\, j_1$, $i_1\, j_0$, and $i_0\, j_0$. The texels are then converted to texture base colors (R_s, G_s, B_s, A_s) according to table 15.1, followed by application of the texture swizzle as described in section 15.2.1 ÒTexture AccessÓ of the OpenGL Specification. A four-component vector is assembled by taking the selected component from each of the post-swizzled texture source colors in the order ($i_0\, j_1$, $i_1\, j_1$, $i_1\, j_0$, $i_0\, j_0$).

For texture gather functions using a shadow sampler type, each of the four texel lookups perform a depth comparison against the depth reference value passed in (*refZ*), and returns the result of that comparison in the appropriate component of the result vector.

As with other texture lookup functions, the results of a texture gather are undefined for shadow samplers if the texture referenced is not a depth texture or has depth comparisons disabled; or for non-shadow samplers if the texture referenced is a depth texture with depth comparisons enabled.

Syntax	Description
gvec4 textureGather(gsampler2D *sampler*, vec2 *P* [, int *comp*]) gvec4 textureGather(gsampler2DArray *sampler*, vec3 *P* [, int *comp*]) gvec4 textureGather(gsamplerCube *sampler*, vec3 *P* [, int *comp*]) gvec4 textureGather(gsamplerCubeArray *sampler*, vec4 *P*[, int *comp*]) gvec4 textureGather(gsampler2DRect *sampler*, vec2 *P*[, int *comp*]) vec4 textureGather(sampler2DShadow *sampler*, vec2 *P*, float *refZ*) vec4 textureGather(sampler2DArrayShadow *sampler*, vec3 *P*, float *refZ*) vec4 textureGather(samplerCubeShadow *sampler*, vec3 *P*, float *refZ*) vec4 textureGather(samplerCubeArrayShadow *sampler*, vec4 *P*, float *refZ*) vec4 textureGather(sampler2DRectShadow *sampler*, vec2 *P*, float *refZ*)	Returns the value If specified, the value of *comp* must be a constant integer expression with a value of 0, 1, 2, or 3, identifying the *x*, *y*, *z*, or *w* post-swizzled component of the four-component vector lookup result for each texel, respectively. If *comp* is not specified, it is treated as 0, selecting the *x* component of each texel to generate the result.
gvec4 textureGatherOffset(gsampler2D *sampler*, vec2 *P*, ivec2 *offset*, [int *comp*]) gvec4 textureGatherOffset(gsampler2DArray *sampler*, vec3 *P*, ivec2 *offset* [int *comp*]) vec4 textureGatherOffset(sampler2DShadow *sampler*, vec2 *P*, float *refZ*, ivec2 *offset*) vec4 textureGatherOffset(sampler2DArrayShadow *sampler*, vec3 *P*, float *refZ*, ivec2 *offset*) gvec4 textureGatherOffset(gsampler2DRect *sampler*, vec2 *P*, ivec2 *offset* [int *comp*]) vec4 textureGatherOffset(sampler2DRectShadow *sampler*, vec2 *P*, float *refZ*, ivec2 *offset*)	Perform a texture gather operation as in textureGather by *offset* as described in textureOffset except that the *offset* can be variable (non constant) and the implementation-dependent minimum and maximum offset values are given by MIN_PROGRAM_TEXTURE_GATHER_OFFSET and MAX_PROGRAM_TEXTURE_GATHER_OFFSET, respectively.
gvec4 textureGatherOffsets(gsampler2D *sampler*, vec2 *P*, ivec2 *offsets*[4] [, int *comp*]) gvec4 textureGatherOffsets(gsampler2DArray *sampler*, vec3 *P*, ivec2 *offsets*[4] [, int *comp*]) vec4 textureGatherOffsets(sampler2DShadow *sampler*, vec2 *P*, float *refZ*, ivec2 *offsets*[4]) vec4 textureGatherOffsets(sampler2DArrayShadow *sampler*, vec3 *P*, float *refZ*, ivec2 *offsets*[4]) gvec4 textureGatherOffsets(gsampler2DRect *sampler*, vec2 *P*, ivec2 *offsets*[4] [, int *comp*]) vec4 textureGatherOffsets(sampler2DRectShadow *sampler*, vec2 *P*, float *refZ*, ivec2 *offsets*[4])	Operate identically to textureGatherOffset except that *offsets* is used to determine the location of the four texels to sample. Each of the four texels is obtained by applying the corresponding offset in *offsets* as a (*u*, *v*) coordinate offset to *P*, identifying the four-texel LINEAR footprint, and then selecting the texel i_0 j_0 of that footprint. The specified values in *offsets* must be constant integral expressions.

8.9.5. Compatibility Profile Texture Functions

The following texture functions are only in the compatibility profile.

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*)	See corresponding signature above without ò1Dó 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*)	See corresponding signature above without ò2Dó 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*)	See corresponding signature above without ò3Dó in the name. Use the texture coordinate *coord* to do a texture lookup in the 3D texture currently bound to *sampler*. For the projective (òProjó) 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*)	See corresponding signature above without òCubeó 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*)	Same functionality as the ÒtextureÓ based names above with the same signature.

8.10. Atomic Counter Functions

The atomic-counter operations in this section operate atomically with respect to each other. They are atomic for any single counter, meaning any of these operations on a specific counter in one shader instantiation will be indivisible by any of these operations on the same counter from another shader instantiation. There is no guarantee that these operations are atomic with respect to other forms of access to the counter or that they are serialized when applied to separate counters. Such cases would require additional use of fences, barriers, or other forms of synchronization, if atomicity or serialization is desired.

The underlying counter is a 32-bit unsigned integer. The result of operations will wrap to [0, 2^{32}-1].

Syntax	Description
uint atomicCounterIncrement(atomic_uint *c*)	Atomically 1. increments the counter for *c*, and 2. returns its value prior to the increment operation. These two steps are done atomically with respect to the atomic counter functions in this table.

Syntax	Description
uint atomicCounterDecrement(atomic_uint *c*)	Atomically 1. decrements the counter for *c*, and 2. returns the value resulting from the decrement operation. These two steps are done atomically with respect to the atomic counter functions in this table.
uint atomicCounter(atomic_uint *c*)	Returns the counter value for *c*.
uint atomicCounterAdd(atomic_uint *c*, uint *data*)	Atomically 1. adds the value of *data* to the counter for *c*, and 2. returns its value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.
uint atomicCounterSubtract(atomic_uint *c*, uint *data*)	Atomically 1. subtracts the value of *data* from the counter for *c*, and 2. returns its value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.
uint atomicCounterMin(atomic_uint *c*, uint *data*)	Atomically 1. sets the counter for *c* to the minimum of the value of the counter and the value of *data*, and 2. returns the value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.

Syntax	Description
uint atomicCounterMax(atomic_uint *c*, uint *data*)	Atomically 1. sets the counter for *c* to the maximum of the value of the counter and the value of *data*, and 2. returns the value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.
uint atomicCounterAnd(atomic_uint *c*, uint *data*)	Atomically 1. sets the counter for *c* to the bitwise AND of the value of the counter and the value of *data*, and 2. returns the value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.
uint atomicCounterOr(atomic_uint *c*, uint *data*)	Atomically 1. sets the counter for *c* to the bitwise OR of the value of the counter and the value of *data*, and 2. returns the value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.
uint atomicCounterXor(atomic_uint *c*, uint *data*)	Atomically 1. sets the counter for *c* to the bitwise XOR of the value of the counter and the value of *data*, and 2. returns the value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.

Syntax	Description
uint atomicCounterExchange(atomic_uint *c*, uint *data*)	Atomically 1. sets the counter value for *c* to the value of *data*, and 2. returns its value prior to the operation. These two steps are done atomically with respect to the atomic counter functions in this table.
uint atomicCounterCompSwap(atomic_uint *c*, uint *compare*, uint *data*)	Atomically 1. compares the value of *compare* and the counter value for *c* 2. if the values are equal, sets the counter value for *c* to the value of *data*, and 3. returns its value prior to the operation. These three steps are done atomically with respect to the atomic counter functions in this table.

8.11. Atomic Memory Functions

Atomic memory functions perform atomic operations on an individual signed or unsigned integer stored in buffer object or shared variable storage. All of the atomic memory operations read a value from memory, compute a new value using one of the operations described below, write the new value to memory, and return the original value read. The contents of the memory being updated by the atomic operation are guaranteed not to be modified by any other assignment or atomic memory function in any shader invocation between the time the original value is read and the time the new value is written.

Atomic memory functions are supported only for a limited set of variables. A shader will fail to compile if the value passed to the *mem* argument of an atomic memory function does not correspond to a buffer or shared variable. It is acceptable to pass an element of an array or a single component of a vector to the *mem* argument of an atomic memory function, as long as the underlying array or vector is a buffer or shared variable.

All the built-in functions in this section accept arguments with combinations of restrict, coherent, and volatile memory qualification, despite not having them listed in the prototypes. The atomic operation will operate as required by the calling argument's memory qualification, not by the built-in function's formal parameter memory qualification.

Syntax	Description
uint atomicAdd(inout uint *mem*, uint *data*) int atomicAdd(inout int *mem*, int *data*)	Computes a new value by adding the value of *data* to the contents *mem*.
uint atomicMin(inout uint *mem*, uint *data*) int atomicMin(inout int *mem*, int *data*)	Computes a new value by taking the minimum of the value of *data* and the contents of *mem*.
uint atomicMax(inout uint *mem*, uint *data*) int atomicMax(inout int *mem*, int *data*)	Computes a new value by taking the maximum of the value of *data* and the contents of *mem*.
uint atomicAnd(inout uint *mem*, uint *data*) int atomicAnd(inout int *mem*, int *data*)	Computes a new value by performing a bit-wise AND of the value of *data* and the contents of *mem*.
uint atomicOr(inout uint *mem*, uint *data*) int atomicOr(inout int *mem*, int *data*)	Computes a new value by performing a bit-wise OR of the value of *data* and the contents of *mem*.
uint atomicXor(inout uint *mem*, uint *data*) int atomicXor(inout int *mem*, int *data*)	Computes a new value by performing a bit-wise EXCLUSIVE OR of the value of *data* and the contents of *mem*.
uint atomicExchange(inout uint *mem*, uint *data*) int atomicExchange(inout int *mem*, int *data*)	Computes a new value by simply copying the value of *data*.
uint atomicCompSwap(inout uint *mem*, uint *compare*, uint *data*) int atomicCompSwap(inout int *mem*, int *compare*, int *data*)	Compares the value of *compare* and the contents of *mem*. If the values are equal, the new value is given by *data*; otherwise, it is taken from the original contents of *mem*.

8.12. Image Functions

Variables using one of the image basic types may be used by the built-in shader image memory functions defined in this section to read and write individual texels of a texture. Each image variable references an image unit, which has a texture image attached.

When image memory functions below access memory, an individual texel in the image is identified using an (*i*), (*i*, *j*), or (*i*, *j*, *k*) coordinate corresponding to the values of *P*. For image2DMS and image2DMSArray variables (and the corresponding int/unsigned int types) corresponding to multisample textures, each texel may have multiple samples and an individual sample is identified using the integer *sample* parameter. The coordinates and sample number are used to select an individual texel in the manner described in section 8.26 ÒTexture Image Loads and StoresÓ of the OpenGL Specification.

Loads and stores support float, integer, and unsigned integer types. The data types below starting Ò*gimage*Ó serve as placeholders meaning types starting either ÒimageÓ, ÒiimageÓ, or ÒuimageÓ in the same way as *gvec* or *gsampler* in earlier sections.

The *IMAGE_PARAMS* in the prototypes below is a placeholder representing 33 separate functions, each for a different type of image variable. The *IMAGE_PARAMS* placeholder is replaced by one of the following parameter lists:

gimage2D *image*, ivec2 *P*

gimage3D *image*, ivec3 *P*

gimageCube *image*, ivec3 *P*

gimageBuffer *image*, int *P*

gimage2DArray *image*, ivec3 *P*

gimageCubeArray *image*, ivec3 *P*

gimage1D *image*, int *P*

gimage1DArray *image*, ivec2 *P*

gimage2DRect *image*, ivec2 *P*

gimage2DMS *image*, ivec2 *P*, int *sample*

gimage2DMSArray *image*, ivec3 *P*, int *sample*

where each of the lines represents one of three different image variable types, and *image*, *P*, and *sample* specify the individual texel to operate on. The method for identifying the individual texel operated on from *image*, *P*, and *sample*, and the method for reading and writing the texel are specified in section 8.26 ÒTexture Image Loads and StoresÓ of the OpenGL Specification.

The atomic functions perform operations on individual texels or samples of an image variable. Atomic memory operations read a value from the selected texel, compute a new value using one of the operations described below, write the new value to the selected texel, and return the original value read. The contents of the texel being updated by the atomic operation are guaranteed not to be modified by any other image store or atomic function between the time the original value is read and the time the new value is written.

Atomic memory operations are supported on only a subset of all image variable types; *image* must be either:

- a signed integer image variable (type starts ÒiimageÓ) and a format qualifier of r32i, used with a *data* argument of type int, or
- an unsigned integer image variable (type starts ÒuimageÓ) and a format qualifier of r32ui, used with a *data* argument of type uint, or
- a float image variable (type starts ÒimageÓ) and a format qualifier of r32f, used with a *data* argument of type float (imageAtomicExchange only).

All the built-in functions in this section accept arguments with combinations of restrict, coherent, and volatile memory qualification, despite not having them listed in the prototypes. The image operation will operate as required by the calling argumentÕs memory qualification, not by the built-in functionÕs formal parameter memory qualification.

Syntax	Description
int imageSize(readonly writeonly gimage1D *image*) ivec2 imageSize(readonly writeonly gimage2D *image*) ivec3 imageSize(readonly writeonly gimage3D *image*) ivec2 imageSize(readonly writeonly gimageCube *image*) ivec3 imageSize(readonly writeonly gimageCubeArray *image*) ivec3 imageSize(readonly writeonly gimage2DArray *image*) ivec2 imageSize(readonly writeonly gimageRect *image*) ivec2 imageSize(readonly writeonly gimage1DArray *image*) ivec2 imageSize(readonly writeonly gimage2DMS *image*) ivec3 imageSize(readonly writeonly gimage2DMSArray *image*) int imageSize(readonly writeonly gimageBuffer *image*)	Returns the dimensions of the image or images bound to *image*. For arrayed images, the last component of the return value will hold the size of the array. Cube images only return the dimensions of one face, and the number of cubes in the cube map array, if arrayed. Note: The qualification readonly writeonly accepts a variable qualified with readonly, writeonly, both, or neither. It means the formal argument will be used for neither reading nor writing to the underlying memory.
int imageSamples(readonly writeonly gimage2DMS *image*) int imageSamples(readonly writeonly gimage2DMSArray *image*)	Returns the number of samples of the image or images bound to *image*.
gvec4 imageLoad(readonly *IMAGE_PARAMS*)	Loads the texel at the coordinate *P* from the image unit *image* (in *IMAGE_PARAMS*). For multisample loads, the sample number is given by *sample*. When *image*, *P*, and *sample* identify a valid texel, the bits used to represent the selected texel in memory are converted to a vec4, ivec4, or uvec4 in the manner described in section 8.26 ÒTexture Image Loads and StoresÓ of the OpenGL Specification and returned.
void imageStore(writeonly *IMAGE_PARAMS*, gvec4 *data*)	Stores *data* into the texel at the coordinate *P* from the image specified by *image*. For multisample stores, the sample number is given by *sample*. When *image*, *P*, and *sample* identify a valid texel, the bits used to represent *data* are converted to the format of the image unit in the manner described in section 8.26 ÒTexture Image Loads and StoresÓ of the OpenGL Specification and stored to the specified texel.
uint imageAtomicAdd(*IMAGE_PARAMS*, uint *data*) int imageAtomicAdd(*IMAGE_PARAMS*, int *data*)	Computes a new value by adding the value of *data* to the contents of the selected texel.

Syntax	Description
uint imageAtomicMin(*IMAGE_PARAMS*, uint *data*) int imageAtomicMin(*IMAGE_PARAMS*, int *data*)	Computes a new value by taking the minimum of the value of *data* and the contents of the selected texel.
uint imageAtomicMax(*IMAGE_PARAMS*, uint *data*) int imageAtomicMax(*IMAGE_PARAMS*, int *data*)	Computes a new value by taking the maximum of the value *data* and the contents of the selected texel.
uint imageAtomicAnd(*IMAGE_PARAMS*, uint *data*) int imageAtomicAnd(*IMAGE_PARAMS*, int *data*)	Computes a new value by performing a bit-wise AND of the value of *data* and the contents of the selected texel.
uint imageAtomicOr(*IMAGE_PARAMS*, uint *data*) int imageAtomicOr(*IMAGE_PARAMS*, int *data*)	Computes a new value by performing a bit-wise OR of the value of *data* and the contents of the selected texel.
uint imageAtomicXor(*IMAGE_PARAMS*, uint *data*) int imageAtomicXor(*IMAGE_PARAMS*, int *data*)	Computes a new value by performing a bit-wise EXCLUSIVE OR of the value of *data* and the contents of the selected texel.
uint imageAtomicExchange(*IMAGE_PARAMS*, uint *data*) int imageAtomicExchange(*IMAGE_PARAMS*, int *data*) float imageAtomicExchange(*IMAGE_PARAMS*, float *data*)	Computes a new value by simply copying the value of *data*.
uint imageAtomicCompSwap(*IMAGE_PARAMS*, uint *compare*, uint *data*) int imageAtomicCompSwap(*IMAGE_PARAMS*, int *compare*, int *data*)	Compares the value of *compare* and the contents of the selected texel. If the values are equal, the new value is given by *data*; otherwise, it is taken from the original value loaded from the texel.

8.13. Geometry Shader Functions

These functions are only available in geometry shaders. They are described in more depth following the table.

Syntax	Description
void EmitStreamVertex(int *stream*)	Emits the current values of output variables to the current output primitive on stream *stream*. The argument to *stream* must be a constant integral expression. On return from this call, the values of all output variables are undefined. Can only be used if multiple output streams are supported.
void EndStreamPrimitive(int *stream*)	Completes the current output primitive on stream *stream* and starts a new one. The argument to *stream* must be a constant integral expression. No vertex is emitted. Can only be used if multiple output streams are supported.

Syntax	Description
void EmitVertex()	Emits the current values of output variables to the current output primitive. When multiple output streams are supported, this is equivalent to calling EmitStreamVertex(0). On return from this call, the values of output variables are undefined.
void EndPrimitive()	Completes the current output primitive and starts a new one. When multiple output streams are supported, this is equivalent to calling EndStreamPrimitive(0). No vertex is emitted.

The function EmitStreamVertex() specifies that a vertex is completed. A vertex is added to the current output primitive in vertex stream *stream* using the current values of all built-in and user-defined output variables associated with *stream*. The values of all output variables for all output streams are undefined after a call to EmitStreamVertex(). If a geometry shader invocation has emitted more vertices than permitted by the output layout qualifier max_vertices, the results of calling EmitStreamVertex() are undefined.

The function EndStreamPrimitive() specifies that the current output primitive for vertex stream *stream* is completed and a new output primitive (of the same type) will be started by any subsequent EmitStreamVertex(). This function does not emit a vertex. If the output layout is declared to be points, calling EndStreamPrimitive() is optional.

A geometry shader starts with an output primitive containing no vertices for each stream. When a geometry shader terminates, the current output primitive for each stream is automatically completed. It is not necessary to call EndStreamPrimitive() if the geometry shader writes only a single primitive.

Multiple output streams are supported only if the output primitive type is declared to be points. It is a compile-time or link-time error if a program contains a geometry shader calling EmitStreamVertex() or EndStreamPrimitive() if its output primitive type is not points.

8.14. Fragment Processing Functions

Fragment processing functions are only available in fragment shaders.

8.14.1. Derivative Functions

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:

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

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

Backward differencing:

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

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

With single-sample rasterization, $dx \leq 1.0$ in equations 1b and 2b. For multisample rasterization, $dx < 2.0$ in equations 1b and 2b.

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

With multisample rasterization, for any given fragment or sample, either neighboring fragments or samples may be considered.

It is typical to consider a 2x2 square of fragments or samples, and compute independent dFdxFine per row and independent dFdyFine per column, while computing only a single dFdxCoarse and a single dFdyCoarse for the entire 2x2 square. Thus, all second-order coarse derivatives, e.g. dFdxCoarse(dFdxCoarse(*x*)), may be 0, even for non-linear arguments. However, second-order fine derivatives, e.g. dFdxFine(dFdyFine(*x*)) will properly reflect the difference between the independent fine derivatives computed within the 2x2 square.

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 14.2 ÒInvarianceÓ of the OpenGL Specification, is relaxed for derivative calculations, because the method may be a function of fragment location.

In some implementations, varying degrees of derivative accuracy for dFdx and dFdy may be obtained by providing GL hints (see section 21.4 ÒHintsÓ of the OpenGL Specification), allowing a user to make an image quality versus speed trade off. These hints have no effect on dFdxCoarse, dFdyCoarse, dFdxFine and dFdyFine.

Syntax	Description
genFType dFdx(genFType *p*)	Returns either dFdxFine(*p*) or dFdxCoarse(*p*), based on implementation choice, presumably whichever is the faster, or by whichever is selected in the API through quality-versus-speed hints.
genFType dFdy(genFType *p*)	Returns either dFdyFine(*p*) or dFdyCoarse(*p*), based on implementation choice, presumably whichever is the faster, or by whichever is selected in the API through quality-versus-speed hints.
genFType dFdxFine(genFType *p*)	Returns the partial derivative of *p* with respect to the window x coordinate. Will use local differencing based on the value of *p* for the current fragment and its immediate neighbor(s).

Syntax	Description
genFType dFdyFine(genFType *p*)	Returns the partial derivative of *p* with respect to the window y coordinate. Will use local differencing based on the value of *p* for the current fragment and its immediate neighbor(s).
genFType dFdxCoarse(genFType *p*)	Returns the partial derivative of *p* with respect to the window x coordinate. Will use local differencing based on the value of *p* for the current fragment's neighbors, and will possibly, but not necessarily, include the value of *p* for the current fragment. That is, over a given area, the implementation can x compute derivatives in fewer unique locations than would be allowed for dFdxFine(*p*).
genFType dFdyCoarse(genFType *p*)	Returns the partial derivative of *p* with respect to the window y coordinate. Will use local differencing based on the value of *p* for the current fragment's neighbors, and will possibly, but not necessarily, include the value of *p* for the current fragment. That is, over a given area, the implementation can compute y derivatives in fewer unique locations than would be allowed for dFdyFine(*p*).
genFType fwidth(genFType *p*)	Returns abs(dFdx(*p*)) + abs(dFdy(*p*)).
genFType fwidthFine(genFType *p*)	Returns abs(dFdxFine(*p*)) + abs(dFdyFine(*p*)).
genFType fwidthCoarse(genFType *p*)	Returns abs(dFdxCoarse(*p*)) + abs(dFdyCoarse (*p*)).

8.14.2. Interpolation Functions

Built-in interpolation functions are available to compute an interpolated value of a fragment shader input variable at a shader-specified (*x*, *y*) location. A separate (*x*, *y*) location may be used for each invocation of the built-in function, and those locations may differ from the default (*x*, *y*) location used to produce the default value of the input.

For all of the interpolation functions, *interpolant* must be an l-value from an in declaration; this can include a variable, a block or structure member, an array element, or some combination of these. Additionally, component selection operators (e.g. .xy, .xxz) may be applied to *interpolant*, in which case the interpolation function will return the result of applying the component selection operator to the interpolated value of *interpolant* (for example, interpolateAt(v.xxz) is defined to return interpolateAt(v).xxz). Arrayed inputs can be indexed with general (nonuniform) integer expressions.

If *interpolant* is declared with the flat qualifier, the interpolated value will have the same value everywhere for a single primitive, so the location used for interpolation has no effect and the functions just return that same value. If *interpolant* is declared with the centroid qualifier, the value returned by interpolateAtSample() and interpolateAtOffset() will be evaluated at the specified location, ignoring the location normally used with the centroid qualifier. If *interpolant* is

declared with the noperspective qualifier, the interpolated value will be computed without perspective correction.

Syntax	Description
float interpolateAtCentroid(float *interpolant*) vec2 interpolateAtCentroid(vec2 *interpolant*) vec3 interpolateAtCentroid(vec3 *interpolant*) vec4 interpolateAtCentroid(vec4 *interpolant*)	Returns the value of the input *interpolant* sampled at a location inside both the pixel and the primitive being processed. The value obtained would be the same value assigned to the input variable if declared with the centroid qualifier.
float interpolateAtSample(float *interpolant*, int *sample*) vec2 interpolateAtSample(vec2 *interpolant*, int *sample*) vec3 interpolateAtSample(vec3 *interpolant*, int *sample*) vec4 interpolateAtSample(vec4 *interpolant*, int *sample*)	Returns the value of the input *interpolant* variable at the location of sample number *sample*. If multisample buffers are not available, the input variable will be evaluated at the center of the pixel. If sample *sample* does not exist, the position used to interpolate the input variable is undefined.
float interpolateAtOffset(float *interpolant*, vec2 offset) vec2 interpolateAtOffset(vec2 *interpolant*, vec2 offset) vec3 interpolateAtOffset(vec3 *interpolant*, vec2 offset) vec4 interpolateAtOffset(vec4 *interpolant*, vec2 offset)	Returns the value of the input *interpolant* variable sampled at an offset from the center of the pixel specified by *offset*. The two floating-point components of *offset*, give the offset in pixels in the *x* and *y* directions, respectively. An offset of (0, 0) identifies the center of the pixel. The range and granularity of offsets supported by this function is implementation-dependent.

8.15. Noise Functions

The noise functions noise1, noise2, noise3, and noise4 have been deprecated starting with version 4.4 of GLSL. When not generating SPIR-V they are defined to return the value 0.0 or a vector whose components are all 0.0. When generating SPIR-V the noise functions are not declared and may not be used.

As in previous releases, the noise functions are not semantically considered to be compile-time constant expressions.

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

8.16. Shader Invocation Control Functions

The shader invocation control function is only available in tessellation control and compute shaders. It is used to control the relative execution order of multiple shader invocations used to process a patch (in the case of tessellation control shaders) or a local work group (in the case of compute shaders), which are otherwise executed with an undefined relative order.

Syntax	Description
void barrier()	For any given static instance of barrier(), all tessellation control shader invocations for a single input patch must enter it before any will be allowed to continue beyond it, or all compute shader invocations for a single work group must enter it before any will continue beyond it.

The function barrier() provides a partially defined order of execution between shader invocations. This ensures that values written by one invocation prior to a given static instance of barrier() can be safely read by other invocations after their call to the same static instance barrier(). Because invocations may execute in undefined order between these barrier calls, the values of a per-vertex or per-patch output variable or shared variables for compute shaders will be undefined in a number of cases enumerated in ÒOutput VariablesÓ (for tessellation control shaders) and ÒShared VariablesÓ (for compute shaders).

For tessellation control shaders, the barrier() function may only be placed inside the function main() of the shader and may not be called within any control flow. Barriers are also disallowed after a return statement in the function main(). Any such misplaced barriers result in a compile-time error.

For compute shaders, the barrier() function may be placed within control flow, but that control flow must be uniform control flow. That is, all the controlling expressions that lead to execution of the barrier must be dynamically uniform expressions. This ensures that if any shader invocation enters a conditional statement, then all invocations will enter it. While compilers are encouraged to give warnings if they can detect this might not happen, compilers cannot completely determine this. Hence, it is the authorÔs responsibility to ensure barrier() only exists inside uniform control flow. Otherwise, some shader invocations will stall indefinitely, waiting for a barrier that is never reached by other invocations.

8.17. Shader Memory Control Functions

Within a single shader invocation, the visibility and order of writes made by that invocation are well-defined. However, the relative order of reads and writes to a single shared memory address from multiple separate shader invocations is largely undefined. Additionally, the order of accesses to multiple memory addresses performed by a single shader invocation, as observed by other shader invocations, is also undefined.

The following built-in functions can be used to control the ordering of reads and writes:

Syntax	Description
void memoryBarrier()	Control the ordering of memory transactions issued by a single shader invocation.
void memoryBarrierAtomicCounter()	Control the ordering of accesses to atomic-counter variables issued by a single shader invocation.
void memoryBarrierBuffer()	Control the ordering of memory transactions to buffer variables issued within a single shader invocation.
void memoryBarrierShared()	Control the ordering of memory transactions to shared variables issued within a single shader invocation, as viewed by other invocations in the same work group. Only available in compute shaders.
void memoryBarrierImage()	Control the ordering of memory transactions to images issued within a single shader invocation.
void groupMemoryBarrier()	Control the ordering of all memory transactions issued within a single shader invocation, as viewed by other invocations in the same work group. Only available in compute shaders.

The memory barrier built-in functions can be used to order reads and writes to variables stored in memory accessible to other shader invocations. When called, these functions will wait for the completion of all reads and writes previously performed by the caller that access selected variable types, and then return with no other effect. The built-in functions memoryBarrierAtomicCounter(), memoryBarrierBuffer(), memoryBarrierImage(), and memoryBarrierShared() wait for the completion of accesses to atomic counter, buffer, image, and shared variables, respectively. The built-in functions memoryBarrier() and groupMemoryBarrier() wait for the completion of accesses to all of the above variable types. The functions memoryBarrierShared() and groupMemoryBarrier() are available only in compute shaders; the other functions are available in all shader types.

When these functions return, the effects of any memory stores performed using coherent variables prior to the call will be visible to any future[1] coherent access to the same memory performed by any other shader invocation. In particular, the values written this way in one shader stage are guaranteed to be visible to coherent memory accesses performed by shader invocations in subsequent stages when those invocations were triggered by the execution of the original shader invocation (e.g. fragment shader invocations for a primitive resulting from a particular geometry shader invocation).

1

An access is only a *future* access if a *happens-before* relation can be established between the store and the load.

Additionally, memory barrier functions order stores performed by the calling invocation, as observed by other shader invocations. Without memory barriers, if one shader invocation performs two stores to coherent variables, a second shader invocation might see the values written

by the second store prior to seeing those written by the first. However, if the first shader invocation calls a memory barrier function between the two stores, selected other shader invocations will never see the results of the second store before seeing those of the first. When using the functions groupMemoryBarrier() or memoryBarrierShared(), this ordering guarantee applies only to other shader invocations in the same compute shader work group; all other memory barrier functions provide the guarantee to all other shader invocations. No memory barrier is required to guarantee the order of memory stores as observed by the invocation performing the stores; an invocation reading from a variable that it previously wrote will always see the most recently written value unless another shader invocation also wrote to the same memory.

8.18. Shader Invocation Group Functions

Implementations of the OpenGL Shading Language may optionally group multiple shader invocations for a single shader stage into a single SIMD invocation group, where invocations are assigned to groups in an undefined, implementation-dependent manner. Shader algorithms on such implementations may benefit from being able to evaluate a composite of Boolean values over all active invocations in a group.

Syntax	Description
bool anyInvocation(bool *value*)	Returns true if and only if *value* is true for at least one active invocation in the group.
bool allInvocations(bool *value*)	Returns true if and only if *value* is true for all active invocations in the group.
bool allInvocationsEqual(bool *value*)	Returns true if *value* is the same for all active invocations in the group.

For all of these functions, the same result is returned to all active invocations in the group.

These functions may be called within conditionally executed code. In groups where some invocations do not execute the function call, the value returned by the function is not affected by any invocation not calling the function, even when value is well-defined for that invocation.

Because these functions depend on the values of *value* in an undefined group of invocations, the value returned by these functions is largely undefined. However, anyInvocation() is guaranteed to return true if *value* is true, and allInvocations() is guaranteed to return false if *value* is false.

Because implementations are not required to combine invocations into groups, simply returning *value* for anyInvocation() and allInvocations() and returning true for allInvocationsEqual() is a legal implementation of these functions.

For fragment shaders, invocations in a SIMD invocation group may include invocations corresponding to pixels that are covered by a primitive being rasterized, as well as invocations corresponding to neighboring pixels not covered by the primitive. *Helper invocations* (see ÒBuilt-In Language VariablesÓ) may be created and the value of *value* for such helper-invocation pixels may affect the value returned by anyInvocation(), allInvocations(), and allInvocationsEqual().

Chapter 9. Shading Language Grammar

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

editing-note

(Jon) Experimental way of indenting a bunch of stuff. Maybe a different coderay language would help?

The following describes the grammar for the OpenGL Shading Language in terms of the above tokens. The starting rule is *translation_unit*. An empty shader (one having no tokens to parse, after pre-processing) is valid, resulting in no compile-time errors, even though the grammar below does not have a rule to accept an empty token stream.

variable_identifier :

IDENTIFIER

primary_expression :

variable_identifier
INTCONSTANT
UINTCONSTANT
FLOATCONSTANT
BOOLCONSTANT
DOUBLECONSTANT
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

> **!** FIELD_SELECTION includes members in structures, component selection for vectors and the 'length' identifier for the length() method

integer_expression :

expression

function_call :

function_call_or_method

function_call_or_method :

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*.

> ! Methods (.length), subroutine array calls, and identifiers are recognized through *postfix_expression*.

function_identifier :

type_specifier
postfix_expression

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 SEMICOLON
type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE SEMICOLON
type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE IDENTIFIER SEMICOLON
type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE IDENTIFIER array_specifier SEMICOLON
type_qualifier SEMICOLON
type_qualifier IDENTIFIER SEMICOLON
type_qualifier IDENTIFIER identifier_list SEMICOLON

identifier_list :

COMMA IDENTIFIER
identifier_list COMMA IDENTIFIER

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 array_specifier

parameter_declaration :

type_qualifier parameter_declarator
parameter_declarator
type_qualifier parameter_type_specifier
parameter_type_specifier

parameter_type_specifier :

type_specifier

init_declarator_list :

single_declaration
init_declarator_list COMMA IDENTIFIER
init_declarator_list COMMA IDENTIFIER array_specifier
init_declarator_list COMMA IDENTIFIER array_specifier EQUAL initializer
init_declarator_list COMMA IDENTIFIER EQUAL initializer

single_declaration :

fully_specified_type
fully_specified_type IDENTIFIER
fully_specified_type IDENTIFIER array_specifier
fully_specified_type IDENTIFIER array_specifier EQUAL initializer
fully_specified_type IDENTIFIER EQUAL initializer

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

fully_specified_type :

type_specifier

type_qualifier type_specifier

invariant_qualifier :

INVARIANT

interpolation_qualifier :

SMOOTH
FLAT
NOPERSPECTIVE

layout_qualifier :

LAYOUT LEFT_PAREN layout_qualifier_id_list RIGHT_PAREN

layout_qualifier_id_list :

layout_qualifier_id
layout_qualifier_id_list COMMA layout_qualifier_id

layout_qualifier_id :

IDENTIFIER
IDENTIFIER EQUAL constant_expression
SHARED

precise_qualifier :

PRECISE

type_qualifier :

single_type_qualifier
type_qualifier single_type_qualifier

single_type_qualifier :

storage_qualifier
layout_qualifier
precision_qualifier
interpolation_qualifier
invariant_qualifier
precise_qualifier

storage_qualifier :

CONST
IN
OUT
INOUT
CENTROID
PATCH
SAMPLE
UNIFORM
BUFFER
SHARED

COHERENT
VOLATILE
RESTRICT
READONLY
WRITEONLY
SUBROUTINE
SUBROUTINE LEFT_PAREN type_name_list RIGHT_PAREN

type_name_list :

TYPE_NAME
type_name_list COMMA TYPE_NAME

type_specifier :

type_specifier_nonarray
type_specifier_nonarray array_specifier

array_specifier :

LEFT_BRACKET RIGHT_BRACKET
LEFT_BRACKET constant_expression RIGHT_BRACKET
array_specifier LEFT_BRACKET RIGHT_BRACKET
array_specifier LEFT_BRACKET constant_expression RIGHT_BRACKET

type_specifier_nonarray :

VOID
FLOAT
DOUBLE
INT
UINT
BOOL
VEC2
VEC3
VEC4
DVEC2
DVEC3
DVEC4
BVEC2
BVEC3
BVEC4
IVEC2
IVEC3
IVEC4
UVEC2
UVEC3
UVEC4
MAT2
MAT3
MAT4
MAT2X2

MAT2X3
MAT2X4
MAT3X2
MAT3X3
MAT3X4
MAT4X2
MAT4X3
MAT4X4
DMAT2
DMAT3
DMAT4
DMAT2X2
DMAT2X3
DMAT2X4
DMAT3X2
DMAT3X3
DMAT3X4
DMAT4X2
DMAT4X3
DMAT4X4
ATOMIC_UINT
SAMPLER2D
SAMPLER3D
SAMPLERCUBE
SAMPLER2DSHADOW
SAMPLERCUBESHADOW
SAMPLER2DARRAY
SAMPLER2DARRAYSHADOW
SAMPLERCUBEARRAY
SAMPLERCUBEARRAYSHADOW
ISAMPLER2D
ISAMPLER3D
ISAMPLERCUBE
ISAMPLER2DARRAY
ISAMPLERCUBEARRAY
USAMPLER2D
USAMPLER3D
USAMPLERCUBE
USAMPLER2DARRAY
USAMPLERCUBEARRAY
SAMPLER1D
SAMPLER1DSHADOW
SAMPLER1DARRAY
SAMPLER1DARRAYSHADOW
ISAMPLER1D
ISAMPLER1DARRAY
USAMPLER1D
USAMPLER1DARRAY

SAMPLER2DRECT
SAMPLER2DRECTSHADOW
ISAMPLER2DRECT
USAMPLER2DRECT
SAMPLERBUFFER
ISAMPLERBUFFER
USAMPLERBUFFER
SAMPLER2DMS
ISAMPLER2DMS
USAMPLER2DMS
SAMPLER2DMSARRAY
ISAMPLER2DMSARRAY
USAMPLER2DMSARRAY
IMAGE2D
IIMAGE2D
UIMAGE2D
IMAGE3D
IIMAGE3D
UIMAGE3D
IMAGECUBE
IIMAGECUBE
UIMAGECUBE
IMAGEBUFFER
IIMAGEBUFFER
UIMAGEBUFFER
IMAGE1D
IIMAGE1D
UIMAGE1D
IMAGE1DARRAY
IIMAGE1DARRAY
UIMAGE1DARRAY
IMAGE2DRECT
IIMAGE2DRECT
UIMAGE2DRECT
IMAGE2DARRAY
IIMAGE2DARRAY
UIMAGE2DARRAY
IMAGECUBEARRAY
IIMAGECUBEARRAY
UIMAGECUBEARRAY
IMAGE2DMS
IIMAGE2DMS
UIMAGE2DMS
IMAGE2DMSARRAY
IIMAGE2DMSARRAY
UIMAGE2DMSARRAY
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
type_qualifier type_specifier struct_declarator_list SEMICOLON

struct_declarator_list :

struct_declarator
struct_declarator_list COMMA struct_declarator

struct_declarator :

IDENTIFIER
IDENTIFIER array_specifier

initializer :

assignment_expression
LEFT_BRACE initializer_list RIGHT_BRACE
LEFT_BRACE initializer_list COMMA RIGHT_BRACE

initializer_list :

initializer
initializer_list COMMA initializer

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
SEMICOLON

function_definition :

function_prototype compound_statement_no_new_scope

In general the above grammar describes a super set of the OpenGL Shading Language. Certain constructs that are valid purely in terms of the grammar are disallowed by statements elsewhere in this specification.

Chapter 10. Acknowledgments

This specification is based on the work of those who contributed to past versions of the Open GL and Open GL ES Language Specifications and the following contributors to this version:

editing-note

(Jon) Would like to make multicolumnar, at least in PDF output.

Pat Brown, NVIDIA
Jeff Bolz, NVIDIA
Frank Chen
Pierre Boudier, AMD
Piers Daniell, NVIDIA
Chris Dodd, NVIDIA
Nick Haemel, NVIDIA
Jason Green, Transgaming
Brent Insko, Intel
Jon Leech
Bill Licea-Kane, Qualcomm
Daniel Koch, NVIDIA
Graeme Leese, Broadcom
Barthold Lichtenbelt, NVIDIA
Bruce Merry, ARM
Robert Ohannessian
Tom Olson, ARM
Brian Paul, VMware
Acorn Pooley, NVIDIA
Daniel Rakos, AMD
Christophe Riccio, AMD
Kevin Rogovin
Ian Romanick, Intel
Greg Roth, NVIDIA
Graham Sellers, AMD
Dave Shreiner, ARM
Jeremy Sandmel, Apple
Robert Simpson, Qualcomm
Eric Werness, NVIDIA
Mark Young, AMD

Chapter 11. Normative References

1. ÒOpenGL® ES, Version 3.2Ó, https://www.khronos.org/registry/OpenGL/index_es.php, November 3, 2016.
2. ÒThe OpenGL® Graphics System: A Specification, Version 4.6 (Core Profile)Ó, https://www.khronos.org/registry/OpenGL/index_gl.php, June 1, 2016.
3. International Standard ISO/IEC 14882:1998(E). Programming Languages - C++.
4. International Standard ISO/IEC 646:1991. Information technology - ISO 7-bit coded character set for information interchange
5. The Unicode Standard Version 6.0 - Core Specification.
6. IEEE 754-2008. *IEEE Standard for Floating-Point Arithmetic*