The OpenGL ES® Shading Language

Language Version: 3.00
Document Revision: 3
11 July 2012

Editor: Robert J. Simpson, Qualcomm

OpenGL GLSL editor: John Kessenich, LunarG

GLSL version 1.1 Authors: John Kessenich, Dave Baldwin, Randi Rost

Table of Contents

1 Introduction

This document specifies only version 3.0 of the OpenGL ES Shading Language. It requires __VERSION__ to substitute 300, and requires **#version** to accept only 300 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 Graphics System Specification, Version 3.0, for details on what language versions are supported.

All OpenGL ES Graphics System Specification references in this specification are to version 3.0

1.1   Changes

This specification is derived from OpenGL GLSL 3.3 revision 7.

1.1.1   Changes from GLSL ES 3.0 revision 2

- Clarified that the paramter for switch statements can be a signed or unsigned integer
- Clarified that all integer vertex shader outputs and fragment shader inputs must be qualified 'flat'
- Invalid layout qualifiers must generate an error
- Layout qualifier IDs are case sensitive
- Precision of packing and unpacking functions
- Precision of return type of textureSize()
- Precision requirements for built-in functions
- Conversion between precisions
- Removed default precision for sampler types introduced in GLSL ES 3.0
- gl_DepthRange members should be highp
- Clarified that the layout qualifier ID values can be signed or unsigned.
- Clarified that use of reserved features is an error
- Clarified description of sampling of projected textures
- Vertex shader outputs with integer type must be qualified as flat
- Clarified that digraphs and trigraphs are disallowed
- The maximum length of an identifier is 1024 characters
- The maximum length of a macro name is 1024 characters
- Added explicit statement that the precision of a variable cannot be changed

- Regions of scope in loop statements, including corrections to grammar
- Range of lowp integers
- Correction to the counting algorithm example for varyings
- The precision statement can be used to set the default precision for sampler types
- The default precision for unsigned integers cannot be set independently from signed integers
- Added default precision for built-in variables
- Clarified that a macro with an empty replacement list does not default to '0' in a pre-processor expression
- Inputs cannot be declared invariant
- Errors may be reported at compile time or link time
- Clarified that layout qualifier parameters may be either signed or unsigned integer constants

1.1.2 Changes from GLSL ES 3.0 revision 1:

- Clarified that mediump and lowp integers wrap on overflow
- Range checking of literal integers
- Redefinition of built-in macros not allowed
- Version directive must be the first line of a shader

1.1.3 Changes from OpenGL GLSL 3.3:

Removed:

- Profiles and deprecation
- Geometry shaders
- Multiple compilation units
- Shared globals (except for uniforms)
- in and out blocks
- vertex array inputs (attribute arrays)
- Layout qualifiers: index, origin_upper_left and pixel_center_integer
- CPP token pasting
- Unsized arrays.
- Implicit type conversion.
- Overloading built-in functions
- noperspective
- Multi-sample textures

- Rectangular textures

- Texture buffers

- 1D textures

- Noise

- Outer scope for built-in functions.

- Redeclaring built-in variables.

Added:

- Line continuation and UTF-8 in GLSL ES 1.00 when used with OpenGL ES 3.0

- Array length operator returns an unsigned integer-constant. The precision is determined using the rules for literal integers

- Clarified that source code lines may be of arbitrary length

- Line continuation

- Extended character set for comments

- Built-in constants: gl_MinProgramTexelOffset, gl_MaxProgramTexelOffset

- Handling and reporting of errors

- GLES macro

- Use of an undefined macro is an error

- Numeric precision of variables and operations

- Default precisions

- Definitions and behavior for precision qualifiers lowp, mediump and highp

- Invariance within a shader

- Relaxation of the order of evaluation of expressions

- Pack and unpack built-in functions

- List of errors

- Normative references

- Extension macro names always defined if the extension is available

- Clarified that for the operators << and >>, if both operands are vectors, they must have the same size

- GLSL ES 1.00 compatibility

- Vertex output, fragment input counting algorithm

1.2   Overview

This document describes *The OpenGL ES Shading Language, version 3.00*

Independent compilation units written in this language are called *shaders*. A *program* is a complete set of shaders that are compiled and linked together. The aim of this document is to thoroughly specify the programming language. The OpenGL ES Graphics System 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 vertex and fragment shader (link 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* or *undefined behavior* there is no such requirement but compilers are encouraged to report possible violations.

A distinction is made between *undefined behavior* and an *undefined value* (or *result*). Undefined behavior includes system instability and/or termination of the application. It is expected that systems will be designed to handle these cases gracefully but specification of this is outside the scope of OpenGL ES.

If a value or result is undefined, the system may behave as if the value or result had been assigned a random value. For example, an undefined gl_Position may cause a triangle to be drawn with a random size and position. The value may not be consistent. For example an undefined boolean value may cause both sub-statements in an if-then-else statement to be executed (see section 4.6.4 Invariance of Undefined Values). The implementation may also detect the generation and/or use of undefined values and behave accordingly (for example causing a trap). Undefined values must not by themselves cause system instability. However undefined values may lead to other more serious conditions such as infinite loops or out of bounds array accesses.

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 values 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 compiler 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 section 9 "Shading Language Grammar" uses all capitals for terminals and lower case for non-terminals.

1.5 Compatibility

The OpenGL ES 3.0 API is designed to work with both GLSL ES v1.00 and GLSL ES 3.00. In general a shader written for OpenGL ES 2.0 should work without modification in OpenGL ES 3.0.

When porting applications from OpenGL ES 2.0 to OpenGL ES 3.0, the following points should be noted:

- Not all language constructs present in v1.00 of the language are available in v3.00. E.g. attribute and varying qualifiers. However, the functionality of GLSL ES 3.00 is a super-set of GLSL ES 1.00.

- Some features of the OpenGL ES 3.0 API require language features that are present in GLSL ES 3.00 but not present in GLSL ES 1.00.

- It is an error to link a vertex shader and a fragment shader if they are written in different versions of the language.

- The OpenGL ES 2.0 API does not support shaders written in GLSL ES 3.0.

5

- Using GLSL ES 1.00 shaders within OpenGL ES 3.0 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:

Uniforms

The number of uniforms specified by gl_MaxVertexUniformVectors and returned by the corresponding API query is the same for GLSL ES versions 1.00 and 3.00 when used as part of OpenGL ES 3.0.

Varyings, vertex outputs and **fragment inputs**

These are specified differently in the two versions of the language and may be different. For GLSL ES 1.00, the maximum number of varyings is specified by gl_MaxVaryingVectors. For GLSL ES 3.00, the maximum number of vertex outputs and fragment inputs is independently specified by gl_MaxVertexOutputVectors and gl_MaxFragmentInputVectors.

In GLSL ES 1.00, only varyings which are statically used in both the vertex and fragment shaders are counted. This applies when GLSL ES 1.00 is used in OpenGL ES 3.0

Multiple Render Targets

Although gl_FragData is declared as an array in GLSL ES 1.00, multiple render targets are not supported in OpenGL ES 2.0 and are therefore not available when using GLSL ES 1.00 in OpenGL ES 3.0.

- 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 3.0 API.

2 Overview of OpenGL ES Shading

The OpenGL ES Shading Language is actually two 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 and fragment processors.

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

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

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

2.2 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*. When a complete set of fragment shaders are compiled and linked, they result in a *fragment shader executable* that runs on the fragment processor.

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

3 Basics

3.1   Character Set

The source character set used for the OpenGL ES shading languages is a subset of UTF-8. It comprises the following characters:

The letters **a-z**, **A-Z,** and the underscore (_)**.**

The numbers **0-9**.

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

The number sign (**#**) for preprocessor use.

Backslash ('\'), used to indicate line continuation when immediately preceding a new-line.

White space: the space character, horizontal tab, vertical tab, form feed, carriage-return, and line-feed.

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.

Inside comments, the character set is extended to allow any byte values to be used but with the exception that a byte with the value zero is always interpreted as the end of the string. The character encoding is assumed to be UTF-8 but no checking is performed for invalid characters.

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.

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 300 es**" is required in any shader that uses version 3.00 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 3.00 of the shading language cannot be linked with shaders declaring version 1.00.

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.

version-declaration:

> *whitespace$_{opt}$* POUND *whitespace$_{opt}$* VERSION *whitespace number whitespace$_{opt}$* 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.

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 a diagnostic message and make the implementation treat the shader as ill-formed.

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

The following predefined macros are available

```
__LINE__
__FILE__
__VERSION__
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 300.

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.

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

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

The maximum length of a macro name is 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 for C++ except for the following:

- Expressions following **#if** and **#elif** are restricted to expressions operating on literal integer constants, plus identifiers consumed by the **defined** operator.

- 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 pre-processor expression.

The operators available are as follows:

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

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

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

There are no number sign based operators (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.

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

Preprocessor expressions will be evaluated at compile time.

#error will cause the implementation to put a diagnostic message into the shader object's information log (see section 6.1.12 "Shader and Program Queries" in the OpenGL ES Graphics System Specification for how to access a shader object's information log). The message will be the tokens following the **#error** directive, up to the first new-line. The implementation must then consider the shader to be ill-formed.

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

```
#pragma STDGL
```

The **STDGL** pragma is used to reserve pragmas for use by this and future revisions of the 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.

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

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

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

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

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

The initial state of the compiler is as if the directive

```
#extension all : disable
```

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

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

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

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

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

3.6   Tokens

The language 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 cannot be used for any other purpose than that defined by this document:

> **const   uniform**
>
> **layout**
>
> **centroid   flat   smooth**
>
> **break   continue   do   for   while   switch   case   default**
>
> **if   else**
>
> **in   out   inout**
>
> **float   int   void   bool   true   false**
>
> **invariant**
>
> **discard   return**
>
> **mat2   mat3   mat4**
>
> **mat2x2   mat2x3   mat2x4**
>
> **mat3x2   mat3x3   mat3x4**
>
> **mat4x2   mat4x3   mat4x4**
>
> **vec2   vec3   vec4   ivec2   ivec3   ivec4   bvec2   bvec3   bvec4**
>
> **uint   uvec2   uvec3   uvec4**
>
> **lowp   mediump   highp   precision**
>
> **sampler2D   sampler3D   samplerCube**
>
> **sampler2DShadow   samplerCubeShadow**
>
> **sampler2DArray**
>
> **sampler2DArrayShadow**
>
> **isampler2D   isampler3D   isamplerCube**
>
> **isampler2DArray**

usampler2D   usampler3D   usamplerCube

usampler2DArray

struct

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

attribute   varying

coherent   volatile   restrict   readonly   writeonly

resource   atomic_uint

noperspective

patch   sample

subroutine

common   partition   active

asm

class   union   enum   typedef   template   this   packed

goto

inline   noinline   volatile   public   static   extern   external   interface

long   short   double   half   fixed   unsigned   superp

input   output

hvec2   hvec3   hvec4   dvec2   dvec3   dvec4   fvec2   fvec3   fvec4

sampler3DRect

filter

image1D   image2D   image3D   imageCube

iimage1D   iimage2D   iimage3D   iimageCube

uimage1D   uimage2D   uimage3D   uimageCube

image1DArray   image2DArray

iimage1DArray   iimage2DArray   uimage1DArray   uimage2DArray

image1DShadow   image2DShadow

image1DArrayShadow   image2DArrayShadow

imageBuffer   iimageBuffer   uimageBuffer

sampler1D   sampler1DShadow   sampler1DArray   sampler1DArrayShadow

isampler1D   isampler1DArray   usampler1D   usampler1DArray

sampler2DRect   sampler2DRectShadow   isampler2DRect   usampler2DRect

samplerBuffer   isamplerBuffer   usamplerBuffer

sampler2DMS   isampler2DMS   usampler2DMS

sampler2DMSArray   isampler2DMSArray   usampler2DMSArray

sizeof   cast

namespace   using

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

3.8   Identifiers

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

identifier
  nondigit
  identifier nondigit
  identifier digit

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

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

Identifiers starting with "gl_" are reserved for use by OpenGL ES, and may not be declared in a shader as either a variable or a function. It is an error to redeclare a variable, including those starting "gl_".

The maximum length of an identifier is 1024 characters. It is an error to declare a variable with a length greater than this.

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

3.9.2 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.

The definition is not used in this version of GLSL ES but may be referenced by extensions.

3.10 Logical Phases of Compilation

The compilation process is based on a subset of the C++ standard (see section 14: Normative References). The compilation units for the vertex and fragment processor are processed separately before being linked together in the final stage of compilation. The logical phases of compilation are:

1. Source strings are concatenated.

2. The source string is converted into a sequence of preprocessing tokens. These tokens include preprocessing numbers, identifiers and preprocessing operations. Comments are each replaced by one space character. Line breaks are retained.

3. The preprocessor is run. Directives are executed and macro expansion is performed.

4. Preprocessing tokens are converted into tokens.

5. White space and line breaks are discarded.

6. The syntax is analyzed according to the GLSL ES grammar.

7. The result is checked according to the semantic rules of the language.

8. The vertex and fragment shaders are linked together. Any vertex outputs and corresponding fragment inputs not used in both the vertex and fragment shaders may be discarded.

9. The binary is generated.

4 Variables and Types

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

There are no default types. All variable and function declarations must have a declared type, and optionally qualifiers. A variable is declared by specifying its type followed by one or more names separated by commas. In many cases, a variable can be initialized as part of its declaration by using the assignment operator (=). The grammar near the end of this document provides a full reference for the syntax of declaring variables.

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

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

4.1   Basic Types

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

Transparent types

Type	Meaning
void	for functions that do not return a value
bool	a conditional type, taking on values of true or false
int	a signed integer
uint	an unsigned integer
float	a single floating-point scalar
vec2	a two-component floating-point vector
vec3	a three-component floating-point vector
vec4	a four-component floating-point vector
bvec2	a two-component Boolean vector
bvec3	a three-component Boolean vector
bvec4	a four-component Boolean vector
ivec2	a two-component signed integer vector
ivec3	a three-component signed integer vector
ivec4	a four-component signed integer vector
uvec2	a two-component unsigned integer vector
uvec3	a three-component unsigned integer vector

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

Floating Point Sampler Types (opaque)

Type	Meaning
sampler2D	a handle for accessing a 2D texture
sampler3D	a handle for accessing a 3D texture
samplerCube	a handle for accessing a cube mapped texture
samplerCubeShadow	a handle for accessing a cube map depth texture with comparison
sampler2DShadow	a handle for accessing a 2D depth texture with comparison
sampler2DArray	a handle for accessing a 2D array texture
sampler2DArrayShadow	a handle for accessing a 2D array depth texture with comparison

Signed Integer Sampler Types (opaque)

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

Unsigned Integer Sampler Types (opaque)

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

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

There are no pointer types.

4.1.1 Void

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

4.1.2 Booleans

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

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

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

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

4.1.3 Integers

Signed and unsigned integer variables are fully supported. In this document, the term *integer* is meant to generally include both signed and unsigned integers. 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. 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. See section 4.5.1 "Range and Precision" for details.

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

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

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

integer-constant:

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

integer-suffix: one of

> **u U**

decimal-constant:

> *nonzero-digit*
> *decimal-constant digit*

octal-constant:

> **0**
> *octal-constant octal-digit*

hexadecimal-constant:

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

digit:

> **0**
> *nonzero-digit*

nonzero-digit: one of

> **1 2 3 4 5 6 7 8 9**

octal-digit **:** one of

> **0 1 2 3 4 5 6 7**

hexadecimal-digit: one of

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

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

It is an error to provide a literal integer whose value would be too large to store in a highp uint variable. Note that this only applies to literals; no error checking is performed on the result of a constant expression.

4 Variables and Types

Examples

```
1          // signed integer, value 1

1u         // unsigned integer, value 1

-1         // unary minus applied to signed integer.
           // result is a signed integer, value -1

-1u        // unary minus applies to unsigned integer
           // result is an unsigned integer, value 0xffffffff

0xffffffff // signed integer, value -1

0xffffffffu // unsigned integer, value 0xffffffff

0xffffffff0 // error: values of signed integer is too large
```

4.1.4 Floats

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

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

As an input value to one of the processing units, a floating-point variable is expected to match the IEEE 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 section 4.5.1 "Range and Precision" for more details on precision and usage of NaNs (Not a Number) and Infs (positive or negative infinities).

Floating-point constants are defined as follows.

> *floating-constant:*
> > *fractional-constant exponent-part$_{opt}$ floating-suffix$_{opt}$*
> > *digit-sequence exponent-part floating-suffix$_{opt}$*
>
> *fractional-constant:*
> > *digit-sequence **.** digit-sequence*
> > *digit-sequence **.***
> > ***.** digit-sequence*
>
> *exponent-part:*
> > ***e** sign$_{opt}$ digit-sequence*
> > ***E** sign$_{opt}$ digit-sequence*
>
> *sign:* one of
> > **+ −**
>
> *digit-sequence:*
> > *digit*
> > *digit-sequence digit*
>
> *floating-suffix:* one of
> > **f F**

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.

4.1.5   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 declaration are:

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

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

4.1.6   Matrices

The OpenGL 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. Example matrix declarations:

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

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

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

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

4.1.7   Samplers

Sampler types (e.g., **sampler2D**) are effectively opaque handles to textures and their filters. They are used with the built-in texture functions (described in section 8.7 "Texture Lookup Functions") to specify which texture to access and how it is to be filtered. They can only be declared as function parameters or **uniform** variables (see section 4.3.5 "Uniform"). Except for array indexing, structure field selection, and parentheses, samplers are not allowed to be operands in expressions. Samplers aggregated into arrays within a shader (using square brackets **[ ]**) can only be indexed with constant integral expressions (see section 4.3.3 "Constant Expressions"). Samplers cannot be treated as l-values; hence cannot be used as **out** or **inout** function parameters, nor can they be assigned into. As uniforms, they are initialized only with the OpenGL ES API; they cannot be declared with an initializer in a shader. As function parameters, only samplers may be passed to samplers of matching type. This enables consistency checking between shader texture accesses and OpenGL ES texture state before a shader is run.

4.1.8   Structures

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

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

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

```
light lightVar2;
```

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

struct-definition:
        qualifier **struct** *name* **{** *member-list* **}** *declarators* *;*

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

member-declaration:
        basic-type declarators;

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

Structures must have at least one member declaration. Member declarators may contain precision qualifiers, but may not contain any other qualifiers. Bit fields are not supported. Member types must be already defined (there are no forward references). Member declarations cannot contain initializers. Member declarators can contain arrays. Such arrays must have a size specified, and the size must be a constant integral expression that's greater than zero (see section 4.3.3 "Constant Expressions"). Each level of structure has its own name space for names given in member declarators; such names need only be unique within that name space.

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

```
struct S { float f; }; // 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 section 5.4.3 "Structure Constructors" .

Any restrictions on the usage of a type or qualifier also apply to a structure that contains that type or qualifier. This applies recursively.

27

4.1.9 Arrays

Variables of the same type can be aggregated into arrays by declaring a name followed by brackets (**[ ]**) enclosing a size. The array size must be a constant integral expression (see section 4.3.3 "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. It is illegal to index an array with a constant integral expression greater than or equal to the declared size. It is also illegal to index an array with a negative constant expression. Arrays declared as formal parameters in a function declaration must 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. Only one-dimensional arrays may be declared. All basic types and structures can be formed into arrays. Some examples are:

```
float frequencies[3];
uniform vec4 lightPosition[4u];

const int numLights = 2;
light lights[numLights];
```

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

```
float[5]
```

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

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

as a constructor of an array

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

as an unnamed parameter

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

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

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

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.

It is an error to declare arrays of arrays:

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

Arrays can have initializers formed from array constructors:

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

An array declaration which leaves the size unspecified is an error.

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

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

The return value is a constant signed integral expression. The precision is determined using the same rules as for literal integers.

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 may be defined and 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.

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
```

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 section 6.1 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);        // 1st 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 field names to be considered the same type. This rule applies recursively for nested or embedded types.

4.3 Storage Qualifiers

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

Qualifier	Meaning
< none: default >	local read/write memory, or an input parameter to a function
const	a compile-time constant, or a function parameter that is read-only
in **centroid in**	linkage into a shader from a previous stage, variable is copied in linkage with centroid based interpolation
out **centroid out**	linkage out of a shader to a subsequent stage, variable is copied out linkage with centroid based interpolation
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL ES, and the application

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

Qualifier	Meaning
smooth	perspective correct interpolation
flat	no interpolation

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

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

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

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

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

Initializers may only be used in declarations of globals with no storage qualifier or with a **const** qualifier. Such initializers must be a constant expression. 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 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 fields may not be qualified with **const**. Structure variables can be declared as **const**, and initialized with a structure constructor.

Initializers for const declarations must be constant expressions, as defined in section 4.3.3 "Constant Expressions."

4.3.3   Constant Expressions

A *constant expression* is one of

- a literal value (e.g., **5** or **true**)

- a global or local variable qualified as **const** (i.e., not including function parameters)

- an expression formed by an operator on operands that are all constant expressions, including getting an element of a constant array, or a field of a constant structure, or components of a constant vector. However, the sequence operator (,) and the assignment operators (=, +=, ...) are not included in the operators that can create a constant expression.

- the length() method on an 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. 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. Sampler 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 section 4.6.1 "The Invariant Qualifier" for more details on how to create invariant expressions.

4.3.4   Input Variables

Shader input variables are declared with the **in** storage qualifier or the **centroid 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. Variables declared as **in** or **centroid in** may not be written to during shader execution. 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 section 7 "Built-in Variables" for a list of the built-in input names.

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

Example declarations in a vertex shader:

```
in vec4 position;
in vec3 normal;
```

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 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.

Fragment shader inputs get per-fragment values, typically interpolated from a previous stage's outputs. They are declared in fragment shaders with the **in** storage qualifier or the **centroid in** storage qualifier. Fragment inputs can only be signed and unsigned integers and integer vectors, **float**, floating-point vectors, matrices, or arrays or structures of these. Fragment shader inputs that are, or contain, signed or unsigned integers or integer vectors must be qualified with the interpolation qualifier **flat**.

Fragment inputs are declared as in the following examples:

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

flat in vec3 myColor;
```

The output of the vertex shader and the input of the fragment shader form an interface. For this interface, vertex shader output variables and fragment shader input variables of the same name must match in type and qualification (other than **out** matching to **in**).

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. They are initialized to 0 at link timeand 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 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 the vertex and fragment shaders share a single global name space. Hence, the types and precisions of uniform variables with the same name must match across shaders that are linked into a single program.

4.3.6 **Output Variables**

Shader output variables are declared with the **out** or **centroid out** storage qualifiers. They form the output interface between the declaring shader and the subsequent stages of the OpenGL 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 at global scope for declaring a single variable name as both input and output to a shader. Output variables must be declared with different names than input variables.

Vertex output variables output per-vertex data and are declared using the **out** storage qualifier or the **centroid out** storage qualifier. They can only be **float**, floating-point vectors, matrices, signed or unsigned integers or integer vectors, or arrays or structures of any these. Vertex shader outputs that are, or contain, signed or unsigned integers or integer vectors must be qualified with the interpolation qualifier **flat**.

Individual vertex outputs are declared as in the following examples:

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

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

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

4.3.7 Interface Blocks

Uniform variable declarations can be grouped into named interface blocks to provide coarser granularity backing than is achievable with individual declarations. They can have an optional instance name, used in the shader to reference their members. A uniform block is backed by the application with a buffer object.

GLSL ES 3.0 does not support interface blocks for shader inputs or outputs.

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

> *interface-block:*
> > *layout-qualifier*$_{opt}$ **uniform** *block-name* **{** *member-list* **}** *instance-name*$_{opt}$ **;**
>
> *layout-qualifier:*
> > **layout (** *layout-qualifier-id-list* **)**
>
> *layout-qualifier-id-list*
> > comma separated list of *layout-qualifier-id*
>
> *member-list:*
> > *member-declaration*
> > *member-declaration member-list*
>
> *member-declaration:*
> > *layout-qualifier*$_{opt}$ *qualifiers*$_{opt}$ *type declarators* **;**
>
> *instance-name:*
> > *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 uniform variable declarations outside blocks, with these exceptions:

- sampler types are not allowed

- structure definitions cannot be nested inside a block

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

Repeating the **uniform** interface qualifier for a member's storage qualifier is optional. For example,

```
uniform Transform
{
    uniform mat4 model_view; // legal, uniform inside a uniform block.
    mat4 projection;         // legal, 'uniform' inherited from block.
    in bool transform_flag;  // illegal, member is not a uniform.
}
```

For uniform blocks, the application uses the block name to identify the block. Block names have no other use within a shader beyond interface matching; it is an error to use a block name at global scope for anything other than as a block name (e.g., use of a block name for a global variable name or function name is currently reserved).

Matched block names within an interface (as defined above) must match in terms of having the same number of declarations with the same sequence of types, precisions and the same sequence of member names, as well as having the same member-wise layout qualification (see next section). Furthermore, if a matching block is declared as an array, then the array sizes must also match.

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,

```
uniform Transform_1
{
    mat4 modelview;
}

uniform Transform_2
{
    mat4 projection;
} transform_2;

mat4 modelview;  // illegal as modelview already defined at this scope
mat4 projection; // legal as projection and transform_2.projection are
                 // distinct.
```

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

```
uniform Transform_1
{
    mat4 modelview; // API will use "modelview"
}

uniform Transform_2
{
    mat4 projection; // API will use "Transform_2.projection"
} transform_2;
```

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

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

For uniform blocks declared as an array, each individual array element corresponds to a separate buffer object backing one instance of the block. As the array size indicates the number of buffer objects needed, uniform block array declarations must specify an array size. All indexes used to index a uniform block array must be constant integral expressions.

When using OpenGL ES API entry points to identify the name of an individual block in an array of blocks, the name string must include an array index (e.g., *Transform[2]*). When using OpenGL 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*).

There is an implementation dependent limit on the number of uniform blocks that can be used per stage. If this limit is exceeded, it will cause a link error.

4.3.8   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 a uniform to establish layouts of other uniform declarations:

   layout-qualifier **uniform ;**

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

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

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

Interface qualifiers are a subset of storage qualifiers:

> *interface-qualifier:*
>> **in**
>> **out**
>> **uniform**

As shown in the previous section, *layout-qualifier* expands to:

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

The tokens in any *layout-qualifier-id-list* are identifiers, not keywords. 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.

4.3.8.1 Input Layout Qualifiers

Vertex shaders allow input layout qualifiers on input variable declarations. The layout qualifier identifier for vertex shader inputs is:

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

Only one argument is accepted. For example,

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

will establish that the vertex shader input *normal* is copied in from vector location number 3.

If an 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 2.11.5 "Vertex Attributes" of the OpenGL ES 3.0 Graphics System Specification for more details.

Fragment shaders cannot have input layout qualifiers.

4.3.8.2 Output Layout Qualifiers

Vertex shaders cannot have output layout qualifiers.

In the fragment shader, a binding between an output variable and a numbered draw buffer is established by the location layout qualifier in the output declaration. The location of each output corresponds to the draw buffer the data is written to. Locations are integral values in the range [0, MAX_DRAW_BUFFERS – 1].

Fragment shaders allow output layout qualifiers only on the interface qualifier **out**. The layout qualifier identifier for fragment shader outputs is:

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

The qualifier may appear at most once within a declaration. For example,

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

will establish that the fragment shader output *color* is copied out to draw buffer 3.

If the named fragment shader 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 copied out to draw buffers 2, 3, and 4.

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. For example,

```
out vec4 my_FragColor; // must be the only output declaration
```

will establish that the fragment shader output *my_FragColor* is copied out to draw buffer 0. Likewise,

```
out vec4 my_FragData[4]; // must be the only output declaration
```

will establish that the fragment shader outputs *my_FragData[0]* to *my_FragData[3]* is copied out to draw buffers 0 through 3 respectively.

If there is more than one output, the location must be specified for all outputs. It is an error if any of the following occur:

- The location of any output or element of an array output, is greater or equal to the value of MAX_DRAW_BUFFERS.

- More than one output or element of an array output is bound to the same location.

 See section 3.9.2 "Shader Execution" of the OpenGL ES 3.0 Graphics System Specification for more details.

4.3.8.3 Uniform Block Layout Qualifiers

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

> *layout-qualifier-id*
> > **shared**
> > **packed**
> > **std140**
> > **row_major**
> > **column_major**

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

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

Default layouts are established at global scope for uniform blocks as

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

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

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

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

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

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

The *shared* qualifier overrides only the *std140* and *packed* qualifiers; other qualifiers are inherited. The compiler/linker will ensure that multiple programs and programmable stages containing this definition will share the same memory layout for this block, as long as they also matched in their *row_major* and/or *column_major* qualifications. This allows use of the same buffer to back the same block definition across different programs.

The *packed* qualifier overrides only *std140* and *shared*; other qualifiers are inherited. When *packed* is used, no shareable layout is guaranteed. The compiler and linker can optimize memory use based on what variables actively get used and on other criteria. Offsets must be queried, as there is no other way of guaranteeing where (and which) variables reside within the block. Attempts to share a packed uniform block across programs or stages will generally fail. However, implementations may aid application management of packed blocks by using canonical layouts for packed blocks.

The *std140* qualifier overrides only the *packed* and *shared* qualifiers; other qualifiers are inherited. The layout is explicitly determined by this, as described in section 2.11.5 "Uniform Variables" under "Standard Uniform Block Layout" of the OpenGL ES Graphics System Specification. Hence, as in *shared* above, the resulting layout is shareable across programs.

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

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

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

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.

Layout qualifiers are identifiers, not keywords and they have their own name space.

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
};
```

4.3.9   Interpolation

The presence of and type of interpolation is controlled by the storage qualifiers **centroid in** and **centroid out**, and by the optional interpolation qualifiers **smooth** and **flat**. When no interpolation qualifier is present, smooth interpolation is used. It is a compile-time error to use more than one interpolation qualifier.

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

A variable qualified as **smooth** will be interpolated in a perspective-correct manner over the primitive being rendered. Interpolation in a perspective correct manner is specified in equations 3.4 in the OpenGL ES 3.0 Graphics System Specification, section 3.5 "Line Segments".

This paragraph only applies if interpolation is being done: If single-sampling, the value is interpolated to the pixel's center, and the **centroid** qualifier, if present, is ignored. If multi-sampling and the variable is not qualified with **centroid**, then the value must be interpolated to the pixel's center, or anywhere within the pixel, or to one of the pixel's samples. If multi-sampling and the variable is qualified with **centroid**, then the value must be interpolated to a point that lies in both the pixel and in the primitive being rendered, or to one of the pixel's samples that falls within the primitive. Due to the less regular location of centroids, their derivatives may be less accurate than non-centroid interpolated variables.

The type and presence of the interpolation qualifiers and storage qualifiers and **invariant** qualifiers of variables with the same name declared in all linked shaders must match, otherwise the link command will fail.

4.3.10 Linking of Vertex Outputs and Fragment Inputs

The type of vertex outputs and fragment input with the same name must match, otherwise the link command will fail. The precision does not need to match. 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 vertex outputs with fragment inputs:

		Fragment Shader Inputs		
		No reference	Declares; no static use	Declares and static use
Vertex Shader Outputs	No reference	Allowed	Allowed	error
	Declares; no static use	Allowed	Allowed	Allowed (values are undefined)
	Declares and static use	Allowed	Allowed	Allowed (values are potentially undefined)

The term *static use* means that after preprocessing the shader includes at least one statement that accesses the input or output, even if that statement is never actually executed.

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. In this case, the precision of the interpolated value is defined by a range and resolution as below:

The precision of values exported to a transform feedback buffer is the precision of the outputs of the vertex shader.

4.4 Parameter Qualifiers

Parameters can have these qualifiers.

Qualifier	Meaning
< none: default >	same is **in**
in	for function parameters passed into a function
out	for function parameters passed back out of a function, but not initialized for use when passed in
inout	for function parameters passed both into and out of a function

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

4.5 Precision and Precision Qualifiers

4.5.1 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).

The following rules apply to highp operations: Infinities and zeros are generated as dictated by IEEE, but subject to the precisions allowed in the following table and subject to allowing positive and negative zeros to be interchanged. However, dividing a non-zero by 0 results in the appropriately signed IEEE Inf: If both positive and negative zeros are implemented, the correctly signed Inf will be generated, otherwise positive Inf is generated. Any 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. NaNs are not required to be generated. Support for signaling NaNs is not required and exceptions are never raised. Operations and built-in functions that operate on a NaN are not required to return a NaN as the result. However if NaNs are generated, isnan() should 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.
pow(x, y)	Inherited from **exp2** $(x *$ **log2** $(y))$.
exp (x), **exp2** (x)	$(3 + 2 * |x|)$ ULP.
log (), **log2** ()	3 ULP outside the range [0.5, 2.0]. Absolute error $< 2^{-21}$ inside the range [0.5, 2.0].
sqrt ()	Inherited from $1.0 /$ **inversesqrt**().
inversesqrt ()	2 ULP.
explicit conversions between types	Correctly rounded.

The rounding mode is not defined but must not affect the result by more than 1 ULP.

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 integer 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	Integer Range	
				Signed	**Unsigned**
highp	As IEEE-754 $(-2^{126}, 2^{127})$	As IEEE-754 $0.0, (2^{-126}, 2^{127})$	As 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^{-9} signed/unsigned	$[-2^{8}, 2^{8}-1]$	$[0, 2^{9}-1]$

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 3.0 specification for details on how to do this.

4.5.2   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.

4.5.3   Precision Qualifiers

Any floating point or any integer 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 floating point constructors nor integer constructors when none of the constructor arguments have precision qualifiers.

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.

For constant expressions and sub-expressions, where the precision is not defined, the evaluation is performed at or above the highest supported precision of the target (either mediump or highp). The evaluation of constant expressions must be invariant and will usually be performed at compile time.

In other 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 the next chapter, function input and output is done through copies, and therefore qualifiers do not have to match.

The same uniform declared in different shaders that are linked together must have the same precision qualification.

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

4.5.4   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** or **float** or any of the sampler 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 innermost statement it was declared in. Precision statements in nested scopes override precision statements in outer scopes. Multiple precision statements for the same basic type can appear inside the same scope, with later statements overriding earlier statements within that scope.

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

```
precision highp float;
precision highp int;

precision lowp sampler2D;
precision lowp samplerCube;
```

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

```
precision mediump int;

precision lowp sampler2D;
precision lowp samplerCube;
```

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 for the following sampler types in either the vertex or fragment language:

```
sampler3D;
samplerCube;
samplerCubeShadow;
sampler2DShadow;
sampler2DArray;
sampler2DArrayShadow;
isampler2D;
isampler3D;
isamplerCube;
isampler2DArray;
usampler2D;
usampler3D;
usamplerCube;
usampler2DArray;
```

4.6   Variance and the Invariant Qualifier

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

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

4.6.1   The Invariant Qualifier

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

```
invariant gl_Position;   // make built-in gl_Position be invariant

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

invariant Color_2;       // error: Color_2 has not been declared
```

or as part of a declaration when a variable is declared

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

The invariant qualifier must appear before any interpolation qualifiers or storage qualifiers when combined with a declaration. Only variables output from a shader can be candidates for invariance. This includes user-defined output variables and the built-in output variables.   As only outputs can be declared as invariant, an invariant 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 the global scope, and before any use of the variables being declared as invariant.

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

- The output variable is declared as invariant in both programs.

- The same values must be input to all shader input variables consumed by expressions and control flow contributing to the value assigned to the output variable.

- The texture formats, texel values, and texture filtering are set the same way for any texture function calls contributing to the value of the output variable.

- All input values are all operated on in the same way. All operations in the consuming expressions and any intermediate expressions must be the same, with the same order of operands and same associativity, to give the same order of evaluation. Intermediate variables and functions must be declared as the same type with the same explicit or implicit precision qualifiers. Any control flow affecting the output value must be the same, and any expressions consumed to determine this control flow must also follow these invariance rules.

- All the data flow and control flow leading to setting the invariant output variable reside in a single compilation unit.

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

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

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

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

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

4.6.2   Invariance 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.

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

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

4.6.4 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.

4.7 Order of Qualification

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

> *invariant-qualifier   interpolation-qualifier storage-qualifier   precision-qualifier*
>
> *storage-qualifier   parameter-qualifier precision-qualifier*

5 Operators and Expressions

5.1   Operators

The OpenGL 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, swizzler post fix increment and decrement	[ ] ( ) . ++ --	Left to Right
3	prefix increment and decrement unary	++ -- + - ~ !	Right to Left
4	multiplicative	* / %	Left to Right
5	additive	+ -	Left to Right
6	bit-wise shift	<< >>	Left to Right
7	relational	< > <= >=	Left to Right
8	equality	== !=	Left to Right
9	bit-wise and	&	Left to Right
10	bit-wise exclusive or	^	Left to Right
11	bit-wise inclusive or	|	Left to Right
12	logical and	&&	Left to Right
13	logical exclusive or	^^	Left to Right
14	logical inclusive or	||	Left to Right
15	selection	? :	Right to Left
16	Assignment arithmetic assignments	= += -= *= /= %= <<= >>= &= ^= |=	Right to Left
17 (lowest)	sequence	,	Left to Right

There is no address-of operator nor a dereference operator. There is no typecast operator; constructors are used instead.

5.2 Array Operations

These are now described in section 5.7 "Structure and Array Operations".

5.3 Function Calls

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

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

5.4 Constructors

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

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

5.4.1 Conversion and Scalar Constructors

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

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

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

When a constructor is used to convert an **int**, **uint**, or a **float** to a **bool**, 0 and 0.0 are converted to **false**, and non-zero values are converted to **true**. When a constructor is used to convert a **bool** to an **int**, **uint**, or **float**, **false** is converted to 0 or 0.0, and **true** is converted to 1 or 1.0.

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

Identity constructors, like **float**(**float**) are also legal, but of little use.

Scalar constructors with non-scalar parameters can be used to take the first element from a non-scalar. For example, the constructor **float**(**vec3**) will select the first component of the **vec3** parameter.

5.4.2   Vector and Matrix Constructors

Constructors can be used to create vectors or matrices from a set of scalars, vectors, or matrices. This includes the ability to shorten vectors.

If there is a single scalar parameter to a vector constructor, it is used to initialize all components of the constructed vector to that scalar's value. If there is a single scalar parameter to a matrix constructor, it is used to initialize all the components on the matrix's diagonal, with the remaining components initialized to 0.0.

If a vector is constructed from multiple scalars, one or more vectors, or one or more matrices, or a mixture of these, the 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 $i \neq j$.

58

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

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

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

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

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

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

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

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

5.4.3 Structure Constructors

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

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

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

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

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. 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.

5.5   Vector Components

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

The component names supported are:

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

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

Note that the third component of the texture coordinate set, *r* in OpenGL 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 vector type is an error so, for example:

```
vec2 pos;
pos.x  // is legal
pos.z  // is illegal
```

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

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

No more than 4 components can be selected.

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

vec2 v2;
v2.xyxy;        // is legal. It evaluates to a vec4.
```

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

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

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

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

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

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

Array subscripting syntax can also be applied to vectors to provide numeric indexing. So in

```
vec4   pos;
```

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

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

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

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

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

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

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

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

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

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

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

5.8   Assignments

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

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

The *lvalue-expression* evaluates to an l-value. The assignment operator stores the value of *rvalue-expression* into the l-value and returns an r-value with the type and precision of *lvalue-expression*. The *lvalue-expression* and *rvalue-expression* must have the same type. Any type-conversions must be specified explicitly via constructors. L-values must be writable. Variables that are built-in types, entire structures or arrays, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[ ]**) are all l-values. Other binary or unary expressions, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (**?:**) is also not allowed as an l-value.

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

The other assignment operators are

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

where the general expression

```
lvalue op= expression
```

is equivalent to

```
lvalue = lvalue op expression
```

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

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

5.9   **Expressions**

Expressions in the shading language are built from the following:

- Constants of type **bool, int, uint, float,** all vector types, and all matrix types.

- Constructors of all types.

- Variable names of all types.

- An array name with the length method applied.

- Subscripted array names.

- Function calls that return values.

- Component field selectors and array subscript results.

- Parenthesized expression. Any expression can be parenthesized. Parentheses can be used to group operations. Operations within parentheses are done before operations across parentheses.

- The arithmetic binary operators add (**+**), subtract (**-**), multiply (*****), and divide (**/**) operate on integer and floating-point scalars, vectors, and matrices. . If the operands are integer types, they must both be signed or both be unsigned. All arithmetic binary operators result in the same fundamental type (signed integer, unsigned integer, or floating-point) as the operands they operate on, after operand type conversion. After conversion, the following cases are valid

  - The two operands are scalars. In this case the operation is applied, resulting in a scalar.

  - One operand is a scalar, and the other is a vector or matrix. In this case, the scalar operation is applied independently to each component of the vector or matrix, resulting in the same size vector or matrix.

  - The two operands are vectors of the same size. In this case, the operation is done component-wise resulting in the same size vector.

  - The operator is add (**+**), subtract (**-**), or divide (**/**), and the operands are matrices with the same number of rows and the same number of columns. In this case, the operation is done component-wise resulting in the same size matrix.

  - The operator is multiply (*****), where both operands are matrices or one operand is a vector and the other a matrix. A right vector operand is treated as a column vector and a left vector operand as a row vector. In all these cases, it is required that the number of columns of the left operand is equal to the number of rows of the right operand. Then, the multiply (*****) operation does a linear algebraic multiply, yielding an object that has the same number of rows as the left operand and the same number of columns as the right operand. Section 5.10 "Vector and Matrix Operations" explains in more detail how vectors and matrices are operated on.

All other cases are illegal.

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

- The operator modulus (**%**) operates on signed or unsigned integers or integer vectors. The operand types must both be signed or both be unsigned. The operands cannot be vectors of differing size. If one operand is a scalar and the other vector, then the scalar is applied component-wise to the vector, resulting in the same type as the vector. If both are vectors of the same size, the result is computed component-wise. The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero second operands remain defined. If both operands are non-negative, then the remainder is non-negative. Results are undefined if one or both operands are negative. The operator modulus (**%**) is not defined for any other data types (non-integer types).

- The arithmetic unary operators negate (-), post- and pre-increment and decrement (**--** and **++**) operate on integer or floating-point values (including vectors and matrices). All unary operators work component-wise on their operands. These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract 1 or 1.0 to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 or 1.0 to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.

- The relational operators greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate only on scalar integer and scalar floating-point expressions. The result is scalar Boolean. 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. They result in a scalar Boolean. The types of the operands must match. For vectors, matrices, structures, and arrays, all components, fields, or elements of one operand must equal the corresponding components, fields, or elements in the other operand for the operands to be considered equal. To get a vector of component-wise equality results for vectors, use the built-in functions **equal** and **notEqual**.

- The logical binary operators and (**&&**), or (**||**), and exclusive or (^^) operate only on two Boolean expressions and result in a Boolean expression. And (**&&**) will only evaluate the right hand operand if the left hand operand evaluated to **true**. Or (**||**) will only evaluate the right hand operand if the left hand operand evaluated to **false**. Exclusive or (^^) will always evaluate both operands.

- The logical unary operator not (**!**). It operates only on a Boolean expression and results in a Boolean expression. To operate on a vector, use the built-in function **not**.

- The sequence (**,**) operator that operates on expressions by returning the type and value of the right-most expression in a comma separated list of expressions. All expressions are evaluated, in order, from left to right.

- The ternary selection operator (**?:**). It operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression, which must result in a scalar Boolean. If the result is true, it selects to evaluate the second expression, otherwise it selects to evaluate the third expression. Only one of the second and third expressions is evaluated. The second and third expressions can be any type, as long their types match. 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. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The fundamental types of the operands (signed or unsigned) must match, and will be the resulting fundamental type. For and (**&**), the result is the bitwise-and function of the operands. For exclusive-or (^), the result is the bitwise exclusive-or function of the operands. For inclusive-or (|), the result is the bitwise inclusive-or function of the operands.

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

5.10   Vector and Matrix Operations

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

```
vec3 v, u;
float f;

v = u + f;
```

will be equivalent to

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

And

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

66

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

The C++ standard requires that 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 in the following ways:

- Addition and multiplication are assumed to be associative.

- Multiplication may be replaced by repeated addition

- Floating point division may be replaced by reciprocal and multiplication:

- Within the constraints of invariance (where applicable), the precision used may vary.

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 section 9 "Shading Language Grammar" should be used as the definitive specification.

Declarations and expressions have already been discussed.

6.1   Function Definitions

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

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

and a function is defined like

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

where *returnType* must be present and 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 explicitly sized. An array is passed or returned by using just its name, without brackets, and the size of the array must match the size specified in the function's declaration.

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

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

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

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

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

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

Only a precision qualifier is allowed on the return type of a function. Formal parameters can have parameter and precision 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 ivec4 y); // allowed, different argument type
int  f(in vec4 x, out ivec4 y); // error, only return type differs
vec4 f(in vec4 x, in  ivec4 y); // error, only qualifier differs
int  f(const in vec4 x, out ivec4 y); // error, only qualifier differs
```

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

```
f(vec4, vec4)   // exact match of vec4 f(in vec4 x, out vec4 y)
f(vec4, ivec4)  // exact match of vec4 f(in vec4 x, out ivec4 y)
f(ivec4, vec4)  // error, no exact match.
f(ivec4, ivec4) // 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. Both the vertex and fragment 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 section 6.4 "Jumps" for more details.

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

6.1.1   Function Calling Conventions

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

- The keyword **in** is used as a qualifier to denote a parameter is to be copied in, but not copied out.

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

- The keyword **inout** is used as a qualifier to denote the parameter is to be both copied in and copied out.

- A function parameter declared with no such qualifier means the same thing as specifying **in**.

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

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

In a function, writing to an input-only parameter is allowed. Only the function's copy is modified. This can be prevented by declaring a parameter with the **const** qualifier.

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

No qualifier is allowed on the return type of a function.

> *function-prototype:*
> > *precision-qualifier type function-name(const-qualifier parameter-qualifier precision-qualifier type name array-specifier, ...)*
>
> *type:*
> > any basic type, array type, structure name, *or structure definition*
>
> *const-qualifier:*
> > empty
> > **const**
>
> *parameter-qualifier:*
> > empty
> > **in**
> > **out**
> > **inout**
>
> *name:*
> > empty

          identifier

array-specifier:
          empty
          [*constant-integral-expression* **]**

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

Static and dynamic recursion is not allowed Static recursion is present if the static function call graph of the 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 list of zero or more *switch-statement* and other statements defined by the language, where *switch-statement* adds some forms of labels. That is

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

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

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

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

Conditionals can be nested.

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

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

6.3 Iteration

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

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

while (condition-expression)
    sub-statement

do
    statement
while (condition-expression)
```

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

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

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

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

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

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

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

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

Loops can be nested.

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

6.4 Jumps

These are the jumps:

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

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

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

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

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

7 Built-in Variables

7.1   Vertex Shader Special Variables

Some OpenGL ES operations occur in fixed functionality between the vertex processor and the fragment processor. Shaders communicate with the fixed functionality of OpenGL ES through the use of built-in variables.

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

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

out highp vec4  gl_Position;
out highp float gl_PointSize;
```

Unless otherwise noted elsewhere, these variables are only available in the vertex language as declared above.

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 2.11.9 "Shader Execution" in the OpenGL ES Graphics System 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 2.11.9 "Shader Execution" in the OpenGL ES 3.0 Graphics System Specification). If the current primitive does not come from an instanced draw call, the value of *gl_InstanceID* is zero.

7.2 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;
```

Except as noted below, they behave as other input and output variables.

The output of the fragment shader executable is processed by the fixed function operations at the back end of the OpenGL 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, 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 in** does not further restrict this value to be inside the current primitive. This value is the result of the fixed functionality that interpolates primitives after vertex processing to generate fragments. The *z* component is the depth value that would be used for the fragment's depth if no shader contained any writes to *gl_FragDepth*. This is useful for invariance if a shader conditionally computes *gl_FragDepth* but otherwise wants the fixed functionality fragment depth.

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

The 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.

7.3 Built-In Constants

The following built-in constants are provided to all shaders. The actual values used are implementation dependent, but must be at least the value shown.

```
//
```

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

const mediump int  gl_MaxVertexAttribs = 16;
const mediump int  gl_MaxVertexUniformVectors = 256;
const mediump int  gl_MaxVertexOutputVectors = 16;
const mediump int  gl_MaxFragmentInputVectors = 15;
const mediump int  gl_MaxVertexTextureImageUnits = 16;
const mediump int  gl_MaxCombinedTextureImageUnits = 32;
const mediump int  gl_MaxTextureImageUnits = 16;
const mediump int  gl_MaxFragmentUniformVectors = 224;
const mediump int  gl_MaxDrawBuffers = 4;
const mediump int  gl_MinProgramTexelOffset = -8;
const mediump int  gl_MaxProgramTexelOffset = 7;
```

7.4   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 2.13.1 "Controlling the Viewport" in the
// OpenGL ES Graphics System Specification.
//
struct gl_DepthRangeParameters {
    highp float near;       // n
    highp float far;        // f
    highp float diff;       // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;
```

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**, *genType* is used as the argument. Where the input arguments (and corresponding output) can be **int**, **ivec2**, **ivec3**, or **ivec4**, *genIType* is used as the argument. Where the input arguments (and corresponding output) can be **uint**, **uvec2**, **uvec3**, or **uvec4**, *genUType* is used as the argument. Where the input arguments (or corresponding output) can be **bool**, **bvec2**, **bvec3**, or **bvec4**, *genBType* is used as the argument. For any specific use of a function, the actual types substituted for *genType*, *genIType*, *genUType, or genBType* have to have the same number of components for all arguments and for the return type. Similarly for *mat,* which can be any matrix basic type.

The precision of built-in functions is dependent on the function and arguments. There are three categories:

- Some functions have predefined precisions. The precision is specified
  e.g.

  highp vec2 **textureSize** (gsampler2D *sampler*, int *lod*)

- For the texture sampling 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
```

- For other built-in functions, a call will return a precision qualification matching the highest precision qualification of the call's input arguments. See Section 4.5.2 "Precision Qualifiers" for more detail.

The built-in functions are assumed to be implemented according to the equations specified in the following sections. The precision at which the calculations are performed follows the general rules for precision of operations as specified in section 4.5.3 "Precision Qualifiers".

Example:

$$normalize\left|\overset{..}{y}\right|=\frac{1}{\sqrt{2\quad 2\quad 2}}\left|\overset{..}{y}\right|$$

If the input vector is lowp, the entire calculation is performed at lowp. For some inputs, this will cause the calculation to overflow, even when the correct result is within the range of lowp.

8.1   Angle and Trigonometry Functions

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

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

Syntax	Description
genType **radians** (genType *degrees*)	Converts *degrees* to radians, i.e., $\dfrac{\pi}{180} \cdot degrees$
genType **degrees** (genType *radians*)	Converts *radians* to degrees, i.e., $\dfrac{180}{\pi} \cdot radians$
genType **sin** (genType *angle*)	The standard trigonometric sine function.
genType **cos** (genType *angle*)	The standard trigonometric cosine function.
genType **tan** (genType *angle*)	The standard trigonometric tangent.
genType **asin** (genType *x*)	Arc sine. Returns an angle whose sine is *x*. The range of values returned by this function is $\left[-\dfrac{\pi}{2}, \dfrac{\pi}{2}\right]$ Results are undefined if $\lvert x \rvert > 1$.
genType **acos** (genType *x*)	Arc cosine. Returns an angle whose cosine is *x*. The range of values returned by this function is [0, π]. Results are undefined if $\lvert x \rvert > 1$.
genType **atan** (genType *y*, genType *x*)	Arc tangent. Returns an angle whose tangent is *y/x*. The signs of *x* and *y* are used to determine what quadrant the angle is in. The range of values returned by this function is $\left[-\pi, \pi\right]$. Results are undefined if *x* and *y* are both 0.
genType **atan** (genType *y_over_x*)	Arc tangent. Returns an angle whose tangent is *y_over_x*. The range of values returned by this function is $\left[-\dfrac{\pi}{2}, \dfrac{\pi}{2}\right]$.

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

8.2   Exponential Functions

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

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

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

8.3   Common Functions

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

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

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

Syntax	Description
genType **mix** (genType *x*, genType *y*, genType *a*) genType **mix** (genType *x*, genType *y*, float *a*)	Returns the linear blend of *x* and *y,* i.e., $x \cdot (1-a) + y \cdot a$
genType **mix** (genType *x*, genType y, 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 genType **mix**(genType *x*, genType *y*, genType(*a*)) where *a* is a Boolean vector.
genType **step** (genType *edge*, genType *x*) genType **step** (float *edge*, genType *x*)	Returns 0.0 if *x* < *edge*, otherwise it returns 1.0.

genType **smoothstep** (genType *edge0*,
 genType *edge1*,
 genType *x*)
 genType **smoothstep** (float *edge0*,
 float *edge1*,
 genType *x*)
	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0 and 1 when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to:
	
	    genType t;
	    t = clamp ((x – edge0) / (edge1 – edge0), 0, 1);
	    return t * t * (3 – 2 * t);
	
	 Results are undefined if edge0 >= edge1.

genBType **isnan** (genType *x*)	Returns **true** if *x* holds a NaN. Returns **false** otherwise.
genBType **isinf** (genType *x*)	Returns **true** if *x* holds a positive infinity or negative infinity. Returns **false** otherwise.

Syntax	Description
genIType **floatBitsToInt** (genType value) genUType **floatBitsToUint** (genType value)	Returns a signed or unsigned highp integer value representing the encoding of a floating-point value. For highp floating point, the value's bit level representation is preserved. For mediump and lowp, the value is first converted to highp floating point and the encoding of that value is returned.
genType **intBitsToFloat** (genIType value) genType **uintBitsToFloat** (genUType value)	Returns a highp 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. For lowp and mediump, the value is first converted to the corresponding signed or unsigned highp integer and then reinterpreted as a highp floating point value as before.

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 **packSnorm2x16** (vec2 v)	First, converts each component of the normalized floating-point value v into 16-bit integer values. Then, the results are packed into the returned 32-bit unsigned integer. The conversion for component c of v to fixed point is done as follows: **packSnorm2x16**: round(clamp(c, -1, +1) * 32767.0) 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 **unpackSnorm2x16** (highp uint p)	First, unpacks a single 32-bit unsigned integer p into a pair of 16-bit unsigned integers. Then, each component is converted to a normalized floating-point value to generate the returned two-component vector. The conversion for unpacked fixed-point value f to floating point is done as follows: **unpackSnorm2x16:** clamp(f / 32767.0, -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 **packUnorm2x16** (vec2 v)	First, converts each component of the normalized floating-point value v into 16-bit integer values. Then, the results are packed into the returned 32-bit unsigned integer. The conversion for component c of v to fixed point is done as follows: **packUnorm2x16**: round(clamp(c, 0, +1) * 65535.0) 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.

Syntax	Description
highp vec2 **unpackUnorm2x16** (highp uint *p*)	First, unpacks a single 32-bit unsigned integer *p* into a pair of 16-bit unsigned integers. Then, each component is converted to a normalized floating-point value to generate the returned two-component vector. The conversion for unpacked fixed-point value *f* to floating point is done as follows: **unpackUnorm2x16:** $f/65535.0$ 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** (mediump 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.
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** (genType *x*)	Returns the length of vector *x*, i.e., $\sqrt{x[0]^2 + x[1]^2 + ...}$

89

Syntax	Description
float **distance** (genType *p0*, genType *p1*)	Returns the distance between *p0* and *p1*, i.e., **length** $(p0 - p1)$
float **dot** (genType *x*, genType *y*)	Returns the dot product of *x* and *y*, i.e., $x[0] \cdot y[0] + x[1] \cdot y[1] + ...$
vec3 **cross** (vec3 *x*, vec3 *y*)	Returns the cross product of x and y, i.e., $$\begin{bmatrix} x[1] \cdot y[2] - y[1] \cdot x[2] \\ x[2] \cdot y[0] - y[2] \cdot x[0] \\ x[0] \cdot y[1] - y[0] \cdot x[1] \end{bmatrix}$$
genType **normalize** (genType *x*)	Returns a vector in the same direction as *x* but with a length of 1 i.e. $\dfrac{x}{length(x)}$
genType **faceforward**(genType *N*, genType *I*, genType *Nref*)	If **dot**(*Nref*, *I*) < 0 return *N,* otherwise return –*N*.
genType **reflect** (genType *I*, genType *N*)	For the incident vector *I* and surface orientation *N*, returns the reflection direction: $I - 2 * \textbf{dot}(N, I) * N$ *N* must already be normalized in order to achieve the desired result.

genType **refract**(genType *I*, genType *N*, float *eta*)	For the incident vector *I* and surface normal *N*, and the ratio of indices of refraction *eta,* return the refraction vector. The result is computed by
	
	k = 1.0 - *eta* * *eta* * (1.0 - **dot**(*N*, *I*) * **dot**(*N*, *I*))
	if (k < 0.0)
	    return genType(0.0)
	else
	    return *eta* * *I* - (*eta* * **dot**(*N*, *I*) + **sqrt**(k)) * *N*
	
	The input parameters for the incident vector *I* and the surface normal *N* must already be normalized to get the desired results.

90

8.6   Matrix Functions

Syntax	Description
mat **matrixCompMult** (mat *x*, mat *y*)	Multiply matrix *x* by matrix *y* component-wise, i.e., result[i][j] is the scalar product of *x*[i][j] and *y*[i][j]. Note: to get linear algebraic matrix multiplication, use the multiply operator (*****).
mat2 **outerProduct**(vec2 *c*, vec2 *r*) mat3 **outerProduct**(vec3 *c*, vec3 *r*) mat4 **outerProduct**(vec4 *c*, vec4 *r*) mat2x3 **outerProduct**(vec3 *c*, vec2 *r*) mat3x2 **outerProduct**(vec2 *c*, vec3 *r*) mat2x4 **outerProduct**(vec4 *c*, vec2 *r*) mat4x2 **outerProduct**(vec2 *c*, vec4 *r*) mat3x4 **outerProduct**(vec4 *c*, vec3 *r*) mat4x3 **outerProduct**(vec3 *c*, vec4 *r*)	Treats the first parameter *c* as a column vector (matrix with one column) and the second parameter *r* as a row vector (matrix with one row) and does a linear algebraic matrix multiply *c * r*, yielding a matrix whose number of rows is the number of components in *c* and whose number of columns is the number of components in *r*.
mat2 **transpose**(mat2 *m*) mat3 **transpose**(mat3 *m*) mat4 **transpose**(mat4 *m*) mat2x3 **transpose**(mat3x2 *m*) mat3x2 **transpose**(mat2x3 *m*) mat2x4 **transpose**(mat4x2 *m*) mat4x2 **transpose**(mat2x4 *m*) mat3x4 **transpose**(mat4x3 *m*) mat4x3 **transpose**(mat3x4 *m*)	Returns a matrix that is the transpose of *m*. The input matrix *m* is not modified.
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 produce scalar Boolean results. For vector results, use the following built-in functions. Below, "bvec" is a placeholder for one of **bvec2**, **bvec3**, or **bvec4**, "ivec" is a placeholder for one of **ivec2**, **ivec3**, or **ivec4**, "uvec" is a placeholder for **uvec2**, **uvec3**, or **uvec4**, and "vec" is a placeholder for **vec2**, **vec3**, or **vec4**. In all cases, the sizes of the input and return vectors for any particular call must match.

Syntax	Description
bvec **lessThan**(vec x, vec y) bvec **lessThan**(ivec x, ivec y) bvec **lessThan**(uvec x, uvec y)	Returns the component-wise compare of $x < y$.
bvec **lessThanEqual**(vec x, vec y) bvec **lessThanEqual**(ivec x, ivec y) bvec **lessThanEqual**(uvec x, uvec y)	Returns the component-wise compare of $x <= y$.
bvec **greaterThan**(vec x, vec y) bvec **greaterThan**(ivec x, ivec y) bvec **greaterThan**(uvec x, uvec y)	Returns the component-wise compare of $x > y$.
bvec **greaterThanEqual**(vec x, vec y) bvec **greaterThanEqual**(ivec x, ivec y) bvec **greaterThanEqual**(uvec x, uvec y)	Returns the component-wise compare of $x >= y$.
bvec **equal**(vec x, vec y) bvec **equal**(ivec x, ivec y) bvec **equal**(uvec x, uvec y) bvec **equal**(bvec x, bvec y) bvec **notEqual**(vec x, vec y) bvec **notEqual**(ivec x, ivec y) bvec **notEqual**(uvec x, uvec y) bvec **notEqual**(bvec x, bvec y)	Returns the component-wise compare of $x == y$. Returns the component-wise compare of $x != y$.
bool **any**(bvec x)	Returns true if any component of x is **true**.
bool **all**(bvec x)	Returns true only if all components of x are **true**.
bvec **not**(bvec x)	Returns the component-wise logical complement of x.

8.8   Texture Lookup Functions

Texture lookup functions are available to vertex and fragment shaders. However, level of detail is not implicitly computed for vertex shaders. The functions in the table below provide access to textures through samplers, as set up through the OpenGL ES API. Texture properties such as size, pixel format, number of dimensions, filtering method, number of mip-map levels, depth comparison, and so on are also defined by OpenGL 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 floating point, unsigned normalized integer, unsigned integer or signed integer data. This is determined by the type of the internal format of the texture. Texture lookups on unsigned normalized integer and floating point data return floating point values in the range [0, 1].

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

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

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

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

For shadow forms (the sampler parameter is a shadow-type), a depth comparison lookup on the depth texture bound to *sampler* is done as described in section 3.8.16 "Texture Comparison Modes" of the OpenGL ES Graphics System Specification. See the table below for which component specifies D_{ref}. The texture bound to *sampler* must be a depth texture, or results are undefined. If a non-shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned on, then results are undefined. If a shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned off, then results are undefined. If a shadow texture call is made to a sampler that does not represent a depth texture, then results are undefined.

In all functions below, the *bias* parameter is optional for fragment shaders. The *bias* parameter is not accepted in a vertex shader. For a fragment shader, if *bias* is present, it is added to the implicit level of detail prior to performing the texture access operation.

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

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

For **Cube** forms, the direction of *P* is used to select which face to do a 2-dimensional texture lookup in, as described in section 3.8.10 "Cube Map Texture Selection" in the OpenGL ES Graphics System Specification.

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

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

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

94

Syntax	Description
highp vec2 **textureSize** (gsampler2D *sampler*, int *lod*) highp vec3 **textureSize** (gsampler3D *sampler*, int *lod*) highp vec2 **textureSize** (gsamplerCube *sampler*, int *lod*) highp vec2 **textureSize** (sampler2DShadow *sampler*, int *lod*) highp vec2 **textureSize** (samplerCubeShadow *sampler*, int *lod*) highp vec3 **textureSize** (gsampler2DArray *sampler*, int *lod*) highp vec3 **textureSize** (sampler2DArrayShadow *sampler*, int *lod*)	Returns the dimensions of level *lod* for the texture bound to *sampler*, as described in section 2.11.9 "Shader Execution" of the OpenGL ES 3.0 Graphics System Specification, under "Texture Size Query". The components in the return value are filled in, in order, with the width, height, depth of the texture. For the array forms, the last component of the return value is the number of layers in the texture array.
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*]) float **texture** (sampler2DArrayShadow *sampler,* vec4 *P*)	Use the texture coordinate *P* to do a texture lookup in the texture currently bound to *sampler*. The last component of *P* is used as D_{ref} for the shadow forms. For array forms, the array layer comes from the last component of *P* in the non-shadow forms, and the second to last component of *P* in the shadow forms.
gvec4 **textureProj** (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**.

Syntax	Description
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*)	Do a texture lookup as in **texture** but with explicit LOD; *lod* specifies λ_{base} and sets the partial derivatives as follows. (See section 3.8.9 "Texture Minification" and equation 3.14 in the OpenGL ES 3.0 Graphics System Specification.) $$\frac{\partial u}{\partial x} = 0 \quad \frac{\partial v}{\partial x} = 0 \quad \frac{\partial w}{\partial x} = 0$$ $$\frac{\partial u}{\partial y} = 0 \quad \frac{\partial v}{\partial y} = 0 \quad \frac{\partial w}{\partial y} = 0$$

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 MIN_PROGRAM_TEXEL_OFFSET and MAX_PROGRAM_TEXEL_OFFSET, respectively.
	
	Note that *offset* does not apply to the layer coordinate for texture arrays. This is explained in detail in section 3.8.9 "Texture Minification" of the OpenGL ES Graphics System Specification, where *offset* is $(\delta_u, \delta_v, \delta_w)$. Note that texel offsets are also not supported for cube maps.

Syntax	Description
gvec4 **texelFetch** (gsampler2D *sampler*, ivec2 *P*, int *lod*) gvec4 **texelFetch** (gsampler3D *sampler*, ivec3 *P*, int *lod*) gvec4 **texelFetch** (gsampler2DArray *sampler*, ivec3 *P*, int *lod*)	Use integer texture coordinate *P* to lookup a single texel from *sampler*. The array layer comes from the last component of *P* for the array forms. The level-of-detail *lod* is as described in sections 2.11.9 "Shader Execution" under Texel Fetches and 3.8 "Texturing" of the OpenGL ES 3.0 Graphics System Specification.

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,* vec3 *P*,
                    float *lod*, ivec2 *offset*)
 	
gvec4 **textureLodOffset** (gsampler2DArray *sampler*, vec3 *P*,
                    float *lod*, ivec2 *offset*)
	Do an offset texture lookup with explicit LOD. See **textureLod** and **textureOffset**.

Syntax	Description
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 LOD. 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 LOD. See **textureProj**, **textureLod**, and **textureOffset**.

gvec4 **textureGrad** (gsampler2D *sampler,* vec2 *P*,
          vec2 *dPdx*, vec2 *dPdy*)
gvec4 **textureGrad** (gsampler3D *sampler,* vec3 *P*,
          vec3 *dPdx*, vec3 *dPdy*)
gvec4 **textureGrad** (gsamplerCube *sampler*, vec3 *P*,
          vec3 *dPdx*, vec3 *dPdy*)
 	
float **textureGrad** (sampler2DShadow *sampler*, vec3 *P*,
          vec2 *dPdx*, vec2 *dPdy*)
float **textureGrad** (samplerCubeShadow *sampler*, vec4 *P*,
          vec3 *dPdx*, vec3 *dPdy*)
 	
gvec4 **textureGrad** (gsampler2DArray *sampler,* vec3 *P*,
          vec2 *dPdx*, vec2 *dPdy*)
 	
float **textureGrad** (sampler2DArrayShadow *sampler*, vec4 *P*,
          vec2 *dPdx*, vec2 *dPdy*)
	Do a texture lookup as in **texture** but with explicit gradients. The partial derivatives of *P* are with respect to window x and window y. Set
	
	$\dfrac{\partial s}{\partial x} = \dfrac{\partial P.s}{\partial x}$
	
	$\dfrac{\partial s}{\partial y} = \dfrac{\partial P.s}{\partial y}$
	
	$\dfrac{\partial t}{\partial x} = \dfrac{\partial P.t}{\partial x}$
	
	$\dfrac{\partial t}{\partial y} = \dfrac{\partial P.t}{\partial y}$
	
	$\dfrac{\partial r}{\partial x} = \dfrac{\partial P.p}{\partial x}$ (cube)
	
	$\dfrac{\partial r}{\partial y} = \dfrac{\partial P.p}{\partial y}$ (cube)
	
	For the cube version, the partial derivatives of *P* are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face.

Syntax	Description
gvec4 **textureGradOffset** (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 Fragment Processing Functions

Fragment processing functions are only available in fragment shaders.

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:

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

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

Backward differencing:

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

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

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

dFdy is approximated similarly, with *y* replacing *x*.

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 3.2 "Invariance" of the OpenGL ES Graphics System Specification, is relaxed for derivative calculations, because the method may be a function of fragment location.

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

4. Functions should be evaluated within the interior of a primitive (interpolated, not extrapolated).

5. Functions for **dFdx** should be evaluated while holding y constant. Functions for **dFdy** should be evaluated while holding x constant. However, mixed higher order derivatives, like **dFdx**(**dFdy**(*y*)) and **dFdy**(**dFdx**(*x*)) are undefined.

6. Derivatives of constant arguments should be 0.

In some implementations, varying degrees of derivative accuracy may be obtained by providing GL hints (section 5.3 "Hints" of the OpenGL ES 3.0 Graphics System Specification), allowing a user to make an image quality versus speed trade off.

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

9 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
BREAK CONTINUE DO ELSE FOR IF DISCARD RETURN SWITCH CASE DEFAULT
BVEC2 BVEC3 BVEC4 IVEC2 IVEC3 IVEC4 UVEC2 UVEC3 UVEC4 VEC2 VEC3 VEC4
MAT2 MAT3 MAT4 CENTROID IN OUT INOUT UNIFORM

FLAT SMOOTH LAYOUT
MAT2X2 MAT2X3 MAT2X4
MAT3X2 MAT3X3 MAT3X4
MAT4X2 MAT4X3 MAT4X4
SAMPLER2D SAMPLER3D SAMPLERCUBE  SAMPLER2DSHADOW
SAMPLERCUBESHADOW  SAMPLER2DARRAY
SAMPLER2DARRAYSHADOW  ISAMPLER2D ISAMPLER3D ISAMPLERCUBE
ISAMPLER2DARRAY  USAMPLER2D USAMPLER3D
USAMPLERCUBE  USAMPLER2DARRAY

STRUCT VOID WHILE

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

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

INVARIANT
HIGH_PRECISION MEDIUM_PRECISION LOW_PRECISION PRECISION
```

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

variable_identifier:
    IDENTIFIER

primary_expression:
    variable_identifier

INTCONSTANT
UINTCONSTANT
FLOATCONSTANT
BOOLCONSTANT
LEFT_PAREN expression RIGHT_PAREN

postfix_expression:
    primary_expression
    postfix_expression LEFT_BRACKET integer_expression RIGHT_BRACKET
    function_call
    postfix_expression DOT FIELD_SELECTION
    postfix_expression INC_OP
    postfix_expression DEC_OP

integer_expression:
    expression

function_call:
    function_call_or_method

function_call_or_method:
    function_call_generic
    postfix_expression DOT function_call_generic

function_call_generic:
    function_call_header_with_parameters RIGHT_PAREN
    function_call_header_no_parameters RIGHT_PAREN

function_call_header_no_parameters:
    function_call_header VOID
    function_call_header

function_call_header_with_parameters:
    function_call_header assignment_expression
    function_call_header_with_parameters COMMA assignment_expression

function_call_header:
    function_identifier LEFT_PAREN

// Grammar Note: Constructors look like functions, but lexical analysis recognized most of them as
// keywords. They are now recognized through "type_specifier".
// Methods (.length) and identifiers are recognized through postfix_expression.

function_identifier:
    type_specifier
    IDENTIFIER
    FIELD_SELECTION

unary_expression:
    postfix_expression
    INC_OP unary_expression
    DEC_OP unary_expression
    unary_operator unary_expression

// Grammar Note: No traditional style type casts.

unary_operator:
    PLUS
    DASH
    BANG
    TILDE

// Grammar Note: No '*' or '&' unary ops. Pointers are not supported.

multiplicative_expression:
    unary_expression
    multiplicative_expression STAR unary_expression
    multiplicative_expression SLASH unary_expression
    multiplicative_expression PERCENT unary_expression

additive_expression:
    multiplicative_expression
    additive_expression PLUS multiplicative_expression
    additive_expression DASH multiplicative_expression

shift_expression:
    additive_expression
    shift_expression LEFT_OP additive_expression
    shift_expression RIGHT_OP additive_expression

relational_expression:

    shift_expression

    relational_expression LEFT_ANGLE shift_expression

    relational_expression RIGHT_ANGLE shift_expression

    relational_expression LE_OP shift_expression

    relational_expression GE_OP shift_expression

equality_expression:

    relational_expression

    equality_expression EQ_OP relational_expression

    equality_expression NE_OP relational_expression

and_expression:

    equality_expression

    and_expression AMPERSAND equality_expression

exclusive_or_expression:

    and_expression

    exclusive_or_expression CARET and_expression

inclusive_or_expression:

    exclusive_or_expression

    inclusive_or_expression VERTICAL_BAR exclusive_or_expression

logical_and_expression:

    inclusive_or_expression

    logical_and_expression AND_OP inclusive_or_expression

logical_xor_expression:

    logical_and_expression

    logical_xor_expression XOR_OP logical_and_expression

logical_or_expression:

    logical_xor_expression

    logical_or_expression OR_OP logical_xor_expression

conditional_expression:

    logical_or_expression

    logical_or_expression QUESTION expression COLON assignment_expression

assignment_expression:

    conditional_expression

    unary_expression assignment_operator assignment_expression

assignment_operator:

    EQUAL

    MUL_ASSIGN

    DIV_ASSIGN

    MOD_ASSIGN

    ADD_ASSIGN

    SUB_ASSIGN

    LEFT_ASSIGN

    RIGHT_ASSIGN

    AND_ASSIGN

    XOR_ASSIGN

    OR_ASSIGN

expression:

    assignment_expression

    expression COMMA assignment_expression

constant_expression:

    conditional_expression

declaration:

    function_prototype SEMICOLON

    init_declarator_list SEMICOLON

    PRECISION precision_qualifier type_specifier_no_prec SEMICOLON

    type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE SEMICOLON

    type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
                                               IDENTIFIER SEMICOLON

    type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
             IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET SEMICOLON

    type_qualifier SEMICOLON

function_prototype:

    function_declarator RIGHT_PAREN

function_declarator:
    function_header
    function_header_with_parameters

function_header_with_parameters:
    function_header parameter_declaration
    function_header_with_parameters COMMA parameter_declaration

function_header:
    fully_specified_type IDENTIFIER LEFT_PAREN

parameter_declarator:
    type_specifier IDENTIFIER
    type_specifier IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

parameter_declaration:
    parameter_type_qualifier parameter_qualifier parameter_declarator
    parameter_qualifier parameter_declarator
    parameter_type_qualifier parameter_qualifier parameter_type_specifier
    parameter_qualifier parameter_type_specifier

parameter_qualifier:
    / empty */*
    IN
    OUT
    INOUT

parameter_type_specifier:
    type_specifier

init_declarator_list:
    single_declaration
    init_declarator_list COMMA IDENTIFIER

    init_declarator_list COMMA IDENTIFIER LEFT_BRACKET constant_expression
                                  RIGHT_BRACKET
    init_declarator_list COMMA IDENTIFIER LEFT_BRACKET
                                    RIGHT_BRACKET EQUAL initializer
    init_declarator_list COMMA IDENTIFIER LEFT_BRACKET constant_expression
                                    RIGHT_BRACKET EQUAL initializer

    init_declarator_list COMMA IDENTIFIER EQUAL initializer

single_declaration:
    fully_specified_type
    fully_specified_type IDENTIFIER

    fully_specified_type IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET
    fully_specified_type IDENTIFIER LEFT_BRACKET RIGHT_BRACKET EQUAL initializer
    fully_specified_type IDENTIFIER LEFT_BRACKET constant_expression
                                        RIGHT_BRACKET EQUAL initializer
    fully_specified_type IDENTIFIER EQUAL initializer
    INVARIANT IDENTIFIER

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

fully_specified_type:
    type_specifier
    type_qualifier type_specifier

invariant_qualifier:
    INVARIANT

interpolation_qualifier:
    SMOOTH
    FLAT

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

parameter_type_qualifier:
    CONST

type_qualifier:

    storage_qualifier

    layout_qualifier

    layout_qualifier storage_qualifier

    interpolation_qualifier storage_qualifier

    interpolation_qualifier

    invariant_qualifier storage_qualifier

    invariant_qualifier interpolation_qualifier storage_qualifier

storage_qualifier:

    CONST

    IN

    OUT

    CENTROID IN

    CENTROID OUT

    UNIFORM

type_specifier:

    type_specifier_no_prec

    precision_qualifier type_specifier_no_prec

type_specifier_no_prec:

    type_specifier_nonarray

    type_specifier_nonarray LEFT_BRACKET RIGHT_BRACKET

    type_specifier_nonarray LEFT_BRACKET constant_expression RIGHT_BRACKET

type_specifier_nonarray:

    VOID

    FLOAT

    INT

    UINT

    BOOL

    VEC2

    VEC3

    VEC4

    BVEC2

    BVEC3

BVEC4

IVEC2

IVEC3

IVEC4

UVEC2

UVEC3

UVEC4

MAT2

MAT3

MAT4

MAT2X2

MAT2X3

MAT2X4

MAT3X2

MAT3X3

MAT3X4

MAT4X2

MAT4X3

MAT4X4

SAMPLER2D

SAMPLER3D

SAMPLERCUBE

SAMPLER2DSHADOW

SAMPLERCUBESHADOW

SAMPLER2DARRAY

SAMPLER2DARRAYSHADOW

ISAMPLER2D

ISAMPLER3D

ISAMPLERCUBE

ISAMPLER2DARRAY

USAMPLER2D

USAMPLER3D

USAMPLERCUBE

USAMPLER2DARRAY

struct_specifier

TYPE_NAME

precision_qualifier:
    HIGH_PRECISION
    MEDIUM_PRECISION
    LOW_PRECISION

struct_specifier:
    STRUCT IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
    STRUCT LEFT_BRACE struct_declaration_list RIGHT_BRACE

struct_declaration_list:
    struct_declaration
    struct_declaration_list struct_declaration

struct_declaration:
    type_specifier struct_declarator_list SEMICOLON
    type_qualifier type_specifier struct_declarator_list SEMICOLON

struct_declarator_list:
    struct_declarator
    struct_declarator_list COMMA struct_declarator

struct_declarator:
    IDENTIFIER
    IDENTIFIER LEFT_BRACKET RIGHT_BRACKET
    IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

initializer:
    assignment_expression

declaration_statement:
    declaration

statement:
    compound_statement_with_scope
    simple_statement

statement_no_new_scope:
    compound_statement_no_new_scope
    simple_statement

statement_with_scope:

compound_statement_no_new_scope
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_with_scope:
    LEFT_BRACE RIGHT_BRACE
    LEFT_BRACE statement_list RIGHT_BRACE

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_with_scope ELSE statement_with_scope
    statement_with_scope

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_with_scope 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 GLSL ES language. Certain constructs that are valid purely in terms of the grammar are disallowed by statements elsewhere in this specification.

Rules specifying the scoping are present only to assist the understanding of scoping and they do not affect the language accepted by the grammar. If required, the grammar can be simplified by making the following substitutions:

- Replace *compound_statement_with_scope* and *compound_statement_no_new_scope* with a new rule *compound_statement*
- Replace *statement_with_scope* and *statement_no_new_scope* with the existing rule *statement*.

10 Errors

This section lists errors that must be detected by the compiler or linker. Development systems must report all grammatical errors are compile time but otherwise, it is implementation-dependent whether an error is reported at compile time or link time and there is no guarantee of consistency.

The error string returned is implementation-dependent.

10.1 Preprocessor Errors

P0001: Preprocessor syntax error

P0002: #error

P0003: #extension if a required extension extension_name is not supported, or if all is specified.

P0005: Invalid #version construct

P0006: #line has wrong parameters

P0007: Language version not supported

P0008: Use of undefined macro

P0009: Macro name too long

10.2 Lexer/Parser Errors

Grammatical errors occurs whenever the grammar rules are not followed. They are not listed individually here.

L0001: Syntax error

The parser also detects the following errors:

L0002: Undefined identifier.

L0003: Use of reserved keywords

L0004: Identifier too long

L0005: Integer constant too long

10.3 Semantic Errors

S0001: Type mismatch in expression e.g. $1 + 1.0$

S0002: Array parameter must be an integer

S0003: Conditional jump parameter (**if**, **for**, **while**, **do-while**) must be a boolean

S0004: Operator not supported for operand types (e.g. mat4 * vec3)

S0005: **?:** parameter must be a boolean

S0006: 2nd and 3rd parameters of **?:** must have the same type

S0007: Wrong arguments for constructor

S0008: Argument unused in constructor

S0009: Too few arguments for constructor

S0011: Arguments in wrong order for structure constructor

S0012: Expression must be a constant expression

S0013: Initializer for constant variable must be a constant expression

S0015: Expression must be a constant integral expression

S0017: Array size must be greater than zero

S0018: Array size not defined

S0020: Indexing an array with a constant integral expression greater than its declared size

S0021: Indexing an array with a negative constant integral expression

S0022: Redefinition of variable in same scope

S0023: Redefinition of function in same scope

S0024: Redefinition of name in same scope (e.g. declaring a function with the same name as a struct)

S0025: Field selectors must be from the same set (cannot mix xyzw with rgba)

S0026: Illegal field selector (e.g. using .z with a **vec2**)

S0027: Target of assignment is not an l-value

S0028: Precision used with type other than **int**, **float** or sampler type

S0029: Declaring a main function with the wrong signature or return type

S0031: **const** variable does not have initializer

S0032: Use of **float** or **int** without a precision qualifier where the default precision is not defined

S0033: Expression that does not have an intrinsic precision where the default precision is not defined

S0034: Variable cannot be declared invariant

S0035: All uses of invariant must be at the global scope

S0037: L-value contains duplicate components (e.g. v.xx = q;)

S0038: Function declared with a return value but return statement has no argument

S0039: Function declared void but return statement has an argument

S0040: Function declared with a return value but not all paths return a value

S0042: Return type of function definition must match return type of function declaration.

S0043: Parameter qualifiers of function definition must match parameter qualifiers of function declaration.

S0045: Declaring an input inside a function

S0046: Declaring a uniform inside a function

S0047: Declaring an output inside a function

S0048: Illegal data type for vertex output or fragment input

S0049: Illegal data type for vertex input (can only use **float**, floating-point vectors, matrices, signed and unsigned integers and integer vectors)

S0050: Initializer for input

S0051: Initializer for output

S0052: Initializer for uniform

S0053: Static recursion present

S0054: Overloading built-in functions not allowed.

S0055: Vertex output with integer type must be declared as flat

S0056: Fragment intput with integer type must be declared as flat

S0057: *init-expression* in switch statement must be a scalar integer

S0058: Illegal data type for fragment output

S0059: Invalid layout qualifier

S0060: Invalid use of layout qualifier (e.g. on vertex shader outputs or fragment shader inputs)

10.4 Linker

L0001: Global variables must have the same type (including the same names for structure and field names and the same size for arrays) and precision.

L0003: Too many vertex input values

L0004: Too many vertex output values

L0005: Too many uniform values

L0006: Too many vertex output values

L0007: Fragment shader uses an input where there is no corresponding vertex output

L0008: Type mismatch between vertex output and fragment input

L0009: Missing main function for shader

11 Counting of Inputs and Outputs

This section applies to vertex shader outputs and fragment shader inputs.

GLSL ES 3.0 specifies the storage available for vertex shader outputs and fragment shader inputs 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 3.0 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.

In all cases, failing resource allocation for variables 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 for vertex output and fragment input variables. Each register can contain a float value.

- 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. Arrays of mat4 and mat4

    2. Arrays of mat2 and mat2 (since they occupy full rows)

    3. Arrays of vec4 and vec4

    4. Arrays of mat3 and mat3

    5. Arrays of vec3 and vec3

    6. Arrays of vec2 and vec2

    7. Arrays of float and float

- 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], ve4[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 (i.e. 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.

- If at any time the packing of a variable fails, the compiler or linker must report an error.

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.

If referenced in the fragment shader (after preprocessing), the built-in special variables (gl_FragCoord, gl_FrontFacing and gl_PointCoord) are included when calculating the storage requirements of fragment inputs.

Vertex outputs and fragment inputs are counted separately. They are only counted if they are statically used within the shader.

12 Issues

12.1   Compatibility with OpenGL ES 2.0

How should OpenGL ES 3.0 support shaders written for OpenGL ES 2.0?

Option 1: Retain all GLSL ES 1.0 constructs in the new language.

Option 2: Allow GLSL ES 1.0 shaders to run in the OpenGL ES 3.0 API.

RESOLUTION: Option 2. This minimizes the complexity of the language with only a small increase in system complexity. It also leaves open the option of deprecating the old language in future versions of the API.

12.2   Convergence with OpenGL

How much should GLSL ES be influenced by the GLSL specification?

OpenGL ES 3.0 is principally targeted at mobile devices such as smartphones and tablets. As such, it is expected that the major use-cases will include gaming and user-interfaces. It is to be expected that content will be ported to and from desktop devices.

RESOLUTION: In the absence of any other requirements, GLSL ES 3.0 should follow GLSL 3.3. The main exceptions to this are:

- • The specification should adhere to the principle that functionality should not be duplicated.

- • Functionality specific to mobile devices (such as reduced precision) can be added.

- • Improvements found in later versions of GLSL can be considered for inclusion.

12.3   Numeric Precision

Should the Open GL ES 2.0 precision requirements be increased?

Most current implementations support a subset of IEEE 754 32-bit floating point. Many implementations also support reduced precision.

RESOLUTIONS:

- • highp float should be specified as a subset of IEEE 754 floating point.

- • highp int should be exactly 32 bits.

- • lowp and mediump should be retained. Mediump to have increased precision.

Should there be a defined format for mediump?

Option: Yes, this would increase portability and encourage the use of mediump on mobile devices.

Option: No, this would be expensive to implement on devices that do not natively support it.

RESOLUTION: No. The specification should allow efficient implementation of mediump float on 16-bit floating point hardware but must also be implementable on devices which only natively support 32-bit floating point.

Should the fragment shader have a default precision?

Vertex shaders have a default high precision because lower precisions are not sufficient for the majority of graphics applications. However, many fragment shader operations do not benefit from high precision and developers should be encouraged to use lower precision where possible as this may increases performance or reduce power consumption. In particular, blend operations normally only require low precision and many texture address calculations can be performed at medium precision.

However OpenGL ES may also be used in higher performance devices where the benefit is limited. Therefore there appears to be no single precision that would be applicable to all situations.

RESOLUTION:No, there will be no default precision for fragment shaders.

12.4   Floating Point Representation and Functionality

Should IEEE 754 representation be mandated?

The internal format used by an implementation might not be visible to an application so it is meaningless to specify this. Certain functionality IEEE 754 must be present though.

RESOLUTION: In general, highp float must behave *as if* it is in IEEE 754 format.

Which features should be mandated?

Most of the IEEE 754 is relatively inexpensive to implement given that 32-bit floating point is a requirement. However some implementations do not implement signed zeros, rounding modes and NaNs because of hardware cost. In addition, there are certain compiler optimizations that the IEEE 745 specification prohibits.

RESOLUTION: Mandate support of signed infinities. Support of signed zeros, NaNs.

Should the support of NaNs be consistent?

Should the specification allow either full IEEE NaN support or no support but nothing in between?

RESOLUTION: No, implementations may have partial support and there is no guarantee of consistency. The only requirement is that isnan() must return false if NaNs are not supported.

Should subnormal numbers (also known as 'denorms') be supported?

RESOLUTION: No, subnormal numbers maybe flushed to zero at any time.

How should the rounding mode be specified?

Most current implementations support round-to-nearest. Some but not all also support round-to-nearest-even.

RESOLUTION: Within the accuracy specification, the rounding mode should be undefined.

Should there be general invariance rules for numeric formats and operations?

The GLSL ES specification allows the implementation a degree of flexibility. Consequently the results of a computation may be different on different implementations. However, it is not stated whether a single implementation is allowed to vary the results of a given computation, either in different shaders or different parts of the same shader. OpenGL has a general invariance rule that prevents the results of a computation varying if no state (including the choice of shader) is unchanged.

RESOLUTION: Operations and formats are in general considered to be variant.

12.5   Precision Qualifiers

Should the precisions be specified as float16, float32 etc.? This would help portability. It implies different types rather than hints. It will require all implementations to use the same or similar algorithms and reduces the scope for innovation.

RESOLUTION: No, the precision should not specify a format. Standardized arithmetic is not (yet) a requirement for graphics.

Do integers have precision qualifiers? OpenGL ES 3.0 hardware is expected to have native integer support and some implementations may have reduced precision available.

RESOLUTION: Yes, integers have precision qualifiers.

How should wrapping behavior of integers be defined? If an application relies on wrapping on one implementation this may cause portability problems.

Option: The standard should specify either wrapping or clamping. This allows for maximum implementation flexibility.

Option: Mandate wrapping. There is a trend towards more complex shaders and developers will expect integers to behave as in C++.

RESOLUTION: Mandate wrapping.

Are precision qualifiers available in the vertex shader?

RESOLUTION: Yes. Reduced precision may be available in the vertex shader in some implementations and it keeps the languages consistent.

Should different precisions create different types and e.g. require explicit conversion between them?

Option1: No, they are just hints. But hinting high precision is meaningless if the implementation can ignore it.

Option 2: Yes they are different types. But this introduces complexity.

RESOLUTION: The precision qualifier can significantly affect behavior in many implementations. **highp** means 32-bit IEEE 743 floating pointis used but **mediump** means that at least medium precision is used (and similarly for **lowp)** so precision qualifiers are more than just hints. As far as the language is concerned it doesn't affect the behavior so they can either be considered as hints or as different types with implicit type conversion. In any case, implementations are free to calculate everything at high precision.

Should precisions be considered when resolving function calls?

RESOLUTION: No, they should be considered more as hints. Function declarations cannot be overloaded based on precision.

How should precisions be propagated in an expression?

Option 1: Only consider the inputs to an operation. For operands that have no defined precision, determination of precision starts at the leaf nodes of the expression tree and proceeds to the root until the precision is found. If necessary this includes the l-value in an assignment. Constant expressions must be invariant and it is expected that they will be evaluated at compile time. Therefore they must be evaluated at the highest precision (either **lowp** or **highp**) supported by the target, or above.

Option 2: Always take the target of the expression into account. The compiler should be able to work out how to avoid losing precision.

RESOLUTION: Option 1. This makes it easier for the developer to specify which precisions are used in a complex expression.

What if there is no precision in an expression?

Option 1: Leave this as undefined.

Option 2: Use the default precision.

RESOLUTION: Use the default precision. It is an error if this is not defined (in the fragment shader).

Do precision qualifiers for uniforms need to match?

Option 1: Yes.

Uniforms are defined to behave as if they are using the same storage in the vertex and fragment processors and may be implemented this way.

If uniforms are used in both the vertex and fragment shaders, developers should be warned if the precisions are different. Conversion of precision should never be implicit.

Option 2: No.

Uniforms may be used by both shaders but the same precision may not be available in both so there is a justification for allowing them to be different.

Using the same uniform in the vertex and fragment shaders will always require the precision to be specified in the vertex shader (since the default precision is highp). This is an unnecessary burden on developers.

RESOLUTION: Yes, precision qualifiers for uniforms must match.

Do precision qualifiers for vertex outputs and the corresponding fragment inputs (previously known as 'varyings') need to match?

Option 1: Yes. Varyings are written by the vertex shader and read by the fragment shader so there are no situations where the precision needs to be different.

Option 2: No, the vertex outputs written by the vertex shader should not be considered to be the same variables as those read by the fragment shader (there can be no shared storage). Hence they can be specified to have different precisions.

RESOLUTION Precision qualifiers for vertex outputs and fragment inputs do not need to match.

lowp int

lowp float has a range of +/- 2.0 but **lowp int** has a range of +/- 256. This becomes problematic if conversion form **lowp float** to **lowp int** is required. Direct conversion i.e. **lowp int** = **int**(**lowp float**) loses almost all the precision and multiplying before conversion e.g. **lowp int** = **int**(**lowp float** * 256) causes an overflow and hence an undefined result. The only way to maintain precision is to first convert to **mediump float**.

Option 1: Keep this behavior. Accept that conversion of **lowp float** to **low int** loses precision and is therefore not useful.

Options 2: Make **lowp int** consistent with **mediump** and **highp int** by setting its range to +/- 1

Options 3: Redefine the conversion of **lowp float** to **lowp int** to include an 8-bit left shift. The conversion of **lowp int** to **lowp float** then contains an 8-bit right shift.

Option 4: Option 1 but add built-in functions to shift-convert between the two formats.

Option 5: Redefine the **lowp float** to be a true floating point format. It would then be equivalent to a floating point value with a 10 bit mantissa and a 3 bit unsigned exponent.

RESOLUTION: Option 1 Conversion will lose most of the precision.

Precision of built-in texture functions.

Most built-in functions take a single parameter and it is sensible for the precision of the return value to be the same as the precision of the parameter. The texture functions take sampler and coordinate parameters. The return value should be completely independent of the precision of the coordinates. How should the precision of the return value be specified?

RESOLUTION: Allow sampler types to take a precision qualifier. The return value of the texture functions have the same precision as the precision of the sampler parameter.

What should the default precision of sampler types be?

Option 1: **lowp**. This will be faster on some implementations. In general, OpenGL ES should default to fast operation rather than precise operation. It is usually easier to detect and correct a functional error than a performance issue.

Option 2: **lowp** for textures that are expected to contain color values. **highp** for textures that are expected to contain other values e.g. depth.

Option 2: No default precision. Although this requires that the precision be specified in every shader, it will force the developer to consider the requirements.

RESOLUTION: The default precision of all sampler types present in GLSL ES 1.0 should also be **lowp** in GLSL ES 3.0. New sampler types in GLSL ES 3.0 should have no default precision.

12.6   **Function and Variable Name Spaces**

Do variables and functions share the same name space? GLSL ES doesn't support function pointers so the grammar can always be used to distinguish cases. However this is a departure from C++.

RESOLUTION: Functions and variables share the same name space.

Should redeclarations of the same names be permitted within the same scope? This would be compatible with C. There are several cases e.g.:

1. Redeclaring a function. A function prototype is a declaration but not a definition. A function definition is both a declaration and a definition. Consequently a function prototype and a function definition (of the same function) within the same scope qualifies as redeclaration.

2. Declaring a name as a function and then redeclaring it as a structure.

3. Declaring a name as a variable and then redeclaring it as a structure.

Disallowing multiple function declarations (including allowing a separate function prototype and function definition) would prevent static recursion by design. However it imposes constraints on the structure of shaders.

GLSL ES 1.00 allows a single function definition plus a single optional function declaration.

RESOLUTION: Multiple definitions are disallowed. Multiple function declarations (function prototypes) are allowed. This is in line with C++.

12.7 Local Function Declarations and Function Hiding

Should local functions hide all functions of the same same?

This is considered useful if local function declarations are allowed. However, the only use for local function declarations in GLSL ES is to unhide functions that have been hidden by variable or structure declarations. This is not a compelling reason to include them.

RESOLUTION: Disallow local function declarations.

12.8 Overloading main()

Should it be possible for the user to overload the main() function?

RESOLUTION: No. The main function cannot be overloaded.

12.9 Error Reporting

In general which errors *must* be reported by the compiler?

Some errors are easy to detect. All grammar errors and type matching errors will normally be detected as part of the normal compilation process. Other semantic errors will require specific code in the compiler. The bulk of the work in a compiler occurs after parsing so adding some error detection should not increase the total cost of compilation significantly. However, it is expected that development systems will have sophisticated error and warning reporting and it is not necessary to repeat this process for on-target compilers.

RESOLUTION: All grammar, type mismatch and other specific semantic errors as listed in this specification must be reported. Reporting of other errors or warnings is optional.

Should compilers report if maxima are exceeded, even if the implementation supports them? This could aid portability.

RESOLUTION: No, high-end implementations may quite legitimately go beyond the specification in these areas and mandating the use of the extension mechanism would cause needless complexity. Development systems should issue portability warnings.

Should static recursion be detected?

RESOLUTION: Yes, the compiler will normally generate the necessary control flow graph so detection is easy.

12.10 Structure Declarations

Should structures with the same name and same member variables be considered as the same type?

RESOLUTION: No, follow the C++ rules. Variables only have the same type if they have been declared with the same type and not if they have been declared with different types that have the same name. This does not apply to linking (for uniforms and varyings) which has its own rules.

Should structure declarations be allowed in function parameters?

RESOLUTION: No, following the previous resolution it would be impossible to call such a function because it would be impossible to declare a variable with the same structure type.

12.11 Embedded Structure Definitions

Should embedded structure definitions be allowed?

e.g.

```
struct S
{
  struct T
  {
    int a;
  } t;
  int b:
};
```

In order to access the constructor, the structure name would have to be scoped at the same level as the outer level structure. This is inconsistent.

Option 1: Disallow embedded structure definitions.

Option 2: Allow embedded structure definitions but accept that the constructor is not accessible.

Option 3: Scope embedded structure names at the same level as the outermost scope name.

RESOLUTION: Remove embedded structure definitions.

12.12 Redefining Built-in Functions

Should it be possible to redefine or overload built-in functions?

There may be some applications where it is useful to redefine the built-in functions but the language does not include the required functionality for all cases. Built-in functions are likely to be efficiently mapped to the hardware. User-defined functions may not be as efficient but may be able to offer greater precision (e.g. for the trig functions). The application may then want access to both the original and new function. Some user-defined functions would benefit from access to the original function. Once the new function has been declared, the original function is hidden so both these use cases are impossible with the current specification.

Option 1: Allow both redefinition and overloading of built-in functions.

Option 2: Disallow redefinition of built-in functions. Allow them to be overloaded. This may be useful where it is required to extend the functionality of a built-in function. However it creates a subtle incompatibility with the desktop:

```
int sin(int x) {return x;}
void main()
{
  float a = sin(1.0);  // legal in GLSL ES, not legal in desktop GLSL.
}
```

It is also a potential source of backwards-incompatibility if a future version of the language introduces new overloads.

Option 3: Remove the ability to redefine or overload functions.

RESOLUTION: Disallow both overloading and redefining built-in functions. There is no compelling use case.

12.13 Global Scope

How should the scoping levels for user-defined and built-in names be defined?

GLSL ES 1.00 and most versions of GLSL have a global scope for user-defined functions and variables and a distinct 'outer' scope where the built-in functions reside. This is different from C++. Since GLSL ES 3.00 does not allow the redefinition of built-in functions, a single global scope is sufficient.

RESOLUTION: A single global scope will be used for user-defined and built-in names.

12.14 Constant Expressions

Should user and built-in functions be allowed in constant expressions? e.g.

```
const float a = sin(1.0);
```

The compiler must be able to evaluate all possible constant expressions as they can potentially be used to size arrays and functions resolution is dependent on array size. Compile-time evaluation of built-in functions is expensive in terms of code size. The complexity of compile-time evaluation of user-defined functions is potentially unbounded.

RESOLUTION: Allow built-in functions to be included in constant expressions. Redefinition of built-in functions is prohibited. User-defined functions are not allowed in constant expressions.

12.15 Varying Linkage

In the vertex shader, a particular varying may be either 1) not declared, 2) declared but not written, 3) declared and written but not in all possible paths or 4) declared and written in all paths. Likewise a varying in a fragment shader may be either a) not declared, b) declared but not read, c) declared and read in some paths or d) declared and read in all paths. Which of these 16 combinations should generate an error?

The compiler should not attempt to discover if a varying is read or written in all possible paths. This is considered too complex for OpenGL ES.

The same vertex shader may be paired with different fragment shaders. These fragment shaders may use a subset of the available input varyings. This behavior should be supported without causing errors. Therefore if the vertex shader writes to a varying that the fragment shader doesn't declare or declared but doesn't read then this is not an error.

If the vertex shader declares but doesn't write to a varying and the fragment shader declares and reads it, is this an error?

RESOLUTION: No.

RESOLUTION: The only error case is when a varying is declared and read by the fragment shader but is not declared in the vertex shader.

12.16 gl_Position

Is it an error if the vertex shader doesn't write to gl_Position? Whether a shader writes to gl_Position cannot always be determined e.g. if there is dependence on an attribute.

Option 1: No it is not an error. The behavior is undefined in this case. Development systems should issue a warning in this case but the on-target compiler should not have to detect this.

Option 2: It is an error if the vertex shader does not statically write to gl_Position

Option 3: It is an error if there is any static path through the shader where gl_Position is not written.

RESOLUTION: No error (option 1). The nature of the undefined behavior must be specified.

12.17 Pre-processor

Is the preprocessor necessary?

Arguments for removing or simplifying the preprocessor:

- The preprocessor is moderately complex to implement. In particular, function-like macros may have arbitrary complexity and require significant resources to compile.

- The C++ standard does not fully specify the preprocessor. In particular, the situations where preprocessor tokens are subject to macro expansion are not fully defined. Neither is the effect of macro definitions encountered during macro expansion.

- Over-use of the preprocessor is a common source of programming errors because there is limited compile-time checking.

Arguments for retaining the preprocessor:

- The extension mechanism relies on the preprocessor so this would need to be replaced.

- The **#define, #ifdef**, **#ifndef**, **#elsif** and **#endif** constructs are commonly used for managing different versions and for include guards.

- There is no template mechanism in GLSL ES so macros are often used instead.

GLSL ES 1.00 removed token pasting and other functionality.

RESOLUTION: Keep the basic preprocessor as defined in the GLSL ES 1.00 specification.

12.18 Character set

GLSL ES 1.00 only allowed a subset of the ascii character set to be used in shaders. That included names and comments. The written languages of many countries include other characters or use a completely different character set. This makes it difficult or impossible to write comments in those languages.

Where should the new characters be allowed? It would be possible to decide independently for comments, identifiers and macros. For macros, they could be allowed as part of macro definitions but prohibited in the final output of macro expansion.

RESOLUTION: The new characters are only allowed inside comments.

Which character set should be used to define the new characters.

UTF-8 has the advantage that it is backwards-compatible with ASCII. All ASCII characters are valid UTF-8 single-byte characters and UTF-8 multi-byte characters all have the highest bit set to '1' in each byte. The disadvantage is that UTF-8 is variable length.

RESOLUTION: UTF-8

How should the extended character set be specified?

Options include full UTF-8 or by explicitly listing the allowed characters.

RESOLUTION: Full UTF-8

Should the compiler check for the presence of invalid UTF-8 byte sequences?

Since any multi-byte characters will only occur within comments and so not required further processing, it would be inexpensive to check for valid UTF-8 characters. Conversely, there appears to be no advantage to doing so. The issue of validity is only of concern to text editors.

RESOLUTION: The compiler must not check for invalid UTF-8 characters. Bytes '0' and newline characters will be interpreted as such wherever they occur.

How does the #version directive interact with the use of UTF-8 in comments?

Following C++, the 'phases of translation' specification defines comment processing to be performed before macro directives are processed. However UTF-8 is legal in GLSL ES 3.00, identified by #version 300 but not in GLSL ES 1.00, identified by #version 100 (or by absence of a #version directive).

Option: The shader is processed in 2 passes. The first determines the shader version and the second performs compilation as before.

Option: Replace the current version directive mechanism with a byte sequence that must always occur at the start of the shader. This is similar to other standards that have multiple versions e.g. http.

Option: Make UTF-8 characters an optional feature of GLSL ES 1.00

RESOLUTION: UTF-8 will be made optional in GLSL ES 1.00

12.19 Line Continuation

Should the line continuation character '\' be included in the specification?

Line continuation was deliberately excluded from previous versions of GLSL and GLSL ES in order to discourage excessive use of the pre-processor. However, function-like macros are commonly used because there is no 'template' mechanism, which would allow functions to be parametrized by a type. Long macro definitions are therefore not uncommon and the line-continuation character may aid readability.

Given that shader source is stored in a list of character strings, the newline character can be omitted and this has the same effect as a newline followed by a line-continuation.

RESOLUTION: Include line-continuation.

How does this interact with #version?

RESOLUTION: Same issue as with UTF-8in general. Line-continuation to be made optional in GLSL ES 1.00

12.20 Phases of Compilation

Should the preprocessor run as the very first stage of compilation or after conversion to preprocessor tokens as with C/C++?

The cases where the result is different are not common.

```
#define e +1
int n = 1e;
```

According to the c++ standard, '1e' should be converted to a preprocessor token which then fails conversion to a number. If the preprocessor is run first, '1e' is expanded to '1+1' which is then parsed successfully.

RESOLUTION: Follow c++ rules.

12.21 Maximum Number of Varyings

How should gl_MaxVaryingFloats be defined? Originally this was specified as 32 floats but currently some desktop implementations fail to implement this correctly. Many implementations use 8 vec4 registers and it is difficult to split varyings across multiple registers without losing performance.

Option 1: Specify the maximum as 8 4-vectors. It is then up to the application to pack varyings. Other languages require the packing to be done by the application. Developers have not reported this as a problem.

Option 2: Specify the maximum according to a packing rule. The developer may use a non-optimal packing so it is better to do this in the driver. Requiring the application to pack varyings is problematic when shaders are automatically generated. It is easier for the driver to implement this.

RESOLUTION: The maximum will be specified according to a packing rule.

Should attributes and uniforms follow this rule?

RESOLUTION: Attributes should not follow this rule. They will be continued to be specified as vec4s.

RESOLUTION: Uniforms should not follow this rule for GLSL ES 3.00. Implementations are expected to virtualize such resources.

Should the built-in special variables (gl_FragCoord, gl_FrontFacing, gl_PointCoord) be included in this packing algorithm? Built-in special variables are implemented in a variety of ways. Some implementations keep them in separate hardware, some do not.

RESOLUTION: Any built-in special variables that are statically used in the shader should be included in the packing algorithm.

Should gl_FragCoord be included in the packing algorithm? The x and y components will always be required for rasterization. The z and w components will often be required.

RESOLUTION: gl_FragCoord is included in the count of varyings.

How should **mat2** varyings be packed?

Option 1: Pack them as 2x2.

Option 2: Pack them as 4 columns x 1 row. This is usually more efficient for an implementation.

Option 3: Allocate a 4 column x 2 row space. This is inefficient but allows flexibility in how implementations map them to registers.

Option 4: As above but pack 2 **mat2** varyings into each 4 column x 2 row block. Any unpaired **mat2** takes a whole 4x2 block.

RESOLUTION: Option 3

Should **mat3** take 3 whole rows?

This would again allow flexibility in implementation but it wastes space that could be used for floats or float arrays.

RESOLUTION: No, **mat3** should take a 3x3 block.

Should **vec3** take a whole row?

RESOLUTION: No.

Should gl_MaxVertexUniformsComponents be changed (from desktop GLSL) to reflect the packing rules?

RESOLUTION: Rename gl_MaxVertexUniformComponents to gl_MaxVertexUniformVectors. Rename gl_MaxFragmentUniformComponents to gl_MaxFragmentUniformVectors.

12.22 Array Declarations

Unsized array declarations.

Desktop GLSL allows arrays to be declared without a size and these can then be accessed with constant integral expressions. The size never needs to be declared. This was to support gl_Texcoord e.g.

```
varying vec4 gl_TexCoord[];
...
gl_FragColor = texture (tex, gl_TexCoord[0].xy);
```

This allows gl_TexCoord to be used without having to declare the number of texture units.

gl_TexCoord is part of the fixed functionality so unsized arrays should be removed for GLSL ES

RESOLUTION: Remove unsized array declarations.

Which forms of array declarations should be permitted?

```
    float a[5];
    ...
    float b[] = a;  // b is explicitly size 5
or
    float a[] = float[] (1.0, 2.0, 3.0);
```

RESOLUTION: All above constructs are valid. However, any declaration that leaves the size undefined is disallowed as this would add complexity and there are no use-cases.

12.23 Invariance

How should invariance between shaders be handled?

Version 1.10 of desktop GLSL uses ftransform() to guarantee that gl_Position can be guaranteed to be calculated the same way in different vertex shaders. This relies on the fixed function that has been removed from ES. It is also very restrictive in that it only allows vertex transforms based on matrices. It does not apply to other values such as those used to generate texture coordinates.

Option 1: Specify all operations to be invariant. No, this is too restrictive. Optimum use of resources becomes impossible for some implementations.

136

Option 2: Add an invariance qualifier to functions that require invariance. No, this does not work as the inputs to the functions and operations performed on the outputs may not be invariant.

Option 3: Add an invariance qualifier to all variables (including shader outputs).

RESOLUTION: Add an invariance qualifier to variables but permit its use only for outputs from the vertex and fragment shaders. Add a global invariance option for use when complete invariance is required.

Should the invariance qualifier be permitted on parameters to texture functions?

Many algorithms rely on two or more textures being exactly aligned, either within a single invocation of a shader or using multi-pass techniques. This could be guaranteed by using the invariant qualifier on variables that are used as parameters to the texture function.

Using the global invariance pragma also guarantees alignment of the textures. It is not clear whether allowing finer control of invariance is useful in practice. Compilers may revert to global invariance and there may be other specific cases that need to be considered.

RESOLUTION: Use of a variable as a parameter to a texture function does not imply that it may be qualified as invariant.

Do invariance qualifiers for declarations in the vertex and fragment shaders need to match?

Option 1: Only allow invariance declarations on varyings in the vertex shader. The invariance of the varying in the fragment shader should then be guaranteed automatically.

Option 2: Specify that they must match.

RESOLUTION: Invariance qualifiers for varying declarations must match.

Should this rule apply if the varying is declared but not used?

RESOLUTION: Yes, this rule applies for declarations, independent of usage.

How does this rule apply to the built-in special variables.

Option 1: It should be the same as for varyings. But gl_Position is used internally by the rasterizer as well as for gl_FragCoord so there may be cases where rasterization is required to be invariant but gl_FragCoord is not.

Option 2: gl_FragCoord and gl_PointCoord can be qualified as invariance if and only if gl_Position and gl_PointSize are qualified invariant, respectively.

Can undefined values be made invariant?

If a type is implemented by a larger native type and due to lack of initialization, a variable of that type has an illegal value, it is possible for variant behaviour to occur.

For example suppose a boolean is represented by a 32-bit integer with 'false' represented as 0 and 'true' represented as '1'. If the compiler uses both an 'equals 0' and an 'equals 1' test, the following may occur:

```
bool b; // The implementation sets this to an illegal value e.g. 3

if (b) // implementation tests 'b == 1' which is false
{
  f();
}
else  // implementation  tests 'b == 0' which is also false
{
  g();
}
```

Neither f() nor g() are executed which is unexpected behaviour. Such cases could be made invariant but would for example require the compiler to initialize undefined values which is a performance cost.

RESOLUTION: Undefined values cannot be made invariant. These shaders are malformed and therefore have undefined behavior.

12.24 Invariance Within a shader

How should invariance within a shader be specified?

Compilers may decide to recalculate a value rather than store it in a register (rematerialization). The new value may not be exactly the same as the original value.

Option 1: Prohibit this behavior.

Option 2: Use the invariance qualifier on variables to control this. This is consistent with the desktop.

RESOLUTION: Values with in a shader are in variant be default. The invariance qualifier or pragma may be used to make them invariant.

Should constant expressions be invariant? In the following example, it is not defined whether the literal expression should always evaluate to the same value.

```
precision mediump int;
precision mediump float;
const int size = int(ceil(4.0/3.0 - 0.333333));
int a[size];
for (int i=0; i<int(ceil(4.0/3.0 - 0.333333)); i++) {a [i] = i;}
```

Implementations must usually be able to evaluate constant expressions at compile time since they can be used to declare the size of arrays. Hardware may compute a less accurate value compared with maths libraries available in C. It would however be expected that functions such as sine and cosine return similar results whether or not they are part of a constant expression. This suggests that the implementation might want to evaluate these functions only on the hardware. However, there are no situations, even with global invariance, where compile time evaluation and runtime evaluation must match exactly.

RESOLUTION: Yes, constant expressions must be invariant.

12.25 While-loop Declarations

What is the purpose of allowing variable declarations in a while statement?

```
while (bool b = f()) {...}
```

Boolean b will always be true until the point where it is destroyed. It is useful in C++ since integers are implicitly converted to booleans.

RESOLUTION: Keep this behavior. Will be required if implicit type conversion is added to a future version.

A similar issue exists in for-loops. The grammar allows constructs such as

```
for(;bool x = a < b;) ;
```

12.26 Cross Linking Between Shaders

Should it be permissible for a fragment shader to call a function defined in a vertex shader or *vice versa*?

RESOLUTION: No, there is no need for this behavior.

12.27 Visibility of Declarations

At what point should a declaration take effect?

```
int x=1;
{
  int x=2, y=x; // case A
  int z=z;      // case B
}
```

Option 1: The name should be visible immediately after the identifier. Both cases above are legal. In case A, y is initialized to the value 2. This is consistent with c++. For case B, the use case is to initialize a variable to point to itself e.g. void* p = &p; This is not relevant to GLSL ES.

Option 2: The name should be visible after the initializer (if present), otherwise immediately after the identifier. In case A, y is initialized to 2. Case B is an error (assuming no prior declaration of z).

Option 3: The name should be visible after the declaration. In case A, y is initialized to 1. Case B is an error if z is has no prior declaration.

RESOLUTION: Option 2. Declarations are visible after the initializer if present, otherwise after the identifier.

12.28 Language Version

What version number should the language have? This version of the language is based on version 3.30 of the desktop GLSL. However it includes a number of features that are in version 4.20 but not 3.30. The previous version of GLSL ES was version 1.00 so this version could be called version 2.00.

RESOLUTION: Follow the desktop GLSL convention so that the language version matches the API version. Hence this version will be called 3.00

12.29 Samplers

Should samplers be allowed as l-values? The specification already allows an equivalent behavior:

Current specification:

```
uniform sampler2D sampler[8];
int index = f(...);
vec4 tex = texture(sampler[index], xy); // allowed
```

Using assignment of sampler types:

```
uniform sampler2D s;
s = g(...);
vec4 tex = texture(s, xy); //  not allowed
```

RESOLUTION: Dynamic indexing of sampler arrays is now prohibited by the specification. Restrict indexing of sampler arrays to constant integral expressions.

12.30 Dynamic Indexing

For GLSL ES 1.00, support of dynamic indexing of arrays, vectors and matrices was not mandated because it was not directly supported by some implementations. Software solutions (via program transforms) exist for a subset of cases but lead to poor performance. Should support for dynamic indexing be mandated for GLSL ES 3.00?

RESOLUTION: Mandate support for dynamic indexing of arrays except for sampler arrays, fragment output arrays and uniform block arrays.

Should support for dynamic indexing of vectors and matrices be mandated in GLSL ES 3.00?

RESOLUTION: Yes.

Indexing of arrays of samplers by constant-index-expressions is supported in GLSL ES 1.00. A constant-index-expression is an expression formed from constant-expressions and certain loop indices, defined for a subset of loop constructs. Should this functionality be included in GLSL ES 3.00?

RESOLUTION: No. Arrays of samplers may only be indexed by constant-integral-expressions.

12.31 Maximum Number of Texture Units

The minimum number of texture units that must be supported in the fragment shader is currently 2 as defined by gl_MaxTextureImageUnits = 8. Is this too low for GLSL ES 3.0?

Option 1: Yes, the number of texturing units is the limiting factor for fragment shaders. The number of texture units was increased from 1 to 2 going from OpenGL ES 1.0 to OpenGL ES 1.1 and increased to 8 for OpenGL ES 2.0

RESOLUTION: Increase to 16

12.32 On-target Error Reporting

Should compilers be required to report any errors at compile time or can errors be deferred until link time?

RESOLUTION: If a program cannot be compiled, on-target compilers are only required to report that an error has occurred. This error may be reported at compile time or link time or both. Development systems must generate grammar errors at compile time.

12.33 Rounding of Integer Division

Should the rounding mode be specified for integer division?

The rounding mode for division is related to the definition of the remainder operator. The important relation in most languages (but not relevant in this version of GLSL ES) is:

$$(a / b) * b + a \% b = a \qquad \text{(a and b are integers)}$$

Usually the remainder operator is defined to have the same sign as the dividend which implies that divide must round towards zero. (Note that the modulo function is not the same as the remainder function. Modulo is defined to have the same sign as the divisor).

The remainder operator was not part of GLSL ES 1.00, so it was not necessary to specify the rounding mode. In GLSL ES 3.00, the remainder operator is included but the results are undefined if either or both operands are negative.

RESOLUTION: The rounding mode is undefined for this version of the specification.

12.34 Undefined Return Values

If a function is declared with a non-void return type, any return statements within the definition must specify a return expression with a type matching the return type. However if the function returns without executing a return statement the behaviour is undefined. Should the compiler attempt to check for these cases and report them as an error?

Example:

```
int f()
{
  // no return statement
}

...

int a = f();
```

Option 1: An undefined value is returned to the caller. No error is generated. This is what most c++ compilers do in practice (although the c++ standard actually specifies 'undefined behaviour').

Option 2: There must be a return statement at the end of all function definitions that return a value.

No, this requires statements to be added that may be impossible to execute.

Option 3: A return statement at the end of a function definition is required only if it is possible for execution to reaches the end of the function:

E.g.

```
int f(bool b)
{
  if (b)
    return 1;
  else
    return 0;
  // No error.  The execution can never reach the end of the function so
  // the implicit return statement is never executed.
}
```

This becomes impossible to determine in the presence of loops.

Option 4: All finite static paths through a function definition must end with a return statement. A static path is a path that could potentially be taken if each branch in the code could be controlled independently.

RESOLUTION: Option 1: The function returns an undefined value.

12.35 Precisions of Operations

Should the precision of operations such as add and multiply be defined?

These are not defined by the C++ standard but it is generally assumed that C++ implementations will use IEEE 754 arithmetic. This is not true for GPUs which generally support only a subset of IEEE 754. In addition, many operations such as the transcendental functions are considered too expensive to implement with more than 10 significant bits of precision. Division is commonly implemented by reciprocal and multiplication.

RESOLUTION: Include a table of precisions for operations.

12.36 Compiler Transforms

What compiler transforms should be allowed?

C++ prohibits compiler transforms of expressions that alter the final result. (Note that C++ allows higher precisions than specified to be used but this is a different issue.) GPUs commonly make use of such transforms, for example when mapping sequential code to vector-based architectures.

RESOLUTION: A specified set of transforms (in addition to those permitted by C++) are allowed.

12.37 Expansion of Function-like Macros in the Preprocessor

When expanding macros, each macro can only be applied once to the original token or any token generated from that token. To implement this, the expansion of function-like macros requires a list of applied macros for each token to be maintained. This is a large overhead.

RESOLUTION: Follow the C++ specification.

What should the behaviour be if a directive is encountered during expansion of function-like macros?

This is currently specified as undefined in C++ although several compilers implement the expected behavior.

RESOLUTION: Leave as undefined behavior.

12.38 Should Extension Macros be Globally Defined?

For each extension there is an associated macro that the shader can use to determine if an extension is available on a given implementation. Should this macro be defined globally or should it be defined when the extension is (successfully) enabled?

Both alternatives are usable since attempting to enable an unimplemented extension only results in a warning.

Option 1: Globally defined

```
#ifdef GL_OES_<extension-name>
  #extension GL_OES_<extension-name> : enable
  ...
#endif
```

Option 2: Defined as part of #extension

```
#extension GL_OES_<extension-name> : enable // warning if not available
#ifdef GL_OES_<extension-name>
  ...
#endif
```

RESOLUTION: The macros are defined globally. There should be a warning-free path for all legal cases.

12.39 Minimum Requirements

GLSL ES 1.00 specified a set of minimum requirements that effectively made parts of the specification optional. The purpose was to enable low cost implementations while allowing higher performance devices to expose features without recourse to extensions. That flexibility came at the cost of portability. Should the minimum requirements section be included as part of GLSL ES 3.00?

RESOLUTION: No, except for the section on counting of varyings.

12.40 Packing Functions

12.41 Boolean logical vector operations

The logical binary operators and (&&), or (||), and exclusive or (^^) operate only on two boolean expressions and result in a boolean expression. Should they be extended to operate on boolean vectors?

The 2nd operand is conditionally evaluated for these operators.

```
bvec4 f();
bvec4 g();

f() && g(); // g() gets 'run' for some components but not others.
            // This isn't well defined.
```

RESOLUTION: No, these should not be part of the language.

12.42 Range Checking of literals

Should an error be generated if a literal integer is outside the range of a 32-bit integer?

This can be easily checked by the compiler. However, there is a complication because the literal does not include the minus sign for negative constants. Signed integers can be distinguished from unsigned integers by the 'u' suffix but the value 0x8000000 is only valid if preceded by a unary minus.

Option: Check only that the numeric part of a literal integer (signed or unsigned) is representable by 32 bits.

Option: Include any preceding unary minus and check that the literal is within the range of a signed or unsigned integer as appropriate.

Option: Extend the checking to any constant integral expression.

RESOLUTION: It is an error to have a literal unsigned integer outside the range of a 32-bit integer.

Should this apply to floating-point numbers?

The GLSL spec allows an arbitrary number of digits before the decimal point. It therefore possible for a float literal to have an arbitrarily large number of characters but still be representable e.g.

```
1<1 million zeros>.0e-1000000
```

1. Parsing constraints. Should the number of characters in each field be limited in some way?

    1. Should the mantissa be limited to e.g. 16 characters?

    2. Should the unsigned part of the mantissa be required to fit into a 32 bit integer?

2. Range checks.

    1. If the value is larger than 3.40282347e38, should it be required to return INF? Or return an error?

RESOLUTION: No limit on the number of characters in the mantissa or exponent in a float literal.

RESOLUTION: Values larger than representable in a float 32 must return INF (+ or - as appropriate). Values with a magnitude too small to be representable in a float 32 must return zero.

12.43 Sequence operator and constant expressions

Should the following construct be allowed?

```
float a[2,3];
```

The expression within the brackets uses the sequence operator (',') and returns the integer 3 so the construct is declaring a single-dimensional array of size 3. In some languages, the construct declares a two-dimensional array. It would be preferable to make this construct illegal to avoid confusion.

One possibility is to change the definition of the sequence operator so that it does not return a constant-expression and hence cannot be used to declare an array size.

RESOLUTION: The result of a sequence operator is not a constant-expression.

12.44 Version Directive

The version directive in GLSL ES 1.00 has been found to be unsuitable in cases where certain features of the language specification are changed. The existing mechanism relies on a pre-processor directive but, following the order of operations specified by the 'phases of translation' section in the C++ specification, it is difficult or perhaps impossible to change features of the language that are processed before such directives are invoked. Such features include the introduction of the line-continuation character ('\') and the extension of the character set.

There are several options for an improved version mechanism. All specify the version in the first line of the shader and require that the version directive is followed by a newline.

Option 1: Add a byte sequence to the start of the shader. This would allow any change to be made to the language, including changing the character set. This mechanism is often used in file formats for images.

Option 2: Add a character string sequence to the start of the shader. Define it to appear to be a pre-processor directive e.g.

```
#version 300 es
```

Option 3: As option 2 but allow some flexibility in the format so that extra white-space would still be allowed.

Option 4: As option 2 but use a distinctive non-pre-processor format e.g.

```
version-300-es
```

Option 5: As option 4 but include the characters 'glsl' to aid identification e.g.

```
glsl-version-300-es
```

RESOLUTION: Option 3. The version directive is a string, present as the only non-white-space in the first line of the shader. It is very unlikely that the character set will be changed in an incompatible way from UTF-8 in the future. Option 3 is the closest in appearance to the current mechanism.

12.45 Use of Unsigned Integers

Should functions that can only return a positive value e.g. textureSize() and the length() method, return signed or unsigned values?

Option 1: Unsigned integer. This allows for some degree of compile-time checking. For example it would be impossible to accidentally access an array element with a negative index in a typical initialization loop such as:

```
float a[5];
for (unit i=0u; i<a.length (); i++)
  a[i] = 0.0;
```

Option 2: Signed integer. This allows greater flexibility in calculating array indices without the need for type conversions e.g.

```
float a[SIZE];
...
int index = a.length() - 3; // Library code. SIZE may not be known when
                            // this code is written
if (index >= 0)             // would not work with an unsigned integer
  f(a[index]);
```

RESOLUTION: Option 2. The principle is that integers that represent values and hence may form part of arithmetic expressions should always be signed, even if it is known that they will always be positive. Values that represent bit vectors should always be unsigned.

The extra checking made available by the use of unsigned integers for values known to be positive is minimal. It would be preferable to include a *range* mechanism in a future version of the language.

13 Acknowledgments

14 Normative References

1. The OpenGL® ES Graphics System Version 3.00

2. The OpenGL® ES Shading Language Version 1.00

3. The OpenGL® Graphics System: A Specification (Versions 3.3 – 4.2)

4. International Standard ISO/IEC 14882:1998(E). Programming Languages – C++

5. International Standard ISO/IEC 646:1991. Information technology - ISO 7-bit coded character set for information interchange The Unicode Standard Version 6.0 – Core Specification