The OpenGL® Shading Language

Language Version: 4.20
Document Revision: 11
12-Dec-2011

Editor: John Kessenich

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

Copyright (c) 2008-2011 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group web-site should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos, OpenKODE, OpenKOGS, OpenVG, OpenMAX, OpenSL ES and OpenWF are trademarks of the Khronos Group Inc. COLLADA is a trademark of Sony Computer Entertainment Inc. used by permission by Khronos. OpenGL and OpenML are registered trademarks and the OpenGL ES logo is a trademark of Silicon Graphics Inc. used by permission by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Table of Contents

1 Introduction

This document specifies only version 4.20 of the OpenGL Shading Language. It requires __VERSION__ to substitute 420, and requires **#version** to accept only 420. 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 Graphics System Specification, Version 4.2, 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.

All OpenGL Graphics System Specification references in this specification are to version 4.2.

1.1 Acknowledgments

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

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, AMD
Daniel Koch, TransGaming
Barthold Lichtenbelt, NVIDIA
Bruce Merry, ARM
Robert Ohannessian
Acorn Pooley, NVIDIA
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

1.2 Changes

1.2.1 Summary of Changes from Version 4.10

Note: No features were deprecated between versions 4.10 and 4.20.

- Move these previously deprecated features to be only in the compatibility profile:
  - The keyword **attribute** for vertex shader inputs. (Use **in** instead.)
  - The keyword **varying** for inputs and outputs. (Use **in** and **out** instead.)
  - The original texturing built-in functions. (Use the new forms instead.)
  - The built-in variables *gl_FragColor* and *gl_FragData*. (Use **out** instead.)
  - Built-in constants related to these.

- Change from ASCII to UTF-8 for the language character set and also allow any characters inside comments (except the byte value 0), including '\'.
- Add line-continuation using '\', as in C++.
- ES convergence
  - Add table showing "Treatment of Mismatched Input/Output Variables" in section 4.3.4.
  - NaNs are not required to be generated, dividing a non-zero by zero makes inf, see 2nd paragraph 4.7.1 for full modification.
  - Clarify that .xyzwxy.xy is illegal, is it temporarily makes a "vec6".
  - Clarify that return statements only accept values (no return of a void function).
- Add image types (GL_ARB_shader_image_load_store)
  - 33 new types, all with "**image**" in their name, correspond to the non-shadow texture types
  - addition of memory qualifiers: **coherent**, **volatile**, **restrict**, **readonly**, and **writeonly**
  - can read/write/modify images from a shader, through new built-in functions
  - qualifiers can act independently on the opaque shader variable and the backing image, so extra qualifiers can be used to separately qualify these
- Variables declared in **if** and **else** statements are scoped only to the end of those statements. Note, this is not backward compatible, it may depend on #**version**.
- Allow implicit conversions of return values to the declared type of the function.
- The **const** keyword can be used to declare variables within a function body with initializer expressions that are not constant expressions.
- Qualifiers on variable declarations no longer have to follow a strict order. The **layout** qualifier can be used multiple times, and multiple parameter qualifiers can be used.
- Parameter qualifiers can include precision and memory qualifiers.
- Add a new **atomic_uint** type to support atomic counters. Also, add built-in functions for manipulating atomic counters.
  - **atomicCounterIncrement**, **atomicCounterDecrement**, and **atomicCounter**
- Add layout qualifier identifiers *binding* and *offset* to bind units to sampler and image variable declarations, atomic counters, and uniform blocks.
- Add built-in functions to pack/unpack 16 bit floating-point numbers (ARB_shading_language_pack2f).
  - **packHalf2x16** and **unpackHalf2x16**
  - **packSnorm2x16** and **unpackSnorm2x16**
- Add **gl_FragDepth** layout qualifiers to communicate what kind of changes will be made to **gl_FragDepth** (GL_AMD_conservative depth).

- Add C-style curly brace initializer lists syntax for initializers. Full initialization of aggregates is required when these are used.
- Allow **.length**() to be applied to vectors and matrices, returning the number of components or columns.
- Clarify that **.length**() returns an **int** type and can be used as a constant integer expression.
- Allow swizzle operations on scalars.
- Even the potential for recursion through subroutine uniforms is an error.
- Positive signed decimal literals, as well as octal and hexadecimal, can set all 32 bits. This includes setting the sign bit to create a negative value.
- Make GLSL consistent with the API regarding user clipping, by no longer referring to *gl_Position* when *gl_ClipVertex* is not written. Rather, user clipping becomes undefined.
- Minor consistency fixes, corrections
    - Consistently state structures have members not fields. The period (**.**) is still called the field selector, for all its multiple uses.
    - Remove comment that there is no communication between instantiations of a shader.
    - Clarified that a comma sequence-operator expression cannot be a constant expression. E.g., “(2,3)” is not allowed, semantically, as a valid constant expression 3, even though it is an expression that will evaluate to 3.
    - Use **vec2** instead of **vec3** for coordinate in textureGather*(sampler2DRect,...).
    - Clarify that **textureGatherOffset**() can take non-constants for the offsets.
- Fix typo to allow integer cube-array textures in the built-in function textureSize().
- Correctly restrict indexes of uniform blocks to being dynamically-uniform integral expressions. This correction also applies to earlier releases (4.00 and 4.10), which inadvertently allowed non-uniform indexes.
- Update the grammar in the last section.

1.3 Overview

This document describes *The OpenGL Shading Language,* version 4.20.

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

1.4 Error Handling

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

1.5 Typographical Conventions

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

1.6 Deprecation

Note that while this section is true of GLSL in general, version 4.2 has no deprecated features in it. Previously deprecated features have been moved to the compatibility profile and no new features have been deprecated.

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

2 Overview of OpenGL Shading

The OpenGL Shading Language is actually 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, and fragment 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, or fragment.

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

2.1 Vertex Processor

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

The vertex processor operates on one vertex at a time. It does not replace graphics operations that require knowledge of several vertices at a time.

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 complete set of tessellation control shaders are 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 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 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 complete set of tessellation evaluation shaders are 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 complete set of geometry shaders are 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 complete set of fragment shaders are compiled and linked, they result in a *fragment shader executable* that runs on the fragment processor.

A fragment shader cannot change a fragment's (x, y) position. Access to neighboring fragments is not allowed. The values computed by the fragment shader are ultimately used to update framebuffer memory or texture memory, depending on the current OpenGL state and the OpenGL command that caused the fragments to be generated.

3 Basics

3.1 Character Set

The source character set used for the OpenGL shading languages, outside of comments, is a subset of UTF-8. It includes the following characters:

> The letters **a-z**, **A-Z,** and the underscore (_).
>
> The numbers **0-9**.
>
> The symbols period (**.**), plus (+), dash (**-**), slash (/), asterisk (*), percent (**%**), angled brackets (< and >), square brackets (**[** and **]**), parentheses (**(** and **)**), braces (**{** and **}**), caret (^), vertical bar (|), ampersand (**&**), tilde (~), equals (=), exclamation point (**!**), colon (**:**), semicolon (**;**), comma (**,**), and question mark (**?**).
>
> The number sign (#) for preprocessor use.
>
> The backslash (\ ) as the line-continuation character when used as the last character of a line, just before a new line.
>
> White space: the space character, horizontal tab, vertical tab, form feed, carriage-return, and line-feed.

An error will be given if any other character is used outside a comment.

There are no 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.

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

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

There is no end-of-file character.

3.2 Source Strings

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

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

```
float f\
oo;
// forms a single line equivalent to "float foo;"
// (assuming '\' is the last character before the new line and "oo" are
// the first two characters of the next line)
```

3.3 Preprocessor

There is a preprocessor that processes the source strings as part of the compilation process.

The complete list of preprocessor directives is as follows.

```
#
#define
#undef

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

#error
#pragma

#extension
#version

#line
```

The following operators are also available

```
defined
##
```

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

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

The following predefined macros are available

```
__LINE__
__FILE__
__VERSION__
```

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

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

__*VERSION*__ will substitute a decimal integer reflecting the version number of the OpenGL shading language. The version of the shading language described in this document will have __*VERSION*__ substitute the decimal integer 420.

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

#if, #ifdef, #ifndef, #else, #elif, and **#endif** are defined to operate as is standard for C++ preprocessors. Expressions following **#if** and **#elif** are further restricted to expressions operating on literal integer constants, plus identifiers consumed by the **defined** operator. It is an error to use **#if** or **#elif** on expressions containing undefined macro names, other than as arguments to the **defined** operator. Character constants are not supported. The operators available are as follows.

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

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

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

Two tokens in a macro can be concatenated into one token using the token pasting (**##**) operator, as is standard for C++ preprocessors. The result must be a valid single token, which will then be subject to macro expansion. That is, macro expansion happens 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 diagnostic message into the shader object's information log (section 6.1.12 "Shader and Program Queries" in the OpenGL Graphics System Specification for how to access a shader object's information log). The message will be the tokens following the **#error** directive, up to the first new line. The implementation must then consider the shader to be ill-formed.

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

```
#pragma STDGL
```

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

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

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

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

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

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

```
#version number profile_opt
```

where *number* must be a version of the language, following the same convention as *__VERSION__* above. The directive "**#version 420**" is required in any shader that uses version 4.20 of the language. Any *number* representing a version of the language a compiler does not support will cause an error to be generated. Version 1.10 of the language does not require shaders to include this directive, and shaders that do not include a **#version** directive will be treated as targeting version 1.10. Shaders that specify **#version** 100 will be treated as targeting version 1.00 of the OpenGL ES Shading Language.

Shaders declaring version 1.40, 1.50, 3.30, 4.0, or 4.1 of the shading language can be linked with shaders declaring version 4.20 in the same program. Shaders targeting earlier versions (1.30 or earlier) of the shading language cannot be linked with version 4.20 shaders.

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

```
core
compatibility
```

If no *profile* argument is provided, the default is **core**. 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.

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

```
#define GL_core_profile 1
```

Implementations providing the compatibility profile provide the following macro:

```
#define GL_compatibility_profile 1
```

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

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

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

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

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

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

The initial state of the compiler is as if the directive

```
#extension all : disable
```

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

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

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

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

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

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

3.4 Comments

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

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.

```
// a single-line comment containing the next line \
a = b; // this is still in the first comment
```

3.5 Tokens

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

token:
keyword
identifier
integer-constant
floating-constant
operator
; { }

3.6 Keywords

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

attribute const uniform varying

coherent volatile restrict readonly writeonly

atomic_uint

layout

centroid flat smooth noperspective

patch sample

break continue do for while switch case default

if else

subroutine

in out inout

float double int void bool true false

invariant

discard return

mat2 mat3 mat4 dmat2 dmat3 dmat4

mat2x2 mat2x3 mat2x4 dmat2x2 dmat2x3 dmat2x4

mat3x2 mat3x3 mat3x4 dmat3x2 dmat3x3 dmat3x4

mat4x2 mat4x3 mat4x4 dmat4x2 dmat4x3 dmat4x4

vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4 dvec2 dvec3 dvec4

uint uvec2 uvec3 uvec4

lowp mediump highp precision
sampler1D sampler2D sampler3D samplerCube
sampler1DShadow sampler2DShadow samplerCubeShadow
sampler1DArray sampler2DArray
sampler1DArrayShadow sampler2DArrayShadow
isampler1D isampler2D isampler3D isamplerCube
isampler1DArray isampler2DArray
usampler1D usampler2D usampler3D usamplerCube
usampler1DArray usampler2DArray
sampler2DRect sampler2DRectShadow isampler2DRect usampler2DRect
samplerBuffer isamplerBuffer usamplerBuffer
sampler2DMS isampler2DMS usampler2DMS
sampler2DMSArray isampler2DMSArray usampler2DMSArray
samplerCubeArray samplerCubeArrayShadow isamplerCubeArray usamplerCubeArray
image1D iimage1D uimage1D
image2D iimage2D uimage2D
image3D iimage3D uimage3D
image2DRect iimage2DRect uimage2DRect
imageCube iimageCube uimageCube
imageBuffer iimageBuffer uimageBuffer
image1DArray iimage1DArray uimage1DArray
image2DArray iimage2DArray uimage2DArray
imageCubeArray iimageCubeArray uimageCubeArray
image2DMS iimage2DMS uimage2DMS
image2DMSArray iimage2DMSArray uimage2DMSArray
struct

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

common partition active
asm
class union enum typedef template this packed
resource

goto

inline noinline public static extern external interface

long short half fixed unsigned superp

input output

hvec2 hvec3 hvec4 fvec2 fvec3 fvec4

sampler3DRect

filter

sizeof cast

namespace using

row_major

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

3.7 Identifiers

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

identifier
    nondigit
    identifier nondigit
    identifier digit

nondigit: one of
    _ a b c d e f g h i j k l m n o p q r s t u v w x y z
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

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

Identifiers starting with “gl_” are reserved for use by OpenGL, and may not be declared in a shader as either a variable or a function. However, as noted in the specification, there are some cases where previously declared variables can be redeclared, and predeclared "gl_" names are allowed to be redeclared in a shader only for these specific purposes. More generally, it is an error to redeclare a variable, including those starting “gl_”.

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

3.8.2 Uniform and Non-Uniform Control Flow

When executing statements in a fragment shader, control flow starts as *uniform control flow*; all fragments enter the same control path into *main*(). Control flow becomes *non-uniform* when different fragments take different paths through control-flow statements (selection, iteration, and jumps). Control flow subsequently returns to being uniform after such divergent sub-statements or skipped code completes, until the next time different control paths are taken.

For example:

```
main()
{
    float a = ...;// this is uniform flow control
    if (a < b) {  // this expression is true for some fragments, not all
        ....;     // non-uniform flow control
    } else {
        ....;     // non-uniform flow control
    }
    ....;         // uniform flow control again
}
```

Other examples of non-uniform flow control can occur within switch statements and after conditional breaks, continues, early returns, and after fragment discards, when the condition is true for some fragments but not others. Loop iterations that only some fragments execute are also non-uniform flow control.

This is similarly defined for other shader stages, based on the per-instance data items they process.

3.8.3 Dynamically Uniform Expressions

A fragment-shader expression is *dynamically uniform* if all fragments evaluating it get the same resulting value. When loops are involved, this refers to the expression's value for the same loop iteration. When functions are involved, this refers to calls from the same call point.

This is similarly defined for other shader stages, based on the per-instance data they process.

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

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 4.1.10 "Implicit Conversions" and as referenced by other sections in this specification.

4.1 Basic Types

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

Transparent types

Type	Meaning
void	for functions that do not return a value
bool	a conditional type, taking on values of true or false
int	a signed integer
uint	an unsigned integer
float	a single floating-point scalar
double	a single double-precision floating point scalar
vec2	a two-component floating-point vector
vec3	a three-component floating-point vector
vec4	a four-component floating-point vector
dvec2	a two-component double-precision floating-point vector
dvec3	a three-component double-precision floating-point vector
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

Type	**Meaning**
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 floating-point matrix
mat3	a 3×3 floating-point matrix
mat4	a 4×4 floating-point matrix
mat2x2	same as a **mat2**
mat2x3	a floating-point matrix with 2 columns and 3 rows
mat2x4	a floating-point matrix with 2 columns and 4 rows
mat3x2	a floating-point matrix with 3 columns and 2 rows
mat3x3	same as a **mat3**
mat3x4	a floating-point matrix with 3 columns and 4 rows
mat4x2	a floating-point matrix with 4 columns and 2 rows
mat4x3	a floating-point matrix with 4 columns and 3 rows
mat4x4	same as a **mat4**
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
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**

Floating Point Opaque Types

Type	**Meaning**
sampler1D **image1D**	a handle for accessing a 1D texture
sampler2D **image2D**	a handle for accessing a 2D texture
sampler3D **image3D**	a handle for accessing a 3D texture
samplerCube **imageCube**	a handle for accessing a cube mapped texture
sampler2DRect **image2DRect**	a handle for accessing a rectangular texture
sampler1DArray **image1DArray**	a handle for accessing a 1D array texture
sampler2DArray **image2DArray**	a handle for accessing a 2D array texture
samplerBuffer **imageBuffer**	a handle for accessing a buffer texture
sampler2DMS **image2DMS**	a handle for accessing a 2D multi-sample texture
sampler2DMSArray **image2DMSArray**	a handle for accessing a 2D multi-sample array texture
samplerCubeArray **imageCubeArray**	a handle for accessing a cube map array texture
sampler1DShadow	a handle for accessing a 1D depth texture with comparison
sampler2DShadow	a handle for accessing a 2D depth texture with comparison
sampler2DRectShadow	a handle for accessing a rectangular texture with comparison
sampler1DArrayShadow	a handle for accessing a 1D array depth texture with comparison
sampler2DArrayShadow	a handle for accessing a 2D array depth texture with comparison
samplerCubeShadow	a handle for accessing a cube map depth texture with comparison
samplerCubeArrayShadow	a handle for accessing a cube map array depth texture with comparison

Signed Integer Opaque Types

Type	**Meaning**
isampler1D **iimage1D**	a handle for accessing an integer 1D texture

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

Unsigned Integer Opaque Types

Type	Meaning
atomic_uint	a handle for accessing an unsigned integer atomic counter
usampler1D **uimage1D**	a handle for accessing an unsigned integer 1D texture
usampler2D **uimage2D**	a handle for accessing an unsigned integer 2D texture
usampler3D **uimage3D**	a handle for accessing an unsigned integer 3D texture
usamplerCube **uimageCube**	a handle for accessing an unsigned integer cube mapped texture
usampler2DRect **uimage2DRect**	a handle for accessing an unsigned integer rectangular texture
usampler1DArray **uimage1DArray**	a handle for accessing an unsigned integer 1D array texture
usampler2DArray **uimage2DArray**	a handle for accessing an unsigned integer 2D array texture

Type	Meaning
usamplerBuffer **uimageBuffer**	a handle for accessing an unsigned integer buffer texture
usampler2DMS **uimage2DMS**	a handle for accessing an unsigned integer 2D multi-sample texture
usampler2DMSArray **uimage2DMSArray**	a handle for accessing an unsigned integer 2D multi-sample texture array
usamplerCubeArray **uimageCubeArray**	a handle for accessing an unsigned integer cube map array texture

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

There are no pointer types.

4.1.1 Void

Functions that do not return a value must be declared as **void**. There is no default function return type. The keyword **void** cannot be used in any other declarations (except for empty formal or actual parameter lists).

4.1.2 Booleans

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

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

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

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

4.1.3 Integers

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

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

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

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

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

integer-suffix: one of
  u U

decimal-constant :
  nonzero-digit
  decimal-constant digit

octal-constant :
  0
  octal-constant octal-digit

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

digit :
  0
  nonzero-digit

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

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

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

No white space is allowed between the digits of an integer constant, including after the leading **0** or after the leading **0x** or **0X** of a constant, or before the suffix **u** or **U**. When 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 an 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. For example,

```
int  a = 0xffffffff;  // 32 bits, a gets the value -1
int  b = 0xffffffffU; // ERROR: can't convert uint to int
uint c = 0xffffffff;  // 32 bits, c gets the value 0xFFFFFFFF
uint d = 0xffffffffU; // 32 bits, d gets the value 0xFFFFFFFF
int  e = -1;          // the literal is "1", then negation is performed,
                      //   and the resulting non-literal 32-bit signed
                      //   bit pattern of 0xFFFFFFFF is assigned, giving e
                      //   the value of -1.
uint f = -1u;         // the literal is "1u", then negation is performed,
                      //   and the resulting non-literal 32-bit unsigned
                      //   bit pattern of 0xFFFFFFFF is assigned, giving f
                      //   the value of 0xFFFFFFFF.
int  g = 3000000000;  // a signed decimal literal taking 32 bits,
                      //   setting the sign bit, g gets -1294967296
int  h = 0xA0000000;  // okay, 32-bit signed hexadecimal
int  i = 5000000000;  // ERROR: needs more than 32 bits
int  j = 0xFFFFFFFFF; // ERROR: needs more that 32 bits
int  k = 0x80000000;  // k gets -2147483648 == 0x80000000
int  l = 2147483648;  // l gets -2147483648 (the literal set the sign bit)
```

Despite all these examples initializing variables, literals are recognized and given values and types independently of their context.

4.1.4 Floats

Single-precision and double-precision floating point variables are available for use in a variety of scalar calculations. Floating-point variables are defined as in the following example:

```
float a, b = 1.5;
double c, d = 2.0LF;
```

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 section 4.7.1 "Range and Precision" 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

4.1.5 Vectors

The OpenGL Shading Language includes data types for generic 2-, 3-, and 4-component vectors of floating-point values, integers, or Booleans. Floating-point vector variables can be used to store colors, normals, positions, texture coordinates, texture lookup results and the like. Boolean vectors can be used for component-wise comparisons of numeric vectors. Some examples of vector declaration are:

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

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

4.1.6 Matrices

The OpenGL Shading Language has built-in types for 2×2, 2×3, 2×4, 3×2, 3×3, 3×4, 4×2, 4×3, and 4×4 matrices of floating-point numbers. 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:

```
mat2 mat2D;
mat3 optMatrix;
mat4 view, projection;
mat4x4 view;  // an alternate way of declaring a mat4
mat3x2 m;     // a matrix with 3 columns and 2 rows
dmat4 highPrecisionMVP;
dmat2x4 dm;
```

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

4.1.7 Opaque Types

The opaque types 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 **uniform-**qualified variables. Except for array indexing, structure member selection, and parentheses, opaque variables are not allowed to be operands in expressions.

Opaque variables cannot be treated as l-values; hence cannot be used as **out** or **inout** function parameters, nor can they be assigned into. However, they can be passed as **in** parameters with matching type and memory qualifiers. They are initialized only through the OpenGL API; they cannot be declared with an initializer in a shader.

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 as a handle to.

4.1.7.1 Samplers

Sampler types (e.g., **sampler2D**) are opaque types, declared and behaving as described above for opaque types. They do not use memory qualifiers. 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 section 8.9 “Texture Functions” ”) and samplers are used to specify which texture to access and how it is to be filtered.

4.1.7.2 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 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 3.9.20 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 Section 8.11 "Image Functions" to specify an image to access.

4.1.7.3 Atomic Counters

Atomic counter types (**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 section 8.10 “Atomic Counter Functions”. They are bound to buffers as described in section 4.4.4.1 “Atomic Counter Layout Qualifiers”. When aggregated into arrays within a shader, atomic counters can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.

4.1.8 Structures

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

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

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

```
light lightVar2;
```

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

struct-definition :
    $qualifier_{opt}$ **struct** $name_{opt}$ { *member-list* } $declarators_{opt}$;

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

member-declaration :

basic-type declarators;

where *name* becomes the user-defined type, and can be used to declare variables to be of this new type. The *name* shares the same name space as other variables, types, and functions. 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 may not contain any other qualifiers. Bit fields are not supported. Member types must be already defined (there are no forward references). Member declarations cannot contain initializers. Member declarators can contain arrays. Such arrays must have a size specified, and the size must be an integral constant expression that's greater than zero (see section 4.3.3 “Constant Expressions”). Each level of structure has its own name space for names given in member declarators; such names need only be unique within that name space.

Anonymous structures are not supported. Embedded structure definitions are not supported.

```
struct S { float f; };

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

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

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.

4.1.9 Arrays

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

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

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

```
float[5]
```

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

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

as a constructor of an array

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

as an unnamed parameter

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

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

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

It is an error to declare arrays of arrays:

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

Arrays can have initializers formed from array constructors:

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

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

```
float a[5];
...
float b[] = a;  // b is explicitly size 5
float b[5] = a; // means the same thing
float b[] = float[](1,2,3,4,5);  // also explicitly sizes to 5
```

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

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

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

This returns a type **int**. The **length** method cannot be called on an array that has not been explicitly sized.

4.1.10 Implicit Conversions

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

Type of expression	Can be implicitly converted to
int	**uint**
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 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.

4.1.11 Initializers

At declaration, an initial value for an aggregate 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 aggregate 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* 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 section 4.1.10 "Implicit Conversions". Since these include constructors, an aggregate can be initialized by either a constructor or an initializer list; 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.

```
int i = { 1 };       // illegal, i is not an aggregate
```

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

If the aggregate 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 aggregate 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 aggregate 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:

```
mat2x2 a = mat2(  vec2( 1.0, 0.0 ), vec2( 0.0, 1.0 ) );
mat2x2 b =      { vec2( 1.0, 0.0 ), vec2( 0.0, 1.0 ) };
mat2x2 c =      {     { 1.0, 0.0 },     { 0.0, 1.0 } };
```

All of the following declarations are illegal.

```
float a[2] = { 3.4, 4.2, 5.0 };          // illegal
vec2 b = { 1.0, 2.0, 3.0 };              // illegal
mat3x3 c = { vec3(0.0), vec3(1.0), vec3(2.0), vec3(3.0) };    // illegal
mat2x2 d = { 1.0, 0.0, 0.0, 1.0 };       // illegal, can't flatten nesting
struct {
    float a;
    int   b;
} e = { 1.2, 2, 3 };                     // illegal
```

In all cases, the innermost 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 section 4.1.10 "Implicit Conversions". In the latter case, an implicit conversion will be done on the initializer before the assignment is done.

```
struct {
    float a;
    int   b;
} e = { 1.2, 2 };             // legal, all types match
```

```
struct {
    float a;
    int   b;
} e = { 1, 3 };               // legal, first initializer is converted
```

All of the following declarations are illegal.

```
int a = true;                         // illegal
vec4 b[2] = { vec4(0.0), 1.0 };       // illegal
mat4x2 c = { vec3(0.0), vec3(1.0) };  // illegal

struct S1 {
    vec4 a;
    vec4 b;
};

struct {
    float s;
    float t;
} d[] = { S1(vec4(0.0), vec4(1.1)) };  // illegal
```

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:

```
float a[] = float[](3.4, 4.2, 5.0, 5.2, 1.1);
float b[] = { 3.4, 4.2, 5.0, 5.2, 1.1 };
float c[] = a;                        // c is explicitly size 5
float d[5] = b;                       // means the same thing
```

It is an error to have too few or too many initializers in an initializer list for the aggregate 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 section 6.2 “Selection” and section 6.3 “Iteration” 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 function body has a scope nested inside the function’s definition. The **if** statement’s expression does not allow new variables to be declared, hence does not form a new scope.

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

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

struct S
{
      int x;
};

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

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

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

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

Shared globals are global variables declared with the same name in independently compiled units (shaders) within the same language (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. If no shader has an explicit size for the array, the largest implicit size is used. Scalars must have exactly the same type name and type definition. Structures must have the same name, sequence of type names, and type definitions, and 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 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
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

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. Which variable types can have which qualifiers are specifically defined in upcoming sections.

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

Function parameters can use **const**, **in**, and **out** qualifiers, but as *parameter qualifiers*. Parameter qualifiers are discussed in section 6.1.1 “Function Calling Conventions”.

Function return types and structure 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 don't 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 an error to write to a **const** variable outside of its declaration, so they must be initialized when declared. For example,

```
const vec3 zAxis = vec3 (0.0, 0.0, 1.0);
const float ceiling = a + b; // a and b not necessarily constants
```

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 section 4.3.3 “Constant Expressions.”

4.3.3 Constant Expressions

A *constant expression* is one of

- a literal value (e.g., **5** or **true**)
- a variable declared with the **const** qualifier and an initializer, where the initializer is a constant expression
- 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 (=, +=, ...**)** are not included in the operators that can create a constant expression.

- valid use of the **length**() method on a sized object, whether or not the object itself is constant
- a constructor whose arguments are all constant expressions
- a built-in function call whose arguments are all constant expressions, with the exception of the texture lookup functions and the noise functions. The built-in functions **dFdx**, **dFdy**, and **fwidth** must return 0 when evaluated inside an initializer with an argument that is a constant expression.

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

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

Constant expressions will be evaluated in an invariant way so as to create the same value in multiple shaders when the same constant expressions appear in those shaders. See section 4.8.1 “The Invariant Qualifier” for more details on how to create invariant expressions.

4.3.4 Input Variables

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

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	Error
	Declared but no Static Use	Allowed	Allowed	Allowed (values are undefined)
	Declared and Static Use	Allowed	Allowed	Allowed (values are potentially undefined)

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

Vertex shader input variables (or attributes) receive per-vertex data. They are declared in a vertex shader with the **in** qualifier. It is an error to use any auxiliary 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*. Vertex shader inputs can be single- or double-precision floating-point scalars, vectors, and matrices, or signed- and unsigned-integer scalars and vectors. Vertex shader inputs can also form arrays of these types, but not structures.

Example declarations in a vertex shader:

```
in vec4 position;
in vec3 normal;
in vec2 texCoord[4];
```

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

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,

```
in float foo[];     // geometry shader input for vertex "out float foo"
```

Each element of such an array corresponds to one vertex of the primitive being processed. Each array can optionally have a size declared. 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 section 4.4.1 “Input Layout Qualifiers”.

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 match in type and qualification, except that the vertex shader name cannot be declared as an array while the geometry shader name must be declared as an array, to allow for vertex indexing. It is a link error if a non-arrayed input is not declared with the same type, qualification, and array dimensionality as the matching output. It is an error if an arrayed input is not declared as an array of the same type and qualification as the corresponding (non-array) output. Symmetrically, it is an error if an arrayed output is not declared as an array of the same type and qualification as the corresponding (non-array) input.

If the output corresponding to an arrayed input is itself an array, it must appear in an output block (see interface blocks below) in the outputting shader and in an input block in the inputting shader with a block instance name declared as an array. This is required because two-dimensional arrays are not supported.

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.

Fragment shader inputs get per-fragment values, typically interpolated from a previous stage's outputs. They are declared in fragment shaders with the **in** storage qualifier. The auxiliary storage qualifiers **centroid** and **sample** can also be applied, as well as the interpolation qualifiers **flat**, **noperspective**, and **smooth.** It is an error to use **patch** in a fragment shader. Fragment inputs can only be signed and unsigned integers and integer vectors, floating point scalars, floating-point vectors, matrices, or arrays or structures of these. Fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type must be qualified with the interpolation qualifier **flat**.

Fragment inputs are declared as in the following examples:

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

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 (other than **out** matching to **in**).

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 4.4.1 “Input Layout Qualifiers” and 4.4.2 “Output Layout Qualifiers”) or by using identical input and output declarations of blocks or variables. Complete rules for interface matching between programs are found in the "Shader Interface Matching" portion of section 2.14.PPO of the OpenGL Graphics System Specification.

4.3.5 Uniform

The **uniform** qualifier is used to declare global variables whose values are the same across the entire primitive being processed. All **uniform** variables are read-only and are initialized externally either at link time or through the API. The link time initial value is either the value of the variable's initializer, if present, or 0 if no initializer is present. Opaque types cannot have initializers.

Example declarations are:

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

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

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

If multiple shaders are linked together, then they will share a single global uniform name space, including within a language as well as across languages. Hence, the types and initializers of uniform variables with the same name must match across all shaders that are linked into a single program.

It is legal for some shaders to provide an initializer for a particular uniform variable, while another shader does not, but all provided initializers must be equal.

4.3.6 Output Variables

Shader output variables are declared with a storage qualifier using the keyword **out**. 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 at global scope for declaring a single variable name as both input and output to a shader. A variable also cannot be declared with both the **in** and the **out** qualifiers. 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. Output variables can only be floating-point scalars, floating-point vectors, matrices, signed or unsigned integers or integer vectors, or arrays or structures of any these.

Individual vertex, tessellation evaluation, and geometry outputs are declared as in the following examples:

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

These can also appear in interface blocks, as described in section 4.3.8 “Interface Blocks”. 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 may be used to output per-vertex and per-patch data. Per-vertex output variables are arrayed (see *arrayed* under 4.3.4 Inputs) and declared using the **out** qualifier without the **patch** qualifier. Per-patch output variables are declared using the **patch** and **out** qualifiers. Per-vertex and per-patch output variables can only be floating-point scalars, floating-point vectors, matrices, signed or unsigned integers or integer vectors, or arrays or structures of any these. 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,

```
out float foo[];  // feeds next stage input "in float foo[]"
```

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 section 4.4.2.1 "Tessellation Control Outputs".

As described under the section 4.3.4 "Input Variables" above, if a per-vertex output of the tessellation control shader is itself an array with multiple values per vertex, it must appear in an output block (see interface blocks below) in the tessellation control shader with a block instance name declared as an array.

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 an 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 an error to use auxiliary storage qualifiers or interpolation qualifiers on an output in a fragment shader. Fragment outputs can only be **float**, floating-point vectors, signed or unsigned integers or integer vectors, or arrays of any these. Matrices and structures cannot be output. Fragment outputs are declared as in the following examples:

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

4.3.7 Interface Blocks

Input, output, and uniform 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. It is illegal 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**, or **uniform** keyword, followed by a block name, followed by an open curly brace (**{**) as follows:

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

interface-qualifier :
in
out
uniform

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

member-declaration :
layout-qualifier$_{\textbf{opt}}$ *qualifiers*$_{\textbf{opt}}$ *type declarators* **;**

instance-name :
identifier
identifier [ ]
identifier [integral-constant-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,

```
uniform Transform {
    mat4 ModelViewMatrix;
    mat4 ModelViewProjectionMatrix;
    uniform mat3 NormalMatrix;       // allowed restatement of qualifier
    float Deformation;
};
```

The above establishes a uniform block named “Transform” with four uniforms grouped inside it.

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

- initializers are not allowed
- opaque types are not allowed
- structure definitions cannot be nested inside a block

Otherwise, built-in types, previously declared structures, and arrays of these are allowed as the type of a declarator in the same manner they are allowed outside a block.

If no optional qualifier is used in a member-declaration, the qualification of the variable is just **in**, **out**, or **uniform** 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 variable consistent with the interface qualifier of the block: Input variables, output variables, and uniform variables can only be in **in** blocks, **out** blocks, and **uniform** blocks, respectively. Repeating the **in**, **out**, or **uniform** interface qualifier for a member's storage qualifier is optional. For example,

```
in Material {
    smooth in vec4 Color1; // legal, input inside in block
    smooth vec4 Color2;    // legal, 'in' inherited from 'in Material'
    vec2 TexCoord;         // legal, TexCoord is an input
    uniform float Atten;   // illegal, mismatched  storage qualifier

};
```

For this section, define an *interface* to be one of these

- All the uniforms of a program. This spans all compilation units linked together within one program.
- The boundary between adjacent programmable pipeline stages: This spans all the outputs in all compilation units of the first stage and all the inputs in all compilation units of the second stage.

The block name (*block-name*) is used to match 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 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 an 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). Matched block names within an 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 the same member-wise layout qualification (see next section). Furthermore, if a matching block is declared as an array, then the array sizes must also match (or follow array matching rules for the interface between a vertex and a geometry shader). Any mismatch will generate a link error. A block name is allowed to have different definitions in different 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,

```
in Light {
    vec4 LightPos;
    vec3 LightColor;
};
in ColoredTexture {
    vec4 Color;
    vec2 TexCoord;
} Material;            // instance name
vec3 Color;            // different Color than Material.Color
vec4 LightPos;         // illegal, already defined
...
... = LightPos;        // accessing LightPos
... = Material.Color;  // accessing Color in ColoredTexture block
```

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 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.

```
out Vertex {
    vec4 Position;  // API transform/feedback will use "Vertex.Position"
    vec2 Texture;
} Coords;           // shader will use "Coords.Position"

out Vertex2 {
    vec4 Color;     // API will use "Color"
};
```

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

```
uniform Transform {  // API uses "Transform[2]" to refer to instance 2
    mat4           ModelViewMatrix;
    mat4           ModelViewProjectionMatrix;
    float          Deformation;
} transforms[4];
...
... = transforms[2].ModelViewMatrix;  // shader access of instance 2
// API uses "Transform.ModelViewMatrix" to query an offset or other query
```

For uniform 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 block array declarations must specify an array size. A uniform 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 must 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*).

Geometry shader input blocks must be declared as arrays and follow the array declaration and linking rules for all geometry shader inputs. All other input and output block arrays must specify an array size.

There is an implementation dependent limit on the number of uniform blocks that can be used per stage. If this limit is exceeded, it will cause a link 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 (a storage qualifier that is **in**, **out**, or **uniform**) to establish layouts of other declarations made with that interface 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, 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*

The tokens used for *layout-qualifier-name* are identifiers, not keywords. 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. If the same *layout-qualifier-name* occurs in multiple layout qualifiers for the same declaration, the last one overrides the former ones.

4.4.1 Input Layout Qualifiers

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

All shaders allow input layout qualifiers on input variable declarations. The location layout qualifier identifier for inputs is:

layout-qualifier-id
location = *integer-constant*

Only one argument is accepted. For example,

```
layout(location = 3) in vec4 normal;
```

will establish that the shader input *normal* is assigned to vector location number 3. For vertex shader inputs, the location specifies the number of the generic 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.

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 is an array of size *n* and each element takes *m* locations, it will be assigned *m* * *n* consecutive locations starting with the location specified. For example,

```
layout(location = 6) in vec4 colors[3];
```

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

If the declared input is an *n* x *m* single- or double-precision 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,

```
layout(location = 9) in mat4 transforms[2];
```

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, 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 input variable of the same type. For example,

```
layout(location = 3) struct S {
    vec3 a;
    mat2 b;
    vec4 c[2];
} s;
```

will assign location 3 to *s.a*, locations 4 and 5 to the two column vectors of *s.b*, and locations 6 and 7 to *s.c*.

Location layout qualifiers may be used on input variables declared as structures, but not on individual members. Location layout qualifiers may not be used on input blocks or input block members.

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 any two non-vertex shader input variables are assigned to the same location. For vertex shaders, multiple input variables may be assigned to the same location using either layout qualifiers or via the OpenGL API. However, such aliasing is intended only to support vertex shaders where each execution path accesses at most one input per each location. Implementations are permitted, but not required, to generate link errors if they detect that every path through the vertex shader executable accesses multiple inputs assigned to any single location. 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.

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 2.11.3 “Vertex Attributes” of the OpenGL Graphics System Specification for more details. A link error will occur if an input variable is declared in multiple shaders of the same language with conflicting locations.

4.4.1.1 Tessellation Evaluation Inputs

Additional input layout qualifier identifiers allowed for tessellation evaluation shaders are:

layout-qualifier-id
  triangles
  quads
  isolines
  equal_spacing
  fractional_even_spacing
  fractional_odd_spacing
  cw
  ccw
  point_mode

One subset of these identifiers, *primitive mode*, is used to specify a tessellation primitive mode to be used by the tessellation primitive generator. To specify a primitive mode, the identifier must be one of **triangles**, **quads**, or **isolines**, which specify 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 subset of these identifiers, *vertex spacing*, is used to specify the spacing used by the tessellation primitive generator when subdividing an edge. To specify vertex spacing, the identifier must be one of the following.

equal_spacing signifying that edges should be divided into a collection of equal-sized segments.

fractional_even_spacing signifying that edges should be divided into an even number of equal-length segments plus two additional shorter "fractional" segments.

fractional_odd_spacing signifying that edges should be divided into an odd number of equal-length segments plus two additional shorter "fractional" segments.

A third subset of these 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. The ordering identifiers **cw** and **ccw** indicate clockwise and counter-clockwise triangles, respectively. If the tessellation primitive generator does not produce triangles, ordering is ignored.

Finally, *point mode,* is specified with the identifier **point_mode** indicating the tessellation primitive generator should produce a point for each unique vertex in the subdivided primitive, rather than generating lines or triangles.

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.

4.4.1.2 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 = *integer-constant*

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, it will fail to compile.

For example,

```
layout(triangles, invocations = 6) in;
```

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

```
gl_in.length()
```

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 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:

```
// code sequence within one shader...
in vec4 Color1[];    // legal, size still unknown
...Color1.length()...// illegal, length() unknown
in vec4 Color2[2];   // legal, size is 2
...Color1.length()...// illegal, Color1 still has no size
in vec4 Color3[3];   // illegal, input sizes are inconsistent
layout(lines) in;    // legal for Color2, input size is 2, matching Color2
in vec4 Color4[3];   // illegal, contradicts layout of lines
...Color1.length()...// legal, length() is 2, Color1 sized by layout()
layout(lines) in;    // legal, matches other layout() declaration
layout(triangles) in;// illegal, does not match earlier layout() declaration
```

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.

4.4.1.3 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** identifier, 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

```
in vec4 gl_FragCoord;     // redeclaration that changes nothing is allowed

// All the following are allowed redeclaration that change behavior
layout(origin_upper_left) in vec4 gl_FragCoord;
layout(pixel_center_integer) in vec4 gl_FragCoord;
layout(origin_upper_left, pixel_center_integer) in vec4 gl_FragCoord;
```

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 *gl_FragCoord* 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 will be illegal.

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

Fragment shaders also 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 3.10 of the OpenGL Specification.

For example,

```
layout(early_fragment_tests) in;
```

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.

4.4.2 Output Layout Qualifiers

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

All shaders allow location output layout qualifiers on output variable declarations. The location layout qualifier identifier for outputs is:

layout-qualifier-id
 location = *integer-constant*

Fragment shaders allow an additional **index** output layout qualifiers:

layout-qualifier-id
 location = *integer-constant*
 index = *integer-constant*

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,

```
layout(location = 3) out vec4 color;
```

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,

```
layout(location = 3, index = 1) out vec4 factor;
```

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,

```
layout(location = 2) out vec4 colors[3];
```

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

If the declared output is an *n* x *m* single- or double-precision 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, but not on individual members. Location layout qualifiers may not be used on output blocks or 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. 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 any two fragment shader output variables are assigned to the same location and index, or if any two output variables from the same non-fragment 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 3.9.2 “Shader Execution” of the OpenGL Graphics System Specification for more details. A link 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.

4.4.2.1 Tessellation Control Outputs

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 include the vertex-count layout qualifier:

layout-qualifier-id
  vertices = *integer-constant*

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 an 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

```
gl_out.length()
```

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 must be declared before any use of the method **length**() or other array use requires its size 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.

4.4.2.2 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 = *integer-constant*
  stream = *integer-constant*

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,

```
layout(triangle_strip, max_vertices = 60) out;  // order does not matter
layout(max_vertices = 60) out;      // redeclaration okay
layout(triangle_strip) out;         // redeclaration okay
layout(points) out;                 // error, contradicts triangle_strip
layout(max_vertices = 30) out;      // error, contradicts 60
```

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:

```
layout(stream = 1) out;
```

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:

```
layout(stream=1) out;             // default is now stream 1
out vec4 var1;                    // var1 gets default stream (1)
layout(stream=2) out Block1 {     // "Block1" belongs to stream 2
    layout(stream=2) vec4 var2;   // redundant block member stream decl
    layout(stream=3) vec2 var3;   // ILLEGAL (must match block stream)
    vec3 var4;                    // belongs to stream 2
};
layout(stream=0) out;             // default is now stream 0
out vec4 var5;                    // var5 gets default stream (0)
out Block2 {                      // "Block2" gets default stream (0)
    vec4 var6;
};
layout(stream=3) out vec4 var7;   // var7 belongs to stream 3
```

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.

4.4.2.3 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

For example:

```
layout (depth_greater) out float gl_FragDepth;
```

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). 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:

```
// redeclaration that changes nothing is allowed
out float gl_FragDepth;

// assume it may be modified in any way
layout (depth_any) out float gl_FragDepth;

// assume it may be modified such that its value will only increase
layout (depth_greater) out float gl_FragDepth;

// assume it may be modified such that its value will only decrease
layout (depth_less) out float gl_FragDepth;

// assume it will not be modified
layout (depth_unchanged) out float gl_FragDepth;
```

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 will be illegal.

4.4.3 Uniform Block Layout Qualifiers

Layout qualifiers can be used for uniform blocks, but not for non-block uniform declarations. The layout qualifier identifiers for uniform blocks are

layout-qualifier-id
  shared
  packed
  std140
  row_major
  column_major
  binding = *integer-constant*

None of these have any semantic affect 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 block layout qualifiers can be declared for global scope, on a single uniform block, or on a single block member declaration.

Default layouts are established (except for *binding*) at global scope for uniform blocks as

```
layout(layout-qualifier-id-list) uniform;
```

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 block definitions.

The initial state of compilation is as if the following were declared:

```
layout(shared, column_major) uniform;
```

Explicitly declaring this in a shader will return defaults back to their initial state.

Uniform 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* 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 they also matched in their *row_major* and/or *column_major* qualifications. This allows use of the same buffer to back the same block definition across different programs.

The *packed* qualifier overrides only *std140* 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. Attempts to share a packed uniform block across programs or stages will generally fail. However, implementations may aid application management of packed blocks by using canonical layouts for packed blocks.

The *std140* qualifier overrides only the *packed* and *shared* qualifiers; other qualifiers are inherited. The layout is explicitly determined by this, as described in section 2.11.4 “Uniform Variables” under Standard Uniform Block Layout of the OpenGL Graphics System Specification. Hence, as in *shared* above, the resulting layout is shareable across programs.

Layout qualifiers on member declarations cannot use the *shared*, *packed*, or *std140* qualifiers. These can only be used at global scope or on a block declaration.

The *row_major* qualifier overrides only the *column_major* qualifier; other qualifiers are inherited. It only affects the layout of matrices. Elements within a matrix row will be contiguous in memory.

The *column_major* qualifier overrides only the *row_major* qualifier; other qualifiers are inherited. It only affects the layout of matrices. Elements within a matrix column will be contiguous in memory.

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

If the *binding* identifier is used with a uniform block instanced as an array then the first element of the array takes the specified block binding and each subsequent element takes the next consecutive uniform block binding point.

If the binding point for any uniform block instance is less than zero, or greater than or equal to the implementation-dependent maximum number of uniform buffer bindings, a compilation error will occur. When the *binding* identifier is used with a uniform block instanced as an array of size *N*, all elements of the array from *binding* through *binding + N – 1* must be within this range.

When multiple arguments are listed in a **layout** declaration, the affect 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

```
layout(row_major, column_major)
```

results in the qualification being *column_major*. Other examples:

```
layout(shared, row_major) uniform; // default is now shared and row_major

layout(std140) uniform Transform { // layout of this block is std140
    mat4 M1;                       // row_major
    layout(column_major) mat4 M2;  // column major
    mat3 N1;                       // row_major
};

uniform T2 {  // layout of this block is shared
    ...
};

layout(column_major) uniform T3 {  // shared and column_major
    mat4 M3;                       // column_major
    layout(row_major) mat4 m4;     // row major
    mat3 N2;                       // column_major
};
```

4.4.4 Opaque-Uniform Layout Qualifiers

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

Details for specific to image formats and atomic counter bindings are given in the subsections below.

Image and sampler types both take the uniform layout qualifier identifier for binding:

layout-qualifier-id
    binding = *integer-constant*

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 identifier can be updated by the OpenGL API.

If the *binding* identifier 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 compilation error will occur. When the *binding* identifier is used with an array of size *N*, all elements of the array from *binding* through *binding* + *N* - *1* must be within this range.

A link error will result if two compilation units in a program specify different *integer-constant* 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.

```
// in one compilation unit...
layout(binding=3) uniform sampler2D s; // s bound to unit 3

// in another compilation unit...
uniform sampler2D s;                   // okay, s still bound at 3

// in another compilation unit...
layout(binding=4) uniform sampler2D s; // ERROR: contradictory bindings
```

4.4.4.1 Atomic Counter Layout Qualifiers

The atomic counter qualifiers are

layout-qualifier-id
    binding = *integer-constant*
    offset = *integer-constant*

For example,

```
layout (binding = 2, offset = 4) uniform atomic_uint a;
```

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

```
layout (binding = 2) uniform atomic_uint bar;
```

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,

```
layout (binding = 2, offset = 4) uniform atomic_uint;
```

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

```
layout (binding = 2) uniform atomic_uint bar; // offset is 4
layout (offset = 8) uniform atomic_uint bar;  // error, no 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:

```
layout (binding=3, offset=4) uniform atomic_uint a; // offset = 4
layout (binding=2) uniform atomic_uint b;           // offset = 0
layout (binding=3) uniform atomic_uint c;           // offset = 8
layout (binding=2) uniform atomic_uint d;           // offset = 4
```

Example of an invalid uniform declaration:

```
layout (offset=4) …               // error, must include binding
layout (binding=1, offset=0) … a; // okay
layout (binding=2, offset=0) … b; // okay
layout (binding=1, offset=0) … c; // error, offsets must not be shared
                                  //        between a and c
layout (binding=1, offset=2) … d; // error, overlaps offset 0 of a
```

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

4.4.4.2 Format Layout Qualifiers

Format layout qualifiers can be used on image variable declarations (those declared with a basic type having "**image**" 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 = *integer-constant*

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 "**image**"), signed integer component types (keywords starting with "**iimage**"), or unsigned integer component types (keywords starting with "**uimage**"), 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 an 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 an error to pass an image uniform variable or function parameter declared without a format layout qualifier to an image load or atomic function.

The *binding* identifier was described in section 4.4.4 “Uniform Layout Qualifiers”.

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.

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**. The auxiliary storage qualifier **patch** is not used for interpolation; it is an 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 triangle. This value will come from a single provoking vertex, as described by the OpenGL Graphics System Specification. A variable may be qualified as **flat** can 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 3.6 in the OpenGL Graphics System Specification, section 3.5 “Line Segments”.

A variable qualified as **noperspective** must be interpolated linearly in screen space, as described in equation 3.7 in the OpenGL Graphics System Specification, section 3.5 “Line Segments”.

When multi-sample 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 Graphics System 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 to a location that lies in both the pixel and in the primitive being rendered, including any of the pixel's 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.

The type and presence of interpolation qualifiers of variables with the same name declared in all linked shaders for the same cross-stage interface must match, otherwise the link command will fail.

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

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:

```
gl_FrontColor
gl_BackColor
gl_FrontSecondaryColor
gl_BackSecondaryColor
```

Fragment language:

```
gl_Color
gl_SecondaryColor
```

For example,

```
in vec4 gl_Color;             // predeclared by the fragment language
flat  in vec4 gl_Color;       // redeclared by user to be flat
flat  in vec4 gl_FrontColor;  // input to geometry shader, no "gl_in[]"
flat out vec4 gl_FrontColor;  // output from geometry shader
```

Ideally, these are redeclared as part of the redeclaration of an interface block, as described in section 7.1.1 "Compatibility Profile Built-In Language Variables". However, for the above purpose, they can be redeclared as individual variables at global scope, outside an interface block. A compilation 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 is **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 section 6.1.1 "Function Calling Conventions".

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).

The following rules apply to both single and double-precision operations: Infinities and zeros are generated as dictated by IEEE, but subject to the precisions allowed in the following table and subject to allowing positive and negative zeros to be interchanged. However, dividing a non-zero by 0 results in the appropriately signed IEEE Inf: If both positive and negative zeros are implemented, the correctly signed Inf will be generated, otherwise positive Inf is generated. Any 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. NaNs are not required to be generated. Support for signaling NaNs is not required and exceptions are never raised. Operations and built-in functions that operate on a NaN are not required to return a NaN as the result.

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** (x * **log2** (y)).
exp (x), **exp2** (x)	$(3 + 2 * \|x\|)$ 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 floating point or any integer declaration can have the type preceded by one of these precision qualifiers:

Qualifier	**Meaning**
highp	None.
mediump	None.
lowp	None.

For example:

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

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

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

4.7.3 Default Precision Qualifiers

The precision statement

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

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

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

The vertex, tessellation, and geometry languages have the following predeclared globally scoped default precision statements:

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

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

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

4.7.4 Available Precision Qualifiers

The built-in macro GL_FRAGMENT_PRECISION_HIGH is defined to 1:

```
#define GL_FRAGMENT_PRECISION_HIGH 1
```

This macro is available in the vertex, tessellation, geometry, and fragment languages.

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

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

4.8.1 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

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

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

or as part of a declaration when a variable is declared

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

Only variables output from a shader (including those that are then input to a subsequent shader) can be candidates for invariance. This includes user-defined output variables and the built-in output variables. As only outputs need be declared with **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 the global scope, and before any use of the variables being declared as invariant.

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

- The output variable is declared as invariant in both programs.
- The same values must be input to all shader input variables consumed by expressions and flow control contributing to the value assigned to the output variable.

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

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

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

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

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

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

4.8.2 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 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 "multiply-add" instruction that can compute a floating-point expression such as

```
result = (a * b) + (c * d);
```

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 may 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. Without any qualifiers, implementations are permitted to perform such 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 qualifier **precise** will ensure that operations contributing to a variable's value are done in their stated order and are done with operator consistency. Order is determined by operator precedence and parenthesis, as described in section 5.1 “Operators”. Operator consistency means for each particular operator, for example the multiply operator (*****), its operation is always computed with the same precision. Note this effectively prevents fusing multiple operations into a single operation.

For example,

```
precise out vec4 position;
```

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 (section 4.8.1), the precise qualifier may be used to qualify a built-in or previously declared user-defined variable as being precise:

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

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 an variable qualified as **precise**.

Some examples of the use of **precise**:

```
in vec4 a, b, c, d;
precise out vec4 v;

float func(float e, float f, float g, float h)
{
    return (e*f) + (g*h);             // no constraint on order or
                                      // operator consistency
}

float func2(float e, float f, float g, float h)
{
    precise float result = (e*f) + (g*h);  // ensures same precision for
                                           // the two multiplies
    return result;
}

float func3(float i, float j, precise out float k)
{
    k = i * i + j;                    // precise, due to <k> declaration
}

void main()
{
    vec4 r = vec3(a * b);           // precise, used to compute v.xyz
    vec4 s = vec3(c * d);           // precise, used to compute v.xyz
    v.xyz = r + s;                          // precise
    v.w = (a.w * b.w) + (c.w * d.w);        // precise
    v.x = func(a.x, b.x, c.x, d.x);         // values computed in func()
                                            // are NOT precise
    v.x = func2(a.x, b.x, c.x, d.x);        // precise!
    func3(a.x * b.x, c.x * d.x, v.x);       // 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.

4.10 Memory Qualifiers

Only variables declared as image types (the basic opaque types with “**image**” in their keyword) can be qualified with a memory qualifier.

Variables declared as image types can 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
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 similar accesses 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 the Section 2.11.13 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 an 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 an error to pass an image variable qualified with **writeonly** to **imageLoad**() or other built-in functions that read image memory.

The values of image variables qualified with **coherent**, **volatile**, **restrict**, **readonly**, or **writeonly** may not be passed to functions whose formal parameters lack such qualifiers. (See section 6.1 “Function Definitions” for more detail on function calling.) It is legal to have additional qualifiers on a formal parameter, but not to have fewer.

```
vec4 funcA(restrict image2D a)   { ... }
vec4 funcB(image2D a)            { ... }
layout(rgba32f) uniform image2D img1;
layout(rgba32f) coherent uniform image2D img2;

funcA(img1);              // OK, adding "restrict" is allowed
funcB(img2);              // illegal, stripping "coherent" is not
```

Layout qualifiers cannot be used on formal function parameters, but they are not included in parameter matching.

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

4.11 Order 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. Multiple memory qualifiers can be used. Any violation of these rules will cause a compile-time error.

5 Operators and Expressions

5.1 Operators

The OpenGL Shading Language has the following operators.

Precedence	Operator Class	Operators	Associativity
1 (highest)	parenthetical grouping	( )	NA
2	array subscript function call and constructor structure field or method selector, 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 section 5.7 “Structure and Array Operations”.

5.3 Function Calls

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

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

5.4 Constructors

Constructors use the function call syntax, where the function name is a type, and the call makes an object of that type. Constructors are used the same way in both initializers and expressions. (See section 9 “Shading Language Grammar” for details.) The parameters are used to initialize the constructed value. Constructors can be used to request a data type conversion to change from one scalar type to another scalar type, or to build larger types out of smaller types, or to reduce a larger type to a smaller type.

In general, constructors are not built-in functions with predetermined prototypes. For arrays and structures, there must be exactly one argument in the constructor for each element or member. For the other types, the arguments must provide a sufficient number of components to perform the initialization, and it is an error to include so many arguments that they cannot all be used. Detailed rules follow. The prototypes actually listed below are merely a subset of examples.

5.4.1 Conversion and Scalar Constructors

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

```
int(uint)    // converts an unsigned integer to a signed integer
int(bool)    // converts a Boolean value to an int
int(float)   // converts a float value to an int
int(double)  // converts a double value to a signed integer
uint(int)    // converts a signed integer value to an unsigned integer
uint(bool)   // converts a Boolean value to an unsigned integer
uint(float)  // converts a float value to an unsigned integer
uint(double) // converts a double value to an unsigned integer
bool(int)    // converts a signed integer value to a Boolean
bool(uint)   // converts an unsigned integer value to a Boolean value
bool(float)  // converts a float value to a Boolean
bool(double) // converts a double value to a Boolean
float(int)   // converts a signed integer value to a float
float(uint)  // converts an unsigned integer value to a float value
float(bool)  // converts a Boolean value to a float
float(double)// converts a double value to a float
double(int)  // converts a signed integer value to a double
double(uint) // converts an unsigned integer value to a double
double(bool) // converts a Boolean value to a double
double(float)// converts a float value to a double
```

When constructors are used to convert any 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 an error to provide extra arguments beyond this last used argument.

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

If the basic type (**bool, int, 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:

```
vec3(float)  // initializes each component of the vec3 with the float
vec4(ivec4)  // makes a vec4 with component-wise conversion
vec4(mat2)   // the vec4 is column 0 followed by column 1

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

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

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

Some examples of these are:

```
vec4 color = vec4(0.0, 1.0, 0.0, 1.0);
vec4 rgba  = vec4(1.0);   // sets each component to 1.0
vec3 rgb   = vec3(color); // drop the 4th component
```

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

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

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

To initialize a matrix by specifying vectors or scalars, the components are assigned to the matrix elements in column-major order.

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

dmat2(dvec2, dvec2);
dmat3(dvec3, dvec3, dvec3);
dmat4(dvec4, dvec4, dvec4, dvec4);

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

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

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

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

dmat2x4(dvec3, double,   // first column
        double, dvec3)   // second column
```

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

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

5.4.3 Structure Constructors

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

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

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

The arguments to the constructor will be used to set the structure's 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 4.1.10 “Implicit Conversions.”

Structure constructors can be used as initializers or in expressions.

5.4.4 Array Constructors

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

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

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

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

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

5.5 Vector 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 an error so, for example:

```
vec2 pos;
float height;
pos.x      // is legal
pos.z      // is illegal
height.x  // is legal
height.y  // is illegal
```

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

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

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

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz= pos.wzyx; // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy; // dup = (1.0, 1.0, 2.0, 2.0)
float f = 1.2;
vec4 dup = f.xxxx;   // dup = (1.2, 1.2, 1.2, 1.2)
```

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. Any resulting vector of any operation must be a valid vector in the language; hence the following is illegal:

```
vec4 f;
vec4 g = pos.xyzwxy.xyzw;  // illegal; pos.xyzwxy is non-existent “vec6”
```

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

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
pos.xw = vec2(5.0, 6.0);         // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0);         // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0);         // illegal - 'x' used twice
pos.xy = vec3(1.0, 2.0, 3.0);    // illegal - mismatch between vec2 and vec3
```

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

Array subscripting syntax can also be applied to vectors (but not to scalars) to provide numeric indexing. So in

```
vec4   pos;
```

pos[2] refers to the third element of pos and is equivalent to *pos.z*. This allows variable indexing into a vector, as well as a generic way of accessing components. Any integer expression can be used as the subscript. The first component is at index zero. Reading from or writing to a vector using a constant integral expression with a value that is negative or greater than or equal to the size of the vector is illegal. When indexing with non-constant expressions, behavior is undefined if the index is negative, or greater than or equal to the size of the vector.

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

```
vec3 v;
const int L = v.length();
```

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

5.6 Matrix Components

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

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

Behavior is undefined when accessing a component outside the bounds of a matrix with a non-constant expression. It is an error to access a matrix with a constant expression that is outside the bounds of the matrix.

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

```
mat3x4 v;
const int L = v.length();
```

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

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	=
indexing (arrays only)	[ ]

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

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

Array indices start at zero. Array elements are accessed using an expression whose type is **int** or **uint**.

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

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

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

5.8 Assignments

Assignments of values to variable names are done with the assignment operator (=):

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

The *lvalue-expression* evaluates to an l-value. The assignment operator stores the value of *rvalue-expression* into the l-value and returns an r-value with the type and precision of *lvalue-expression*. The *lvalue-expression* and *rvalue-expression* must have the same type, or the expression must have a type in the table in section 4.1.10 “Implicit Conversions” that converts to the type of *lvalue-expression*, in which case an implicit conversion will be done on the *rvalue-expression* before the assignment is done. Any other desired type-conversions must be specified explicitly via a constructor. L-values must be writable. Variables that are built-in types, entire structures or arrays, structure 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.

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

The other assignment operators are

- add into (+=)
- subtract from (**-**=)
- multiply into (*****=)
- divide into (/=)
- modulus into (**%**=)
- left shift by (<<=)

- right shift by (>>=)
- and into (**&**=)
- inclusive-or into (|=)
- exclusive-or into (^=)

where the general expression

```
lvalue op= expression
```

is equivalent to

```
lvalue = lvalue op expression
```

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

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

5.9 Expressions

Expressions in the shading language are built from the following:

- Constants of type **bool,** all integer types, all floating-point types, all vector types, and all matrix types.
- Constructors of all types.
- Variable names of all types.
- An array name with the length method applied.
- Subscripted array names.
- Function calls that return values.
- Component field selectors and array subscript results.
- Parenthesized expression. Any expression can be parenthesized. Parentheses can be used to group operations. Operations within parentheses are done before operations across parentheses.
- The arithmetic binary operators add (+), subtract (**-**), multiply (*****), and divide (/) operate on integer and floating-point scalars, vectors, and matrices. If the fundamental types in the operands do not match, then the conversions from section 4.1.10 “Implicit Conversions” 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. Section 5.10 "Vector and Matrix Operations" explains in more detail how vectors and matrices are operated on.

  All other cases are illegal.

  Dividing by zero does not cause an exception but does result in an unspecified value. Use the built-in functions **dot**, **cross**, **matrixCompMult**, and **outerProduct**, to get, respectively, vector dot product, vector cross product, matrix component-wise multiplication, and the matrix product of a column vector times a row vector.

- The operator modulus (**%**) operates on signed or unsigned integer scalars or integer vectors. If the fundamental types in the operands do not match, then the conversions from section 4.1.10 "Implicit Conversions" are applied to create matching types. The operands cannot be vectors of differing size. If one operand is a scalar and the other vector, then the scalar is applied component-wise to the vector, resulting in the same type as the vector. If both are vectors of the same size, the result is computed component-wise. The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero second operands remain defined. If both operands are non-negative, then the remainder is non-negative. Results are undefined if one or both operands are negative. The operator modulus (**%**) is not defined for any other data types (non-integer types).
- The arithmetic unary operators negate (-), post- and pre-increment and decrement (**--** and ++) operate on integer or floating-point values (including vectors and matrices). All unary operators work component-wise on their operands. These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract 1 or 1.0 to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 or 1.0 to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.
- The relational operators greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate only on scalar integer and scalar floating-point expressions. The result is scalar Boolean. Either the operands' types must match, or the conversions from section 4.1.10 "Implicit Conversions" will be applied to 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. They result in a scalar Boolean. If the operand types do not match, then there must be a conversion from section 4.1.10 “Implicit Conversions” applied to one operand that can make them match, in which case this conversion is done. For vectors, matrices, structures, and arrays, all components, 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 ternary selection operator (**?:**). It operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression, which must result in a scalar Boolean. If the result is true, it selects to evaluate the second expression, otherwise it selects to evaluate the third expression. Only one of the second and third expressions is evaluated. The second and third expressions can be any type, as long their types match, or there is a conversion in section 4.1.10 “Implicit Conversions” that can be applied to one of the expressions to make their types match. This resulting matching type is the type of the entire expression.
- The one's complement operator (~). The operand must be of type signed or unsigned integer or integer vector, and the result is the one's complement of its operand; each bit of each component is complemented, including any sign bits.
- The shift operators (<<) and (>>). For both operators, the operands must be signed or unsigned integers or integer vectors. One operand can be signed while the other is unsigned. In all cases, the resulting type will be the same type as the left operand. If the first operand is a scalar, the second operand has to be a scalar as well. If the first operand is a vector, the second operand must be a scalar or a vector, and the result is computed component-wise. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's base type. The value of E1 << E2 is E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1 >> E2 is E1 right-shifted by E2 bit positions. If E1 is a signed integer, the right-shift will extend the sign bit. If E1 is an unsigned integer, the right-shift will zero-extend.
- The bitwise operators and (**&**), exclusive-or (**^**), and inclusive-or (**|**). The operands must be of type signed or unsigned integers or integer vectors. The operands cannot be vectors of differing size. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The fundamental types of the operands (signed or unsigned) must match, and will be the resulting fundamental type. For and (**&**), the result is the bitwise-and function of the operands. For exclusive-or (**^**), the result is the bitwise exclusive-or function of the operands. For inclusive-or (**|**), the result is the bitwise inclusive-or function of the operands.

For a complete specification of the syntax of expressions, see section 9 “Shading Language Grammar.”

5.10 Vector and Matrix Operations

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

```
vec3 v, u;
float f;

v = u + f;
```

will be equivalent to

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

And

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

will be equivalent to

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

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

```
vec3 v, u;
mat3 m;

u = v * m;
```

is equivalent to

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

And

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

is equivalent to

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

And

```
mat3 m, n, r;

r = m * n;
```

is equivalent to

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

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

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

and similarly for other sizes of vectors and matrices.

6 Statements and Structure

The fundamental building blocks of the OpenGL Shading Language are:

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

The overall structure of a shader is as follows

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

global-declaration:
    function-definition
    declaration

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

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

statement-list:
    statement
    statement-list statement

statement:
    compound-statement
    simple-statement

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

compound-statement:
    { statement-list }

simple-statement:
    declaration-statement
    expression-statement
    selection-statement

iteration-statement
jump-statement

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

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

Declarations and expressions have already been discussed.

6.1 Function Definitions

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

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

and a function is defined like

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

where *returnType* must be present and include a type. If the type of *returnValue* does not match *returnType,* there must be an implicit conversion in section 4.1.10 “Implicit Conversions” that converts the type of *returnValue* to *returnType,* or a compile 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 function's declaration.

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

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

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

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

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

```
void func1() { }
void func2() { return func1(); } // illegal return statement
```

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 "( )" is declared. The idiom "(**void**)" as a parameter list is provided for convenience.

Function names can be overloaded. The same function name can be used for multiple functions, as long as the parameter types differ. If a function name is declared twice with the same parameter types, then the return types and all qualifiers must also match, and it is the same function being declared. For example,

```
vec4 f(in vec4 x, out vec4  y);   // (A)
vec4 f(in vec4 x, out uvec4 y);   // (B) okay, different argument type
vec4 f(in ivec4 x, out uvec4 y);  // (C) okay, different argument type

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

When function calls are resolved, an exact type match for all the arguments is sought. If an exact match is found, all other functions are ignored, and the exact match is used. If no exact match is found, then the implicit conversions in section 4.1.10 "Implicit Conversions" will be applied to find a match. Mismatched types on input parameters (**in** or **inout** or default**)** must have a conversion from the calling argument type to the formal parameter type. Mismatched types on output parameters (**out** or **inout**) must have a conversion from the formal parameter type to the calling argument type.

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 semantic error for an ambiguous overloaded function call occurs and the shader will fail to compile.

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 (C) above, the following examples show how the rules apply to different sets of calling argument types:

```
f(vec4, vec4);        // exact match of vec4 f(in vec4 x, out vec4 y)
f(vec4, uvec4);       // exact match of vec4 f(in vec4 x, out ivec4 y)
f(vec4, ivec4);       // matched to vec4 f(in vec4 x, out vec4 y)
                      //   (C) not relevant, can't convert vec4 to
                      //   ivec4.  (A) better than (B) for 2nd
                      //   argument (rule 2), same on first argument.
f(ivec4, vec4);       // NOT matched.  All three match by implicit
                      //   conversion.  (C) is better than (A) and (B)
                      //   on the first argument.  (A) is better than
                      //   (B) and (C).
```

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

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

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

The function *main* can contain uses of **return**. See section 6.4 “Jumps” for more details.

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

6.1.1 Function Calling Conventions

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

- The keyword **in** is used as a qualifier to denote a parameter is to be copied in, but not copied out.
- The keyword **out** is used as a qualifier to denote a parameter is to be copied out, but not copied in. This should be used whenever possible to avoid unnecessarily copying parameters in.

- The keyword **inout** is used as a qualifier to denote the parameter is to be both copied in and copied out. 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 function's copy is modified. This can be prevented by declaring a parameter with the **const** qualifier.

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

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

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

parameter-qualifiers :
empty
list of *parameter-qualifier*

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**. The above is used for function declarations (i.e., prototypes) and for function definitions. Hence, function definitions can have unnamed arguments.

Recursion is not allowed, not even statically. Static recursion is present if the static function-call graph of a program contains cycles. This includes all potential function calls through variables declared as **subroutine uniform** (described below). It is an 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 types are declared using a statement similar to a function declaration, with the **subroutine** keyword, as follows:

```
subroutine returnType subroutineTypeName(type0 arg0, type1 arg1,
                                         ..., typen argn);
```

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:

```
subroutine(subroutineTypeName0, ..., subroutineTypeNameN)
returnType functionName(type0 arg0, type1 arg1, ..., typen argn)
{ ... } // function body
```

It is an 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.

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 subroutineTypeName subroutineVarName;
```

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 dynamically indexed at use.

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

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

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

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

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

Conditionals can be nested.

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

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

6.3 Iteration

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

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

while (condition-expression)
    sub-statement

do
    statement
while (condition-expression)
```

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

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

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

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

```
for (int i = 0; i < 10; i++) {
    int i;  // redeclaration error
}
```

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

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

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

Loops can be nested.

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

6.4 Jumps

These are the jumps:

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

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

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

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

The **discard** keyword is only allowed within fragment shaders. It can be used within a fragment shader to abandon the operation on the current fragment. This keyword causes the fragment to be discarded and no updates to any buffers will occur. Control flow exits the shader, and subsequent implicit or explicit derivatives are undefined when this exit is non-uniform. It would typically be used within a conditional statement, for example:

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

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

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

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

7 Built-in Variables

7.1 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.

In the vertex language, the built-ins are intrinsically declared as:

```
in  int   gl_VertexID;
in  int   gl_InstanceID;

out gl_PerVertex {
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```

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

```
in gl_PerVertex {
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];

in int gl_PrimitiveIDIn;
in int gl_InvocationID;

out gl_PerVertex {
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};

out int gl_PrimitiveID;
out int gl_Layer;
out int gl_ViewportIndex;
```

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

```
in gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];

in int gl_PatchVerticesIn;
in int gl_PrimitiveID;
in int gl_InvocationID;

out gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_out[];

patch out float gl_TessLevelOuter[4];
patch out float gl_TessLevelInner[2];
```

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

```
in gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];

in int gl_PatchVerticesIn;
```

```
in int gl_PrimitiveID;
in vec3 gl_TessCoord;
patch in float gl_TessLevelOuter[4];
patch in float gl_TessLevelInner[2];

out gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```

In the fragment language, built-in variables are intrinsically declared as:

```
in  vec4  gl_FragCoord;
in  bool  gl_FrontFacing;
in  float gl_ClipDistance[];
in  vec2  gl_PointCoord;
in  int   gl_PrimitiveID;
in  int   gl_SampleID;
in  vec2  gl_SamplePosition;
in  int   gl_SampleMaskIn[];


out float gl_FragDepth;
out int   gl_SampleMask[];
```

Each of the above variables is discussed below.

The variable *gl_VertexID* is a vertex language input variable that holds an integer index for the vertex, as defined under “Shader Inputs” in section 2.11.7 “Varying Variables” in the OpenGL Graphics System Specification. While the variable *gl_VertexID* is always present, its value is not always defined.

The variable *gl_InstanceID* is a vertex language input variable that holds the instance number of the current primitive in an instanced draw call (see “Shader Inputs” in section 2.11.7 “Varying Variables” in the OpenGL Graphics System Specification). If the current primitive does not come from an instanced draw call, the value of *gl_InstanceID* is zero.

As an output 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*, and it is undefined after geometry processing if the geometry executable calls **EmitVertex**() without having written *gl_Position* since the last **EmitVertex**() (or hasn't written it at all). As an input variable, *gl_Position* reads the output written in the previous shader stage to *gl_Position*.

As an output 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. As an input variable, *gl_PointSize* reads the output written in the previous shader stage to *gl_PointSize* .

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

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

The output variable *gl_PrimitiveID* is available only in the geometry language and provides a single integer that serves as a primitive identifier. This is then available to fragment shaders as the fragment input *gl_PrimitiveID*, which will select the written primitive ID from the provoking vertex in 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 2.13.4 “Geometry Shader Execution Environment” (under “Geometry Shader Outputs”) and section 3.9.2 “Shader Execution” (under “Shader Inputs”) of the OpenGL Graphics System Specification for more information.

For tessellation control and evaluation languages the input variable *gl_PrimitiveID* is filled with the number of primitives processed by the shader since the current set of rendering primitives was started. For the fragment language, it is filled with the value written to the *gl_PrimitiveID* geometry shader output if a geometry shader is present. Otherwise, it is assigned in the same manner as with tessellation control and evaluation shaders.

The geometry language input variable *gl_PrimitiveIDIn* behaves identically to the tessellation control and evaluation language input variable *gl_PrimitiveID*.

The input variable *gl_InvocationID* is available only in the tessellation control and geometry languages. In the tessellation control shader, it identifies the number of the output patch vertex assigned to the tessellation control shader invocation. In the geometry shader, it identifies the invocation number assigned to the geometry shader invocation. In both cases, *gl_InvocationID* is assigned integer values in the range *[0, N-1]*, where *N* is the number of output patch vertices or geometry shader invocations per primitive.

The output variable *gl_Layer* is available only in the geometry language, and 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 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 2.13.4 “Geometry Shader Execution Environment” (under “Geometry Shader Outputs”) and section 4.4.7 “Layered Framebuffers” of the OpenGL Graphics System 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 4.12 of section 4.4.7 “Layered Framebuffers” of the OpenGL Graphics System 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* is available only in the geometry language and 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 2.13, under "Geometry Shader Outputs" of the OpenGL Graphics System Specification (Core Profile) for more information.

The variable *gl_PatchVerticesIn* is available only in the tessellation control and evaluation languages. It is an integer specifying the number of vertices in the input patch being processed by the shader. A single tessellation control or evaluation shader can read patches of differing sizes, so the value of *gl_PatchVerticesIn* may differ between patches.

The output variables *gl_TessLevelOuter[]* and *gl_TessLevelInner[]* are available only in the tessellation control language. The values written to these variables 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.

The variable *gl_TessCoord* is available only in the tessellation evaluation language. It 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.

The input variables *gl_TessLevelOuter[]* and *gl_TessLevelInner[]* are available only in the tessellation evaluation shader. If a tessellation control shader is active, these variables are filled with corresponding outputs written by the tessellation control shader. Otherwise, they are assigned with default tessellation levels specified in section 2.12 "Tessellation" in the OpenGL Graphics System Specification.

Fragment shaders output values to the OpenGL pipeline using declared **out** variables, the built-in variables *gl_FragDepth* and *gl_SampleMask*, unless the **discard** statement is executed.

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 the same size described above. 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 multi-sample 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 *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:

```
out gl_PerVertex {
    vec4 gl_Position;    // will use gl_Position
    float gl_PointSize;  // will use gl_PointSize
    vec4 t;              // error, only gl_PerVertex members allowed
};  // no other members of gl_PerVertex will be used
```

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*.

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 compilation error will result. It is also a compilation 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 section 4.3.8 “Interface Blocks” for interface block matching, or a link error will result. It will also be a link 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.

7.1.1 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:

```
out gl_PerVertex {       // part of the gl_PerVertex block described in 7.1
    // in addition to other gl_PerVertex members...
    vec4  gl_ClipVertex;
    vec4  gl_FrontColor;
    vec4  gl_BackColor;
    vec4  gl_FrontSecondaryColor;
    vec4  gl_BackSecondaryColor;
    vec4  gl_TexCoord[];
    float gl_FogFragCoord;
};
```

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 an error for the set of shaders forming a program to statically read or write both *gl_ClipVertex* and *gl_ClipDistance.* 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:

```
out gl_PerVertex {
    vec4 gl_Position;    // will use gl_Position
    vec4 gl_FrontColor;  // will consume gl_color in the fragment shader
    vec4 gl_BackColor;
    vec4 gl_TexCoord[3]; // 3 elements of gl_TexCoord will be used
};  // no other aspects of the fixed interface will be used
```

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 “c” value in section 3.11 “Fog” of the compatibility profile of the OpenGL Graphics System Specification, by the fixed functionality pipeline. For example, if the z-coordinate of the fragment in eye space is desired as “c”, then that's what the vertex shader executable should write into *gl_FogFragCoord*.

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

```
in vec4 gl_TexCoord[3];
out vec4 gl_TexCoord[4];
```

(This treatment is a special case for *gl_TexCoord[]*, not a general method for redeclaring members of blocks.) It is a compilation 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.

```
in gl_PerVertex {        // part of the gl_PerVertex block described in 7.1
    // in addition to other gl_PerVertex members...
    vec4  gl_ClipVertex;
    vec4  gl_FrontColor;
    vec4  gl_BackColor;
    vec4  gl_FrontSecondaryColor;
    vec4  gl_BackSecondaryColor;
    vec4  gl_TexCoord[];
    float gl_FogFragCoord;
} gl_in[];
```

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 an error to not include the built-in instance name or to change its name. For example,

```
in gl_PerVertex {
    vec4  gl_ClipVertex;
    vec4  gl_FrontColor;
} gl_in[];  // must be present and must be "gl_in[]"
```

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:

```
in gl_PerFragment {
    in float gl_FogFragCoord;
    in vec4  gl_TexCoord[];
    in vec4  gl_Color;
    in vec4  gl_SecondaryColor;
};
```

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 3.11 "Fog" of the compatibility profile of the OpenGL Graphics System 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 2.14 "Vertex Shaders" of the OpenGL Graphics System Specification (Compatibility Profile). If they don't match within a program, a link 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:

```
out vec4 gl_FragColor;
out vec4 gl_FragData[gl_MaxDrawBuffers];
```

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 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.

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

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.

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

const int  gl_MaxVertexAttribs = 16;
const int  gl_MaxVertexUniformComponents = 1024;

const int  gl_MaxVaryingComponents = 60;
const int  gl_MaxVertexOutputComponents = 64;
const int  gl_MaxGeometryInputComponents = 64;
const int  gl_MaxGeometryOutputComponents = 128;
const int  gl_MaxFragmentInputComponents = 128;
const int  gl_MaxVertexTextureImageUnits = 16;
const int  gl_MaxCombinedTextureImageUnits = 80;
const int  gl_MaxTextureImageUnits = 16;
```

```
const int  gl_MaxImageUnits = 8;
const int  gl_MaxCombinedImageUnitsAndFragmentOutputs = 8;
const int  gl_MaxImageSamples = 0;
const int  gl_MaxVertexImageUniforms = 0;
const int  gl_MaxTessControlImageUniforms = 0;
const int  gl_MaxTessEvaluationImageUniforms = 0;
const int  gl_MaxGeometryImageUniforms = 0;
const int  gl_MaxFragmentImageUniforms = 0;
const int  gl_MaxCombinedImageUniforms = 0;
const int  gl_MaxFragmentUniformComponents = 1024;
const int  gl_MaxDrawBuffers = 8;
const int  gl_MaxClipDistances = 8;
const int  gl_MaxGeometryTextureImageUnits = 16;
const int  gl_MaxGeometryOutputVertices = 256;
const int  gl_MaxGeometryTotalOutputComponents = 1024;
const int  gl_MaxGeometryUniformComponents = 1024;
const int  gl_MaxGeometryVaryingComponents = 64;
```

```
const int gl_MaxTessControlInputComponents = 128;
const int gl_MaxTessControlOutputComponents = 128;
const int gl_MaxTessControlTextureImageUnits = 16;
const int gl_MaxTessControlUniformComponents = 1024;
const int gl_MaxTessControlTotalOutputComponents = 4096;

const int gl_MaxTessEvaluationInputComponents = 128;
const int gl_MaxTessEvaluationOutputComponents = 128;
const int gl_MaxTessEvaluationTextureImageUnits = 16;
const int gl_MaxTessEvaluationUniformComponents = 1024;

const int gl_MaxTessPatchComponents = 120;
const int gl_MaxPatchVertices = 32;
const int gl_MaxTessGenLevel = 64;

const int gl_MaxViewports = 16;

const int gl_MaxVertexUniformVectors = 256;
const int gl_MaxFragmentUniformVectors = 256;
const int gl_MaxVaryingVectors = 15;

const int gl_MaxVertexAtomicCounters = 0;
const int gl_MaxTessControlAtomicCounters = 0;
const int gl_MaxTessEvaluationAtomicCounters = 0;
const int gl_MaxGeometryAtomicCounters = 0;
const int gl_MaxFragmentAtomicCounters = 8;
const int gl_MaxCombinedAtomicCounters = 8;
const int gl_MaxAtomicCounterBindings = 1;
const int gl_MaxVertexAtomicCounterBuffers = 0;
const int gl_MaxTessControlAtomicCounterBuffers = 0;
const int gl_MaxTessEvaluationAtomicCounterBuffers = 0;
const int gl_MaxGeometryAtomicCounterBuffers = 0;
const int gl_MaxFragmentAtomicCounterBuffers = 1;
const int gl_MaxCombinedAtomicCounterBuffers = 1;
const int gl_MaxAtomicCounterBufferSize = 16384;

const int gl_MinProgramTexelOffset = -8;
const int gl_MaxProgramTexelOffset = 7;
```

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

7.3.1 Compatibility Profile Built-In Constants

```
const   int  gl_MaxTextureUnits = 2;
const   int  gl_MaxTextureCoords = 8;
const   int  gl_MaxClipPlanes = 8;
const   int  gl_MaxVaryingFloats = 60;
```

7.4 Built-In Uniform State

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

```
//
// Depth range in window coordinates,
// section 2.14.1  “Controlling the Viewport” in the
// OpenGL Graphics System Specification.
//
struct gl_DepthRangeParameters {
    float near;        // n
    float far;         // f
    float diff;        // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;

uniform int gl_NumSamples;
```

7.4.1 Compatibility Profile State

These variables are present only in the compatibility profile.

```
//
// compatibility profile only
//
uniform mat4  gl_ModelViewMatrix;
uniform mat4  gl_ProjectionMatrix;
uniform mat4  gl_ModelViewProjectionMatrix;
uniform mat4  gl_TextureMatrix[gl_MaxTextureCoords];

//
// compatibility profile only
//
uniform mat3  gl_NormalMatrix; // transpose of the inverse of the
                               // upper leftmost 3x3 of gl_ModelViewMatrix

uniform mat4  gl_ModelViewMatrixInverse;
uniform mat4  gl_ProjectionMatrixInverse;
uniform mat4  gl_ModelViewProjectionMatrixInverse;
uniform mat4  gl_TextureMatrixInverse[gl_MaxTextureCoords];

uniform mat4  gl_ModelViewMatrixTranspose;
uniform mat4  gl_ProjectionMatrixTranspose;
uniform mat4  gl_ModelViewProjectionMatrixTranspose;
uniform mat4  gl_TextureMatrixTranspose[gl_MaxTextureCoords];

uniform mat4  gl_ModelViewMatrixInverseTranspose;
uniform mat4  gl_ProjectionMatrixInverseTranspose;
```

```
uniform mat4  gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4  gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];

//
// compatibility profile only
//
uniform float gl_NormalScale;

//
// compatibility profile only
//
uniform vec4  gl_ClipPlane[gl_MaxClipPlanes];

//
// compatibility profile only
//
struct gl_PointParameters {
    float size;
    float sizeMin;
    float sizeMax;
    float fadeThresholdSize;
    float distanceConstantAttenuation;
    float distanceLinearAttenuation;
    float distanceQuadraticAttenuation;
};

uniform gl_PointParameters gl_Point;

//
// compatibility profile only
//
struct gl_MaterialParameters {
    vec4  emission;    // Ecm
    vec4  ambient;     // Acm
    vec4  diffuse;     // Dcm
    vec4  specular;    // Scm
    float shininess;   // Srm
};
uniform gl_MaterialParameters  gl_FrontMaterial;
uniform gl_MaterialParameters  gl_BackMaterial;
```

```
//
// compatibility profile only
//

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

uniform gl_LightSourceParameters  gl_LightSource[gl_MaxLights];

struct gl_LightModelParameters {
    vec4  ambient;       // Acs
};

uniform gl_LightModelParameters  gl_LightModel;

//
// compatibility profile only
//
// Derived state from products of light and material.
//

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

uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

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

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

```
//
// compatibility profile only
//
uniform vec4  gl_TextureEnvColor[gl_MaxTextureUnits];
uniform vec4  gl_EyePlaneS[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneT[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneR[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneQ[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneS[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneT[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneR[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneQ[gl_MaxTextureCoords];

//
// compatibility profile only
//
struct gl_FogParameters {
    vec4 color;
    float density;
    float start;
    float end;
    float scale;   // Derived:   1.0 / (end - start)
};

uniform gl_FogParameters gl_Fog;
```

8 Built-in Functions

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

The built-in functions basically fall into three categories:

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

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

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

User code can replace built-in functions with their own if they choose, by simply re-declaring and defining the same name and argument list. Because built-in functions are in a more outer scope than user built-in functions, doing this will hide all built-in functions with the same name as the re-declared function.

When the built-in functions are specified below, where the input arguments (and corresponding output) can be **float**, **vec2**, **vec3**, or **vec4**, *genType* is used as the argument. Where the input arguments (and corresponding output) can be **int**, **ivec2**, **ivec3**, or **ivec4**, *genIType* is used as the argument. Where the input arguments (and corresponding output) can be **uint**, **uvec2**, **uvec3**, or **uvec4**, *genUType* is used as the argument. 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 *genType*, *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**
genType **radians** (genType *degrees*)	Converts *degrees* to radians, i.e., $\frac{\pi}{180} \cdot degrees$
genType **degrees** (genType *radians*)	Converts *radians* to degrees, i.e., $\frac{180}{\pi} \cdot radians$
genType **sin** (genType *angle*)	The standard trigonometric sine function.
genType **cos** (genType *angle*)	The standard trigonometric cosine function.
genType **tan** (genType *angle*)	The standard trigonometric tangent.
genType **asin** (genType *x*)	Arc sine. Returns an angle whose sine is *x*. The range of values returned by this function is $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ Results are undefined if $\lvert x \rvert > 1$.
genType **acos** (genType *x*)	Arc cosine. Returns an angle whose cosine is *x*. The range of values returned by this function is $[0, \pi]$. Results are undefined if $\lvert x \rvert > 1$.
genType **atan** (genType *y*, genType *x*)	Arc tangent. Returns an angle whose tangent is *y/x*. The signs of *x* and *y* are used to determine what quadrant the angle is in. The range of values returned by this function is $[-\pi, \pi]$. Results are undefined if *x* and *y* are both 0.
genType **atan** (genType *y_over_x*)	Arc tangent. Returns an angle whose tangent is *y_over_x*. The range of values returned by this function is $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

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

8.2 Exponential Functions

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

Syntax	**Description**
genType **pow** (genType x, genType y)	Returns x raised to the y power, i.e., x^y Results are undefined if $x < 0$. Results are undefined if $x = 0$ and $y <= 0$.
genType **exp** (genType x)	Returns the natural exponentiation of x, i.e., e^x.
genType **log** (genType x)	Returns the natural logarithm of x, i.e., returns the value y which satisfies the equation $x = e^y$. Results are undefined if $x <= 0$.
genType **exp2** (genType x)	Returns 2 raised to the x power, i.e., 2^x
genType **log2** (genType x)	Returns the base 2 logarithm of x, i.e., returns the value y which satisfies the equation $x = 2^y$ Results are undefined if $x <= 0$.
genType **sqrt** (genType x) genDType **sqrt** (genDType x)	Returns $\sqrt{x}$. Results are undefined if $x < 0$.
genType **inversesqrt** (genType x) genDType **inversesqrt** (genDType x)	Returns $\frac{1}{\sqrt{x}}$. Results are undefined if $x <= 0$.

8.3 Common Functions

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

Syntax	**Description**
genType **abs** (genType *x*) genIType **abs** (genIType *x*) genDType **abs** (genDType *x*)	Returns *x* if *x* >= 0, otherwise it returns –*x*.
genType **sign** (genType *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.
genType **floor** (genType *x*) genDType **floor** (genDType *x*)	Returns a value equal to the nearest integer that is less than or equal to *x*.
genType **trunc** (genType *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*.
genType **round** (genType *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*.
genType **roundEven** (genType *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.)
genType **ceil** (genType *x*) genDType **ceil** (genDType *x*)	Returns a value equal to the nearest integer that is greater than or equal to *x*.
genType **fract** (genType *x*) genDType **fract** (genDType *x*)	Returns *x* – **floor** (*x*).
genType **mod** (genType *x*, float *y*) genType **mod** (genType *x*, genType *y*) genDType **mod** (genDType *x*, double *y*) genDType **mod** (genDType *x*, genDType *y*)	Modulus. Returns *x* – *y* * **floor** (*x*/*y*).

Syntax	**Description**
genType **modf** (genType *x*, out genType *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*.
genType **min** (genType *x*, genType *y*) genType **min** (genType *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*.
genType **max** (genType *x*, genType *y*) genType **max** (genType *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*.

Syntax	Description
genType **clamp** (genType *x*, genType *minVal*, genType *maxVal*) genType **clamp** (genType *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*.
genType **mix** (genType *x*, genType *y*, genType *a*) genType **mix** (genType *x*, genType *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., $x \cdot (1-a) + y \cdot a$

Syntax	**Description**
genType **mix** (genType *x*, genType y, genBType a) genDType **mix** (genDType *x*, genDType 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, genType **mix**(genType *x*, genType *y*, genType(*a*)) where *a* is a Boolean vector.
genType **step** (genType *edge*, genType *x*) genType **step** (float *edge*, genType *x*) genDType **step** (genDType *edge*, genDType *x*) genDType **step** (double *edge*, genDType *x*)	Returns 0.0 if *x* < *edge*, otherwise it returns 1.0.
genType **smoothstep** (genType *edge0*, genType *edge1*, genType *x*) genType **smoothstep** (float *edge0*, float *edge1*, genType *x*) genDType **smoothstep** (genDType *edge0*, genDType *edge1*, genDType *x*) genDType **smoothstep** (double *edge0*, double *edge1*, genDType *x*)	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0 and 1 when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to: genType t; t = clamp ((x – edge0) / (edge1 – edge0), 0, 1); return t * t * (3 – 2 * t); (And similarly for doubles.) *Results are undefined if edge0 >= edge1.*
genBType **isnan** (genType *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** (genType *x*) genBType **isinf** (genDType *x*)	Returns **true** if *x* holds a positive infinity or negative infinity. Returns **false** otherwise.
genIType **floatBitsToInt** (genType *value*) genUType **floatBitsToUint** (genType *value*)	Returns a signed or unsigned integer value representing the encoding of a floating-point value. The floating-point value's bit-level representation is preserved.

Syntax	**Description**
genType **intBitsToFloat** (genIType *value*) genType **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 floating point value is unspecified. If an Inf is passed in, the resulting floating-point value is the corresponding Inf.
genType **fma** (genType *a*, genType *b*, genType *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 "*a*b + c*" consumed by a variable declared **precise** is considered two operations. • The precision of **fma**() can differ from the precision of the the expression "*a*b + c*". • **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 "*a*b + c*".
genType **frexp** (genType *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: $x = significand \cdot 2^{exponent}$ The significand is returned by the function and the exponent is returned in the parameter *exp*. For a floating-point value of zero, the significant and exponent are both zero. For a floating-point value that is an infinity or is not a number, the results are undefined.
genType **ldexp** (genType *x*, in genIType *exp*) genDType **ldexp** (genDType *x*, in genIType *exp*)	Builds a floating-point number from *x* and the corresponding integral exponent of two in *exp*, returning: $significand \cdot 2^{exponent}$ If this product is too large to be represented in the floating-point type, the result is undefined.

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 8- or 16-bit 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, four 8-bit unsigned integers, or four 8-bit signed integers. 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
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 the 32 most significant bits.
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*.

8.5 Geometric Functions

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

Syntax	Description
float **length** (genType *x*) double **length** (genDType *x*)	Returns the length of vector *x*, i.e., $\sqrt{x[0]^2+x[1]^2+\ldots}$
float **distance** (genType *p0*, genType *p1*) double **distance** (genDType *p0*, genDType *p1*)	Returns the distance between *p0* and *p1*, i.e., **length** (*p0 – p1*)
float **dot** (genType *x*, genType *y*) double **dot** (genDType *x*, genDType *y*)	Returns the dot product of *x* and *y*, i.e., $x[0]\cdot y[0]+x[1]\cdot y[1]+\ldots$
vec3 **cross** (vec3 *x*, vec3 *y*) dvec3 **cross** (dvec3 *x*, dvec3 *y*)	Returns the cross product of x and y, i.e., $\begin{bmatrix} x[1]\cdot y[2]-y[1]\cdot x[2] \\ x[2]\cdot y[0]-y[2]\cdot x[0] \\ x[0]\cdot y[1]-y[0]\cdot x[1] \end{bmatrix}$
genType **normalize** (genType *x*) genDType **normalize** (genDType *x*)	Returns a vector in the same direction as *x* but with a length of 1.
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.

Syntax	Description
genType **faceforward** (genType *N*, genType *I*, genType *Nref*) genDType **faceforward** (genDType *N*, genDType *I*, genDType *Nref*)	If **dot**(*Nref*, *I*) < 0 return *N,* otherwise return –*N.*
genType **reflect** (genType *I*, genType *N*) genDType **reflect** (genDType *I*, genDType *N*)	For the incident vector *I* and surface orientation *N*, returns the reflection direction: *I* – 2 * **dot**(*N*, *I*) * *N* *N* must already be normalized in order to achieve the desired result.

genType **refract** (genType *I*, genType *N*,
float *eta*)
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
	k = 1.0 - *eta* * *eta* * (1.0 - **dot**(*N*, *I*) * **dot**(*N*, *I*))
	if (k < 0.0)
	    return genType(0.0) // or genDType(0.0)
	else
	    return *eta* * *I* - (*eta* * **dot**(*N*, *I*) + **sqrt**(k)) * *N*
	The input parameters for the incident vector *I* and the surface normal *N* must already be normalized to get the desired results.

8.6 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 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 $x <= y$.
bvec **greaterThan** (vec x, vec y) bvec **greaterThan** (ivec x, ivec y) bvec **greaterThan** (uvec x, uvec y)	Returns the component-wise compare of $x > y$.
bvec **greaterThanEqual** (vec x, vec y) bvec **greaterThanEqual** (ivec x, ivec y) bvec **greaterThanEqual** (uvec x, uvec y)	Returns the component-wise compare of $x >= y$.
bvec **equal** (vec x, vec y) bvec **equal** (ivec x, ivec y) bvec **equal** (uvec x, uvec y) bvec **equal** (bvec x, bvec y) bvec **notEqual** (vec x, vec y) bvec **notEqual** (ivec x, ivec y) bvec **notEqual** (uvec x, uvec y) bvec **notEqual** (bvec x, bvec y)	Returns the component-wise compare of $x == y$. Returns the component-wise compare of $x != y$.
bool **any** (bvec x)	Returns true if any component of *x* is **true**.

Syntax	**Description**
bool **all** (bvec x)	Returns true only if all components of *x* are **true**.
bvec **not** (bvec x)	Returns the component-wise logical complement of *x*.

8.8 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. “Bit number” 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 integer *x* and *y*, returning the sum modulo 2^{32}. The value *carry* is set to 0 if the sum was less than 2^{32}, or to 1 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 0 if $x >= y$, or to 1 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 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.

Syntax	Description
genIType **bitfieldInsert** (genIType *base*, genIType *insert*, int *offset*, int *bits*) genUType **bitfieldInsert** (genUType *base*, genUType *insert*, int *offset*, int *bits*)	Returns the insertion 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.
genIType **bitfieldReverse** (genIType *value*) genUType **bitfieldReverse** (genUType *value*)	Returns the reversal of 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 bits set to 1 in the binary representation of *value*.
genIType **findLSB** (genIType *value*) genIType **findLSB** (genUType *value*)	Returns the bit number of the least significant bit set to 1 in the binary representation of *value*. If *value* is zero, -1will 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 bit set to 1. For negative integers, the result will be the bit number of the most significant bit set to 0. 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, automatic level of detail is 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 mip-map levels, depth comparison, and so on are also defined by OpenGL API calls. Such properties are taken into account as the texture is accessed via the built-in functions defined below.

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

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

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

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

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

For shadow forms (the sampler parameter is a shadow-type), a depth comparison lookup on the depth texture bound to *sampler* is done as described in section 3.8.16 "Texture Comparison Modes" of the OpenGL Graphics System 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 shaders. 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 rectangular textures, multi-sample textures, or texture buffers are supported because mip-maps are not allowed for these types of textures.

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

Some texture functions (non-"**Lod**" and non-"**Grad**" versions) may require implicit derivatives. Implicit derivatives are undefined within non-uniform control flow and for 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 3.8.10 "Cube Map Texture Selection" in the OpenGL Graphics System Specification.

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

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

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

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 in the OpenGL Graphics System Specification) is obtained after any LOD 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:

```
float ComputeAccessedLod(float computedLod)
{
    // Clamp the computed LOD according to the texture LOD clamps.
    if (computedLod < TEXTURE_MIN_LOD) computedLod = TEXTURE_MIN_LOD;
    if (computedLod > TEXTURE_MAX_LOD) computedLod = TEXTURE_MAX_LOD;

    // Clamp the computed LOD to the range of accessible levels.
    if (computedLod < 0.0)
        computedLod = 0.0;
    if (computedLod > (float)
        maxAccessibleLevel) computedLod = (float) maxAccessibleLevel;

    // Return a value according to the min filter.
    if (TEXTURE_MIN_FILTER is LINEAR or NEAREST) {
        return 0.0;
    } else if (TEXTURE_MIN_FILTER is NEAREST_MIPMAP_NEAREST
               or LINEAR_MIPMAP_NEAREST) {
        return ceil(computedLod + 0.5) - 1.0;
    } else {
        return computedLod;
    }
}
```

The value *maxAccessibleLevel* is the level number of the smallest accessible level of the mipmap array (the value *q* in section 3.8.9 “Texture Minification” of the OpenGL Graphics System 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*) ivec3 **textureSize** (gsampler2DArray *sampler*, int *lod*) ivec2 **textureSize** (sampler1DArrayShadow *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 2.11.7 “Shader Execution” of the OpenGL Graphics System Specification, under “Texture Size Query”. The components in the return value are filled in, in order, with the width, height, depth of the texture. For the array forms, the last component of the return value is the number of layers in the texture array.

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.

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** (gsampler1DArray *sampler,* vec2 *P* [, float *bias*]) gvec4 **texture** (gsampler2DArray *sampler*, vec3 *P* [, float *bias*]) gvec4 **texture** (gsamplerCubeArray *sampler*, vec4 *P* [, float *bias*]) float **texture** (sampler1DArrayShadow *sampler,* vec3 *P* [, float *bias*]) float **texture** (sampler2DArrayShadow *sampler,* vec4 *P*) gvec4 **texture** (gsampler2DRect *sampler*, vec2 *P*) float **texture** (sampler2DRectShadow *sampler*, vec3 *P*) float **texture** (gsamplerCubeArrayShadow *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 *P.w*. 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*.

Syntax	**Description**
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*. The resulting 3rd component of *P* in the shadow forms is used as D_{ref}. After these values are computed, texture lookup proceeds as in **texture**.
gvec4 **textureLod** (gsampler1D *sampler*, float *P*, float *lod*) gvec4 **textureLod** (gsampler2D *sampler*, vec2 *P*, float *lod*) gvec4 **textureLod** (gsampler3D *sampler*, vec3 *P*, float *lod*) gvec4 **textureLod** (gsamplerCube *sampler*, vec3 *P*, float *lod*) float **textureLod** (sampler1DShadow *sampler,* vec3 *P*, float *lod*) float **textureLod** (sampler2DShadow *sampler,* vec3 *P*, float *lod*) gvec4 **textureLod** (gsampler1DArray *sampler*, vec2 *P*, float *lod*) gvec4 **textureLod** (gsampler2DArray *sampler*, vec3 *P*, float *lod*) float **textureLod** (sampler1DArrayShadow *sampler,* vec3 *P*, float *lod*) gvec4 **textureLod** (gsamplerCubeArray *sampler*, vec4 *P*, float *lod*)	Do a texture lookup as in **texture** but with explicit LOD; *lod* specifies λ_{base} and sets the partial derivatives as follows. (See section 3.8.11 "Texture Minification" and equation 3.17 in the OpenGL Graphics System Specification.) $\frac{\partial u}{\partial x} = 0 \quad \frac{\partial v}{\partial x} = 0 \quad \frac{\partial w}{\partial x} = 0$ $\frac{\partial u}{\partial y} = 0 \quad \frac{\partial v}{\partial y} = 0 \quad \frac{\partial w}{\partial y} = 0$

Syntax	**Description**
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*]) 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*]) float **textureOffset** (sampler2DShadow *sampler,* vec3 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureOffset** (gsampler1DArray *sampler,* vec2 *P,* int *offset* [, float *bias*]) gvec4 **textureOffset** (gsampler2DArray *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]) float **textureOffset** (sampler1DArrayShadow *sampler,* vec3 *P*, int *offset* [, float *bias*])	Do a texture lookup as in **texture** but with *offset* added to the (u,v,w) texel coordinates before looking up each texel. The offset value must be a constant expression. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by 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 3.8.11 “Texture Minification” of the OpenGL Graphics System Specification, where *offset* is $(\delta_u, \delta_v, \delta_w)$. Note that texel offsets are also not supported for cube maps.
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 2.11.8 “Shader Execution” under Texel Fetches and 3.8 “Texturing” of the OpenGL Graphics System Specification.

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

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

Syntax	**Description**
gvec4 **textureGradOffset** (gsampler1D *sampler,* float *P,* float *dPdx*, float *dPdy,* int *offset*) gvec4 **textureGradOffset** (gsampler2D *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy,* ivec2 *offset*) gvec4 **textureGradOffset** (gsampler3D *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy,* ivec3 *offset*) 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** (gsampler1DArray *sampler,* vec2 *P,* float *dPdx*, float *dPdy,* int *offset*) gvec4 **textureGradOffset** (gsampler2DArray *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy,* ivec2 *offset*) float **textureGradOffset** (sampler1DArrayShadow *sampler,* vec3 *P*, float *dPdx*, float *dPdy,* int *offset*) float **textureGradOffset** (sampler2DArrayShadow *sampler,* vec4 *P*, vec2 *dPdx*, vec2 *dPdy,* ivec2 *offset*)	Do a texture lookup with both explicit gradient and offset, as described in **textureGrad** and **textureOffset**.
gvec4 **textureProjGrad** (gsampler1D *sampler,* vec2 *P*, float *dPdx*, float *dPdy*) gvec4 **textureProjGrad** (gsampler1D *sampler,* vec4 *P*, float *dPdx*, float *dPdy*) gvec4 **textureProjGrad** (gsampler2D *sampler,* vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureProjGrad** (gsampler2D *sampler,* vec4 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureProjGrad** (gsampler3D *sampler,* vec4 *P*, vec3 *dPdx*, vec3 *dPdy*) 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** (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*) gvec4 **textureProjGradOffset** (gsampler3D *sampler,* vec4 *P*, vec3 *dPdx*, vec3 *dPdy*, ivec3 *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 Texture Gather Instructions

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_0j_1, i_1j_1, i_1j_0, and i_0j_0. The texels are then converted to texture source colors (R_s, G_s, B_s, A_s) according to Table 3.20, followed by application of the texture swizzle as described in section 3.9.2 “Shader Execution” of the OpenGL Graphics System 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_0j_1, i_1j_1, i_1j_0, i_0j_0).

For texture gather functions using a shadow sampler type, each of the four texel lookups performs 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 vec4(Sample_i0_j1(*P*, base).*comp*, Sample_i1_j1(*P*, base).*comp*, Sample_i1_j0(*P*, base).*comp*, Sample_i0_j0(*P*, base).*comp*) 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*]) gvec4 **textureGatherOffset**(gsampler2DRect *sampler,* vec2 *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*) 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.

Syntax	**Description**
gvec4 **textureGatherOffsets**(gsampler2D *sampler*, vec2 *P*, ivec2 *offsets*[4] [, *int comp*]) gvec4 **textureGatherOffsets**(gsampler2DArray *sampler*, vec3 *P*, ivec2 *offsets*[4] [, *int comp*]) gvec4 **textureGatherOffsets**(gsampler2DRect *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]) 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_0j_0 of that footprint. The specified values in *offsets* must be set with constant integral expressions.

8.9.4 Compatibility Profile Texture Functions

The following texture functions are only in the compatibility profile.

Syntax (deprecated)	**Description (deprecated)**
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 "1D" in the name.
vec4 **texture2D** (sampler2D *sampler*, vec2 *coord* [, float *bias*]) vec4 **texture2DProj** (sampler2D *sampler*, vec3 *coord* [, float *bias*]) vec4 **texture2DProj** (sampler2D *sampler*, vec4 *coord* [, float *bias*]) vec4 **texture2DLod** (sampler2D *sampler*, vec2 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*, vec3 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*, vec4 *coord*, float *lod*)	See corresponding signature above without "2D" in the name.
vec4 **texture3D** (sampler3D *sampler*, vec3 *coord* [, float *bias*]) vec4 **texture3DProj** (sampler3D *sampler*, vec4 *coord* [, float *bias*]) vec4 **texture3DLod** (sampler3D *sampler*, vec3 *coord*, float *lod*) vec4 **texture3DProjLod** (sampler3D *sampler*, vec4 *coord*, float *lod*)	See corresponding signature above without "3D" in the name. Use the texture coordinate *coord* to do a texture lookup in the 3D texture currently bound to *sampler*. For the projective ("**Proj**") versions, the texture coordinate is divided by *coord.q*.
vec4 **textureCube** (samplerCube *sampler*, vec3 *coord* [, float *bias*]) vec4 **textureCubeLod** (samplerCube *sampler*, vec3 *coord*, float *lod*)	See corresponding signature above without "Cube" in the name.

Syntax (deprecated)	**Description (deprecated)**
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 "**texture"** 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 a synchronization, if atomicity or serialization is desired.

The value returned by an atomic-counter function is the value of an atomic counter, which may be

- returned and incremented in an atomic operation, or
- decremented and returned in an atomic operation, or
- simply returned.

The underlying counter is a 32-bit unsigned integer. Increments and decrements at the limit of the range 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 the atomic counter functions in this table.
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*.

8.11 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 multi-sample 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 3.9.X of the OpenGL specification.

Loads and stores support float, integer, and unsigned integer types. The data types below starting "*gimage*" serve as placeholders meaning types starting either "**image**", "**iimage**", or "**uimage**" 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:

gimage1D *image*, int *P*

gimage2D *image*, ivec2 *P*

gimage3D *image*, ivec3 *P*

gimage2DRect *image*, ivec2 *P*

gimageCube *image*, ivec3 *P*

gimageBuffer *image*, int *P*

gimage1DArray *image*, ivec2 *P*

gimage2DArray *image*, ivec3 *P*

gimageCubeArray *image*, ivec3 *P*

gimage2DMS *image*, ivec2 *P*, int *sample*

gimage2DMSArray *image*, ivec3 *P*, int *sample*

where each of the line 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 3.9.X of the OpenGL specification ref .

The atomic functions perform atomic 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 "**iimage**") and a format qualifier of *r32i*, used with a *data* argument of type **int***,* or
- an unsigned image variable (type starts "**uimage**") and a format qualifier of *r32ui*, used with a *data* argument of type **uint**.

Syntax	**Description**
gvec4 **imageLoad** (readonly *IMAGE_PARAMS*)	Loads the texel at the coordinate *P* from the image unit *image* (in *IMAGE_PARAMS*). For multi-sample loads, the sample number is given by *sample*. When *image*, *P*, *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 3.9.20 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 multi-sample 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 3.9.20 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.
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.

Syntax	Description
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*)	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.

8.12 Fragment Processing Functions

Fragment processing functions are only available in fragment shaders.

8.12.1 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:

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

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

Backward differencing:

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

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

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

dFdy is approximated similarly, with y replacing x.

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

1. The method may use piecewise linear approximations. Such linear approximations imply that higher order derivatives, **dFdx**(**dFdx**(*x*)) and above, are undefined.
2. The method may assume that the function evaluated is continuous. Therefore derivatives within non-uniform control flow are undefined.
3. The method may differ per fragment, subject to the constraint that the method may vary by window coordinates, not screen coordinates. The invariance requirement described in section 3.2 "Invariance" of the OpenGL Graphics System Specification, is relaxed for derivative calculations, because the method may be a function of fragment location.

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

4. Functions should be evaluated within the interior of a primitive (interpolated, not extrapolated).
5. Functions for **dFdx** should be evaluated while holding y constant. Functions for **dFdy** should be evaluated while holding x constant. However, mixed higher order derivatives, like **dFdx**(**dFdy**(*y*)) and **dFdy**(**dFdx**(*x*)) are undefined.
6. Derivatives of constant arguments should be 0.

In some implementations, varying degrees of derivative accuracy may be obtained by providing GL hints (section 5.4 "Hints" of the OpenGL Graphics System Specification), allowing a user to make an image quality versus speed trade off.

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

Syntax	Description
genType **fwidth** (genType *p*)	Returns the sum of the absolute derivative in x and y using local differencing for the input argument *p*, i.e., **abs** (**dFdx** (*p*)) + **abs** (**dFdy** (*p*));

8.12.2 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 input variable or an element of an input variable declared as an array. Component selection operators (e.g., *.xy*) may not be used when specifying *interpolant*. If *interpolant* is declared with a **flat** or **centroid** qualifier, the qualifier will have no effect on the interpolated value. 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*)	Return the value of the input *interpolant* sampled at a location inside the 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*)	Return 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)	Return 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.

8.13 Noise Functions

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

- The return value(s) are always in the range [-1.0,1.0], and cover at least the range [-0.6, 0.6], with a Gaussian-like distribution.
- The return value(s) have an overall average of 0.0
- They are repeatable, in that a particular input value will always produce the same return value
- They are statistically invariant under rotation (i.e., no matter how the domain is rotated, it has the same statistical character)
- They have a statistical invariance under translation (i.e., no matter how the domain is translated, it has the same statistical character)
- They typically give different results under translation.
- The spatial frequency is narrowly concentrated, centered somewhere between 0.5 to 1.0.
- They are C^1 continuous everywhere (i.e., the first derivative is continuous)

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

8.14 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*)	Emit 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.
void **EmitVertex** ()	Emit the current values of output variables to the current output primitive. On return from this call, the values of output variables are undefined. When multiple output streams are supported, this is equivalent to calling **EmitStreamVertex**(0).
void **EndPrimitive** ()	Completes the current output primitive and starts a new one. No vertex is emitted. When multiple output streams are supported, this is equivalent to calling **EndStreamPrimitive**(0).

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 output variables associated with *stream*. These include *gl_PointSize*, *gl_ClipDistance*, *gl_Layer*, *gl_Position*, *gl_PrimitiveID* and *gl_ViewportIndex*. 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 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*. A program will fail to link if it contains a geometry shader calling **EmitStreamVertex**() or **EndStreamPrimitive**() if its input primitive type is not *points*.

8.15 Shader Invocation Control Functions

The shader invocation control function is available only in tessellation control shaders. It is used to control the relative execution order of multiple shader invocations used to process a patch, 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.

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 will be undefined in a number of cases enumerated in section 4.3.7 “Output Variables”.

The **barrier**() function may only be placed inside the function **main**() of the tessellation control shader and may not be called within any control flow. Barriers are also disallowed after a return statement in the function **main**().

8.16 Shader Memory Control Function

Shaders of all types may read and write the contents of textures and buffer objects using image variables. While the order of reads and writes within a single shader invocation is well-defined, the relative order of reads and writes to a single shared memory address from multiple separate shader invocations is largely undefined. The order of memory accesses performed by one shader invocation, as observed by other shader invocations, is also largely undefined but can be controlled through memory control functions.

Syntax	Description
void **memoryBarrier** ()	Control the ordering of memory transactions issued by a single shader invocation.

The built-in **memoryBarrier**() waits on the completion of all memory accesses resulting from the use of image variables or atomic counters (by the same shader invocation calling **memoryBarrier()**) and then returns with no other effect. When this function returns, the results of any memory stores performed using coherent variables performed prior to the call will be visible to any future coherent memory access to the same addresses from other shader invocations. 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).

9 Shading Language Grammar for Core Profile

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

```
CONST BOOL FLOAT DOUBLE INT UINT
BREAK CONTINUE DO ELSE FOR IF DISCARD RETURN SWITCH CASE DEFAULT SUBROUTINE
BVEC2 BVEC3 BVEC4 IVEC2 IVEC3 IVEC4 UVEC2 UVEC3 UVEC4 VEC2 VEC3 VEC4
MAT2 MAT3 MAT4 CENTROID IN OUT INOUT UNIFORM PATCH SAMPLE
DVEC2 DVEC3 DVEC4 DMAT2 DMAT3 DMAT4

NOPERSPECTIVE FLAT SMOOTH LAYOUT PATCH SAMPLE
MAT2X2 MAT2X3 MAT2X4
MAT3X2 MAT3X3 MAT3X4
MAT4X2 MAT4X3 MAT4X4
DMAT2X2 DMAT2X3 DMAT2X4
DMAT3X2 DMAT3X3 DMAT3X4
DMAT4X2 DMAT4X3 DMAT4X4
SAMPLER1D SAMPLER2D SAMPLER3D SAMPLERCUBE SAMPLER1DSHADOW SAMPLER2DSHADOW
SAMPLERCUBESHADOW SAMPLER1DARRAY SAMPLER2DARRAY SAMPLER1DARRAYSHADOW
SAMPLER2DARRAYSHADOW ISAMPLER1D ISAMPLER2D ISAMPLER3D ISAMPLERCUBE
ISAMPLER1DARRAY ISAMPLER2DARRAY USAMPLER1D USAMPLER2D USAMPLER3D
USAMPLERCUBE USAMPLER1DARRAY USAMPLER2DARRAY
SAMPLER2DRECT SAMPLER2DRECTSHADOW ISAMPLER2DRECT USAMPLER2DRECT
SAMPLERBUFFER ISAMPLERBUFFER USAMPLERBUFFER
SAMPLERCUBEARRAY SAMPLERCUBEARRAYSHADOW
ISAMPLERCUBEARRAY USAMPLERCUBEARRAY
SAMPLER2DMS ISAMPLER2DMS USAMPLER2DMS
SAMPLER2DMSArray ISAMPLER2DMSArray USAMPLER2DMSArray
IMAGE1D IIMAGE1D UIMAGE1D IMAGE2D IIMAGE2D
UIMAGE2D IMAGE3D IIMAGE3D UIMAGE3D
IMAGE2DRECT IIMAGE2DRECT UIMAGE2DRECT
IMAGECUBE IIMAGECUBE UIMAGECUBE
IMAGEBUFFER IIMAGEBUFFER UIMAGEBUFFER
IMAGE1DARRAY IIMAGE1DARRAY UIMAGE1DARRAY
IMAGE2DARRAY IIMAGE2DARRAY UIMAGE2DARRAY
IMAGECUBEARRAY IIMAGECUBEARRAY UIMAGECUBEARRAY
IMAGE2DMS IIMAGE2DMS UIMAGE2DMS
IMAGE2DMSARRAY IIMAGE2DMSARRAY UIMAGE2DMSARRAY
```

```
STRUCT VOID WHILE

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

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

INVARIANT PRECISE
HIGH_PRECISION MEDIUM_PRECISION LOW_PRECISION PRECISION
```

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

variable_identifier:
IDENTIFIER

primary_expression:
variable_identifier
INTCONSTANT
UINTCONSTANT
FLOATCONSTANT
BOOLCONSTANT
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

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 LEFT_BRACKET RIGHT_BRACKET SEMICOLON
type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET 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 LEFT_BRACKET constant_expression RIGHT_BRACKET

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 LEFT_BRACKET RIGHT_BRACKET

init_declarator_list COMMA IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

init_declarator_list COMMA IDENTIFIER LEFT_BRACKET RIGHT_BRACKET EQUAL initializer

init_declarator_list COMMA IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET EQUAL initializer

init_declarator_list COMMA IDENTIFIER EQUAL initializer

single_declaration:

fully_specified_type

fully_specified_type IDENTIFIER

fully_specified_type IDENTIFIER LEFT_BRACKET RIGHT_BRACKET

fully_specified_type IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

fully_specified_type IDENTIFIER LEFT_BRACKET RIGHT_BRACKET EQUAL initializer

fully_specified_type IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET EQUAL initializer

fully_specified_type IDENTIFIER EQUAL initializer

INVARIANT IDENTIFIER

// 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 INTCONSTANT

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
    INOUT
    IN

OUT
CENTROID
PATCH
SAMPLE
UNIFORM
SUBROUTINE
SUBROUTINE LEFT_PAREN IDENTIFIER type_name_list RIGHT_PAREN

type_name_list:
TYPE_NAME
type_name_list COMMA TYPE_NAME

type_specifier:
type_specifier_nonarray
type_specifier_nonarray LEFT_BRACKET RIGHT_BRACKET
type_specifier_nonarray 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
SAMPLER1D
SAMPLER2D
SAMPLER3D
SAMPLERCUBE
SAMPLER1DSHADOW
SAMPLER2DSHADOW
SAMPLERCUBESHADOW
SAMPLER1DARRAY
SAMPLER2DARRAY
SAMPLER1DARRAYSHADOW
SAMPLER2DARRAYSHADOW
SAMPLERCUBEARRAY

SAMPLERCUBEARRAYSHADOW
ISAMPLER1D
ISAMPLER2D
ISAMPLER3D
ISAMPLERCUBE
ISAMPLER1DARRAY
ISAMPLER2DARRAY
ISAMPLERCUBEARRAY
USAMPLER1D
USAMPLER2D
USAMPLER3D
USAMPLERCUBE
USAMPLER1DARRAY
USAMPLER2DARRAY
USAMPLERCUBEARRAY
SAMPLER2DRECT
SAMPLER2DRECTSHADOW
ISAMPLER2DRECT
USAMPLER2DRECT
SAMPLERBUFFER
ISAMPLERBUFFER
USAMPLERBUFFER
SAMPLER2DMS
ISAMPLER2DMS
USAMPLER2DMS
SAMPLER2DMSARRAY
ISAMPLER2DMSARRAY
USAMPLER2DMSARRAY
IMAGE1D
IIMAGE1D
UIMAGE1D
IMAGE2D
IIMAGE2D
UIMAGE2D
IMAGE3D
IIMAGE3D
UIMAGE3D

IMAGE2DRECT
IIMAGE2DRECT
UIMAGE2DRECT
IMAGECUBE
IIMAGECUBE
UIMAGECUBE
IMAGEBUFFER
IIMAGEBUFFER
UIMAGEBUFFER
IMAGE1DARRAY
IIMAGE1DARRAY
UIMAGE1DARRAY
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 LEFT_BRACKET RIGHT_BRACKET

IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

initializer:

assignment_expression

declaration_statement:

declaration

statement:

compound_statement

simple_statement

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

simple_statement:

declaration_statement

expression_statement

selection_statement

switch_statement

case_label

iteration_statement

jump_statement

compound_statement:

LEFT_BRACE RIGHT_BRACE

LEFT_BRACE statement_list RIGHT_BRACE

statement_no_new_scope:

compound_statement_no_new_scope

simple_statement

compound_statement_no_new_scope:
LEFT_BRACE RIGHT_BRACE
LEFT_BRACE statement_list RIGHT_BRACE

statement_list:
statement
statement_list statement

expression_statement:
SEMICOLON
expression SEMICOLON

selection_statement:
IF LEFT_PAREN expression RIGHT_PAREN selection_rest_statement

selection_rest_statement:
statement ELSE statement
statement

condition:
expression
fully_specified_type IDENTIFIER EQUAL initializer

switch_statement:
SWITCH LEFT_PAREN expression RIGHT_PAREN LEFT_BRACE switch_statement_list RIGHT_BRACE

switch_statement_list:
/ nothing */*
statement_list

case_label:
CASE expression COLON
DEFAULT COLON

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

for_init_statement:
expression_statement
declaration_statement

conditionopt:
condition
/ empty */*

for_rest_statement:
conditionopt SEMICOLON
conditionopt SEMICOLON expression

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

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

translation_unit:
external_declaration
translation_unit external_declaration

external_declaration:
function_definition
declaration

function_definition:
function_prototype compound_statement_no_new_scope