The OpenGL ES® Shading Language, Version 3.20.5

Robert J. Simpson, Qualcomm (Editor) ; John Kessenich, Dave Baldwin and Randi Rost (Version 1.1 Authors)

Version 3.20.5, Wed, 12 Dec 2018 23:38:17 +0000

Table of Contents

Chapter 1. Introduction

This document specifies only version 3.20 of the OpenGL ES Shading Language. It requires __VERSION__ to substitute 320, and requires **#version** to accept only `320 es`. 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 ES API. See the OpenGL ES Specification for details on what language versions are supported.

All references in this specification to the OpenGL ES Specification are to version 3.2.

1.1. Changes

1.1.1. Changes from GLSL ES 3.2 revision 4

- Clarified that this specification completely defines the OpenGL ES Shading Language. Normatively reference C++ only for the preprocessor.
- Private GLSL issues 7, 38: Corrected the values of some builtin constants. The values were given correctly in the OpenGL ES Specification.
- Private GLSL issue 30: Clarify that output packing rules apply to the last vertex pipeline stage, not necessarily the vertex stage.
- Private GLSL issue 15: Clarify the ordering of bindings for arrays of arrays.
- Private GLSL issue 14: Uniform variables need only match at link time if they are statically used.
- For **precise** computations, the controlling expressions for control flow and ternary operators (**?:**) are not included.

1.1.2. Changes from GLSL ES 3.2 revision 3

- Matching of default uniforms when shaders are linked.
- *gl_DepthRange* is only guaranteed to be available in the fragment stage.
- Clarification of definition of static use.
- Sampling behavior in the absence of **sample** and **centroid**.
- Clarified the requirements when the specification uses the terms *should*/*should not* and *undefined behavior*.
- Arrayed blocks cannot have layout location qualifiers on members
- **barrier**() defines a partial order which includes tessellation control shader outputs.
- Vertex shader integer output qualification.
- Incorrect use of predefined pragmas.
- Clarified use of **readonly** and **writeonly** qualifiers.
- USAMPLERBUFFER added to grammar.
- Clarified precision qualifiers can be used in interface blocks.

- Clarified **memoryBarrierShared** only applies to the current workgroup.
- The layout qualifier *invocations* must not be zero.
- The layout qualifier *local_size* must not be zero.
- Clarified the definition of static assignment.
- Removed list of types with no default precision.
- Removed scoping rules from the grammar. Refer instead to the scoping section.
- Require a statement after the final label of a switch.
- Define **gl_BoundingBox**.
- **length**() expressions returning a constant-value may not include side effects.
- Clarified that variables may be declared **readonly writeonly**.
- Use of constant expressions within **#line** directives is undefined.
- **gl_in** can be redeclared using unsized-array syntax.
- Clarified which sampler types may be used for depth and stencil textures.
- Added order-of-operation and other explanations to the Precise Qualifier section.
- The **precise** qualifier applied to a block/struct applies recursively to the members.

1.1.3. Changes from GLSL ES 3.2 revision 2

- Updated value for *gl_MaxTessControlTotalOutputComponents*
- Clarified the allowed character set for pre-processing
- Integer division wrapping behavior
- Clarified pre-processor expressions (*pp-constant-expression*)
- UBO and SSBO precisions do not need to match for linked shaders (consistent with GLSL ES 3.1)
- **modf** function
- Sequence and ternary operators with **void** type
- Sequence and ternary operators with array types

1.1.4. Changes from GLSL ES 3.2 revision 1

- Signed zeros must be supported
- Layout qualifier table
- Allowed optimizations when evaluating expressions
- Updated value for *gl_MaxTessControlInputComponents*
- Updated value for *gl_MaxTessControlOutputComponents*
- Updated value for *gl_MaxTessEvaluationInputComponents*
- Updated value for *gl_MaxTessEvaluationOutputComponents*
- Updated value for *gl_MaxGeometryOutputComponents*

- Require precisions in blocks to match when linking
- Updated conclusions in issues section

1.1.5. Changes from GLSL ES 3.1 revision 4

- Added the following extensions:
    - KHR_blend_equation_advanced
    - OES_sample_variables
    - OES_shader_image_atomic
    - OES_shader_multisample_interpolation
    - OES_texture_storage_multisample_2d_array
    - OES_geometry_shader
    - OES_gpu_shader5
    - OES_primitive_bounding_box
    - OES_shader_io_blocks
    - OES_tessellation_shader
    - OES_texture_buffer
    - OES_texture_cube_map_array
    - KHR_robustness

1.2. Overview

This document describes *The OpenGL ES Shading Language, version 3.20*.

Independent compilation units written in this language are called *shaders*. A *program* is a set of shaders that are compiled and linked together. The aim of this document is to thoroughly specify the programming language. The OpenGL ES Specification will specify the OpenGL ES entry points used to manipulate and communicate with programs and shaders.

1.3. Error Handling

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

The compilation process is implementation-dependent but is generally split into a number of stages, each of which occurs at one of the following times:

- A call to *glCompileShader*
- A call to *glLinkProgram*
- A draw call or a call to *glValidateProgram*

The implementation should report errors as early a possible but in any case must satisfy the following:

- All lexical, grammatical and semantic errors must have been detected following a call to *glLinkProgram*
- Errors due to mismatch between the shaders (link-time errors) must have been detected following a call to *glLinkProgram*
- Errors due to exceeding resource limits must have been detected following any draw call or a call to *glValidateProgram*
- A call to *glValidateProgram* must report all errors associated with a program object given the current GL state.

Where the specification uses the terms *required*, *must*/*must not*, *does*/*does not*, *disallowed*, or *not supported*, the compiler or linker is required to detect and report any violations. Similarly when a condition or situation is an **error**, it must be reported. Use of any feature marked as *reserved* is an error. Where the specification uses the terms *should*/*should not*, *undefined behavior*, *undefined value* or *undefined *results**, implementations will not produce a compile-time error but are encouraged to issue a warning for violations. The run-time behavior of the program in these cases is not constrained (and so may include termination or system instability). It is expected that systems will be designed to handle these cases gracefully but specification of this is outside the scope of OpenGL ES.

Implementations may not in general support functionality beyond the mandated parts of the specification without use of the relevant extension. The only exceptions are:

1. If a feature is marked as optional.
2. Where a maximum value is stated (e.g. the maximum number of vertex outputs), the implementation may support a higher value than that specified.

Where the implementation supports more than the mandated specification, off-target compilers are encouraged to issue warnings if these features are used.

The compilation process is split between the compiler and linker. The allocation of tasks between the compiler and linker is implementation dependent. Consequently there are many errors which may be detected either at compile or link time, depending on the implementation.

1.4. Typographical Conventions

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

1.5. Compatibility

The OpenGL ES 3.2 API is designed to work with GLSL ES v1.00, GLSL ES 3.00, GLSL ES 3.10 and GLSL ES 3.20. In general a shader written for versions prior to OpenGL ES 3.2 should work without modification in OpenGL ES 3.2.

When porting applications from an earlier to later version of the API, the following points should be noted:

- Not all language constructs present in earlier versions of the language are available in later versions e.g. attribute and varying qualifiers are present in v1.00 but not v3.00. However, the functionality of GLSL ES 3.20 is a super-set of GLSL ES 3.10.
- Some features of later versions of the API require language features that are not present in earlier version of the language.
- It is an error to link shaders if they are written in different versions of the language.
- The OpenGL ES 2.0 and 3.0 APIs do not support shaders written in GLSL ES 3.20.
- Using GLSL ES 1.00 shaders within OpenGL ES 3.x may extend the resources available beyond the minima specified in GLSL ES 1.0. Shaders which make use of this will not necessarily run on an OpenGL ES 2.0 implementation: Similarly for GLSL ES 3.00 shaders running within OpenGL ES 3.2.
- Support of line continuation and support of UTF-8 characters within comments is optional in GLSL ES 1.00 when used with the OpenGL ES 2.0 API. However, support is mandated for both of these when a GLSL ES 1.00 shader is used with the OpenGL ES 3.x APIs.

Chapter 2. Overview of OpenGL ES Shading

The OpenGL ES 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 ES processing pipeline. Currently, these processors are the vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute processors.

Compilation units for these processors are referred to as *shaders* and the processors themselves are also referred to as *shader stages*. Only one shader can be run on a processor at any one time; there is no support for linking multiple compilation units together for a single shader stage. One or more shaders are linked together to form a single program and each program contains shader *executables* for one or more consecutive shader stages.

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

Most OpenGL ES state is not tracked or made available to shaders. Typically, user-defined variables will be used for communicating between different stages of the OpenGL ES 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 ES 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 ES Shading Language to run on this processor are called *vertex shaders*.

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 ES Shading Language to run on this processor are called tessellation control shaders.

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

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

Tessellation control shader invocations run mostly independently, with undefined relative execution order. However, the built-in function **barrier**() can be used to control execution order by synchronizing invocations, effectively dividing tessellation control shader execution into a set of phases. Tessellation control shaders will get undefined results if one invocation reads from a per-vertex or per-patch attribute written by another invocation at any point during the same phase, or if two invocations attempt to write different values to the same per-patch output 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 ES Shading Language to run on this processor are called tessellation evaluation shaders.

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 ES Shading Language to run on this processor are called *geometry shaders*.

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 ES Shading Language to run on this processor are called *fragment shaders*.

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 ES state and the OpenGL ES command that caused the fragments to be generated.

2.6. Compute Processor

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

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

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

Chapter 3. Basics

3.1. Character Set

The source character set used for the OpenGL ES Shading Language is Unicode in the UTF-8 encoding scheme. Invalid UTF-8 characters are ignored. During pre-processing, the following applies:

- A byte with the value zero is always interpreted as the end of the string
- Backslash ('\'), is used to indicate line continuation when immediately preceding a new-line.
- White space consists of one or more of the following characters: the space character, horizontal tab, vertical tab, form feed, carriage-return, line-feed.
- The number sign (**#**) is used for preprocessor directives
- Macro names are restricted to:
    - The letters **a-z**, **A-Z**, and the underscore (**_**).
    - The numbers **0-9**, except for the first character of a macro name.

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

- The letters **a-z**, **A-Z**, and the underscore (**_**).
- The numbers **0-9**.
- The symbols period (**.**), plus (**+**), dash (**-**), slash (**/**), asterisk (*****), percent (**%**), angled brackets (**<** and **>**), square brackets (**[** and **]**), parentheses (**(** and **)**), braces (**{** and **}**), caret (**^**), vertical bar (**|**), ampersand (**&**), tilde (**~**), equals (**=**), exclamation point (**!**), colon (**:**), semicolon (**;**), comma (**,**), and question mark (**?**).

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

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

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

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

There is no end-of-file character.

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.

Diagnostic messages returned from compiling a shader must identify both the line number within a string and which source string the message applies to. Source strings are counted sequentially with the first string being string 0. Line numbers are one more than the number of new-lines that have been processed, including counting the new-lines that will be removed by the line-continuation character (\).

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

```
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. Version Declaration

Shaders must declare the version of the language they are written to. The version is specified in the first line of a shader by a character string:

```
#version number es
```

where *number* must be a version of the language, following the same convention as __VERSION__ above. The directive "**#version 320 es**" is required in any shader that uses version 3.20 of the language. Any *number* representing a version of the language a compiler does not support will cause an error to be generated. Version 1.00 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 1.00.

Shaders declaring version 3.20 of the shading language cannot be linked with shaders declaring a previous version.

The **#version** directive must be present in the first line of a shader and must be followed by a newline. It may contain optional white-space as specified below but no other characters are allowed. The directive is only permitted in the first line of a shader.

Processing of the #version directive occurs before all other preprocessing, including line concatenation and comment processing.

version-declaration **:**

whitespace$_{opt}$ POUND *whitespace*$_{opt}$ VERSION *whitespace number whitespace* ES *whitespace*$_{opt}$

Tokens:

POUND #
VERSION **version**
ES **es**

3.4. Preprocessor

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

The complete list of preprocessor directives is as follows.

#
#define
#undef

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

#error
#pragma

#extension

#line

The following operator is also available:

defined

Note that the version directive is not considered to be a preprocessor directive and so is not listed here.

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

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

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

The following predefined macros are available:

> __LINE__
> __FILE__
> __VERSION__
> GL_ES

__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 ES Shading Language. The version of the shading language described in this document will have __VERSION__ substitute the decimal integer 320.

GL_ES will be defined and set to 1. This is not true for the non-ES OpenGL Shading Language, so it can be used to do a compile time test to determine if a shader is running on an ES system.

By convention, all macro names containing two consecutive underscores (__) are reserved for use by underlying software layers. Defining such a name in a shader does not itself result in an error, but may result in unintended behaviors that stem from having multiple definitions of the same name. All macro names prefixed with “GL_” (“GL” followed by a single underscore) are also reserved, and defining such a name results in a compile-time error.

It is an error to undefine or to redefine a built-in (pre-defined) macro name.

Implementations must support macro-name lengths of up to 1024 characters. It is an error to declare a name with a length greater than this.

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

- Expressions following **#if** and **#elif** are restricted to *pp-constant-expressions* as defined below.
- Undefined identifiers not consumed by the **defined** operator do not default to '0'. Use of such identifiers causes an error.
- Character constants are not supported.

As in C++, a macro name defined with an empty replacement list does not default to '0' when used in a preprocessor expression.

A *pp-constant-expression* is an integral expression, evaluated at compile-time during preprocessing and formed from literal integer constants and the following operators:

Precedence	**Operator class**	**Operators**	**Associativity**
1 (highest)	parenthetical grouping	( )	NA
2	unary	defined + - ~ !	Right to Left

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

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

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

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

The semantics of applying operators in the preprocessor match those standard in the C++ preprocessor with the following exceptions:

- The 2nd operand in a logical and ('&&') operation is evaluated if and only if the 1st operand evaluates to non-zero.
- The 2nd operand in a logical or ('||') operation is evaluated if and only if the 1st operand evaluates to zero.
- There is no boolean type and no boolean literals. A *true* or *false* result is returned as integer *one* or *zero* respectively. Wherever a boolean operand is expected, any non-zero integer is interpreted as *true* and a zero integer as *false*.

If an operand is not evaluated, the presence of undefined identifiers in the operand will not cause an error.

#error will cause the implementation to put a compile-time diagnostic message into the shader object's information log (see section 7.12 "Shader, Program and Program Pipeline Queries" of the OpenGL ES 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 treat the presence of a **#error** directive as a compile-time error.

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

```
#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.

The scope as well as the effect of the optimize and debug pragmas is implementation-dependent except that their use must not generate an error. Incorrect use of predefined pragmas does not cause an error.

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

```
#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 a compile-time error on the **#extension** if the extension *extension_name* is not supported, or if **all** is specified.

Behavior	Effect
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.

For each extension there is an associated macro. The macro is always defined in an implementation that supports the extension. This allows the following construct to be used:

```
#ifdef OES_extension_name
    #extension OES_extension_name : enable
    // code that requires the extension
#else
    // alternative code
#endif
```

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

Note

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

If during macro expansion a preprocessor directive is encountered, the results are undefined; the compiler may or may not report an error in such cases.

3.5. Comments

Comments are delimited by /* and */, or by // and a new-line. // style comments include the initial // marker and continue up to, but not including, the terminating newline. /*…*/ comments include both the start and end marker. The begin comment delimiters (/* or //) are not recognized as comment delimiters inside of a comment, hence comments cannot be nested. Comments are treated syntactically as a single space.

3.6. Tokens

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

token :
  keyword
  identifier
  integer-constant
  floating-constant
  operator

; { }

3.7. Keywords

The following are the keywords in the language and (after preprocessing) can only be used as described in this specification, or an error results:

const uniform buffer shared

coherent volatile restrict readonly writeonly

atomic_uint

layout

centroid flat smooth

patch sample

invariant precise

break continue do for while switch case default

if else

in out inout

int void bool true false float

discard return

vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4

uint uvec2 uvec3 uvec4

mat2 mat3 mat4

mat2x2 mat2x3 mat2x4

mat3x2 mat3x3 mat3x4

mat4x2 mat4x3 mat4x4

lowp mediump highp precision

sampler2D sampler2DShadow sampler2DArray sampler2DArrayShadow

isampler2D isampler2DArray usampler2D usampler2DArray

sampler2DMS isampler2DMS usampler2DMS

sampler2DMSArray isampler2DMSArray usampler2DMSArray

sampler3D isampler3D usampler3D

samplerCube samplerCubeShadow isamplerCube usamplerCube

samplerCubeArray samplerCubeArrayShadow

isamplerCubeArray usamplerCubeArray

samplerBuffer isamplerBuffer usamplerBuffer

image2D iimage2D uimage2D

image2DArray iimage2DArray uimage2DArray

image3D iimage3D uimage3D

imageCube iimageCube uimageCube

imageCubeArray iimageCubeArray uimageCubeArray

imageBuffer iimageBuffer uimageBuffer

struct

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

attribute varying

noperspective

subroutine

common partition active

asm

class union enum typedef template this

resource

goto

inline noinline public static extern external interface

long short half fixed unsigned superp double

input output

hvec2 hvec3 hvec4 fvec2 fvec3 fvec4

dvec2 dvec3 dvec4

dmat2 dmat3 dmat4

dmat2x2 dmat2x3 dmat2x4

dmat3x2 dmat3x3 dmat3x4

dmat4x2 dmat4x3 dmat4x4

filter

sizeof cast

namespace using

sampler1D sampler1DShadow sampler1DArray sampler1DArrayShadow

isampler1D isampler1DArray usampler1D usampler1DArray

sampler2DRect sampler2DRectShadow isampler2DRect usampler2DRect

sampler3DRect

image1D iimage1D uimage1D

image1DArray iimage1DArray uimage1DArray

image2DRect iimage2DRect uimage2DRect

image2DMS iimage2DMS uimage2DMS

image2DMSArray iimage2DMSArray uimage2DMSArray

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

3.8. Identifiers

Identifiers are used for variable names, function names, structure names, and field selectors (field selectors select components of `vectors` and `matrices`, similarly to structure members). Identifiers have the form:

identifier :

  nondigit
  identifier nondigit
  identifier digit

nondigit : **one of**

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

digit : **one of**

  0 1 2 3 4 5 6 7 8 9

Identifiers starting with “gl_” are reserved for use by OpenGL ES, and in general, may not be declared in a shader; this results in an error. 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.

Implementations must support identifier lengths of up to 1024 characters. It is an error if the length exceeds this value.

3.9. Definitions

Some language rules described below depend on the following definitions.

3.9.1. Static Use

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

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

Other examples of non-uniform control flow 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 control flow.

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

3.9.3. 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.

3.10. Logical Phases of Compilation

The compilation units for the shader processors are processed separately before optionally being linked together in the final stage of compilation. The logical phases of compilation are:

1. Source strings are input as byte sequences. The value 'zero' is interpreted as a terminator.
2. Source strings are concatenated to form a single input. Zero bytes are discarded but all other values are retained.
3. Each string is interpreted according to the UTF-8 standard, with the exception that all invalid byte sequences are retained in their original form for subsequent processing.
4. Each {carriage-return, line-feed} and {line-feed, carriage return} sequence is replaced by a single newline. All remaining carriage-return and line-feed characters are then each replaced by a newline.
5. Line numbering for each character, which is equal to the number of preceding newlines plus one, is noted. Note this can only be subsequently changed by the #line directive and is not affected by the removal of newlines in phase 6 of compilation.
6. Wherever a backslash ('\') occurs immediately before a newline, both are deleted. Note that no

whitespace is substituted, thereby allowing a single preprocessing token to span a newline. This operation is not recursive; any new {backslash newline} sequences generated are not removed.

7. All comments are replaced with a single space. All (non-zero) characters and invalid UTF-8 byte sequences are allowed within comments. '//' style comments include the initial '//' marker and continue up to, but not including, the terminating newline. '/**...**/' comments include both the start and end marker.
8. The source string is converted into a sequence of preprocessing tokens. These tokens include preprocessing numbers, identifiers and preprocessing operations. The line number associated with each token is copied from the line number of the first character of the token.
9. The preprocessor is run. Directives are executed and macro expansion is performed.
10. White space and newlines are discarded.
11. Preprocessing tokens are converted into tokens.
12. The syntax is analyzed according to the GLSL ES grammar.
13. The result is checked according to the semantic rules of the language.
14. Optionally, the shaders are linked together to form one or more programs or separable programs. When a pair of shaders from consecutive stages are linked into the same program, any outputs and corresponding inputs not used in both shaders may be discarded.
15. The binary is generated.

Chapter 4. Variables and Types

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

There are no default types. All variable and function declarations must have a declared type, and optionally qualifiers. A variable is declared by specifying its type followed by one or more names separated by commas. In many cases, a variable can be initialized as part of its declaration by using the assignment operator (=).

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

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

4.1. Basic Types

Definition

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

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

4.1.1. Transparent Types

Type	Meaning
void	for functions that do not return a value
bool	a conditional type, taking on values of true or false
int	a signed integer
uint	an unsigned integer
float	a single-precision floating-point scalar
vec2	a two-component single-precision floating-point vector
vec3	a three-component single-precision floating-point vector
vec4	a four-component single-precision floating-point vector
bvec2	a two-component Boolean vector
bvec3	a three-component Boolean vector
bvec4	a four-component Boolean vector
ivec2	a two-component signed integer vector
ivec3	a three-component signed integer vector
ivec4	a four-component signed integer vector

Type	Meaning
uvec2	a two-component unsigned integer vector
uvec3	a three-component unsigned integer vector
uvec4	a four-component unsigned integer vector
mat2	a 2 × 2 single-precision floating-point matrix
mat3	a 3 × 3 single-precision floating-point matrix
mat4	a 4 × 4 single-precision floating-point matrix
mat2x2	same as a **mat2**
mat2x3	a single-precision floating-point matrix with 2 columns and 3 rows
mat2x4	a single-precision floating-point matrix with 2 columns and 4 rows
mat3x2	a single-precision floating-point matrix with 3 columns and 2 rows
mat3x3	same as a **mat3**
mat3x4	a single-precision floating-point matrix with 3 columns and 4 rows
mat4x2	a single-precision floating-point matrix with 4 columns and 2 rows
mat4x3	a single-precision floating-point matrix with 4 columns and 3 rows
mat4x4	same as a **mat4**

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

4.1.2. Floating-Point Opaque Types

Type	Meaning
sampler2D **image2D**	a handle for accessing a 2D texture
sampler2DShadow	a handle for accessing a 2D depth texture with comparison
sampler2DArray **image2DArray**	a handle for accessing a 2D array texture
sampler2DArrayShadow	a handle for accessing a 2D array depth texture with comparison
sampler2DMS	a handle for accessing a 2D multisample texture
sampler2DMSArray	a handle for accessing a 2D multisample array texture
sampler3D **image3D**	a handle for accessing a 3D texture

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

4.1.3. Signed Integer Opaque Types

Type	Meaning
isampler2D **iimage2D**	a handle for accessing an integer 2D texture
isampler2DArray **iimage2DArray**	a handle for accessing an integer 2D array texture
isampler2DMS	a handle for accessing an integer 2D multisample texture
isampler2DMSArray	a handle for accessing an integer 2D multisample array texture
isampler3D **iimage3D**	a handle for accessing an integer 3D texture
isamplerCube **iimageCube**	a handle for accessing an integer cube mapped texture
isamplerCubeArray **iimageCubeArray**	a handle for accessing an integer cube map array texture
isamplerBuffer **iimageBuffer**	a handle for accessing an integer buffer texture

4.1.4. Unsigned Integer Opaque Types

Type	Meaning
usampler2D **uimage2D**	a handle for accessing an unsigned integer 2D texture
usampler2DArray **uimage2DArray**	a handle for accessing an unsigned integer 2D array texture
usampler2DMS	a handle for accessing an unsigned integer 2D multisample texture
usampler2DMSArray	a handle for accessing an unsigned integer 2D multisample array texture

Type	Meaning
usampler3D **uimage3D**	a handle for accessing an unsigned integer 3D texture
usamplerCube **uimageCube**	a handle for accessing an unsigned integer cube mapped texture
usamplerCubeArray **uimageCubeArray**	a handle for accessing an unsigned integer cube map array texture
usamplerBuffer **uimageBuffer**	a handle for accessing an unsigned integer buffer texture
atomic_uint	a handle for accessing an unsigned integer atomic counter

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

There are no pointer types.

4.1.5. Void

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

4.1.6. Booleans

Definition

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

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

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

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

4.1.7. Integers

Definitions

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

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

A *vector integral type* is a vector of signed or unsigned integers:

Signed and unsigned integer variables are fully supported. In this document, the term *integer* is meant to generally include both signed and unsigned integers. **highp** unsigned integers have exactly 32 bits of precision. **highp** signed integers use 32 bits, including a sign bit, in two's complement form.

mediump and **lowp** integers have implementation-defined numbers of bits. See "Range and Precision" for details. For all precisions, operations resulting in overflow or underflow will not cause any exception, nor will they saturate, rather they will "wrap" to yield the low-order n bits of the result where n is the size in bits of the integer. However, for the case where the minimum representable value is divided by -1, it is allowed to return either the minimum representable value or the maximum representable value.

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

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

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

integer-constant **:**

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

integer-suffix **: one of**

u U

decimal-constant **:**

nonzero-digit
decimal-constant digit

octal-constant **:**

0
octal-constant octal-digit

hexadecimal-constant **:**

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

digit **:**

0
nonzero-digit

nonzero-digit : **one of**

1 2 3 4 5 6 7 8 9

octal-digit : **one of**

0 1 2 3 4 5 6 7

hexadecimal-digit : **one of**

0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

No white space is allowed between the digits of an integer constant, including after the leading **0** or after the leading **0x** or **0X** of a constant, or before the suffix **u** or **U**. When the suffix **u** or **U** is present, the literal has type **uint**, otherwise the type is **int**. A leading unary minus sign (-) is interpreted as an arithmetic unary negation, not as part of the constant. 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,

```
1             // OK. Signed integer, value 1
1u            // OK. Unsigned integer, value 1
-1            // OK. Unary minus applied to signed integer.
              // result is a signed integer, value -1
-1u           // OK. Unary minus applies to unsigned integer.
              // Result is an unsigned integer, value 0xffffffff
0xA0000000    // OK. 32-bit signed hexadecimal
0xABcdEF00u   // OK. 32-bit unsigned hexadecimal
0xffffffff    // OK. Signed integer, value -1
0x80000000    // OK. Evaluates to -2147483648
0xffffffffu   // OK. Unsigned integer, value 0xffffffff
0xfffffffff   // Error: needs more than 32 bits
3000000000    // OK. A signed decimal literal taking 32 bits.
              // It evaluates to -1294967296
2147483648    // OK. Evaluates to -2147483648 (the literal set the sign bit)
5000000000    // Error: needs more than 32 bits
```

4.1.8. Floats

Definition

A *floating-point type* is any floating-point scalar, vector or matrix type. It excludes arrays and structures.

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

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

As an input value to one of the processing units, a floating-point variable is expected to match the IEEE 754 single precision floating-point definition for precision and dynamic range. **highp** floating-point variables within a shader are 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 "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

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

There is no limit on the number of digits in any *digit-sequence*. If the value of the floating-point number is too large (small) to be stored as a single precision value, it is converted to positive (negative) infinity. A value with a magnitude too small to be represented as a mantissa and exponent is converted to zero. Implementations may also convert subnormal (denormalized) numbers to zero.

4.1.9. Vectors

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

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

Initialization of vectors can be done with constructors. See "Vector and Matrix Constructors".

4.1.10. Matrices

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

Initialization of matrix values is done with constructors (described in "Vector and Matrix Constructors") in column-major order.

mat2 is an alias for **mat2x2**, not a distinct type. Similarly for **mat3** and **mat4.** The following is legal:

```
mat2 a;
mat2x2 b = a;
```

4.1.11. Opaque Types

Definition

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

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

When aggregated into arrays within a shader, opaque types can only be indexed with a dynamically uniform integral expression (see “Dynamically Uniform Expressions”) unless otherwise noted; otherwise, results are undefined.

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

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

Samplers

Sampler types (e.g. **sampler2D**) are opaque types, declared and behaving as described above for opaque types.

Sampler variables are handles to 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 “Texture Functions”) and samplers are used to specify which texture to access and how it is to be filtered.

Images

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

Image variables are handles to two-, or three-dimensional images corresponding to all or a portion of a single level of a texture image bound to an image unit. There are distinct image variable types for each texture target, and for each of float, integer, and unsigned integer data types. Image accesses should use an image type that matches the target of the texture whose level is bound to the image unit, or for non-layered bindings of 3D or array images should use the image type that matches the dimensionality of the layer of the image (i.e., a layer of 3D, 2DArray, Cube, or CubeArray should use **image2D**). If the image target type does not match the bound image in this manner, if the data type does not match the bound image, or if the format layout qualifier does not match the image unit format as described in section 8.22 “Texture Image Loads and Stores” of the OpenGL ES 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 “Image Functions” to specify an image to access.

Atomic Counters

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

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

Members of structures cannot be declared as atomic counter types.

The default precision of all atomic types is **highp**. It is an error to declare an atomic type with a different precision or to specify the default precision for an atomic type to be **lowp** or **mediump**.

4.1.12. Structures

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

```
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 definitive grammar is as given in "Shading Language Grammar".

struct-definition :

  qualifier$_{opt}$ **struct** name$_{\mathrm{opt}}$_ **{** *member-list* **}** *declarators*$_{opt}$;

member-list :

  member-declaration ;
  member-declaration member-list ;

member-declaration :

  basic-type declarators ;

where *name* becomes the user-defined type, and can be used to declare variables to be of this new type. The *name* shares the same name space as other variables, types, and functions. All previously visible variables, types, constructors, or functions with that name are hidden. The optional *qualifier* only applies to any *declarators*, and is not part of the type being defined for *name*.

Structures must have at least one member declaration. Member declarators may contain precision qualifiers, but use of any other qualifier results in an error. 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 a constant integral expression that's greater than zero (see "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; }; // Allowed: S is defined as a structure.

struct T {
    S;              // Error: anonymous structures disallowed
    struct { ... }; // Error: embedded structures disallowed
    S s;            // Allowed: nested structure with a name.
};
```

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

Structures can contain variables of any type except:

- **atomic_uint** (since there is no mechanism to specify the binding)
- image types (since there is no mechanism to specify the format qualifier)

4.1.13. Arrays

Variables of the same type can be aggregated into arrays by declaring a name followed by brackets (**[ ]**) enclosing an optional size.

When present, the array size must be a constant integral expression (see "Constant Expressions") greater than zero. The type of the size parameter can be a signed or unsigned integer and the choice of type does not affect the type of the resulting array. Arrays only have a single dimension (a single number within "[ ]"), however, arrays of arrays can be declared. Any type can be formed into an array.

Arrays are sized either at compile-time or at run-time. To size an array at compile-time, either the size must be specified within the brackets as above or must be inferred from the type of the initializer.

If an array is declared as the last member of a shader storage block and the size is not specified at compile-time, it is sized at run-time. In all other cases, arrays are sized only at compile-time. An array declaration sized at compile-time which leaves the size of the array unspecified is an error.

For compile-time sized arrays, it is illegal to index an array with a constant integral expression greater than or equal to the declared size or with a negative constant expression. Arrays declared as formal parameters in a function declaration must also 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. If robust buffer access is enabled (see section 10.3.5 "Robust Buffer Access" of the OpenGL ES Specification), such indexing must not result in abnormal program termination. The results are still undefined, but implementations are encouraged to produce zero values for such accesses.

Some examples are:

```
float frequencies[3];
uniform vec4 lightPosition[4u];
const int numLights = 2;
light lights[numLights];
vec4 a[3][2];
a.length() // this is 3
a[x].length() // this is 2
// a shader storage block, introduced in section 4.3.7 "Buffer Variables"
buffer b {
    float u[]; // an error
    vec4 v[];  // okay, v will be sized at run-time
} name[3];     // when the block is arrayed, all u will be the same size,
               // but not necessarily all v, if sized dynamically
```

When the **length**() method will return a compile-time constant, the expression in brackets (x above) will be evaluated and subject to the rules required for array indices, but the array will not be dereferenced. Thus, behavior is well defined even if the run-time value of the expression is out of bounds.

An array type can be formed by specifying a non-array type ([type_specifier_nonarray]) followed by an [array_specifier].

```
float[5]
```

Note that the construct *type [size]* does not always result in an array of length *size* of type *type*:

```
float[2][3] // an array of size [2] of array of size [3] of float,
            // not size [3] of float[2]
```

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;
```

An array type can also be formed without specifying a size if the definition includes an initializer:

```
float x[] = float[2] (1.0, 2.0); // declares an array of size 2
float y[] = float[] (1.0, 2.0, 3.0); // declares an array of size 3
float a[5];
float b[] = a;
```

Note that the initializer itself does not need to be a constant expression but the length of the initializer will be a constant expression.

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
```

An array of arrays can be declared as:

```
vec4 a[3][2]; // size-3 array of size-2 array of vec4
```

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

```
vec4[2] a[3]; // size-3 array of size-2 array of vec4
vec4[3][2] a; // size-3 array of size-2 array of vec4
```

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

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

The last member of a shader storage block (see "Buffer Variables"), may be declared without specifying a size. For such arrays, the effective array size is inferred at run-time from the size of the data store backing the shader storage block. Such runtime-sized arrays may be indexed with general integer expressions, but may not be passed as an argument to a function or indexed with a negative constant expression.

```
struct S { float f; };
buffer ShaderStorageBlock1
{
    vec4 a[]; // illegal
    vec4 b[]; // legal, runtime-sized arrays are last member
};
buffer ShaderStorageBlock2
{
    vec4 a[4]; // legal, size declared
    S b[]; // legal, runtime-sized arrays are allowed,
    // including arrays of structures
};
```

However, it is a compile-time error to assign to a runtime-sized array. Assignments to individual elements of the array is allowed.

Arrays have a fixed number of elements. This can be obtained by using the **length**() method:

```
float a[5];
a.length(); // returns 5
```

The return value is a signed integral expression. For compile-time sized arrays, the value returned by the length method is a constant expression. For run-time sized arrays , the value returned will not be constant expression and will be determined at run time based on the size of the buffer object providing storage for the block.

The precision is determined using the same rules as for other cases where there is no intrinsic precision. See “Precision Qualifiers”.

Any restrictions on the usage of a type also apply to arrays of that type. This applies recursively.

4.2. Scoping

The scope of a declaration determines where the declaration is visible. GLSL ES uses a system of statically nested scopes. This allows names to be redefined within a shader.

4.2.1. Definition of Terms

The term *scope* refers to a specified region of the program where names are guaranteed to be visible. For example, a *compound_statement_with_scope* ('{' *statement statement* … '}') defines a scope.

A *nested scope* is a scope defined within an outer scope.

The terms '*same scope*' and '*current scope*' are equivalent to the term '*scope*' but used to emphasize that nested scopes are excluded.

The *scope of a declaration* is the region or regions of the program where that declaration is visible.

A *name space* defines where names may be defined. Within a single name space, a name has at most one entry, specifying it to be one of: structure, variable, or function.

In general, each scope has an associated name space. However, in certain cases e.g. for uniforms, multiple scopes share the same name space. In these cases, conflicting declarations are an error, even though the name is only visible in the scopes where it is declared.

4.2.2. Types of Scope

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 (specified as *statement-no-new-scope* in the grammar). 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's parameter declarations and body together form a single scope.

```
int f( /* nested scope begins here */ int k)
{
    int k = k + 3; // redeclaration error of the name k
    ...
}
int f(int k)
{
    {
       int k = k + 3; // 2nd k is parameter, initializing nested first k
       int m = k // use of new k, which is hiding the parameter
    }
}
```

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

```
for ( /* nested scope begins here */ int i = 0; i < 10; i++)
{
    int i; // redeclaration error
}
```

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

```
int _i_ = 17;
do
    int i = 4; // okay, in nested scope
while (i == 0); // i is 17, scoped outside the do-while body
```

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

Representing the if construct as:

if if-expression **then** if-statement **else** else-statement,

a variable declared in the if-statement is scoped to the end of the if-statement. A variable declared in the else-statement is scoped to the end of the else-statement. This applies both when these statements are simple statements and when they are compound statements. The if-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,/* 2nd x visible here */ y = x; // y is initialized to 2
    int z = z; // error if z not previously defined.
}
{
int x = x; // x is initialized to '1'
}
```

A structure name declaration is visible at the end of the *struct_specifier* in which it was declared:

```
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.
```

4.2.3. Redeclaring Names

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. Otherwise, within a shader, 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.

Names of built-in functions cannot be redeclared as functions. Therefore overloading or redefining built-in functions is an error.

A *declaration* is considered to be a statement that adds a name or signature to the symbol table. A *definition* is a statement that fully defines that name or signature. E.g.

```
int f();// declaration;
int f() {return 0;}// declaration and definition
int x; // declaration and definition
int a[4];// array declaration and definition
struct S {int x;};// structure declaration and definition
```

The determination of equivalence of two declarations depends on the type of declaration. For functions, the whole function signature must be considered (see “Function Definitions”). For variables (including arrays) and structures only the names must match.

Within each scope, a name may be declared either as a variable declaration *or* as function declarations *or* as a structure.

Examples of combinations that are allowed:

1.

```
void f(int) {...}
void f(float) {...}// function overloading allowed
```

2.

```
void f(int);// 1^st^ declaration (allowed)
void f(int);// repeated declaration (allowed)
void f(int) {...}// single definition (allowed)
```

Examples of combinations that are disallowed:

1.

```
void f(int) {...}
void f(int) {...}// Error: repeated definition
```

2.

```
void f(int);
struct f {int x;};// Error: type 'f' conflicts with function 'f'
```

3.

```
struct f {int x;};
int f;// Error: conflicts with the type 'f'
```

4.

```
int a[3];
int a[3];// Error: repeated array definition
```

5.

```
int x;
int x;// Error: repeated variable definition
```

4.2.4. Global Scope

The built-in functions are scoped in the global scope users declare global variables in. That is, a shader's global scope, available for user-defined functions and global variables, is the same as the scope containing the built-in functions. Function declarations (prototypes) cannot occur inside of functions; they must be at global scope. Hence it is not possible to hide a name with a function.

4.2.5. Shared Globals

Shared globals are variables that can be accessed by multiple compilation units. In GLSL ES the only shared globals are uniforms. Vertex shader outputs are not considered to be shared globals since they must pass through the rasterization stage before they are used as input by the fragment shader.

Shared globals share the same name space, and must be declared with the same type and precision. They will share the same storage. Shared global arrays must have the same base type and the same explicit size. Scalars must have exactly the same precision, 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.

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 compile-time constant

Storage Qualifier	Meaning
in	linkage into a shader from a previous stage, variable is copied in
out	linkage out of a shader to a subsequent stage, variable is copied out
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL ES, and the application
buffer	value is stored in a buffer object, and can be read or written both by shader invocations and the OpenGL ES API
shared	compute shader only; variable storage is shared across all work items in a workgroup

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

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

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

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

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

In declarations of global variables with no storage qualifier or with a const qualifier, any initializer must be a constant expression. Declarations of global variables with other storage qualifiers may not contain initializers. Global variables without storage qualifiers that are not initialized in their declaration or by the application will not be initialized by OpenGL ES, but rather will enter *main()* with undefined values.

4.3.1. Default Storage Qualifier

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

4.3.2. Constant Qualifier

Named compile-time constants can be declared using the **const** qualifier. Any variables qualified as constant are read-only variables for that shader. Declaring variables as constant allows more descriptive shaders than using hard-wired numerical constants. The **const** qualifier can be used with any of the 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);
```

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

Initializers for **const** declarations must be constant expressions, as defined in "Constant Expressions".

4.3.3. Constant Expressions

A *constant expression* is one of

- A literal value (e.g. **5** or **true**).
- A global or local variable qualified as **const** (i.e., not including function parameters).
- An expression formed by an operator on operands that are all constant expressions, including getting an element 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.
- The **length**() method on a compile-time sized array, 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. This rule excludes functions with a **void** return or functions that have an **out** parameter. 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.

Scalar, vector, matrix, array and structure variables are constant expressions if qualified as **const**. Opaque types cannot be constant expressions.

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

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

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

Constant expressions may be evaluated by the compiler's host platform, and are therefore not required to compute the same value that the same expression would evaluate to on the shader execution target. However, the host must use the same or greater precision than the target would use. When the precision qualification cannot be determined, the expression is evaluated at **highp**. See "Default Precision Qualifiers".

4.3.4. Input Variables

Shader input variables are declared with the **in** storage qualifier. They form the input interface between previous stages of the OpenGL ES pipeline and the declaring shader. Input variables must be declared at global scope. Values from the previous pipeline stage are copied into input variables at the beginning of shader execution. It is an error to write to a variable declared as an input.

Only the input variables that are actually read need to be written by the previous stage; it is allowed to have superfluous declarations of input variables.

See "Built-In Variables" for a list of the built-in input names.

Vertex shader input variables (or attributes) receive per-vertex data. It is an error to use auxiliary storage or interpolation qualifiers on a vertex shader input. The values copied in are established by the OpenGL ES API or through the use of the layout identifier **location**.

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

- A boolean type
- An opaque type
- An array
- A structure

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

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

```
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. For geometry shaders, the array size will be set by, (or if provided must be consistent with) the input **layout** declaration(s) establishing the type of input primitive, as described later in "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 have matching types, except that the geometry shader will have one more array dimension than the vertex shader, to allow for vertex indexing. If such an arrayed interface variable is not declared with the necessary additional input or output array dimension, a link-time error will result. Geometry shader inputs, tessellation control shader inputs and outputs, and tessellation evaluation inputs all have an additional level of arrayness relative to other shader inputs and outputs. These inputs and outputs are known as *per-vertex-arrayed* inputs and outputs. Component limits for arrayed interfaces (e.g. *gl_MaxTessControlInputComponents*) are limits per vertex, not limits for the entire interface.

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

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

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

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

- A boolean type
- An opaque type
- A structure containing an array
- A structure containing a structure
- For per-vertex-arrayed variables:
    - Per-vertex-arrayed arrays of arrays
    - Per-vertex-arrayed arrays of structures
- For non-per-vertex-arrayed variables:
    - An array of arrays
    - An array of structures

Fragment shader inputs get per-fragment values, typically interpolated from a previous stage's outputs.

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

- A boolean type
- An opaque type
- An array of arrays
- An array of structures
- A structure containing an array
- A structure containing a structure

Fragment shader inputs that are, or contain, integral types must be qualified with the interpolation qualifier **flat**.

Fragment inputs are declared as in the following examples:

```
in vec3 normal;
centroid in vec2 TexCoord;
flat in vec3 myColor;
```

The fragment shader inputs form an interface with the last active shader in the vertex processing pipeline. For this interface, the last active shader stage output variables and fragment shader input variables of the same name must match in type and qualification, with a few exceptions: The storage qualifiers must, of course, differ (one is **in** and one is **out**). Also, auxiliary qualification (e.g. **centroid**) may differ. When auxiliary qualifiers do not match, those provided in the fragment shader supersede those provided in previous stages. If any such qualifiers are completely missing in the fragment shaders, then the default is used, rather than any qualifiers that may have been declared in previous stages. That is, what matters is what is declared in the fragment shaders, not what is declared in shaders in previous stages.

When an interface between shader stages is formed using shaders from two separate program objects, it is not possible to detect mismatches between inputs and outputs when the programs are linked. When there are mismatches between inputs and outputs on such interfaces, attempting to use the two programs in the same program pipeline will result in program pipeline validation failures, as described in section 7.4.1 "Shader Interface Matching" of the OpenGL ES Specification.

Shaders can ensure matches across such interfaces either by using input and output layout qualifiers (sections "Input Layout Qualifiers" and "Output Layout Qualifiers") or by using identical input and output declarations of blocks or variables. Complete rules for interface matching are found in section 7.4.1 "Shader Interface Matching" of the OpenGL ES Specification.

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

4.3.5. Uniform Variables

The **uniform** qualifier is used to declare global variables whose values are the same across the entire primitive being processed. All **uniform** variables are read-only. Except for variables declared within a uniform block, all uniform variables are initialized to 0 at link time and may be updated through the API.

Example declarations are:

```
uniform vec4 lightPosition;
```

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

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

Uniforms in shaders all share a single global name space when linked into a program or separable program. Hence, the types, precisions, and any location specifiers of all statically used uniform variables with the same name must match across all shaders that are linked into a single program. However it is not required to repeat the location specifier in all the linked shaders. While this single uniform name space is cross stage, a uniform variable name's scope is per stage: If a uniform variable name is declared in one stage (e.g. a vertex shader) but not in another (e.g. a fragment shader), then that name is still available in the other stage for a different use.

A compile or link-time error is generated if any of the explicitly given or compiler generated uniform locations is greater than the implementation-defined maximum number of uniform locations minus one.

Unlike locations for inputs and outputs, uniform locations are logical values, not register locations, and there is no concept of overlap. For example:

```
layout(location = 2) uniform mat4 x;
layout(location = 3) uniform mat4 y; // No overlap with x
layout(location = 2) in mat4 x;
layout(location = 3) in mat4 y; // Error, locations conflict with x
```

4.3.6. Output Variables

Shader output variables are declared with the **out** storage qualifier. They form the output interface between the declaring shader and the subsequent stages of the OpenGL ES pipeline. Output variables must be declared at global scope. During shader execution they will behave as normal unqualified global variables. Their values are copied out to the subsequent pipeline stage on shader exit. Only output variables that are read by the subsequent pipeline stage need to be written; it is allowed to have superfluous declarations of output variables.

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

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

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

- A boolean type
- An opaque type
- A structure containing an array
- A structure containing a structure
- For per-vertex-arrayed variables (applies to tessellation control, tessellation evaluation and geometry shaders):
  - Per-vertex-arrayed arrays of arrays
  - Per-vertex-arrayed arrays of structures
- For non-per-vertex-arrayed variables:
  - An array of arrays
  - An array of structures

Vertex shader outputs may be qualified with the interpolation qualifier **flat**[1].

1

Unlike previous versions of the OpenGL ES Shading Language, there is no requirement for outputs containing integers to be qualified as **flat**, since the vertex shader may interface with the tessellation control shader. However, in all cases, the qualifier must match across interfaces.

Individual outputs are declared as in the following examples:

```
out vec3 normal;
centroid out vec2 TexCoord;
invariant centroid out vec4 Color;
flat out vec3 myColor;
sample out vec4 perSampleColor;
```

These can also appear in interface blocks, as described in "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 used to output per-vertex and per-patch data. Per-vertex output variables are arrayed (see *arrayed* under "Input Variables") and declared using the **out** qualifier without the **patch** qualifier. Per-patch output variables are declared using the **patch** and **out** qualifiers.

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

```
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 "Tessellation Control Outputs".

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

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

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

1. At the beginning of execution.
2. At each synchronization point, unless
   - the value was well-defined after the previous synchronization point and was not written by any invocation since, or
   - the value was written by exactly one shader invocation since the previous synchronization point, or
   - the value was written by multiple shader invocations since the previous synchronization point, and the last write performed by all such invocations wrote the same value.
3. When read by a shader invocation, if
   - the value was undefined at the previous synchronization point and has not been written 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 in a fragment shader output declaration. It is a compile-time error to declare a fragment shader output with, or that contains, any of the following types:

- A boolean type
- An opaque type
- A matrix type
- A structure
- An array of arrays

Fragment shader outputs declared as arrays may only be indexed by a constant integral expression.

Fragment outputs are declared as in the following examples:

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

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

4.3.7. Buffer Variables

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

The **buffer** qualifier can be used to declare interface blocks (see "Interface Blocks"), which are then referred to as shader storage blocks. It is a compile-time error to declare buffer variables outside a block. Buffer variables cannot have initializers.

Note

The terms *shader storage buffer (object)*, and *shader storage block* are often used interchangeably. The former generally refers to the underlying storage whereas the latter refers only to the interface definition in the shader. Similarly for *uniform buffer objects* and _uniform blocks.

```
// use buffer to create a buffer block (shader storage block)
buffer BufferName { // externally visible name of buffer
    int count;      // typed, shared memory...
    ...             // ...
    vec4 v[];       // last member may be an array that is not sized
                    // until after link time (dynamically sized)
} Name;             // name of block within the shader
```

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

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

Precision qualifiers for such variables need not match.

4.3.8. Shared Variables

The **shared** qualifier is used to declare global variables that have storage shared between all work items in a compute shader workgroup. Variables declared as **shared** may only be used in compute shaders (see "Compute Processor"). Any other declaration of a **shared** variable is an error. Shared variables are implicitly coherent (see "Memory Qualifiers").

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

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

In order to achieve ordering with respect to reads and writes to **shared** variables, control flow barriers must be employed using the **barrier**() function (see "Shader Invocation Control Functions").

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

4.3.9. Interface Blocks

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

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

interface-block **:**

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

interface-qualifier **:**

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

member-list **:**

  member-declaration
  member-declaration member-list

member-declaration **:**

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

instance-name **:**

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

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

First, an example,

```
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, uniform, and buffer variable declarations outside blocks, with these exceptions:

- Opaque types are not allowed
- Structure definitions cannot be nested inside a block
- Arrays of arrays of blocks are not allowed, except for the case in the tessellation pipe where the declaration is a per-vertex-array of arrays of blocks.

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

Repeating the **in**, **out**, **patch**, **uniform**, or **buffer** interface qualifier for a 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
};
```

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

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

values passed to the fragment shader first pass through the rasterizer and interpolator.

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

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

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

```
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 shader interfaces, not to shaders). If there is no instance name, then the API does not use the block name to access a member, just the member name.

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

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

```
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"
    float Color2;
};

// in same program as Vertex2 above:
out Vertex3 {
    float Intensity;
    vec4 Color;     // ERROR, name collision with Color in Vertex2
};
float Color2;       // ERROR, collides with Color2 in Vertex2
```

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

When using OpenGL ES API entry points to identify the name of an individual block in an array of blocks, the name string may include an array index (e.g. *Transform[2]*). When using OpenGL ES 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*). See section 7.3.1 “Program Interfaces” of the OpenGL ES Specification for details.

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

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

4.4. Layout Qualifiers

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

layout-qualifier interface-qualifier **;**

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

layout-qualifier interface-qualifier declaration **;**

Declarations of layouts can only be made at global scope or block members, and only where indicated in the following subsections; their details are specific to what the interface qualifier is, and are discussed individually.

The *layout-qualifier* expands to:

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

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

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

layout-qualifier-value **:**
  integer-constant

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

The set of allowed layout qualifiers depends on the shader, the interface and the variable type as specified in the following sections. For example, a sampler in the default uniform block in a fragment shader can have **location** and **binding** layout qualifiers but no others. Invalid use of layout qualifiers is an error.

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

Layout Qualifier	Qualifier Only	Individual Variable	Block	Block Member	Allowed Interfaces
shared **packed** **std140** **std430**	X		X		**uniform** / **buffer**
row_major **column_major**	X		X	X	
binding =		opaque types only	X		
offset =		atomic counters only			
location =		X			**uniform** / **buffer**
location =		X[1]	X	X	all **in** / **out**, except for compute
triangles **quads** **isolines**	X				tessellation evaluation **in**
equal_spacing **fractional_even_spacing** **fractional_odd_spacing**	X				tessellation evaluation **in**
cw **ccw**	X				tessellation evaluation **in**
point_mode	X				tessellation evaluation **in**
points	X				geometry **in** /**out**
[**points**] **lines** **lines_adjacency** **triangles** **triangles_adjacency**	X				geometry **in**

Layout Qualifier	Qualifier Only	Individual Variable	Block	Block Member	Allowed Interfaces
invocations =	X				geometry **in**
early_fragment_tests	X				fragment **in**
local_size_x = **local_size_y** = **local_size_z** =	X				compute **in**
vertices =	X				tessellation control **out**
[**points**] **line_strip** **triangle_strip**	X				geometry **out**
max_vertices =	X				geometry **out**
rgba32f **rgba16f** **r32f** **rgba8** **rgba8_snorm** **rgba32i** **rgba16i** **rgba8i** **r32i** **rgba32ui** **rgba16ui** **rgba8ui** **r32ui**		image types only			**uniform**

Layout Qualifier	Qualifier Only	Individual Variable	Block	Block Member	Allowed Interfaces
blend_support_multiply **blend_support_screen** **blend_support_overlay** **blend_support_darken** **blend_support_lighten** **blend_support_colordodge** **blend_support_colorburn** **blend_support_hardlight** **blend_support_softlight** **blend_support_difference** **blend_support_exclusion** **blend_support_hsl_hue** **blend_support_hsl_saturation** **blend_support_hsl_color** **blend_support_hsl_luminosity** **blend_support_all_equations**	X				fragment **out**

1

Location qualifiers are not allowed for members of an arrayed block, except for per-vertex-arrays (see "Interface Blocks").

4.4.1. Input Layout Qualifiers

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

All shaders except compute shaders allow **location** layout qualifiers on input variable declarations, input block declarations, and input block member declarations.

layout-qualifier-id **:**

location = *layout-qualifier-value*

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

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

If a shader input is any scalar or vector type, it will consume a single location.

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

```
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 $n \times m$ matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an n -element array of m-component vectors. For example,

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

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

If a block has no block-level **location** layout qualifier, it is required that either all or none of its members have a **location** layout qualifier, or a compile-time error results.

If an input is declared as an array of blocks, excluding per-vertex-arrays as required for tessellation, it is an error to declare a member of the block with a **location** qualifier.

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

```
layout(location = 3) in struct S
{
    vec3 a;                      // gets location 3
    mat2 b;                      // gets locations 4 and 5
    vec4 c[2];                   // gets locations 6 and 7
    layout(location = 8) vec2 A; // ERROR, can't use on struct member
} s;
layout(location = 4) in block
{
    vec4 d;                      // gets location 4
    vec4 e;                      // gets location 5
    layout(location = 7) vec4 f; // gets location 7
    vec4 g;                      // gets location 8
    layout(location = 1) vec4 h; // gets location 1
    vec4 i;                      // gets location 2
    vec4 j;                      // gets location 3
    vec4 k;                      // ERROR, location 4 already used
};
```

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

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

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

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

If a vertex shader input variable with no location assigned in the shader text has a location specified through the OpenGL ES API, the API-assigned location will be used. Otherwise, such variables will be assigned a location by the linker. See section 11.1.1 “Vertex Attributes” of the OpenGL ES Specification for more details.

It is an error if more than one input or element of a matrix input is bound to the same location.

Tessellation Evaluation Inputs

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

layout-qualifier-id :

primitive_mode
vertex_spacing
ordering
point_mode

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

primitive-mode:

triangles
quads
isolines

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

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

vertex-spacing:

equal_spacing
fractional_even_spacing
fractional_odd_spacing

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

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

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

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

ordering:

cw
ccw

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

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

point-mode:

point_mode

Any or all of these identifiers may be specified one or more times in a single input layout declaration.

The tessellation evaluation shader object in a program must declare a primitive mode in its input layout. Declaring vertex spacing, ordering, or point mode identifiers is optional. If spacing or vertex ordering declarations are omitted, the tessellation primitive generator will use equal spacing or counter-clockwise vertex ordering, respectively. If a point mode declaration is omitted, the tessellation primitive generator will produce lines or triangles according to the primitive mode.

Geometry Shader Inputs

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

layout-qualifier-id :

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

The identifiers **points**, **lines**, **lines_adjacency**, **triangles**, and **triangles_adjacency** are used to specify the type of input primitive accepted by the geometry shader, and only one of these is accepted. The geometry shader must declare this 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 the geometry shader, it will be run once for each input primitive. If an invocation count is declared, all such declarations must specify the same count. If a shader specifies an invocation count greater than the implementation-dependent maximum, or less than or equal to zero, a compile-time error results.

For example,

```
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.

An input can be declared without an array size if there is a previous layout which specifies the size. For built-in inputs (e.g. *gl_in[]*), a layout must be declared before any use.

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

```
// code sequence within one shader...
in vec4 Color2[2];    // legal, size is 2
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
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 in the geometry shader of a program.

Fragment Shader Inputs

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

layout-qualifier-id :
    early_fragment_tests

to request that fragment tests be performed before fragment shader execution, as described in section 13.8.4 "Early Fragment Tests" of the OpenGL ES Specification.

For example,

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

Specifying this will make per-fragment tests be performed before fragment shader execution. In addition it is an error to statically write to *gl_FragDepth* in the fragment shader. If this is not declared, per-fragment tests will be performed after fragment shader execution.

Compute Shader Inputs

There are no layout location qualifiers for compute shader inputs.

Layout qualifier identifiers for compute shader inputs are the workgroup size qualifiers:

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

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

For example, the following declaration in a compute shader

```
layout(local_size_x = 32, local_size_y = 32) in;
```

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

As another example, the declaration

```
layout(local_size_x = 8) in;
```

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

If the fixed workgroup size of the shader in any dimension is less than or equal to zero or greater than the maximum size supported by the implementation for that dimension, a compile-time error results. Also, if such a layout qualifier is declared more than once in the same shader, all those declarations must set the same set of workgroup sizes and set them to the same values; otherwise a compile-time error results.

Furthermore, if a program object contains a compute shader, that shader must contain an input layout qualifier specifying a fixed workgroup size for the program, or a link-time error will occur.

4.4.2. Output Layout Qualifiers

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

As with input layout qualifiers, all shaders except compute shaders allow **location** layout qualifiers on output variable declarations, output block declarations, and output block member declarations.

layout-qualifier-id **:**
  location = *layout-qualifier-value*

The usage and rules for applying the **location** qualifier to blocks and structures are exactly as described in “Input Layout Qualifiers”.

The qualifier may appear at most once within a declaration. 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.

For fragment shader outputs, the location specifies the color output number 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.

Declared outputs of scalar or vector type consume a single location.

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 $n \times m$ matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned will be the same as for an n-element array of m -component vectors.

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

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

If an output is declared as an array of blocks, excluding per-vertex-arrays as required for tessellation, it is an error to declare a member of the block with a **location** qualifier.

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

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

Compile-time errors may also be given if at compile time it is known the link will fail. A negative output location will result in an error.

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

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

If an output variable has no location assigned in the shader text, it will be assigned a location by the linker. See section 11.1.3 “Shader Execution” of the OpenGL ES Specification for more details.

If there is only a single output, the location does not need to be specified, in which case it defaults to zero. This applies for all output types, including arrays. If there is more than one fragment output, the location must be specified for all outputs.

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.

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 are:

layout-qualifier-id **:**
  vertices = *layout-qualifier-value*

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

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

```
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 declared before any use of the method **length**() or other array use that requires its size to be known.

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

All tessellation control shader layout declarations in a program must specify the same output patch vertex count. There must be at least one layout qualifier specifying an output patch vertex count in any program containing a tessellation control shader.

Geometry Outputs

Geometry shaders can have two additional types of output layout identifiers: an output primitive type and a maximum output vertex count. 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 layout qualifier identifiers for geometry shader outputs are

layout-qualifier-id :

  points
  line_strip
  triangle_strip
  max_vertices = *layout-qualifier-value*

The primitive type identifiers **points**, **line_strip**, and **triangle_strip** are used to specify the type of output primitive produced by the geometry shader, and only one of these is accepted. The geometry shader object 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.

The vertex count identifier **max_vertices** is used to specify the maximum number of vertices the shader will ever emit in a single invocation. The geometry shader object 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.

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*.

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.

Fragment Outputs

Fragment shaders can have an output layout for redeclaring the built-in variable *gl_FragDepth*:

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

The built-in *gl_FragDepth* is only predeclared in fragment shaders, so redeclaring it in any other shader language results in an error.

Fragment shaders additionally support layout qualifiers specifying a set of advanced blend equations supported when the fragment shader is used. These layout qualifiers are only permitted on the interface qualifier **out**, and use the identifiers specified in the “Layout Qualifier” column of the table below.

If a layout qualifier in the table below is specified in the fragment shader, the fragment shader may be used with the corresponding advanced blend equation in the “Blend Equation(s) Supported” column. Additionally, the special qualifier **blend_support_all_equations** indicates that the shader may be used with any advanced blending equation supported by the OpenGL ES Specification. It is not an error to specify more than one of these identifiers in any fragment shader. Specifying more than one qualifier or **blend_support_all_equations** means that the fragment shader may be used with multiple advanced blend equations. Additionally, it is not an error to specify any single one of these layout qualifiers more than once.

Layout Qualifier	Blend Equations(s) Supported
blend_support_multiply	MULTIPLY
blend_support_screen	SCREEN
blend_support_overlay	OVERLAY
blend_support_darken	DARKEN
blend_support_lighten	LIGHTEN
blend_support_colordodge	COLORDODGE
blend_support_colorburn	COLORBURN
blend_support_hardlight	HARDLIGHT
blend_support_softlight	SOFTLIGHT
blend_support_difference	DIFFERENCE
blend_support_exclusion	EXCLUSION
blend_support_hsl_hue	HSL_HUE
blend_support_hsl_saturation	HSL_SATURATION
blend_support_hsl_color	HSL_COLOR
blend_support_hsl_luminosity	HSL_LUMINOSITY
blend_support_all_equations	all blend equations

4.4.3. Uniform Variable Layout Qualifiers

The following layout qualifier can be used for all default-block uniform variables but not for variables in uniform or shader storage blocks. The layout qualifier identifier for uniform variables is:

layout-qualifier-id **:**

  location = *layout-qualifier-value*

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

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

4.4.4. Uniform and Shader Storage Block Layout Qualifiers

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

layout-qualifier-id **:**

  shared
  packed
  std140
  std430
  row_major
  column_major
  binding = *layout-qualifier-value*

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

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

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

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

and for shader storage blocks as:

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

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

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

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

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

The **shared** qualifier overrides only the **std140**, **std430**, and **packed** qualifiers; other qualifiers are inherited. The compiler/linker will ensure that multiple programs and programmable stages containing this definition will share the same memory layout for this block, as long as 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**, **std430**, and **shared**; other qualifiers are inherited. When **packed** is used, no shareable layout is guaranteed. The compiler and linker can optimize memory use based on what variables actively get used and on other criteria. Offsets must be queried, as there is no other way of guaranteeing where (and which) variables reside within the block.

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

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

The layout is explicitly determined by this, as described in section 7.6.2.2 “Standard Uniform Block Layout” of the OpenGL ES Specification. Hence, as in **shared** above, the resulting layout is shareable across programs.

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

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

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

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

The **binding** qualifier specifies the uniform buffer binding point corresponding to the uniform or shader storage block, which will be used to obtain the values of the member variables of the block. It is a compile-time error to specify the **binding** qualifier for the global scope or for block member declarations. Any uniform or shader storage block declared without a **binding** qualifier is initially assigned to block binding point zero. After a program is linked, the binding points used for uniform (but not shader storage) blocks declared with or without a **binding** qualifier can be updated by the OpenGL ES API.

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

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

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

For example

```
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
};
```

4.4.5. Opaque Uniform Layout Qualifiers

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

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

layout-qualifier-id **:**

  binding = *layout-qualifier-value*

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

If the **binding** qualifier is used with an array, the first element of the array takes the specified unit and each subsequent element takes the next consecutive unit. For an array of arrays, each element (e.g. 6 elements for a[2][3]) gets a binding point, and they are ordered per the array of array ordering described in "Arrays."

If the **binding** is less than zero, or greater than or equal to the implementation-dependent maximum supported number of units, a compile-time error will occur. When the **binding** qualifier is used with an array of size *N*, all elements of the array from **binding** through *binding* + *N* - *1* must be within this range.

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

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

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

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

4.4.6. Atomic Counter Layout Qualifiers

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

layout-qualifier-id :

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

For example,

```
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*.

4.4.7. 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 = *layout-qualifier-value*

float-image-format-qualifier :

rgba32f
rgba16f
r32f
rgba8
rgba8_snorm

int-image-format-qualifier :

rgba32i
rgba16i
rgba8i
r32i

uint-image-format-qualifier :

rgba32ui
rgba16ui
rgba8ui
r32ui

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 must specify a format layout qualifier.

The **binding** qualifier was described in “Opaque Uniform Layout Qualifiers”.

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

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

A variable qualified as **flat** will not be interpolated. Instead, it will have the same value for every fragment within a primitive. This value will come from a single provoking vertex, as described by the OpenGL ES Specification. A variable qualified as **flat** may also be qualified as **centroid** or **sample**, which will mean the same thing as qualifying it only as **flat**.

A variable qualified as **smooth** will be interpolated in a perspective-correct manner over the primitive being rendered.

Interpolation in a perspective correct manner is specified in equations 13.4 of the OpenGL ES

Specification, section 13.4.1 “Line Segments” and equation 13.7, section 13.5.1 “Polygon Interpolation”.

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

When multisample rasterization is enabled, **centroid** and **sample** may be used to control the location and frequency of the sampling of the qualified fragment shader input. If a fragment shader input is qualified with **centroid**, a single value may be assigned to that variable for all samples in the pixel, but that value must be interpolated at a location that lies in both the pixel and in the primitive being rendered, including any of the 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.

4.6. Parameter Qualifiers

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

Qualifier	Meaning
<none: default>	same as **in**
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 “Function Calling Conventions”.

4.7. Precision and Precision Qualifiers

4.7.1. Range and Precision

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

The following rules apply to **highp** operations: Signed infinities and zeros are generated as dictated by IEEE, but subject to the precisions allowed in the following table. Any subnormal (denormalized) value input into a shader or potentially generated by any operation in a shader can be flushed to 0.

The rounding mode cannot be set and is undefined but must not affect the result by more than 1 ULP. NaNs are not required to be generated. Support for signaling NaNs is not required and exceptions are never raised. Operations including built-in functions that operate on a NaN are not required to return a NaN as the result. However if NaNs are generated, **isnan**() must return the correct value.

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

For single precision operations, precisions are required as follows:

Operation	Precision
a + *b*, *a* - *b*, *a* * *b*	Correctly rounded.
<, <=, ==, >, >=	Correct result.
a / *b*, 1.0 / *b*	2.5 ULP for |b| in the range [2^{-126}, 2^{126}].
a * *b* + *c*	Correctly rounded single operation or sequence of two correctly rounded operations.
fma()	Inherited from *a* * *b* + *c*.
pow(*x*, *y*)	Inherited from **exp2**(*y* * **log2**(*x*)).
exp(*x*), **exp2**(*x*)	(3 + 2 · |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.

Storage requirements are declared through use of *precision qualifiers*. The precision of operations must preserve the storage precisions of the variables involved.

highp floating-point values are stored in IEEE 754 single precision floating-point format. **mediump** and **lowp** floating-point values have minimum range and precision requirements as detailed below and have maximum range and precision as defined by IEEE 754.

All integral types are assumed to be implemented as integers and so may not be emulated by floating-point values. **highp** signed integers are represented as twos-complement 32-bit signed integers. **highp** unsigned integers are represented as unsigned 32-bit integers. **mediump** integers (signed and unsigned) must be represented as an integer with between 16 and 32 bits inclusive. **lowp** integers (signed and unsigned) must be represented as an integer with between 9 and 32 bits inclusive.

The required ranges and precisions for precision qualifiers are:

Qualifier	**Floating-Point Range**	**Floating-Point Magnitude Range**	**Floating-Point Precision**	**Signed Integer Range**	**Unsigned Integer Range**
highp	Subset of IEEE-754 $(-2^{128},2^{128})$	Subset of IEEE-754 0.0, $[2^{-126},2^{128})$	Subset of IEEE-754 relative: 2^{-24}	$[-2^{31},2^{31}-1]$	$[0,2^{32}-1]$
mediump (minimum requirements)	$(-2^{14},2^{14})$	$(2^{14},2^{14})$	Relative: 2^{-10}	$[-2^{15},2^{15}-1]$	$[0,2^{16}-1]$
lowp (minimum requirements)	(-2,2)	$(2^{-8},2)$	Absolute: 2^{-8}	$[-2^{8},2^{8}-1]$	$[0,2^{9}-1]$

Relative precision is defined as the worst case (i.e. largest) ratio of the smallest step in relation to the value for all non-zero values:

$$Precision_{relative} = \left|\frac{|v_1 - v_2|_{min}}{v_1}\right|_{max}, \; v_1, v_2 \neq 0, v_1 \neq v_2$$

It is therefore twice the maximum rounding error when converting from a real number.

In addition, the range and precision of a **mediump** floating-point value must be the same as or greater than the range and precision of a **lowp** floating-point value. The range and precision of a **highp** floating-point value must be the same as or greater than the range and precision of a **mediump** floating-point value.

The range of a **mediump** integer value must be the same as or greater than the range of a **lowp** integer value. The range of a **highp** integer value must be the same as or greater than the range of a **mediump** integer value.

Within the above specification, an implementation is allowed to vary the representation of numeric values, both within a shader and between different shaders. If necessary, this variance can be controlled using the invariance qualifier.

The actual ranges and precisions provided by an implementation can be queried through the API. See the OpenGL ES Specification specification for details on how to do this.

4.7.2. Conversion between precisions

Within the same type, conversion from a lower to a higher precision must be exact. When converting from a higher precision to a lower precision, if the value is representable by the implementation of the target precision, the conversion must also be exact. If the value is not representable, the behavior is dependent on the type:

- For signed and unsigned integers, the value is truncated; bits in positions not present in the target precision are set to zero. (Positions start at zero and the least significant bit is considered to be position zero for this purpose.)

- For floating-point values, the value should either clamp to +Inf or -Inf, or to the maximum or minimum value that the implementation supports. While this behavior is implementation-dependent, it should be consistent for a given implementation.

4.7.3. Precision Qualifiers

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

Qualifier	Meaning
highp	The variable satisfies the minimum requirements for **highp** described above. **highp** variables have the maximum range and precision available but may cause operations to run more slowly on some implementations.
mediump	The variable satisfies the minimum requirements for **mediump** described above. **mediump** variables may typically be used to store high dynamic range colors and low precision geometry.
lowp	The variable satisfies the minimum requirements for **lowp** described above. **lowp** variables may typically be used to store 8-bit color values.

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 constructors.

For this paragraph, “operation” includes operators, built-in functions, and constructors, and “operand” includes function arguments and constructor arguments. The precision used to internally evaluate an operation, and the precision qualification subsequently associated with any resulting intermediate values, must be at least as high as the highest precision qualification of the operands consumed by the operation.

In cases where operands do not have a precision qualifier, the precision qualification will come from the other operands. If no operands have a precision qualifier, then the precision qualifications of the operands of the next consuming operation in the expression will be used. This rule can be applied recursively until a precision qualified operand is found. If necessary, it will also include the precision qualification of l-values for assignments, of the declared variable for initializers, of formal parameters for function call arguments, or of function return types for function return values. If the precision cannot be determined by this method e.g. if an entire expression is composed only of operands with no precision qualifier, and the result is not assigned or passed as an argument, then it is evaluated at the default precision of the type or greater. When this occurs in the fragment shader, the default precision must be defined.

For example, consider the statements:

```
uniform highp float h1;
highp float h2 = 2.3 * 4.7; // operation and result are highp
precision
mediump float m;
m = 3.7 * h1 * h2; // all operations are highp precision
h2 = m * h1; // operation is highp precision
m = h2 - h1; // operation is highp precision
h2 = m + m; // addition and result at mediump precision
void f(highp float p);
f(3.3); // 3.3 will be passed in at highp precision
```

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

Precision qualifiers for outputs in one shader matched to inputs in another shader need not match when both shaders are linked into the same program. When both shaders are in separate programs, mismatched precision qualifiers will result in a program interface mismatch that will result in program pipeline validation failures, as described in section 7.4.1 “Shader Interface Matching” of the OpenGL ES Specification.

The precision of a variable is determined when the variable is declared and cannot be subsequently changed.

Where the precision of a constant integral or constant floating-point expression is not specified, evaluation is performed at **highp**. This rule does not affect the precision qualification of the expression.

The evaluation of constant expressions must be invariant and will usually be performed at compile time.

4.7.4. 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**, **float**, or any of the opaque types, and the *precision-qualifier* can be **lowp**, **mediump**, or **highp**.

Any other types or qualifiers will result in 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 inner-most statement it was declared in. Precision statements in nested scopes override precision statements in outer scopes. Multiple precision statements for the same basic type can appear inside the same scope, with later statements overriding earlier statements within that scope.

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

```
precision highp float;
precision highp int;
precision lowp sampler2D;
precision lowp samplerCube;
precision highp atomic_uint;
```

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

```
precision mediump int;
precision lowp sampler2D;
precision lowp samplerCube;
precision highp atomic_uint;
```

The fragment language has no default precision qualifier for floating-point types. Hence for **float**, floating-point vector and matrix variable declarations, either the declaration must include a precision qualifier or the default float precision must have been previously declared. Similarly, there is no default precision qualifier in any of the languages for any type not listed above.

4.7.5. Available Precision Qualifiers

The built-in macro GL_FRAGMENT_PRECISION_HIGH is defined to one:

```
#define GL_FRAGMENT_PRECISION_HIGH 1
```

This macro is available in all languages except compute.

4.8. Variance and the Invariant Qualifier

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

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

4.8.1. The Invariant Qualifier

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

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

The **invariant** keyword can be followed by a comma separated list of previously declared identifiers. All uses of **invariant** must be at global scope or on block members, and before any use of the variables being declared as invariant.

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

- The output variable is declared as invariant in both programs.
- The same values must be input to all shader input variables consumed by expressions and control flow contributing to the value assigned to the output variable.
- The texture formats, texel values, and texture filtering are set the same way for any texture function calls contributing to the value of the output variable.
- All input values are all operated on in the same way. All operations in the consuming expressions and any intermediate expressions must be the same, with the same order of operands and same associativity, to give the same order of evaluation. Intermediate variables and functions must be declared as the same type with the same explicit or implicit precision qualifiers and the same constant 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.

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 Within a Shader

When a value is stored in a variable, it is usually assumed it will remain constant unless explicitly changed. However, during the process of optimization, it is possible that the compiler may choose to recompute a value rather than store it in a register. Since the precision of operations is not completely specified (e.g. a low precision operation may be done at medium or high precision), it would be possible for the recomputed value to be different from the original value.

Values are allowed to be variant within a shader. To prevent this, the invariant qualifier or invariant pragma must be used.

Within a shader, there is no invariance for values generated by different non-constant expressions, even if those expressions are identical.

Example 1:

```
precision mediump;
vec4 col;
vec2 a = ...
...
col = texture(tex, a);// a has a value _a1 _
...
col = texture(tex, a); // a has a value a2 where possibly a1 != a2
```

To enforce invariance in this example use:

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

Example 2:

```
vec2 m = ...;
vec2 n = ...;
vec2 a = m + n;
vec2 b = m + n; // a and b are not guaranteed to be exactly equal
```

There is no mechanism to enforce invariance between a and b.

4.8.3. Invariance of Constant Expressions

Invariance must be guaranteed for constant expressions. A particular constant expression must evaluate to the same result if it appears again in the same shader or a different shader. This includes the same expression appearing in two shaders of the same language or shaders of two different languages.

Constant expressions must evaluate to the same result when operated on as already described above for invariant variables. Constant expressions are not invariant with respect to equivalent non-constant expressions, even when the **invariant** qualifier or pragma is used.

4.8.4. Invariance of Undefined Values

Undefined values are not invariant nor can they be made invariant by use of the invariant qualifier or pragma. In some implementations, undefined values may cause unexpected behavior if they are used in control-flow expressions e.g. in the following case, one, both or neither functions may be executed and this may not be consistent over multiple invocations of the shader:

```
int x; // undefined value
if (x == 1)
{
    f(); // Undefined whether f() is executed
}
if (x == 2)
{
    g(); // Undefined whether g() is executed.
}
```

Note that an undefined value is a value that has not been specified. A value that has been specified but has a potentially large error due to, for example, lack of precision in an expression, is not undefined and so can be made invariant.

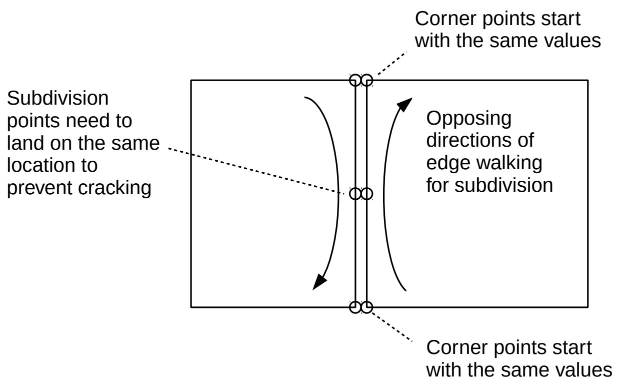
4.9. The Precise Qualifier

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

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


Corner points start with the same values
Subdivision points need to land on the same location to prevent cracking
Opposing directions of edge walking for subdivision
Corner points start with the same values

As stated in "Evaluation of Expressions", the compiler may transform expressions even if this changes the resulting value. Without any qualifiers, implementations are permitted to perform optimizations that effectively modify the order or number of operations used to evaluate an expression, even if those optimizations may produce slightly different results relative to unoptimized code.

The **precise** qualifier ensures that operations contributing to a variable's value are done in their stated order and with operator consistency. The order is determined by operator precedence and parenthesis, as described in "Operators". Operator consistency means for each particular operator, for example the multiply operator (*****), its operation is always computed with the same precision. Specifically, values computed by compiler-generated code must adhere to the following identities:

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

While the following are prevented:

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

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

For example,

```
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 (see "The Invariant Qualifier"), 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
```

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

This qualifier will affect the evaluation of an r-value in a particular function if and only if the result is eventually consumed in the same function by an l-value qualified as **precise**. Any other expressions within a function are not affected, including return values and output parameters not declared as **precise** but that are eventually consumed outside the function by a variable qualified as **precise**. Unaffected expressions also include the controlling expressions in selection and iteration statements and the condition in ternary operators (**?:**).

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()
{
    vec3 r = vec3(a * b);             // precise, used to compute v.xyz
    vec3 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.

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

4.10. Memory Qualifiers

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

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

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

Memory accesses to image variables declared using the **coherent** qualifier are performed coherently with accesses to the same location from other shader invocations.

As described in section 7.11 “Shader Memory Access” of the OpenGL ES 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.

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

Except for image variables qualified with the format qualifiers **r32f**, **r32i**, and **r32ui**, image variables must specify a memory qualifier (**readonly**, **writeonly**, or both).

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

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

```
coherent buffer Block {
    readonly vec4 member1;
    vec4 member2;
};
```

is equivalent to

```
buffer Block {
    coherent readonly vec4 member1;
    coherent vec4 member2;
};
```

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

When calling user-defined functions, variables qualified with **coherent**, **volatile**, **readonly**, or

writeonly may not be passed to functions whose formal parameters lack such qualifiers. (See “Function Definitions” for more detail on function calling.) It is legal to have any additional memory qualifiers on a formal parameter, but only **restrict** can be taken away from a calling argument, by a formal parameter that lacks the **restrict** qualifier.

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

```
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, and layout qualification is not included in parameter matching.

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

4.11. Order and Repetition of Qualification

When multiple qualifiers are present in a declaration, they may appear in any order, but they must all appear before the type. The **layout** qualifier is the only qualifier that can appear more than once. Further, a declaration can have at most one storage qualifier, at most one auxiliary storage qualifier, and at most one interpolation qualifier. Multiple memory qualifiers can be used. Any violation of these rules will cause a compile-time error.

4.12. Empty Declarations

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

```
int;                // No effect
struct S {int x;}; // Defines a struct S
```

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

```
invariant in float x; // Error. An input cannot be invariant.
invariant in float;   // Error even though no variable is declared.
```

Chapter 5. Operators and Expressions

5.1. Operators

The OpenGL ES 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 "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 "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 "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
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
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
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
```

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

Integer values having more bits of precision than a single-precision floating-point mantissa will lose precision when converted to **float**.

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

Some useful vector constructors are as follows:

```
vec3(float)          // initializes each component of 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
mat2(float, float,               // first column
     float, float);              // second column
mat3(float, float, float,        // first column
     float, float, float,        // second column
     float, float, float);       // third column
mat4(float, float, float, float, // first column
     float, float, float, float, // second column
     float, float, float, float, // third column
     float, float, float, float); // fourth column
mat2x3(vec2, float,              // first column
       vec2, float);             // second column
```

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

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

5.4.3. Structure Constructors

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

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

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

The arguments to the constructor will be used to set the structure's members, in order, using one argument per member. Each argument must be the same type as the member it sets.

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.

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

```
vec4 b[2] = ...;
vec4[3][2](b, b, b);    // constructor
vec4[][2](b, b, b);     // constructor, valid, size deduced
vec4[3][](b, b, b);     // constructor, valid, size deduced
vec4[][](b, b, b);      // constructor, valid, both sizes deduced
```

5.5. Vector Components

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

The component names supported are:

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

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

Note that the third component of the texture coordinate set, *r* in OpenGL ES, 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;
pos.x       // is legal
pos.z       // is illegal
```

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

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

No more than 4 components can be selected.

```
vec4 v4;
v4.xyzwxy;      // is illegal since it has 6 components
(v4.xyzwxy).xy; // is illegal since the intermediate value has 6
components
```

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

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

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

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

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

To form an l-value, swizzling must further be applied to an l-value and contain no duplicate components. 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 results in an error. When indexing with non-constant expressions, behavior is undefined if the index is negative, or greater than or equal to the size of the vector.

Note that scalars are not considered to be single-component vectors and therefore the use of component selection operators on scalars is illegal.

5.6. Matrix Components

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

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

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

5.7. Structure and Array Operations

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

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

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

The equality operators and assignment operator are only allowed if the two operands are same size and type. The operands cannot contain any opaque types. Structure types must be of the same declared structure. Both array operands must be compile-time 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. Any type-conversions must be specified explicitly via constructors. It is an error if the l-value is not writable. Variables that are built-in types, entire structures or arrays, structure members, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[ ]**) are all l-values. Other binary or unary expressions, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (**?:**) is also not allowed as an l-value. Using an incorrect expression as an l-value results in an error.

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

The other assignment operators are

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

where the general expression

> *lvalue op*= *expression*

is equivalent to

> *lvalue* = *lvalue op expression*

where *lvalue* is the value returned by *lvalue-expression*, *op* is as described below, and the *lvalue-*

expression and *expression* must satisfy the semantic requirements of both *op* and equals (=).

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

5.9. Expressions

Expressions in the shading language are built from the following:

- Constants of type **bool**, all integral types, all floating-point types, all vector types, and all matrix types.
- Constructors of all types.
- Variable names of all types.
- Arrays with the length method applied.
- Subscripted arrays.
- Function calls that return values. In some cases, function calls returning **void** are also allowed in expressions as specified below.
- Component field selectors and array subscript results.
- Parenthesized expressions. Any expression, including expressions with void type can be parenthesized. Parentheses can be used to group operations. Operations within parentheses are done before operations across parentheses.
- The arithmetic binary operators add (+), subtract (-), multiply (*), and divide (/) operate on integer and floating-point scalars, vectors, and matrices. If the operands are integral types, they must both be signed or both be unsigned. All arithmetic binary operators result in the same fundamental type (signed integer, unsigned integer, or floating-point) as the operands they operate on. 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. “Vector and Matrix Operations” explains in more detail how vectors and matrices are operated on.

  All other cases result in an error.

Use the built-in functions **dot**, **cross**, **matrixCompMult**, and **outerProduct**, to get, respectively, vector dot product, vector cross product, matrix component-wise multiplication, and the matrix product of a column vector times a row vector.

- The operator modulus (**%**) operates on signed or unsigned integers or integer vectors. The operand types must both be signed or both be unsigned. The operands cannot be vectors of differing size; this is an error. If one operand is a scalar and the other vector, then the scalar is applied component-wise to the vector, resulting in the same type as the vector. If both are vectors of the same size, the result is computed component-wise. The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero second operands remain defined. If both operands are non-negative, then the remainder is non-negative. Results are undefined if one or both operands are negative. The operator modulus (**%**) is not defined for any other data types (non-integer types).
- The arithmetic unary operators negate (**-**), post- and pre-increment and decrement (**--** and **++**) operate on integer or floating-point values (including vectors and matrices). All unary operators work component-wise on their operands. These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be a writable l-value. Pre-increment and pre-decrement add or subtract 1 or 1.0 to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 or 1.0 to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.
- The relational operators greater than (**>**), less than (**<**), greater than or equal (**>=**), and less than or equal (**<=**) operate only on scalar integer and scalar floating-point expressions. The result is scalar Boolean. The types of the operands must match. To do component-wise relational comparisons on vectors, use the built-in functions **lessThan**, **lessThanEqual**, **greaterThan**, and **greaterThanEqual.**
- The equality operators **equal** (**==**), and not equal (**!=**) operate on all types except opaque types. They result in a scalar Boolean. The types of the operands must match. For vectors, matrices, structures, and arrays, all components, members, or elements of one operand must equal the corresponding components, members, or elements in the other operand for the operands to be considered equal. To get a vector of component-wise equality results for vectors, use the built-in functions **equal** and **notEqual**.
- The logical binary operators and (**&&**), or (**||**), and exclusive or (**^^**) operate only on two Boolean expressions and result in a Boolean expression. And (**&&**) will only evaluate the right hand operand if the left hand operand evaluated to **true**. Or (**||**) will only evaluate the right hand operand if the left hand operand evaluated to **false**. Exclusive or (**^^**) will always evaluate both operands.
- The logical unary operator not (**!**). It operates only on a Boolean expression and results in a Boolean expression. To operate on a vector, use the built-in function **not**.
- The sequence (**,**) operator that operates on expressions by returning the type and value of the right-most expression in a comma separated list of expressions. All expressions are evaluated, in order, from left to right. The operands to the sequence operator may have **void** type.
- The ternary selection operator (**?:**). It operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression, which must result in a scalar Boolean. If the result is

true, it selects to evaluate the second expression, otherwise it selects to evaluate the third expression. Only one of the second and third expressions is evaluated. The second and third expressions can be any type, including **void**, as long their types match. 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 with the same size as the first operand, and the result is computed component-wise. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left 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; this is an error. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. 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 "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.

5.11. Evaluation of Expressions

In general expressions must be evaluated in the order specified by the precedence of operations and may only be regrouped if the result is the same or where the result is undefined. No other transforms may be applied that affect the result of an operation. GLSL ES relaxes these requirements for scalar operations in the following ways:

- Addition and multiplication are assumed to be associative.
- Multiplication is assumed to be distributive over addition. Therefore expressions may be expanded and re-factored.
- Floating-point division may be replaced by reciprocal and multiplication.
- Multiplication may be replaced by repeated addition.
- Within the constraints of invariance (where applicable), the precision used may vary.

These rules also apply to the built-in functions.

Chapter 6. Statements and Structure

The fundamental building blocks of the OpenGL ES 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 "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 cannot be void, or:

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

Each of the *typeN* must include a type and can optionally include a parameter qualifier and/or **const**.

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

Arrays are allowed as arguments and as the return type. In both cases, the array must be compile-time 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 a structure.

See "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. When function calls are resolved, an exact type match for all the arguments is required.

For example,

```
vec4 f(in vec4 x, out vec4 y);
vec4 f(in vec4 x, out uvec4 y);      // allowed, different argument type
int f(in vec4 x, out vec4 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
```

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

```
f(vec4, vec4)   // exact match of vec4 f(in vec4 x, out vec4 y)
f(vec4, uvec4)  // exact match of vec4 f(in vec4 x, out uvec4 y)
f(ivec4, vec4)  // error, no exact match.
f(ivec4, uvec4) // error, no exact match.
```

User-defined functions can have multiple declarations, but only one definition.

A shader cannot redefine or overload built-in functions.

The function *main* is used as the entry point to a shader executable. All shaders must define a function named *main*. 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 "Jumps" for more details.

It is a compile-time or link-time error to declare or define a function **main** with any other parameters or return type.

6.1.1. Function Calling Conventions

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

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

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

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

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**, or an error results.

Only precision qualifiers are allowed on the return type of a function.

The syntax for function prototypes can be informally expressed as:

function_prototype :
    [*type_qualifier*] *type_specifier* *IDENTIFIER* *LEFT_PAREN* *parameter_declaration* , *parameter_declaration* , ... , *RIGHT_PAREN*

parameter_declaration **:**

  [*type_qualifier*] *type_specifier* [*IDENTIFIER* [*array_specifier*]]

type_qualifier **:**

  single_type_qualifier , *single_type_qualifier* , ...

The qualifiers allowed on formal parameters are:

  empty
  const
  in
  out
  inout
  precise
  memory-qualifier
  precision-qualifier

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

Static, and hence dynamic recursion, are not allowed. Static recursion is present if the static function-call graph of a program contains cycles. Dynamic recursion occurs if at any time control flow has entered but not exited a single function more than once.

6.2. Selection

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

selection-statement **:**

  if (*bool-expression* **)** *statement*
  if (*bool-expression* **)** *statement* **else** *statement*
  switch (*init-expression* **) {** *switch-statement-list*$_{opt}$ **}**

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

switch-statement-list **:**

  switch-statement
  switch-statement-list switch-statement

switch-statement **:**

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

Note the above grammar's purpose is to aid discussion in this section; the normative grammar is in “Shading Language Grammar”.

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

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

Conditionals can be nested.

The type of *init-expression* in a **switch** statement must be a scalar integer. The type of *init-expression* must match the type of the **case** labels within each **switch** statement. Either signed integers or unsigned integers are allowed but there is no implicit type conversion between the two. If a **case** label has a *constant-expression* of equal value to *init-expression*, execution will continue after that label. Otherwise, if there is a **default** label, execution will continue after that label. Otherwise, execution skips the rest of the switch statement. It is 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.

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

6.3. Iteration

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

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

See “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" or other non-structured flow of control.

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

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

The **discard** keyword is only allowed within fragment shaders. It can be used within a fragment shader to abandon the operation on the current fragment. This keyword causes the fragment to be discarded and no updates to any buffers will occur. Any prior writes to other buffers such as shader storage buffers are unaffected. Control flow exits the shader, and subsequent implicit or explicit derivatives are undefined when this control flow is non-uniform (meaning different fragments within the primitive take different control paths). It would typically be used within a conditional statement, for example:

```
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.

Chapter 7. Built-In Variables

7.1. Built-In Language Variables

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

7.1.1. Vertex Shader Special Variables

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

```
in highp int gl_VertexID;
in highp int gl_InstanceID;

out gl_PerVertex {
    out highp vec4 gl_Position;
    out highp float gl_PointSize;
};
```

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

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

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

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

7.1.2. Tessellation Control Shader Special Variables

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

```
in gl_PerVertex {
    highp vec4 gl_Position;
} gl_in[gl_MaxPatchVertices];

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

out gl_PerVertex {
    highp vec4 gl_Position;
} gl_out[];

patch out highp float gl_TessLevelOuter[4];
patch out highp float gl_TessLevelInner[2];
patch out highp vec4 gl_BoundingBox[2];
```

Tessellation Control Input Variables

gl_Position contains the output written in the previous shader stage to *gl_Position*.

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

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

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

Tessellation Control Output Variables

gl_Position is used in the same fashion as the corresponding output variable in the vertex shader.

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

The values written to *gl_BoundingBox* specify the minimum and maximum clip-space extents of a bounding box containing all primitives generated from the patch by the primitive generator, geometry shader, and clipping. Fragments may or may not be generated for portions of these primitives that extend outside the window-coordinate projection of this bounding box.

7.1.3. Tessellation Evaluation Shader Special Variables

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

```
in gl_PerVertex {
    highp vec4 gl_Position;
} gl_in[gl_MaxPatchVertices];

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

out gl_PerVertex {
    highp vec4 gl_Position;
};
```

Tessellation Evaluation Input Variables

gl_Position contains the output written in the previous shader stage to *gl_Position*.

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

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

```
gl_TessCoord.x == 1.0 - (1.0 - gl_TessCoord.x) // two operations performed
gl_TessCoord.y == 1.0 - (1.0 - gl_TessCoord.y) // two operations performed
gl_TessCoord.z == 1.0 - (1.0 - gl_TessCoord.z) // two operations performed
```

to aid in replicating subdivision computations.

gl_TessLevelOuter and *gl_TessLevelInner* are filled with the corresponding outputs written by the active tessellation control shader.

Tessellation Evaluation Output Variables

gl_Position is used in the same fashion as the corresponding output variable in the vertex shader.

7.1.4. Geometry Shader Special Variables

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

```
in gl_PerVertex {
    highp vec4 gl_Position;
} gl_in[];

in highp int gl_PrimitiveIDIn;
in highp int gl_InvocationID;

out gl_PerVertex {
    highp vec4 gl_Position;
};

out highp int gl_PrimitiveID;
out highp int gl_Layer;
```

Geometry Shader Input Variables

gl_Position contains the output written in the previous shader stage to *gl_Position*.

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

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

Geometry Shader Output Variables

gl_Position is used in the same fashion as the corresponding output variable in the vertex shader.

gl_PrimitiveID is filled with a single integer that serves as a primitive identifier to the fragment shader. This is then available to fragment shaders, which will select the written primitive ID from the provoking vertex of the primitive being shaded. If a fragment shader using *gl_PrimitiveID* is active and a geometry shader is also active, the geometry shader must write to *gl_PrimitiveID* or the fragment shader input *gl_PrimitiveID* is undefined. See section 11.3.4.4 “Geometry Shader Outputs” of the OpenGL ES Specification for more information.

gl_Layer is used to select a specific layer (or face and layer of a cube map) of a multi-layer framebuffer attachment. The actual layer used will come from one of the vertices in the primitive being shaded. Which vertex the layer comes from is determined as discussed in section 11.3.4.4 of the OpenGL ES Specification but may be undefined, so it is best to write the same layer value for all vertices of a primitive. If a shader statically assigns a value to *gl_Layer*, layered rendering mode is enabled. See section 11.3.4.4 “Geometry Shader Outputs” and section 9.8 “Layered Framebuffers” of the OpenGL ES 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 8.25 of the OpenGL ES 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*.

7.1.5. Fragment Shader Special Variables

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

```
in highp vec4 gl_FragCoord;
in bool gl_FrontFacing;
out highp float gl_FragDepth;
in mediump vec2 gl_PointCoord;
in bool gl_HelperInvocation;
in highp int gl_PrimitiveID;
in highp int gl_Layer;
in lowp int gl_SampleID;
in mediump vec2 gl_SamplePosition;
in highp int gl_SampleMaskIn[(gl_MaxSamples+31)/32];
out highp int gl_SampleMask[(gl_MaxSamples+31)/32];
```

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

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

Writing to *gl_FragDepth* will establish the depth value for the fragment being processed. If depth buffering is enabled, and no shader writes *gl_FragDepth*, then the fixed function value for depth will be used as the fragment's depth value. If a shader statically assigns a value to *gl_FragDepth*, and there is an execution path through the shader that does not set *gl_FragDepth*, then the value of the fragment's depth may be undefined for executions of the shader that take that path. That is, if the set of linked fragment shaders statically contain a write to *gl_FragDepth*, then it is responsible for always writing it.

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

The variable *gl_FragCoord* is available as an input variable from within fragment shaders and it holds the window relative coordinates (*x*, *y*, *z*, *1/w*) values for the fragment. If multi-sampling, this value can be for any location within the pixel, or one of the fragment samples. The use of **centroid** does not further restrict this value to be inside the current primitive. This value is the result of the fixed functionality that interpolates primitives after vertex processing to generate fragments. The *z* component is the depth value that would be used for the fragment's depth if no shader contained any writes to *gl_FragDepth*. This is useful for invariance if a shader conditionally computes *gl_FragDepth* but otherwise wants the fixed functionality fragment depth.

Fragment shaders have access to the input built-in variable *gl_FrontFacing*, whose value is **true** if the fragment belongs to a front-facing primitive. One use of this is to emulate two-sided lighting by selecting one of two colors calculated by a vertex or geometry shader.

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

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

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

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

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

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

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

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

- Fragments corresponding to helper invocations are discarded when shader execution is complete, without updating the framebuffer.
- Stores to image and buffer variables performed by helper invocations have no effect on the underlying image or buffer memory.
- Atomic operations to image, buffer, or atomic counter variables performed by helper invocations have no effect on the underlying image or buffer memory. The values returned by such atomic operations are undefined.

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

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

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

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

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

7.1.6. Compute Shader Special Variables

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

```
// workgroup dimensions
in uvec3 gl_NumWorkGroups;
const uvec3 gl_WorkGroupSize;

// workgroup and invocation IDs
in uvec3 gl_WorkGroupID;
in uvec3 gl_LocalInvocationID;

// derived variables
in uvec3 gl_GlobalInvocationID;
in uint gl_LocalInvocationIndex;
```

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

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

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

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

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

```
gl_GlobalInvocationID =
    gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID;
```

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

```
gl_LocalInvocationIndex =
    gl_LocalInvocationID.z * gl_WorkGroupSize.x * gl_WorkGroupSize.y +
    gl_LocalInvocationID.y * gl_WorkGroupSize.x +
    gl_LocalInvocationID.x;
```

7.2. 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 mediump int gl_MaxVertexAttribs = 16;
const mediump int gl_MaxVertexUniformVectors = 256;
const mediump int gl_MaxVertexOutputVectors = 16;
const mediump int gl_MaxVertexTextureImageUnits = 16;
const mediump int gl_MaxVertexImageUniforms = 0;
const mediump int gl_MaxVertexAtomicCounters = 0;
const mediump int gl_MaxVertexAtomicCounterBuffers = 0;

const mediump int gl_MaxTessControlInputComponents = 64;
const mediump int gl_MaxTessControlOutputComponents = 64;
const mediump int gl_MaxTessControlTextureImageUnits = 16;
const mediump int gl_MaxTessControlUniformComponents = 1024;
const mediump int gl_MaxTessControlTotalOutputComponents = 2048;
const mediump int gl_MaxTessControlImageUniforms = 0;
const mediump int gl_MaxTessControlAtomicCounters = 0;
const mediump int gl_MaxTessControlAtomicCounterBuffers = 0;

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

const mediump int gl_MaxTessEvaluationInputComponents = 64;
const mediump int gl_MaxTessEvaluationOutputComponents = 64;
const mediump int gl_MaxTessEvaluationTextureImageUnits = 16;
const mediump int gl_MaxTessEvaluationUniformComponents = 1024;
const mediump int gl_MaxTessEvaluationImageUniforms = 0;
const mediump int gl_MaxTessEvaluationAtomicCounters = 0;
const mediump int gl_MaxTessEvaluationAtomicCounterBuffers = 0;

const mediump int gl_MaxGeometryInputComponents = 64;
const mediump int gl_MaxGeometryOutputComponents = 64;
const mediump int gl_MaxGeometryImageUniforms = 0;
const mediump int gl_MaxGeometryTextureImageUnits = 16;
const mediump int gl_MaxGeometryOutputVertices = 256;
const mediump int gl_MaxGeometryTotalOutputComponents = 1024;
const mediump int gl_MaxGeometryUniformComponents = 1024;
const mediump int gl_MaxGeometryAtomicCounters = 0;
const mediump int gl_MaxGeometryAtomicCounterBuffers = 0;

const mediump int gl_MaxFragmentInputVectors = 15;
const mediump int gl_MaxFragmentImageUniforms = 4;
const mediump int gl_MaxFragmentUniformVectors = 256;
const mediump int gl_MaxFragmentAtomicCounters = 8;
const mediump int gl_MaxFragmentAtomicCounterBuffers = 1;

const mediump int gl_MaxDrawBuffers = 4;
const mediump int gl_MaxTextureImageUnits = 16;
const mediump int gl_MinProgramTexelOffset = -8;
const mediump int gl_MaxProgramTexelOffset = 7;
const mediump int gl_MaxImageUnits = 4;
const mediump int gl_MaxSamples = 4;

const mediump int gl_MaxComputeImageUniforms = 4;
const highp ivec3 gl_MaxComputeWorkGroupCount = ivec3(65535, 65535, 65535);
const highp ivec3 gl_MaxComputeWorkGroupSize = ivec3(128, 128, 64);
const mediump int gl_MaxComputeUniformComponents = 1024;
const mediump int gl_MaxComputeTextureImageUnits = 16;
const mediump int gl_MaxComputeAtomicCounters = 8;
const mediump int gl_MaxComputeAtomicCounterBuffers = 1;

const mediump int gl_MaxCombinedTextureImageUnits = 96;
const mediump int gl_MaxCombinedImageUniforms = 4;
const mediump int gl_MaxCombinedShaderOutputResources = 4;
const mediump int gl_MaxCombinedAtomicCounters = 8;
const mediump int gl_MaxCombinedAtomicCounterBuffers = 1;
const mediump int gl_MaxAtomicCounterBindings = 1;
const mediump int gl_MaxAtomicCounterBufferSize = 32;
```

7.3. Built-In Uniform State

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

```
//
// Depth range in window coordinates,
// section 12.5.1 "Controlling the Viewport" in the
// OpenGL ES Specification.
//
struct gl_DepthRangeParameters {
    highp float near; // n
    highp float far;  // f
    highp float diff; // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;
uniform lowp int gl_NumSamples;
```

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

7.4. Redeclaring Built-In Blocks

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

If the *gl_PerVertex* block is not redefined in a given program, the intrinsically declared definition of that block is used for the program interface.

For example:

```
out gl_PerVertex {
    highp vec4 gl_Position; // will use gl_Position
    highp 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*. Such a redeclaration can also add the **invariant** qualifier and interpolation qualifiers.

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

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

Chapter 8. Built-In Functions

The OpenGL ES 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).

When the built-in functions are specified below, where the input arguments (and corresponding output) can be **float**, **vec2**, **vec3**, or **vec4**, *genFType* is used as the argument. Where the input arguments (and corresponding output) can be **int**, **ivec2**, **ivec3**, or **ivec4**, *genIType* is used as the argument. Where the input arguments (and corresponding output) can be **uint**, **uvec2**, **uvec3**, or **uvec4**, *genUType* is used as the argument. Where the input arguments (or corresponding output) can be **bool**, **bvec2**, **bvec3**, or **bvec4**, *genBType* is used as the argument. For any specific use of a function, the actual types substituted for *genFType*, *genIType*, *genUType*, or *genBType* have to have the same number of components for all arguments and for the return type. Similarly, *mat* is used for any matrix basic type.

Built-in functions have an effective precision qualification. This qualification cannot be set explicitly and may be different from the precision qualification of the result.

The precision qualification of the operation of a built-in function is based on the precision qualification of its formal parameters and actual parameters (input arguments): When a formal parameter specifies a precision qualifier, that is used, otherwise, the precision qualification of the actual (calling) argument is used. The highest precision of these will be the precision of the operation of the built-in function. Generally, this is applied across all arguments to a built-in function, with the exceptions being:

- **bitfieldExtract** and **bitfieldInsert** ignore the *offset* and *bits* arguments.
- **interpolateAt** functions only look at the *interpolant* argument.

The precision qualification of the result of a built-in function is determined in one of the following ways:

For the texture sampling, image load and image store functions, the precision of the return type matches the precision of the sampler type:

```
uniform lowp sampler2D sampler;
highp vec2 coord;
...
lowp vec4 col = texture (sampler, coord); // texture() returns lowp
```

Otherwise:

- For prototypes that do not specify a resulting precision qualifier, the precision will be the same as the precision of the operation (as defined earlier).
- For prototypes that do specify a resulting precision qualifier, the specified precision qualifier is the precision qualification of the result.

Where the built-in functions in the following sections specify an equation, the entire equation will be evaluated at the operation's precision. This may lead to underflow or overflow in the result, even when the correct result could be represented in the operation precision.

8.1. Angle and Trigonometry Functions

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

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

Syntax	Description
genFType **radians**(genFType *degrees*)	Converts *degrees* to radians, i.e., (π / 180) · degrees.
genFType **degrees**(genFType *radians*)	Converts *radians* to degrees, i.e., (180 / π) · radians.
genFType **sin**(genFType *angle*)	The standard trigonometric sine function.
genFType **cos**(genFType *angle*)	The standard trigonometric cosine function.
genFType **tan**(genFType *angle*)	The standard trigonometric tangent.
genFType **asin**(genFType *x*)	Arc sine. Returns an angle whose sine is *x*. The range of values returned by this function is [-π / 2, π / 2]. Results are undefined if ∣x∣ > 1.
genFType **acos**(genFType *x*)	Arc cosine. Returns an angle whose cosine is *x*. The range of values returned by this function is [0,π]. Results are undefined if ∣x∣ > 1.

Syntax	**Description**
genFType **atan**(genFType *y*, genFType *x*)	Arc tangent. Returns an angle whose tangent is y / x. The signs of *x* and *y* are used to determine what quadrant the angle is in. The range of values returned by this function is [-π, π. Results are undefined if *x* and *y* are both 0.
genFType **atan**(genFType *y_over_x*)	Arc tangent. Returns an angle whose tangent is *y_over_x*. The range of values returned by this function is [-π / 2, π / 2].
genFType **sinh**(genFType *x*)	Returns the hyperbolic sine function (e^x - e^{-x}) / 2.
genFType **cosh**(genFType *x*)	Returns the hyperbolic cosine function (e^x + e^{-x}) / 2.
genFType **tanh**(genFType *x*)	Returns the hyperbolic tangent function sinh(x) / cosh(x).
genFType **asinh**(genFType *x*)	Arc hyperbolic sine; returns the inverse of **sinh**.
genFType **acosh**(genFType *x*)	Arc hyperbolic cosine; returns the non-negative inverse of **cosh**. Results are undefined if x < 1.
genFType **atanh**(genFType *x*)	Arc hyperbolic tangent; returns the inverse of **tanh**. Results are undefined if x ≥ 1.

8.2. Exponential Functions

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

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

8.3. Common Functions

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

Syntax	Description
genFType **abs**(genFType *x*) genIType **abs**(genIType *x*)	Returns *x* if x ≥ 0; otherwise it returns -*x*.
genFType **sign**(genFType *x*) genIType **sign**(genIType *x*)	Returns 1.0 if *x* > 0, 0.0 if *x* = 0, or -1.0 if *x* < 0.
genFType **floor**(genFType *x*)	Returns a value equal to the nearest integer that is less than or equal to *x*.
genFType **trunc**(genFType *x*)	Returns a value equal to the nearest integer to *x* whose absolute value is not larger than the absolute value of *x*.
genFType **round**(genFType *x*)	Returns a value equal to the nearest integer to *x*. The fraction 0.5 will round in a direction chosen by the implementation, presumably the direction that is fastest. This includes the possibility that **round**(*x*) returns the same value as **roundEven**(*x*) for all values of *x*.
genFType **roundEven**(genFType *x*)	Returns a value equal to the nearest integer to *x*. A fractional part of 0.5 will round toward the nearest even integer. (Both 3.5 and 4.5 for x will return 4.0.)
genFType **ceil**(genFType *x*)	Returns a value equal to the nearest integer that is greater than or equal to *x*.
genFType **fract**(genFType *x*)	Returns *x* - **floor**(*x*).
genFType **mod**(genFType *x*, float *y*) genFType **mod**(genFType *x*, genFType *y*)	Modulus. Returns x - y · **floor**(x / y).
genFType **modf**(genFType *x*, out genFType *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*. If *x* has the value +/- Inf, the return value should be NaN and must be either NaN or 0.0. For **highp** non-constant expressions, the value returned must be consistent.
genFType **min**(genFType *x*, genFType *y*) genFType **min**(genFType *x*, float *y*) genIType **min**(genIType *x*, genIType *y*) genIType **min**(genIType *x*, int *y*) genUType **min**(genUType *x*, genUType *y*) genUType **min**(genUType *x*, uint *y*)	Returns *y* if *y* < *x;* otherwise it returns *x*.
genFType **max**(genFType *x*, genFType *y*) genFType **max**(genFType *x*, float *y*) genIType **max**(genIType *x*, genIType *y*) genIType **max**(genIType *x*, int *y*) genUType **max**(genUType *x*, genUType *y*) genUType **max**(genUType *x*, uint *y*)	Returns *y* if *x* < *y;* otherwise it returns *x*.

Syntax	**Description**
genFType **clamp**(genFType *x*, genFType *minVal*, genFType *maxVal*) genFType **clamp**(genFType *x*, float *minVal*, float *maxVal*) genIType **clamp**(genIType *x*, genIType *minVal*, genIType *maxVal*) genIType **clamp**(genIType *x*, int *minVal*, int *maxVal*) genUType **clamp**(genUType *x*, genUType *minVal*, genUType *maxVal*) genUType **clamp**(genUType *x*, uint *minVal*, uint *maxVal*)	Returns **min**(**max**(*x*, *minVal*), *maxVal*). Results are undefined if *minVal* > *maxVal*.
genFType **mix**(genFType *x*, genFType *y*, genFType *a*) genFType **mix**(genFType *x*, genFType *y*, float *a*)	Returns the linear blend of *x* and *y*, i.e., x · (1 - a) + y · a.
genFType **mix**(genFType *x*, genFType *y*, genBType *a*) genIType **mix**(genIType *x*, genIType *y*, genBType *a*) genUType **mix**(genUType *x*, genUType *y*, genBType *a*) genBType **mix**(genBType *x*, genBType *y*, genBType *a*)	Selects which vector each returned component comes from. For a component of *a* that is **false**, the corresponding component of *x* is returned. For a component of *a* that is **true**, the corresponding component of *y* is returned. Components of *x* and *y* that are not selected are allowed to be invalid floating-point values and will have no effect on the results. Thus, this provides different functionality than, for example, genFType **mix**(genFType *x*, genFType *y*, genFType(*a*)) where *a* is a Boolean vector.
genFType **step**(genFType *edge*, genFType *x*) genFType **step**(float *edge*, genFType *x*)	Returns 0.0 if *x* < *edge;* otherwise it returns 1.0.

genFType **smoothstep**(genFType *edge0*, genFType *edge1*, genFType *x*)
genFType **smoothstep**(float *edge0*, float *edge1*, genFType *x*)
	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:
	
	`genFType 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.

Syntax	**Description**
genBType **isnan**(genFType *x*)	Returns **true** if *x* holds a NaN. Returns **false** otherwise. Always returns **false** if NaNs are not implemented.
genBType **isinf**(genFType *x*)	Returns **true** if *x* holds a positive infinity or negative infinity. Returns **false** otherwise.
genIType **floatBitsToInt**(highp genFType *value*) genUType **floatBitsToUint**(highp genFType *value*)	Returns a signed or unsigned integer value representing the encoding of a floating-point value. The **float** value's bit-level representation is preserved.
genFType **intBitsToFloat**(highp genIType *value*) genFType **uintBitsToFloat**(highp genUType *value*)	Returns a floating-point value corresponding to a signed or unsigned integer encoding of a floating-point value. If an Inf or NaN is passed in, it will not signal, and the resulting floating-point value is unspecified. Otherwise, the bit-level representation is preserved.
genFType **fma**(genFType *a*, genFType *b*, genFType *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 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*".

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

8.4. Floating-Point Pack and Unpack Functions

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

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

Syntax	Description
mediump vec2 **unpackHalf2x16**(highp 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 ES 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**(genFType *x*)	Returns the length of vector *x*, i.e., sqrt(x_0^2 + x_1^2 + ...).
float **distance**(genFType *p0*, genFType *p1*)	Returns the distance between *p0* and *p1*, i.e., **length**(*p0* - *p1*)
float **dot**(genFType *x*, genFType *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*)	Returns the cross product of *x* and *y*, i.e., $(x_1 \cdot y_2 - y_1 \cdot x_2, x_2 \cdot y_0 - y_2 \cdot x_0, x_0 \cdot y_1 - y_0 \cdot x_1)$.
genFType **normalize**(genFType *x*)	Returns a vector in the same direction as *x* but with a length of 1, i.e. *x* / **length**(x).
genFType **faceforward**(genFType *N*, genFType *I*, genFType *Nref*)	If **dot**(*Nref*, *I*) < 0 return *N*, otherwise return -*N*.
genFType **reflect**(genFType *I*, genFType *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.

Syntax	Description
genFType **refract**(genFType *I*, genFType *N*, float *eta*)	For the incident vector *I* and surface normal *N*, and the ratio of indices of refraction *eta*, return the refraction vector. The result is computed by the refraction equation shown below. `The input parameters for the incident vector _I_ and the surface normal _N_ must already be normalized to get the desired results.`

8.5.1. Refraction Equation

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

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

8.6. Matrix Functions

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

Syntax	Description
mat **matrixCompMult**(mat *x*, mat *y*)	Multiply matrix *x* by matrix *y* component-wise, i.e., result[i][j] is the scalar product of *x*[i][j] and *y*[i][j]. Note: to get linear algebraic matrix multiplication, use the multiply operator (*****).
mat2 **outerProduct**(vec2 *c*, vec2 *r*) mat3 **outerProduct**(vec3 *c*, vec3 *r*) mat4 **outerProduct**(vec4 *c*, vec4 *r*) mat2x3 **outerProduct**(vec3 *c*, vec2 *r*) mat3x2 **outerProduct**(vec2 *c*, vec3 *r*) mat2x4 **outerProduct**(vec4 *c*, vec2 *r*) mat4x2 **outerProduct**(vec2 *c*, vec4 *r*) mat3x4 **outerProduct**(vec4 *c*, vec3 *r*) mat4x3 **outerProduct**(vec3 *c*, vec4 *r*)	Treats the first parameter *c* as a column vector (matrix with one column) and the second parameter *r* as a row vector (matrix with one row) and does a linear algebraic matrix multiply *c* * *r*, yielding a matrix whose number of rows is the number of components in *c* and whose number of columns is the number of components in *r*.

Syntax	**Description**
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*.
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

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.

Syntax	Description
bvec **equal**(vec x, vec y) bvec **equal**(ivec x, ivec y) bvec **equal**(uvec x, uvec y) bvec **equal**(bvec x, bvec y)	Returns the component-wise compare of x == y.
bvec **notEqual**(vec x, vec y) bvec **notEqual**(ivec x, ivec y) bvec **notEqual**(uvec x, uvec y) bvec **notEqual**(bvec x, bvec y)	Returns the component-wise compare of x ≠ y.
bool **any**(bvec x)	Returns **true** if any component of *x* is **true**.
bool **all**(bvec x)	Returns **true** only if all components of *x* are **true**.
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**(highp genUType *x*, highp genUType *y*, out lowp genUType *carry*)	Adds 32-bit unsigned integers *x* and *y*, returning the sum modulo 2^{32}. The value *carry* is set to zero if the sum was less than 2^{32}, or one otherwise.
genUType **usubBorrow**(highp genUType *x*, highp genUType *y*, out lowp genUType *borrow*)	Subtracts the 32-bit unsigned integer *y* from *x*, returning the difference if non-negative, or 2^{32} plus the difference otherwise. The value *borrow* is set to zero if x ≥ y, or one otherwise.
void **umulExtended**(highp genUType *x*, highp genUType *y*, out highp genUType *msb*, out highp genUType *lsb*) void **imulExtended**(highp genIType *x*, highp genIType *y*, out highp genIType *msb*, out highp genIType *lsb*)	Multiplies 32-bit unsigned or signed integers *x* and *y*, producing a 64-bit result. The 32 least-significant bits are returned in *lsb*. The 32 most-significant bits are returned in *msb*.

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

Syntax	Description
lowp genIType **findMSB**(highp genIType *value*) lowp genIType **findMSB**(highp genUType *value*)	Returns the bit number of the most significant bit in the binary representation of *value*. For positive integers, the result will be the bit number of the most significant one bit. For negative integers, the result will be the bit number of the most significant zero bit. For a *value* of zero or negative one, -1 will be returned.

8.9. Texture Functions

Texture lookup functions are available in all shading stages. However, level-of-detail is implicitly computed only for fragment shaders. Other shaders operate as though the base level-of-detail were computed as zero. The functions in the table below provide access to textures through samplers, as set up through the OpenGL ES API. Texture properties such as size, pixel format, number of dimensions, filtering method, number of mipmap levels, depth comparison, and so on are also defined by OpenGL ES API calls. Such properties are taken into account as the texture is accessed via the built-in functions defined below.

Texture data can be stored by the GL as single-precision floating-point, unsigned normalized integer, unsigned integer or signed integer data. This is determined by the type of the internal format of the texture.

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

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

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

If an integer sampler type is used, the result of a texture lookup is an **ivec4**. If an unsigned integer sampler type is used, the result of a texture lookup is a **uvec4**. If a floating-point sampler type is used, the result of a texture lookup is a **vec4**.

In the prototypes below, the "*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 8.20 "Texture Comparison Modes" of the OpenGL ES Specification. See the table below for which component specifies D_{ref}. The texture bound to *sampler* must be a depth texture, or results are undefined. If a non-shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned on, then results are undefined. If a shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned off, then results are undefined. If a shadow texture call is made to a sampler that does not represent a depth texture, then results are undefined.

In all functions below, the *bias* parameter is optional for fragment shaders. The *bias* parameter is not accepted in any other shader stage. For a fragment shader, if *bias* is present, it is added to the implicit level-of-detail prior to performing the texture access operation. No *bias* or *lod* parameters for multisample textures, or texture buffers are supported because mipmaps are not allowed for these types of textures.

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

Some texture functions (non-"**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 8.13 "Cube Map Texture Selection" of the OpenGL ES Specification.

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

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

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

8.9.1. Texture Query Functions

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

Syntax	Description
highp ivec2 **textureSize**(gsampler2D *sampler*, int *lod*) highp ivec3 **textureSize**(gsampler3D *sampler*, int *lod*) highp ivec2 **textureSize**(gsamplerCube *sampler*, int *lod*) highp ivec2 **textureSize**(sampler2DShadow *sampler*, int *lod*) highp ivec2 **textureSize**(samplerCubeShadow *sampler*, int *lod*) highp ivec3 **textureSize**(gsamplerCubeArray *sampler*, int *lod*) highp ivec3 **textureSize**(samplerCubeArrayShadow *sampler*, int *lod*) highp ivec3 **textureSize**(gsampler2DArray *sampler*, int *lod*) highp ivec3 **textureSize**(sampler2DArrayShadow *sampler*, int *lod*) highp int **textureSize**(gsamplerBuffer *sampler*) highp ivec2 **textureSize**(gsampler2DMS *sampler*) highp ivec3 **textureSize**(gsampler2DMSArray *sampler*)	Returns the dimensions of level *lod* (if present) for the texture bound to *sampler*, as described in section 8.11 “Texture Queries” of the OpenGL ES Specification. The components in the return value are filled in, in order, with the width, height, and depth of the texture. For the array forms, the last component of the return value is the number of layers in the texture array, or the number of cubes in the texture cube map array.

8.9.2. Texel Lookup Functions

Syntax	Description
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**(sampler2DShadow *sampler*, *vec3 _P* [, float *bias*]) float **texture**(samplerCubeShadow *sampler*, *vec4 _P* [, float *bias*]) gvec4 **texture**(gsampler2DArray *sampler*, vec3 *P* [, float *bias*]) gvec4 **texture**(gsamplerCubeArray *sampler*, vec4 *P* [, float *bias*]) float **texture**(sampler2DArrayShadow *sampler*, vec4 *P*) float **texture**(samplerCubeArrayShadow *sampler*, vec4 *P*, float *compare*)	Use the texture coordinate *P* to do a texture lookup in the texture currently bound to *sampler*. For shadow forms: When *compare* is present, it is used as D_{ref} and the array layer comes from the last component of *P*. When *compare* is not present, the last component of *P* is used as D_{ref} and the array layer comes from the second to last component of *P*. For non-shadow forms: the array layer comes from the last component of *P*.

Syntax	Description
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**(sampler2DShadow *sampler*, vec4 *P* [, float *bias*])	Do a texture lookup with projection. The texture coordinates consumed from *P*, not including the last component of *P*, are divided by the last component of *P* to form projected coordinates *P'*. The resulting third component of *P* in the shadow forms is used as D_{ref}. The third component of *P* is ignored when *sampler* has type **gsampler2D** and *P* has type **vec4**. After these values are computed, texture lookup proceeds as in **texture**.
gvec4 **textureLod**(gsampler2D *sampler*, vec2 *P*, float *lod*) gvec4 **textureLod**(gsampler3D *sampler*, vec3 *P*, float *lod*) gvec4 **textureLod**(gsamplerCube *sampler*, vec3 *P*, float *lod*) float **textureLod**(sampler2DShadow *sampler*, vec3 *P*, float *lod*) gvec4 **textureLod**(gsampler2DArray *sampler*, vec3 *P*, float *lod*) gvec4 **textureLod**(gsamplerCubeArray *sampler*, vec4 *P*, float *lod*)	Do a texture lookup as in **texture** but with explicit level-of-detail; *lod* specifies λ_{base}] and sets the partial derivatives as follows: (See section 8.14 "Texture Minification" and equations 8.4-8.6 of the OpenGL ES Specification.) $\partial u / \partial x = \partial v / \partial x = \partial w / \partial x = 0$ $\partial u / \partial y = \partial v / \partial y = \partial w / \partial y = 0$
gvec4 **textureOffset**(gsampler2D *sampler*, vec2 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureOffset**(gsampler3D *sampler*, vec3 *P*, ivec3 *offset* [, float *bias*]) float **textureOffset**(sampler2DShadow *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureOffset**(gsampler2DArray *sampler*, vec3 *P*, ivec2 *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 8.14.2 "Coordinate Wrapping and Texel Selection" of the OpenGL ES Specification, where *offset* is $(\delta_u, \delta_v, \delta_w)$. Note that texel offsets are also not supported for cube maps.
gvec4 **texelFetch**(gsampler2D *sampler*, ivec2 *P*, int *lod*) gvec4 **texelFetch**(gsampler3D *sampler*, ivec3 *P*, int *lod*) gvec4 **texelFetch**(gsampler2DArray *sampler*, ivec3 *P*, int *lod*) gvec4 **texelFetch**(gsamplerBuffer *sampler*, int *P*) gvec4 **texelFetch**(gsampler2DMS *sampler*, ivec2 *P*, int *sample*) gvec4 **texelFetch**(gsampler2DMSArray *sampler*, ivec3 *P*, int *sample*)	Use integer texture coordinate *P* to lookup a single texel from *sampler*. The array layer comes from the last component of *P* for the array forms. The level-of-detail *lod* (if present) is as described in sections 11.1.3.2 "Texel Fetches" and 8.14.1 "Scale Factor and Level of Detail" of the OpenGL ES Specification.

Syntax	Description
gvec4 **texelFetchOffset**(gsampler2D *sampler*, ivec2 *P*, int *lod*, ivec2 *offset*) gvec4 **texelFetchOffset**(gsampler3D *sampler*, ivec3 *P*, int *lod*, ivec3 *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**(gsampler2D *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureProjOffset**(gsampler2D *sampler*, vec4 *P*, ivec2 *offset* [, float *bias*]) gvec4 **textureProjOffset**(gsampler3D *sampler*, vec4 *P*, ivec3 *offset* [, float *bias*]) float **textureProjOffset**(sampler2DShadow *sampler*, vec4 *P*, ivec2 *offset* [, float *bias*])	Do a projective texture lookup as described in **textureProj**, offset by *offset* as described in **textureOffset**.
gvec4 **textureLodOffset**(gsampler2D *sampler*, vec2 *P*, float *lod*, ivec2 *offset*) gvec4 **textureLodOffset**(gsampler3D *sampler*, vec3 *P*, float *lod*, ivec3 *offset*) float **textureLodOffset**(sampler2DShadow *sampler*, *P*, float *lod*, ivec2 *offset*) gvec4 **textureLodOffset**(gsampler2DArray *sampler*, vec3 *P*, float *lod*, ivec2 *offset*)	Do an offset texture lookup with explicit level-of-detail. See **textureLod** and **textureOffset**.
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**(sampler2DShadow *sampler*, vec4 *P*, float *lod*)	Do a projective texture lookup with explicit level-of-detail. See **textureProj** and **textureLod**.
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**(sampler2DShadow *sampler*, vec4 *P*, float *lod*, ivec2 *offset*)	Do an offset projective texture lookup with explicit level-of-detail. See **textureProj**, **textureLod**, and **textureOffset**.

Syntax	Description
gvec4 **textureGrad**(gsampler2D *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureGrad**(gsampler3D *sampler*, *P*, vec3 *dPdx*, vec3 *dPdy*) gvec4 **textureGrad**(gsamplerCube *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy*) float **textureGrad**(sampler2DShadow *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) float **textureGrad**(samplerCubeShadow *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*) float **textureGrad**(sampler2DArrayShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureGrad**(gsamplerCubeArray *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*)	Do a texture lookup as in **texture** but with explicit gradients as shown below. The partial derivatives of *P* are with respect to window *x* and window *y*. For the cube version, the partial derivatives of *P* are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face.
gvec4 **textureGradOffset**(gsampler2D *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) gvec4 **textureGradOffset**(gsampler3D *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy*, ivec3 *offset*) float **textureGradOffset**(sampler2DShadow *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) gvec4 **textureGradOffset**(gsampler2DArray *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *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**(gsampler2D *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureProjGrad**(gsampler2D *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*) gvec4 **textureProjGrad**(gsampler3D *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*) float **textureProjGrad**(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.
gvec4 **textureProjGradOffset**(gsampler2D *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) gvec4 **textureProjGradOffset**(gsampler2D *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*) gvec4 **textureProjGradOffset**(gsampler3D *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*, ivec3 *offset*) float **textureProjGradOffset**(sampler2DShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*)	Do a texture lookup projectively and with explicit gradient as described in **textureProjGrad**, as well as with offset, as described in **textureOffset**.

8.9.3. Explicit Gradients

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

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

8.9.4. Texture Gather Functions

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

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

The selected component is identified by the optional *comp* argument, where the values zero, one, two, and three identify the R_s, G_s, B_s, or A_s component, respectively. If *comp* is omitted, it is treated as identifying the R_s component.

Incomplete textures (see section 8.16 “Texture Completeness” of the OpenGL ES Specification) return a texture source color of (0,0,0,1) for all four source texels.

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

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

The **textureGatherOffset** built-in functions from the OpenGL ES Shading Language return a vector derived from sampling four texels in the image array of level $level_{base}$. For each of the four texel offsets specified by the *offsets* argument, the rules for the LINEAR minification filter are applied to identify a 2 × 2 texel footprint, from which the single texel T_{i0j0} is selected. A four-component vector is then assembled by taking a single component from each of the four T_{i0j0} texels in the same manner as for the **textureGather** function.

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*])
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*)
	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*]) vec4 **textureGatherOffset**(sampler2DShadow *sampler*, vec2 *P*, float *refZ*, ivec2 *offset*) vec4 **textureGatherOffset**(sampler2DArrayShadow *sampler*, vec3 *P*, float *refZ*, ivec2 *offset*)	Perform a texture gather operation as in **textureGather** by *offset* as described in **textureOffset** except that the *offset* can be variable (non constant) and the implementation-dependent minimum and maximum offset values are given by MIN_PROGRAM_TEXTURE_GATHER_OFFSET and MAX_PROGRAM_TEXTURE_GATHER_OFFSET, respectively.
gvec4 **textureGatherOffsets**(gsampler2D *sampler*, vec2 *P*, ivec2 *offsets*[4] [, int *comp*]) gvec4 **textureGatherOffsets**(gsampler2DArray *sampler*, vec3 *P*, ivec2 *offsets*[4] [, int *comp*]) vec4 **textureGatherOffsets**(sampler2DShadow *sampler*, vec2 *P*, float *refZ*, ivec2 *offsets*[4]) vec4 **textureGatherOffsets**(sampler2DArrayShadow *sampler*, vec3 *P*, float *refZ*, ivec2 *offsets*[4])	Operate identically to **textureGatherOffset** except that *offsets* is used to determine the location of the four texels to sample. Each of the four texels is obtained by applying the corresponding offset in *offsets* as a (*u*, *v*) coordinate offset to *P*, identifying the four-texel LINEAR footprint, and then selecting the texel i_0 j_0 of that footprint. The specified values in *offsets* must be constant integral expressions.

8.10. Atomic Counter Functions

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

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

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

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

Atomic memory functions are supported only for a limited set of variables. A shader will fail to compile if the value passed to the *mem* argument of an atomic memory function does not correspond to a buffer or shared variable. It is acceptable to pass an element of an array or a single component of a vector to the *mem* argument of an atomic memory function, as long as the underlying array or vector is a buffer or shared variable.

All the built-in functions in this section accept arguments with combinations of **restrict**, **coherent**, and **volatile** memory qualification, despite not having them listed in the prototypes. The atomic operation will operate as required by the calling argument's memory qualification, not by the built-in function’s formal parameter memory qualification.

Syntax	**Description**
uint **atomicAdd**(inout uint *mem*, uint *data*) int **atomicAdd**(inout int *mem*, int *data*)	Computes a new value by adding the value of *data* to the contents *mem*.
uint **atomicMin**(inout uint *mem*, uint *data*) int **atomicMin**(inout int *mem*, int *data*)	Computes a new value by taking the minimum of the value of *data* and the contents of *mem*.
uint **atomicMax**(inout uint *mem*, uint *data*) int **atomicMax**(inout int *mem*, int *data*)	Computes a new value by taking the maximum of the value of *data* and the contents of *mem*.
uint **atomicAnd**(inout uint *mem*, uint *data*) int **atomicAnd**(inout int *mem*, int *data*)	Computes a new value by performing a bit-wise AND of the value of *data* and the contents of *mem*.
uint **atomicOr**(inout uint *mem*, uint *data*) int **atomicOr**(inout int *mem*, int *data*)	Computes a new value by performing a bit-wise OR of the value of *data* and the contents of *mem*.
uint **atomicXor**(inout uint *mem*, uint *data*) int **atomicXor**(inout int *mem*, int *data*)	Computes a new value by performing a bit-wise EXCLUSIVE OR of the value of *data* and the contents of *mem*.
uint **atomicExchange**(inout uint *mem*, uint *data*) int **atomicExchange**(inout int *mem*, int *data*)	Computes a new value by simply copying the value of *data*.
uint **atomicCompSwap**(inout uint *mem*, uint *compare*, uint *data*) int **atomicCompSwap**(inout int *mem*, int *compare*, int *data*)	Compares the value of *compare* and the contents of *mem*. If the values are equal, the new value is given by *data*; otherwise, it is taken from the original contents of *mem*.

8.12. Image Functions

Variables using one of the image basic types may be used by the built-in shader image memory functions defined in this section to read and write individual texels of a texture. Each image variable references an image unit, which has a texture image attached.

When image memory functions below access memory, an individual texel in the image is identified using an (*i*), (*i, j*), or (*i, j, k*) coordinate corresponding to the values of *P*. The coordinates are used to select an individual texel in the manner described in section 8.22 “Texture Image Loads and Stores” of the OpenGL ES 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 18 separate functions, each for a different type of image variable. The *IMAGE_PARAMS* placeholder is replaced by one of the following parameter lists:

gimage2D *image*, ivec2 *P*

gimage3D *image*, ivec3 *P*

gimageCube *image*, ivec3 *P*

gimageBuffer *image*, int *P*

gimage2DArray *image*, ivec3 *P*

gimageCubeArray *image*, ivec3 *P*

where each of the lines represents one of three different image variable types, and *image*, *P* specify the individual texel to operate on. The method for identifying the individual texel operated on from *image*, *P*, and the method for reading and writing the texel are specified in section 8.22 “Texture Image Loads and Stores” of the OpenGL ES Specification.

The atomic functions perform operations on individual texels or samples of an image variable. Atomic memory operations read a value from the selected texel, compute a new value using one of the operations described below, write the new value to the selected texel, and return the original value read. The contents of the texel being updated by the atomic operation are guaranteed not to be modified by any other image store or atomic function between the time the original value is read and the time the new value is written.

Atomic memory operations are supported on only a subset of all image variable types; *image* must be either:

- a signed integer image variable (type starts “**iimage**”) and a format qualifier of **r32i**, used with a *data* argument of type **int**, or
- an unsigned integer image variable (type starts “**uimage**”) and a format qualifier of **r32ui**, used with a *data* argument of type **uint**, or
- a float image variable (type starts “**image**”) and a format qualifier of **r32f**, used with a *data* argument of type **float** (**imageAtomicExchange** only).

All the built-in functions in this section accept arguments with combinations of **restrict**, **coherent**, and **volatile** memory qualification, despite not having them listed in the prototypes. The image operation will operate as required by the calling argument’s memory qualification, not by the built-in function’s formal parameter memory qualification.

Syntax	**Description**
highp ivec2 **imageSize**(readonly writeonly gimage2D *image*) highp ivec3 **imageSize**(readonly writeonly gimage3D *image*) highp ivec2 **imageSize**(readonly writeonly gimageCube *image*) highp ivec3 **imageSize**(readonly writeonly gimageCubeArray *image*) highp ivec3 **imageSize**(readonly writeonly gimage2DArray *image*) highp int **imageSize**(readonly writeonly gimageBuffer *image*)	Returns the dimensions of the image or images bound to *image*. For arrayed images, the last component of the return value will hold the size of the array. Cube images only return the dimensions of one face, and the number of cubes in the cube map array, if arrayed. Note: The qualification **readonly writeonly** accepts a variable qualified with **readonly**, **writeonly**, both, or neither. It means the formal argument will be used for neither reading nor writing to the underlying memory.

Syntax	Description
highp gvec4 **imageLoad**(readonly *IMAGE_PARAMS*)	Loads the texel at the coordinate *P* from the image unit *image* (in *IMAGE_PARAMS*). When *image* and *P* identify a valid texel, the bits used to represent the selected texel in memory are converted to a **vec4**, **ivec4**, or **uvec4** in the manner described in section 8.23 "Texture Image Loads and Stores" of the OpenGL ES 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*. When *image* and *P* identify a valid texel, the bits used to represent *data* are converted to the format of the image unit in the manner described in section 8.23 "Texture Image Loads and Stores" of the OpenGL ES Specification and stored to the specified texel.
highp uint **imageAtomicAdd**(*IMAGE_PARAMS*, uint *data*) highp int **imageAtomicAdd**(*IMAGE_PARAMS*, int *data*)	Computes a new value by adding the value of *data* to the contents of the selected texel.
highp uint **imageAtomicMin**(*IMAGE_PARAMS*, uint *data*) highp 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.
highp uint **imageAtomicMax**(*IMAGE_PARAMS*, uint *data*) highp 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.
highp uint **imageAtomicAnd**(*IMAGE_PARAMS*, uint *data*) highp 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.
highp uint **imageAtomicOr**(*IMAGE_PARAMS*, uint *data*) highp 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.
highp uint **imageAtomicXor**(*IMAGE_PARAMS*, uint *data*) highp 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.
highp uint **imageAtomicExchange** (*IMAGE_PARAMS*, uint *data*) highp int **imageAtomicExchange** (*IMAGE_PARAMS*, int *data*) highp float **imageAtomicExchange** (*IMAGE_PARAMS*, float *data*)	Computes a new value by simply copying the value of *data*.

Syntax	Description
highp uint **imageAtomicCompSwap** (*IMAGE_PARAMS*, uint *compare*, uint *data*) highp int **imageAtomicCompSwap** (*IMAGE_PARAMS*, int *compare*, int *data*)	Compares the value of *compare* and the contents of the selected texel. If the values are equal, the new value is given by *data*; otherwise, it is taken from the original value loaded from the texel.

8.13. Geometry Shader Functions

These functions are only available in geometry shaders. They are described in more depth following the table.

Syntax	Description
void **EmitVertex**()	Emits the current values of output variables to the current output primitive. On return from this call, the values of output variables are undefined.
void **EndPrimitive**()	Completes the current output primitive and starts a new one. No vertex is emitted.

The function **EmitVertex**() specifies that a vertex is completed. A vertex is added to the current output primitive using the current values of all built-in and user-defined output variables. The values of all output variables are undefined after a call to **EmitVertex**(). If a geometry shader invocation has emitted more vertices than permitted by the output layout qualifier **max_vertices**, the results of calling **EmitVertex**() are undefined.

The function **EndPrimitive**() specifies that the current output primitive is completed and a new output primitive (of the same type) will be started by any subsequent **EmitVertex**(). This function does not emit a vertex. If the output layout is declared to be **points**, calling **EndPrimitive**() is optional.

A geometry shader starts with an output primitive containing no vertices. When a geometry shader terminates, the current output primitive is automatically completed. It is not necessary to call **EndPrimitive**() if the geometry shader writes only a single primitive.

8.14. Fragment Processing Functions

Fragment processing functions are only available in fragment shaders.

8.14.1. Derivative Functions

Derivatives may be computationally expensive and/or numerically unstable. Therefore, an OpenGL ES implementation may approximate the true derivatives by using a fast but not entirely accurate derivative computation. Derivatives are undefined within non-uniform control flow.

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

Forward differencing:

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

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

Backward differencing:

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

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

With single-sample rasterization, $dx \le 1.0$ in equations 1b and 2b. For multisample rasterization, $dx < 2.0$ in equations 1b and 2b.

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

An OpenGL ES implementation may use the above or other methods to perform the calculation, subject to the following conditions:

1. The method may use piecewise linear approximations. Such linear approximations imply that higher order derivatives, **dFdx(dFdx(***x***))** and above, are undefined.
2. The method may assume that the function evaluated is continuous. Therefore derivatives within the body of a non-uniform conditional are undefined.
3. The method may differ per fragment, subject to the constraint that the method may vary by window coordinates, not screen coordinates. The invariance requirement described in section 13.2 “Invariance” of the OpenGL ES 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:

1. Functions should be evaluated within the interior of a primitive (interpolated, not extrapolated).
2. 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.
3. Derivatives of constant arguments should be 0.

In some implementations, varying degrees of derivative accuracy may be obtained by providing GL hints (see section 19.1 “Hints” of the OpenGL ES Specification), allowing a user to make an image quality versus speed trade off.

Syntax	**Description**
genFType **dFdx**(genFType *p*)	Returns the derivative in x using local differencing for the input argument *p*.

Syntax	Description
genFType **dFdy**(genFType *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.
genFType **fwidth**(genFType *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*));

8.14.2. Interpolation Functions

Built-in interpolation functions are available to compute an interpolated value of a fragment shader input variable at a shader-specified (*x*, *y*) location. A separate (*x*, *y*) location may be used for each invocation of the built-in function, and those locations may differ from the default (*x*, *y*) location used to produce the default value of the input. For the **interpolateAt*** functions, the call will return a precision qualification matching the precision of the *interpolant* argument to the function call.

For all of the interpolation functions, *interpolant* must be an l-value from an **in** declaration; this can be either a variable, or an array element. Component selection operators (e.g. **.xy**) may not be used when specifying *interpolant*.

If *interpolant* is declared with the **flat** qualifier, the interpolated value will have the same value everywhere for a single primitive, so the location used for interpolation has no effect and the functions just return that same value. If *interpolant* is declared with the **centroid** qualifier, the value returned by **interpolateAtSample**() and **interpolateAtOffset**() will be evaluated at the specified location, ignoring the location normally used with the **centroid** qualifier.

Syntax	Description
float **interpolateAtCentroid**(float *interpolant*) vec2 **interpolateAtCentroid**(vec2 *interpolant*) vec3 **interpolateAtCentroid**(vec3 *interpolant*) vec4 **interpolateAtCentroid**(vec4 *interpolant*)	Returns the value of the input *interpolant* sampled at a location inside both the pixel and the primitive being processed. The value obtained would be the same value assigned to the input variable if declared with the **centroid** qualifier.

Syntax	Description
float **interpolateAtSample**(float *interpolant*, int *sample*) vec2 **interpolateAtSample**(vec2 *interpolant*, int *sample*) vec3 **interpolateAtSample**(vec3 *interpolant*, int *sample*) vec4 **interpolateAtSample**(vec4 *interpolant*, int *sample*)	Returns the value of the input *interpolant* variable at the location of sample number *sample*. If multisample buffers are not available, the input variable will be evaluated at the center of the pixel. If sample *sample* does not exist, the position used to interpolate the input variable is undefined.
float **interpolateAtOffset**(float *interpolant*, vec2 offset) vec2 **interpolateAtOffset**(vec2 *interpolant*, vec2 offset) vec3 **interpolateAtOffset**(vec3 *interpolant*, vec2 offset) vec4 **interpolateAtOffset**(vec4 *interpolant*, vec2 offset)	Returns the value of the input *interpolant* variable sampled at an offset from the center of the pixel specified by *offset*. The two floating-point components of *offset*, give the offset in pixels in the *x* and *y* directions, respectively. An offset of (0, 0) identifies the center of the pixel. The range and granularity of offsets supported by this function is implementation-dependent.

8.15. Shader Invocation Control Functions

The shader invocation control function is only available in tessellation control and compute shaders. It is used to control the relative execution order of multiple shader invocations used to process a patch (in the case of tessellation control shaders) or a workgroup (in the case of compute shaders), which are otherwise executed with an undefined relative order.

Syntax	Description
void **barrier**()	For any given static instance of **barrier**(), all tessellation control shader invocations for a single input patch must enter it before any will be allowed to continue beyond it, or all compute shader invocations for a single workgroup must enter it before any will continue beyond it.

The function **barrier**() provides a partially defined order of execution between shader invocations. The ensures that, for some types of memory accesses, 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 an undefined order between these barrier calls, the values of a per-vertex or per-patch output variable for tessellation control shaders, or the values of **shared** variables for compute shaders will be undefined in a number of cases enumerated in "Output Variables" (for tessellation control shaders) and "Shared Variables" (for compute shaders).

For tessellation control shaders, the **barrier**() function may only be placed inside the function **main**() of the shader and may not be called within any control flow. Barriers are also disallowed after a return statement in the function **main**(). Any such misplaced barriers result in a compile-time error.

A **barrier**() affects control flow but only synchronizes memory accesses to **shared** variables and tessellation control output variables. For other memory accesses, it does not ensure 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 of **barrier**(). To achieve this requires the use of both **barrier**() and a memory barrier.

For compute shaders, the **barrier**() function may be placed within control flow, but that control flow must be uniform control flow. That is, all the controlling expressions that lead to execution of the barrier must be dynamically uniform expressions. This ensures that if any shader invocation enters a conditional statement, then all invocations will enter it. While compilers are encouraged to give warnings if they can detect this might not happen, compilers cannot completely determine this. Hence, it is the author's responsibility to ensure **barrier**() only exists inside uniform control flow. Otherwise, some shader invocations will stall indefinitely, waiting for a barrier that is never reached by other invocations.

8.16. Shader Memory Control Functions

Within a single shader invocation, the visibility and order of writes made by that invocation are well-defined. However, the relative order of reads and writes to a single shared memory address from multiple separate shader invocations is largely undefined. Additionally, the order of accesses to multiple memory addresses performed by a single shader invocation, as observed by other shader invocations, is also undefined.

The following built-in functions can be used to control the ordering of reads and writes:

Syntax	Description
void **memoryBarrier**()	Control the ordering of memory transactions issued by a single shader invocation.
void **memoryBarrierAtomicCounter**()	Control the ordering of accesses to atomic-counter variables issued by a single shader invocation.
void **memoryBarrierBuffer**()	Control the ordering of memory transactions to buffer variables issued within a single shader invocation.
void **memoryBarrierShared**()	Control the ordering of memory transactions to shared variables issued within a single shader invocation, as viewed by other invocations in the same workgroup. Only available in compute shaders.
void **memoryBarrierImage**()	Control the ordering of memory transactions to images issued within a single shader invocation.
void **groupMemoryBarrier**()	Control the ordering of all memory transactions issued within a single shader invocation, as viewed by other invocations in the same workgroup. Only available in compute shaders.

The memory barrier built-in functions can be used to order reads and writes to variables stored in memory accessible to other shader invocations. When called, these functions will wait for the completion of all reads and writes previously performed by the caller that access selected variable types, and then return with no other effect. The built-in functions **memoryBarrierAtomicCounter**(), **memoryBarrierBuffer**(), **memoryBarrierImage**(), and **memoryBarrierShared**() wait for the completion of accesses to atomic counter, buffer, image, and shared variables, respectively. The built-in functions **memoryBarrier**() and **groupMemoryBarrier**() wait for the completion of accesses to all of the above variable types. The functions **memoryBarrierShared**() and **groupMemoryBarrier**() are available only in compute shaders; the other functions are available in all shader types.

When these functions return, the effects of any memory stores performed using coherent variables prior to the call will be visible to any future[1] coherent access to the same memory performed by any other shader invocation. In particular, the values written this way in one shader stage are guaranteed to be visible to coherent memory accesses performed by shader invocations in subsequent stages when those invocations were triggered by the execution of the original shader invocation (e.g. fragment shader invocations for a primitive resulting from a particular geometry shader invocation).

1

An access is only a *future* access if a *happens-before* relation can be established between the store and the load.

Additionally, memory barrier functions order stores performed by the calling invocation, as observed by other shader invocations. Without memory barriers, if one shader invocation performs two stores to coherent variables, a second shader invocation might see the values written by the second store prior to seeing those written by the first. However, if the first shader invocation calls a memory barrier function between the two stores, selected other shader invocations will never see the results of the second store before seeing those of the first. When using the functions **groupMemoryBarrier**() or **memoryBarrierShared**(), this ordering guarantee applies only to other shader invocations in the same compute shader workgroup; all other memory barrier functions provide the guarantee to all other shader invocations. No memory barrier is required to guarantee the order of memory stores as observed by the invocation performing the stores; an invocation reading from a variable that it previously wrote will always see the most recently written value unless another shader invocation also wrote to the same memory.

Chapter 9. Shader Interface Matching

As described in chapter 7 of the OpenGL ES Specification, shaders may be linked together to form a *program object* before being bound to the pipeline or may be linked and bound individually as *separable program objects*.

Note

These were previously known as separate shader objects (SSOs) but the mechanism has been extended to support future versions of the specification that have more than two shader stages. It allows a subset of the shaders to be linked together.

Within a *program object* or a *separable program object*, qualifiers for matching variables must themselves match according to the rules specified in this section. There are also matching rules for qualifiers of matching variables between separable program objects but only for variables across an input/output boundary between shader stages. For other shader interface variables such as uniforms, each program object or separable program object has its own name space and so the same name can refer to multiple independent variables. Consequently, there are no matching rules for qualifiers in these cases.

9.1. Input Output Matching by Name in Linked Programs

When linking shaders, the type of declared vertex outputs and fragment inputs with the same name must match, otherwise the link command will fail. Only those fragment inputs statically used (i.e. read) in the fragment shader must be declared as outputs in the vertex shader; declaring superfluous vertex shader outputs is permissible.

The following table summarizes the rules for matching shader outputs to shader inputs in consecutive stages when shaders are linked together.

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
	Declares; no static Use	Allowed	Allowed	Allowed (values are undefined)
	Declares and static Use	Allowed	Allowed	Allowed (values are potentially undefined)

See “Static Use” for the definition of *static use*.

The precision of a vertex output does not need to match the precision of the corresponding fragment input. The minimum precision at which vertex outputs are interpolated is the minimum of the vertex output precision and the fragment input precision, with the exception that for **highp**, implementations do not have to support full IEEE 754 precision.

The precision of values exported to a transform feedback buffer is the precision of the outputs of the vertex shader. However, they are converted to **highp** format before being written.

9.2. Matching of Qualifiers

The following tables summarize the requirements for matching of qualifiers. It applies whenever there are two or more matching variables in a shader interface.

Notes:

1. *Yes* means the qualifiers must match.
2. *No* means the qualifiers do not need to match.
3. *Consistent* means qualifiers may be missing from a subset of declarations but they cannot conflict
4. If there are conflicting qualifiers, only the last of these is significant.
5. Matching is based only on the resulting qualification, not on the presence or otherwise of qualifiers.
6. The rules apply to all declared variables, irrespective of whether they are statically used, with the exception of inputs and outputs when shaders are linked (see "Input Output Matching by Name in Linked Programs").
7. Errors are generated for any conflicts.

9.2.1. Linked Shaders

Qualifier Class	**Qualifier**	**in/out**	**Default Uniforms**	**uniform Block**	**buffer Block**
Storage[1]	**in** **out** **uniform**	N/A	N/A	N/A	N/A
Auxiliary	**centroid** **sample**	No	N/A	N/A	N/A
	patch	Yes	N/A	N/A	N/A
Layout	**location**	Yes[2]	Consistent	N/A	N/A
	Block layout[3,4]	N/A	N/A	Yes	Yes
	binding	N/A	Consistent	Yes	Yes
	offset	N/A	Yes	N/A	N/A
	format	N/A	Yes	N/A	N/A
Interpolation	**smooth** **flat**	Yes	N/A	N/A	N/A

Qualifier Class	Qualifier	in/out	Default Uniforms	uniform Block	buffer Block
Precision	**lowp** **mediump** **highp**	No	Yes	No	No
Variance	**invariant** **precise**	No	N/A	N/A	N/A
Memory	all	N/A	Yes	Yes	Yes

1

Storage qualifiers determine *when* variables match rather than being *required* to match for matching variables. Note also that each shader interface has a separate name space so for example, it is possible to use the same name for a vertex output and fragment uniform.

2

If present, the **location** qualifier determines the matching of inputs and outputs. See section 7.4.1 “Shader interface Matching” of the OpenGL ES Specification for details.]

3

The **row_major** and **column_major** layout qualifiers do not need to match when applied to non-matrix types.

4

In cases where a layout qualifier overrides a previous layout qualifier or a default, only the resulting qualification must match.

9.2.2. Separable Programs

Qualifier Class	Qualifier	in/out
Storage	**in** **out** **uniform**	N/A
Auxiliary	**centroid** **sample**	No
	patch	Yes
Layout	**location**	Yes
	Block layout	N/A
	binding	N/A
	offset	N/A
	format	N/A
Interpolation	**smooth** **flat**	Yes
Precision	**lowp** **mediump** **highp**	Yes

Qualifier Class	**Qualifier**	**in/out**
Variance	**invariant** **precise**	No
Memory	all	N/A

Chapter 10. Shading Language Grammar

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

```
CONST BOOL FLOAT INT UINT

BVEC2 BVEC3 BVEC4 IVEC2 IVEC3 IVEC4 UVEC2 UVEC3 UVEC4 VEC2 VEC3 VEC4

MAT2 MAT3 MAT4 CENTROID IN OUT INOUT

UNIFORM PATCH SAMPLE BUFFER SHARED

COHERENT VOLATILE RESTRICT READONLY WRITEONLY

FLAT SMOOTH LAYOUT

MAT2X2 MAT2X3 MAT2X4

MAT3X2 MAT3X3 MAT3X4

MAT4X2 MAT4X3 MAT4X4




ATOMIC_UINT

SAMPLER2D SAMPLER3D SAMPLERCUBE SAMPLER2DSHADOW

SAMPLERCUBESHADOW SAMPLER2DARRAY SAMPLER2DARRAYSHADOW

ISAMPLER2D ISAMPLER3D ISAMPLERCUBE ISAMPLER2DARRAY

USAMPLER2D USAMPLER3D USAMPLERCUBE USAMPLER2DARRAY




SAMPLERBUFFER ISAMPLERBUFFER USAMPLERBUFFER

SAMPLERCUBEARRAY SAMPLERCUBEARRAYSHADOW

ISAMPLERCUBEARRAY USAMPLERCUBEARRAY

SAMPLER2DMS ISAMPLER2DMS USAMPLER2DMS
```

```
SAMPLER2DMSARRAY ISAMPLER2DMSARRAY USAMPLER2DMSARRAY

IMAGE2D IIMAGE2D UIMAGE2D

IMAGE3D IIMAGE3D UIMAGE3D

IMAGECUBE IIMAGECUBE UIMAGECUBE

IMAGEBUFFER IIMAGEBUFFER UIMAGEBUFFER

IMAGE2DARRAY IIMAGE2DARRAY UIMAGE2DARRAY

IMAGECUBEARRAY IIMAGECUBEARRAY UIMAGECUBEARRAY




STRUCT VOID

WHILE BREAK CONTINUE DO ELSE FOR IF DISCARD RETURN SWITCH CASE DEFAULT

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 ES Shading Language in terms of the above tokens. The starting rule is *translation_unit*.

variable_identifier :

IDENTIFIER

primary_expression :

variable_identifier
INTCONSTANT
UINTCONSTANT
FLOATCONSTANT
BOOLCONSTANT
LEFT_PAREN expression RIGHT_PAREN

postfix_expression :

primary_expression
postfix_expression LEFT_BRACKET integer_expression RIGHT_BRACKET
function_call
postfix_expression DOT FIELD_SELECTION
postfix_expression INC_OP
postfix_expression DEC_OP

FIELD_SELECTION includes members in structures, component selection for vectors and the 'length' identifier for the length() method

integer_expression :

expression

function_call :

function_call_or_method

function_call_or_method :

function_call_generic

function_call_generic :

function_call_header_with_parameters RIGHT_PAREN
function_call_header_no_parameters RIGHT_PAREN

function_call_header_no_parameters :

function_call_header VOID
function_call_header

function_call_header_with_parameters :

function_call_header assignment_expression
function_call_header_with_parameters COMMA assignment_expression

function_call_header :

function_identifier LEFT_PAREN

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

Methods (**.length**) and identifiers are recognized through *postfix_expression*.

function_identifier **:**

type_specifier
postfix_expression

unary_expression **:**

postfix_expression
INC_OP unary_expression
DEC_OP unary_expression
unary_operator unary_expression

Grammar Note: No traditional style type casts.

unary_operator **:**

PLUS
DASH
BANG
TILDE

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

multiplicative_expression **:**

unary_expression
multiplicative_expression STAR unary_expression
multiplicative_expression SLASH unary_expression
multiplicative_expression PERCENT unary_expression

additive_expression **:**

multiplicative_expression
additive_expression PLUS multiplicative_expression
additive_expression DASH multiplicative_expression

shift_expression **:**

additive_expression
shift_expression LEFT_OP additive_expression
shift_expression RIGHT_OP additive_expression

relational_expression **:**

shift_expression
relational_expression LEFT_ANGLE shift_expression
relational_expression RIGHT_ANGLE shift_expression
relational_expression LE_OP shift_expression
relational_expression GE_OP shift_expression

equality_expression :

relational_expression
equality_expression EQ_OP relational_expression
equality_expression NE_OP relational_expression

and_expression :

equality_expression
and_expression AMPERSAND equality_expression

exclusive_or_expression :

and_expression
exclusive_or_expression CARET and_expression

inclusive_or_expression :

exclusive_or_expression
inclusive_or_expression VERTICAL_BAR exclusive_or_expression

logical_and_expression :

inclusive_or_expression
logical_and_expression AND_OP inclusive_or_expression

logical_xor_expression :

logical_and_expression
logical_xor_expression XOR_OP logical_and_expression

logical_or_expression :

logical_xor_expression
logical_or_expression OR_OP logical_xor_expression

conditional_expression :

logical_or_expression
logical_or_expression QUESTION expression COLON assignment_expression

assignment_expression :

conditional_expression
unary_expression assignment_operator assignment_expression

assignment_operator :

EQUAL
MUL_ASSIGN
DIV_ASSIGN
MOD_ASSIGN
ADD_ASSIGN
SUB_ASSIGN
LEFT_ASSIGN
RIGHT_ASSIGN
AND_ASSIGN
XOR_ASSIGN

OR_ASSIGN

expression **:**

assignment_expression
expression COMMA assignment_expression

constant_expression **:**

conditional_expression

declaration **:**

function_prototype SEMICOLON
init_declarator_list SEMICOLON
PRECISION precision_qualifier type_specifier SEMICOLON
type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE SEMICOLON
type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE IDENTIFIER SEMICOLON
type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE IDENTIFIER array_specifier SEMICOLON
type_qualifier SEMICOLON
type_qualifier IDENTIFIER SEMICOLON
type_qualifier IDENTIFIER identifier_list SEMICOLON

identifier_list **:**

COMMA IDENTIFIER
identifier_list COMMA IDENTIFIER

function_prototype **:**

function_declarator RIGHT_PAREN

function_declarator **:**

function_header
function_header_with_parameters

function_header_with_parameters **:**

function_header parameter_declaration
function_header_with_parameters COMMA parameter_declaration

function_header **:**

fully_specified_type IDENTIFIER LEFT_PAREN

parameter_declarator **:**

type_specifier IDENTIFIER
type_specifier IDENTIFIER array_specifier

parameter_declaration **:**

type_qualifier parameter_declarator
parameter_declarator
type_qualifier parameter_type_specifier

parameter_type_specifier

parameter_type_specifier :

type_specifier

init_declarator_list :

single_declaration
init_declarator_list COMMA IDENTIFIER
init_declarator_list COMMA IDENTIFIER array_specifier
init_declarator_list COMMA IDENTIFIER array_specifier EQUAL initializer
init_declarator_list COMMA IDENTIFIER EQUAL initializer

single_declaration :

fully_specified_type
fully_specified_type IDENTIFIER
fully_specified_type IDENTIFIER array_specifier
fully_specified_type IDENTIFIER array_specifier EQUAL initializer
fully_specified_type IDENTIFIER EQUAL initializer

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

fully_specified_type :

type_specifier
type_qualifier type_specifier

invariant_qualifier :

INVARIANT

interpolation_qualifier :

SMOOTH
FLAT

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
IDENTIFIER EQUAL UINTCONSTANT
SHARED

precise_qualifier :

PRECISE

type_qualifier :

single_type_qualifier
type_qualifier single_type_qualifier

single_type_qualifier :

storage_qualifier
layout_qualifier
precision_qualifier
interpolation_qualifier
invariant_qualifier
precise_qualifier

storage_qualifier :

CONST
IN
OUT
INOUT
CENTROID
PATCH
SAMPLE
UNIFORM
BUFFER
SHARED
COHERENT
VOLATILE
RESTRICT
READONLY
WRITEONLY

type_specifier :

type_specifier_nonarray
type_specifier_nonarray array_specifier

array_specifier :

LEFT_BRACKET RIGHT_BRACKET
LEFT_BRACKET constant_expression RIGHT_BRACKET
array_specifier LEFT_BRACKET RIGHT_BRACKET
array_specifier LEFT_BRACKET constant_expression RIGHT_BRACKET

type_specifier_nonarray :

VOID
FLOAT
INT
UINT
BOOL
VEC2
VEC3
VEC4

BVEC2
BVEC3
BVEC4
IVEC2
IVEC3
IVEC4
UVEC2
UVEC3
UVEC4
MAT2
MAT3
MAT4
MAT2X2
MAT2X3
MAT2X4
MAT3X2
MAT3X3
MAT3X4
MAT4X2
MAT4X3
MAT4X4
ATOMIC_UINT
SAMPLER2D
SAMPLER3D
SAMPLERCUBE
SAMPLER2DSHADOW
SAMPLERCUBESHADOW
SAMPLER2DARRAY
SAMPLER2DARRAYSHADOW
SAMPLERCUBEARRAY
SAMPLERCUBEARRAYSHADOW
ISAMPLER2D
ISAMPLER3D
ISAMPLERCUBE
ISAMPLER2DARRAY
ISAMPLERCUBEARRAY
USAMPLER2D
USAMPLER3D
USAMPLERCUBE
USAMPLER2DARRAY
USAMPLERCUBEARRAY
SAMPLERBUFFER
ISAMPLERBUFFER
USAMPLERBUFFER
SAMPLER2DMS
ISAMPLER2DMS
USAMPLER2DMS
SAMPLER2DMSARRAY

ISAMPLER2DMSARRAY
USAMPLER2DMSARRAY
IMAGE2D
IIMAGE2D
UIMAGE2D
IMAGE3D
IIMAGE3D
UIMAGE3D
IMAGECUBE
IIMAGECUBE
UIMAGECUBE
IMAGEBUFFER
IIMAGEBUFFER
UIMAGEBUFFER
IMAGE2DARRAY
IIMAGE2DARRAY
UIMAGE2DARRAY
IMAGECUBEARRAY
IIMAGECUBEARRAY
UIMAGECUBEARRAY
struct_specifier
TYPE_NAME

precision_qualifier :

HIGH_PRECISION
MEDIUM_PRECISION
LOW_PRECISION

struct_specifier :

STRUCT IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
STRUCT LEFT_BRACE struct_declaration_list RIGHT_BRACE

struct_declaration_list :

struct_declaration
struct_declaration_list struct_declaration

struct_declaration :

type_specifier struct_declarator_list SEMICOLON
type_qualifier type_specifier struct_declarator_list SEMICOLON

struct_declarator_list :

struct_declarator
struct_declarator_list COMMA struct_declarator

struct_declarator :

IDENTIFIER
IDENTIFIER array_specifier

initializer **:**

  assignment_expression

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

In general the above grammar describes a super set of the OpenGL ES Shading Language. Certain constructs that are valid purely in terms of the grammar are disallowed by statements elsewhere in this specification.

Chapter 11. Counting of Inputs and Outputs

This section applies to outputs from the last active vertex processing stage and inputs to the fragment stage. GLSL ES 3.20 specifies the storage available for such variables in terms of an array of 4-vectors. The assumption is that variables will be packed into these arrays without wasting space. This places significant burden on implementations since optimal packing is computationally intensive. Implementations may have more internal resources than exposed to the application and so avoid the need to perform packing but this is also considered an expensive solution.

GLSL ES 3.20 therefore relaxes the requirements for packing by specifying a simpler algorithm that may be used. This algorithm specifies a minimum requirement for when a set of variables must be supported by an implementation. The implementation is allowed to support more than the minimum and so may use a more efficient algorithm and/or may support more registers than the virtual target machine.

Outputs from the last active vertex stage and inputs to the fragment stage are counted separately. If statically used in the fragment shader, the built-in special variables (*gl_FragCoord*, *gl_FrontFacing* and *gl_PointCoord*) are included when calculating the storage requirements of fragment inputs.

If the last active vertex-pipeline shader and fragment shader are linked together, inputs and outputs are only counted if they are statically used within the shader. If the shaders are each compiled into a separable program, all declared inputs and outputs are counted.

Note

GLSL ES 3.20 does not require the implementation to remove outputs which are not statically used in the fragment shader.

For the algorithm used, failing resource allocation for a variable must result in an error.

The resource allocation of variables must succeed for all cases where the following packing algorithm succeeds:

- The target architecture consists of a grid of registers, 16 rows by 4 columns. Each register can contain a scalar value, i.e. a float, int or uint.
- Variables with an explicit location are allocated first. When attempting to allocate a location for other variables, if there is a conflict, the search moves to the next available free location.
- Structures are assumed to be flattened. Each data member is treated as if it were at global scope.
- Variables are packed into the registers one at a time so that they each occupy a contiguous sub-rectangle. No splitting of variables is permitted.
- The orientation of variables is fixed. Vectors always occupy registers in a single row. Elements of an array must be in different rows. E.g. **vec4** will always occupy one row; float[16] will occupy one column. Since it is not permitted to split a variable, large arrays e.g. float[32] will always fail with this algorithm.
- Non-square matrices of type **matCxR** consume the same space as a square matrix of type **matN** where N is the greater of C and R. Variables of type **mat2** occupies 2 complete rows. These rules

allow implementations more flexibility in how variables are stored. + Other variables consume only the minimum space required.

- Arrays of size N are assumed to take N times the size of the base type.
- Variables are packed in the following order:
  1. **mat4** and arrays of **mat4**.
  2. **mat2** and arrays of **mat2** (since they occupy full rows)
  3. **vec4** and arrays of **vec4**
  4. **mat3** and arrays of **mat3**
  5. **vec3** and arrays of **vec3**
  6. **vec2** and arrays of **vec2**
  7. Scalar types and arrays of scalar types
- For each of the above types, the arrays are processed in order of size, largest first. Arrays of size 1 and the base type are considered equivalent. The first type to be packed will be mat4[4], mat4[3], mat[2] followed by mat4, mat2[4]…mat2[2], mat2, vec4[8], vec4[7],…vec4[1], vec4, mat3[2], mat3 and so on. The last variables to be packed will be float (and float[1]).
- For 2,3 and 4 component variables packing is started using the 1st column of the 1st row. Variables are then allocated to successive rows, aligning them to the 1st column.
- For 2 component variables, when there are no spare rows, the strategy is switched to using the highest numbered row and the lowest numbered column where the variable will fit. (In practice, this means they will be aligned to the x or z component.) Packing of any further 3 or 4 component variables will fail at this point.
- 1 component variables (e.g. floats and arrays of floats) have their own packing rule. They are packed in order of size, largest first. Each variable is placed in the column that leaves the least amount of space in the column and aligned to the lowest available rows within that column. During this phase of packing, space will be available in up to 4 columns. The space within each column is always contiguous in the case where no variables have explicit locations.
- For each type, variables with the 'smooth' property are packed first, followed by variables with the 'flat' property.
- Each row can contain either values with the 'smooth' property or the 'flat' property but not both. If this situation is encountered during allocation, the algorithm skips the component location and continues with the next available location. These skipped locations may be used for other values later in the allocation process.
- There is no backtracking. Once a value is assigned a location, it cannot be changed, even if such a change is required for a successful allocation.

Example: pack the following types:

```
out vec4 a;      // top left
out mat3 b;      // align to left, lowest numbered rows
out mat2x3 c;    // same size as mat3, align to left
out vec2 d[6];   // align to left, lowest numbered rows
out vec2 e[4];   // Cannot align to left so align to z column, highest
                 // numbered rows
out vec2 f;      // Align to left, lowest numbered rows.
out float g[3]   // Column with minimum space
out float h[2];  // Column with minimum space (choice of 3, any
                 // can be used)
out float i;     // Column with minimum space
```

In this example, the variables happen to be listed in the order in which they are packed. Packing is independent of the order of declaration.

	x	**y**	**z**	**w**
0	**a**	**a**	**a**	**a**
1	**b**	**b**	**b**	
2	**b**	**b**	**b**	
3	**b**	**b**	**b**	
4	**c**	**c**	**c**	
5	**c**	**c**	**c**	
6	**c**	**c**	**c**	
7	**d**	**d**	**g**	
8	**d**	**d**	**g**	
9	**d**	**d**	**g**	
10	**d**	**d**		
11	**d**	**d**		
12	**d**	**d**	**e**	**e**
13	**f**	**f**	**e**	**e**
14	**h**	**i**	**e**	**e**
15	**h**		**e**	**e**

Some types e.g. mat4[8] will be too large to fit. These always fail with this algorithm.

Chapter 12. Acknowledgments

This specification is based on the work of those who contributed to past versions of the Open GL and Open GL ES Language Specifications and the following contributors to this version:

Acorn Pooley, NVIDIA
Alberto Moreira, Qualcomm
Aleksandra Krstic, Qualcomm
Alon Or-bach, Nokia & Samsung
Andrzej Kacprowski, Intel
Arzhange Safdarzadeh, Intel
Aske Simon Christensen, ARM
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, NVIDIA
Ben Bowman, Imagination Technologies
Ben Brierton, Broadcom
Benj Lipchak, Apple
Benson Tao, Vivante
Bill Licea-Kane, Qualcomm
Brent Insko, Intel
Brian Murray, Freescale
Bruce Merry, ARM
Carlos Santa, TI
Cass Everitt, Epic Games & NVIDIA
Cemil Azizoglu, TI
Chang-Hyo Yu, Samsung
Chris Dodd, NVIDIA
Chris Knox, NVIDIA
Chris Tserng, TI
Clay Montgomery, TI
Daniel Kartch, NVIDIA
Daniel Koch, Transgaming& NVIDIA
Daoxiang Gong, Imagination Technologies
Dave Shreiner, ARM
David Garcia, AMD
David Jarmon, Vivante
Derek Cornish, Epic Games
Dominick Witczak, Mobica
Eben Upton, Broadcom
Ed Plowman, ARM
Eisaku Ohbuchi, DMP
Elan Lennard, ARM
Erik Faye-Lund, ARM
Graham Connor, Imagination Technologies
Graham Sellers, AMD
Greg Roth, NVIDIA
Guillaume Portier, Hi Corporation

Guofang Jiao, Qualcomm
Hans-Martin Will, Vincent
Hwanyong Lee, Huone
I-Gene Leong, NVIDIA
Ian Romanick, Intel
Ian South-Dickinson, NVIDIA
Ilan Aelion-Exch, Samsung
Inkyun Lee, Huone
Jacob Strm, Ericsson
James Adams, Broadcom
James Jones, Imagination Technologies
James McCombe, Imagination Technologies
Jamie Gennis, Google
Jan-Harald Fredriksen, ARM
Jani Vaisanen, Nokia
Jarkko Kemppainen, Symbio
Jeff Bolz, NVIDIA
Jeff Leger, Qualcomm
Jeff Vigil, Qualcomm
Jeremy Sandmel, Apple
Jeremy Thorne, Broadcom
Jim Hauxwell, Broadcom
Jinsung Kim, Huone
Jiyoung Yoon, Huone
John Kessenich, Google
Jon Kennedy, 3DLabs
Jon Leech, Khronos
Jonathan Putsman, Imagination Technologies
Joohoon Lee, Samsung
JoukoKylmäoja, Symbio
Jrn Nystad, ARM
Jussi Rasanen, NVIDIA
Kalle Raita, drawElements
Kari Pulli, Nokia
Keith Whitwell, VMware
Kent Miller, Netlogic Microsystems
Kimmo Nikkanen, Nokia
Konsta Karsisto, Nokia
Krzysztof Kaminski, Intel
Larry Seiler, Intel
Lars Remes, Symbio
Lee Thomason, Adobe
Lefan Zhong, Vivante
Marcus Lorentzon, Ericsson
Mark Butler, Imagination Technologies
Mark Callow, Hi Corporation
Mark Cresswell, Broadcom
Mark Snyder, Alt Software

Mark Young, AMD
Mathieu Robart, STM
Matt Netsch, Qualcomm
Matt Russo, Matrox
Maurice Ribble, Qualcomm
Max Kazakov, DMP
Mika Pesonen, Nokia
Mike Cai, Vivante
Mike Weiblen, Zebra Imaging & Qualcomm
Mila Smith, AMD
Nakhoon Baek, Kyungpook Univeristy
Nate Huang, NVIDIA
Neil Trevett, NVIDIA
Nelson Kidd, Intel
Nick Haemel, NVIDIA
Nick Penwarden, Epic Games
Niklas Smedberg, Epic Games
Nizar Romdan, ARM
Oliver Wohlmuth , Fujitsu
Pat Brown, NVIDIA
Paul Ruggieri, Qualcomm
Per Wennersten, Ericsson
Petri Talala, Symbio
Phil Huxley, ZiiLabs
Philip Hatcher, Freescale & Intel
Piers Daniell, NVIDIA
Pyry Haulos, drawElements
Piotr Tomaszewski, Ericsson
Piotr Uminski, Intel
Rami Mayer, Samsung
Rauli Laatikainen, RightWare
Rob Barris, NVIDIA
Rob Simpson, Qualcomm
Roj Langhi, Vivante
Rune Holm, ARM
Sami Kyostila, Nokia
Sean Ellis, ARM
Shereef Shehata, TI
Sila Kayo, Nokia
Slawomir Cygan, Intel
Slawomir Grajewski, Intel
Steve Hill, STM & Broadcom
Steven Olney, DMP
Suman Sharma, Intel
Tapani Palli, Nokia
Teemu Laakso, Symbio
Tero Karras, NVIDIA
Timo Suoranta, Imagination Technologies & Broadcom

Tom Cooksey, ARM
Tom McReynolds, NVIDIA
Tom Olson, TI & ARM
Tomi Aarnio, Nokia
Tommy Asano, Takumi
Wes Bang, Nokia
Yanjun Zhang, Vivante

Chapter 13. Normative References

1. International Standard ISO/IEC 14882:1998(E). Programming Languages - C++. Referenced for preprocessor only
2. “OpenGL^{R} ES, Version 3.2”, https://www.khronos.org/registry/OpenGL/index_es.php, November 3, 2016.
3. “The OpenGL^{R} Graphics System: A Specification, Version 4.6 (Core Profile)”, https://www.khronos.org/registry/OpenGL/index_gl.php, June 1, 2016.
4. International Standard ISO/IEC 646:1991. Information technology - ISO 7-bit coded character set for information interchange
5. The Unicode Standard Version 6.0 - Core Specification.
6. IEEE 754-2008. *IEEE Standard for Floating-Point Arithmetic*