The OpenGL® ES Shading Language

Language Version: 1.10

Document Revision: ~~11~~12

Draft 2

Editor (versions 1-11): John Kessenich

Editor (version 12): Robert Simpson

Table of Contents

1 Introduction

This document restates version 1.10 of the OpenGL Shading Language, document revision number 59. Overview information not relevant to the language specification has been removed from sections 1 and 2. Also, the original issues and acknowledgments sections have been removed. Tables and formulas have been reformatted, and spelling errors corrected. ~~Otherwise, the content is identical.~~ Change bars and markings show the changes made for the ES version of the language. Additions are underlined, deletions are stricken through.

This specification requires __VERSION__ to be 110, and **#version** to accept 110.

1.1 Acknowledgments

This specification contains many contributions from discussions with members of the Khronos OpenGL ES group, including Robert Simpson of Bitboys, John Kessenich, Barthold Lichtenbelt, and Nick Murphy of 3Dlabs, Bill Marshall, John Jarvis, and John Boal of Alt Software, Ed Plowman of ARM, Aaftab Munshi and Chris Grimm of ATI, Eisaku Ohbuchi, Eric Fausett, Yoshihiko Kuwahara, Keisuke Kirii, Yukitaka Takemuta, and Max Kazakov of DMP, Jacob ~~Strom~~Ström of Ericsson, Young Seok Kim of ETRI, Borgar Ljosland, Edvard Sørgård, Mario Blazevic, Justin Radeka, Jorn Nystaa, Remi Pedersen, and Frode Heggelund of Falanx, Brian Murray of Freescale, Tero ~~Sarkinnen~~Sarkkinen and Timo Suoranta of Futuremark, Mark Callow and Stanley Kao of Hi, Petri Kero, Ville Miettinen, and Jasin Bushnaief of Hybrid, Graham Connor, John Howson, Ben Bowman, James McCarthy, and Nicolas Thibieroz of Imagination, Yong Moo Kim and Woo Sedo Kim of LG Electronics, Hiroyasu Negishi and Yoshiyuki Kato of Mitsubishi, Jani Vaarala, Kari Pulli, Tero Pihlajakoski, Jarkko Kemppainen, and Joonas Itäranta of Nokia, Tom McReynolds, Gary King, and Neil Trevett of Nvidia, Andy Methley, Akira Uesaki, and Katzutaka Nishio of Panasonic, Aleksandra Krstic of Qualcomm, Jitaek Lim and Kee Chang Lee of Samsung, Jon Leech and Thomas Tannert of SGI, Steve Lee of SIS, Remi Arnaud, Axel Mamode, and Robin Green of Sony, Dan Rice of Sun, Lane Roberts and Stefan von Cavallar of Symbian, Phil Huxley of Tao, Tom Olson of Texas Instruments, and Hans-Martin Will of Vincent.

1.2 Change History

Changes from Revision 11 of the OpenGL ES Shading Language specification

- Specify no white space in floating point constants.
- Specify how struct constructors use the function name space.
- Prohibit main() function with other signatures.
- Clarify how functions are hidden by other functions.
- Specify that expressions where the precision cannot be defined should be evaluated at the default precision
- Specify the rules for invariance within a shader.

Changes from Revision 10 of the OpenGL ES Shading Language specification:

- The extensions' macros are defined to a value of '1', not just defined. This is to conform to the correct convention.

Changes from Revision 9 of the OpenGL ES Shading Language specification:

- Added formal extension for noise functions.
- Added formal extension for derivative functions.
- Made 3D textures available only if the 3D texture extension is enabled. This is part of an API extension.

Changes from Revision 8 of the OpenGL ES Shading Language specification:

- Added the grammar at the end.
- Correct multiple qualifier order, to match existing parameter qualification order.
- Refined **invariant** declarations: takes a list, is globally scoped, and declared before use.
- Make spec. references refer to the 2.0 OpenGL spec. instead of version 1.4.
- Removed gl_MaxTextureUnits, as it is for fixed function only.
- Removed comment about point sprites being disabled.
- Reserved 'superp' for possible future super precision qualifier.
- Gave specific precision qualifiers to the built-in variables in section 7.
- Added a note in the built-in functions (chapter 8) about precision qualification for parameters and return values.

Changes from Revision 7 of the OpenGL ES Shading Language specification:

- Added actual ranges and precisions.
- Stated what happens on floating point overflow.
- Change intermediate results precision to be based on operands' precision when possible, and not to include the l-values' precision.

- Add the macro GL_ES to test for compilation for an ES system.
- Allow out of bounds array access behavior to be platform dependent.

Changes from Revision 6 of the OpenGL ES Shading Language specification:

- Added precision qualifiers **highp**, **mediump**, and **lowp** for floating point and integer types.
- Added **invariant** qualifier to say an output value is to be invariant. Remove the specific mechanism itransform().
- Grammar was deleted, to be replaced later with correct ES grammar.

Changes from Revision 5 of the OpenGL ES Shading Language specification:

- Fixed a lot of typos and English-level clarifications. Same typos were fixed in Revision 59 of the OpenGL Shading Language specification to form Revision 60. These shared non-functional changes are not identified with change bars or other markings.

Changes from Revision 59 of the OpenGL Shading Language specification:

- Most OpenGL state uniform variables are removed.
- All OpenGL state attribute variables are removed.
- All OpenGL state varying variables are removed.
- The output variables gl_ClipVertex and gl_FragDepth are removed.
- Point sprites are supported with the added built-in gl_PointCoord varying variable.
- The minimum maximum vertex attributes is changed from 16 to 8, the minimum maximum vertex-uniform-components is changed from 512 to 384.
- Removed 1D and shadow textures.
- Proposed precision hints and minimum precisions are specified.
- Generic itransform() is added to replace the removed fix-functionality ftransform().
- dFdx(), dFdy(), and fwidth() are made optional.
- noise() is made optional.
- Other minor language fixes/simplifications. Static recursion is disallowed (dynamic recursion was already disallowed). Error messages can be skipped. Behavior for writing outside an array is limited. c**lamp()** and **smoothstep()** domain descriptions are improved.

1.3 Overview

This document describes *The OpenGL ES Shading Language.*

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~~ entry points used to manipulate and communicate with programs and shaders are defined in a separate specification.

1.4 Error Handling

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

1.5 Typographical Conventions

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

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 the programmable processors contained in the OpenGL processing pipeline.

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

Any OpenGL state used by the shader is automatically tracked and made available to shaders. This automatic state tracking mechanism allows the application to use ~~existing~~ OpenGL state commands for state management and have the current values of such state automatically available for use in a shader.

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*.

A vertex shader operates on one vertex at a time. The vertex processor does not replace graphics operations that require knowledge of several vertices at a time. Vertex shaders must compute the homogeneous position of the incoming vertex.

2.2 Fragment Processor

The *fragment processor* is a programmable unit that operates on fragment values and their associated data. Compilation units written in the OpenGL ES Shading Language to run on this processor are called *fragment* shaders.

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

3 Basics

3.1 Character Set

The source character set used for the OpenGL ES shading languages is a subset of ASCII. It includes the following characters:

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

The numbers **0-9**.

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

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

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

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

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

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

There is no end-of-file character. The end of a source string is indicated to the compiler by a length, not a character.

3.2 Source Strings

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

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

For this version of the OpenGL ES Shading Language, only one vertex and one fragment shader can be linked together. The architecture of the system and this specification are designed to link together multiple vertex and multiple fragment shaders, but this will not be supported until a future version of the specification.

3.3 Preprocessor

There is a preprocessor that processes the source strings before they are compiled.

The complete list of preprocessor directives is as follows.

```
#
#define
#undef

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

#error
#pragma

#extension
#version

#line
```

The following operators are also available

```
defined
```

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

The number sign (#) on a line by itself is ignored. Any directive not listed above will cause a diagnostic message and make the implementation treat the shader as ill-formed.

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

The following predefined macros are available

```
__LINE__
__FILE__
__VERSION__
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 110.

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 see whether a shader is running on 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.

#if, #ifdef, #ifndef, #else, #elif, and **#endif** are defined to operate as is standard for C++ preprocessors. Expressions following **#if** and **#elif** are restricted to expressions operating on literal integer constants, plus identifiers consumed by the **defined** operator. Character constants are not supported. The operators available are as follows.

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

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

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

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

The semantics of applying operators to integer literals in the preprocessor match those standard in the C++ preprocessor. ~~Shading Language.ES , not those in the OpenGL~~

Preprocessor expressions will be evaluated according to the behavior of the host processor, not the processor targeted by the shader.

#error will cause the implementation to put a diagnostic message into the shader's information log (see the API in ~~external~~ the platform documentation for how to access a shader's information log). The message will be the tokens following the **#error** directive, up to the first new-line. The implementation must then consider the shader to be ill-formed.

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

```
#pragma STDGL
```

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

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

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

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

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

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

```
#version number
```

where *number* must be 110 for this specification's version of the language (following the same convention as *__VERSION__* above), in which case the directive will be accepted with no errors or warnings. Any *number* less than 110 will cause an error to be generated. Any *number* greater than the latest version of the language a compiler supports will also cause an error to be generated. Version 110 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 110. Compilers for subsequent versions of this language are guaranteed, on seeing the "**#version** 110" directive in a shader, to either support version 110, or to issue an error that they do not support it.

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

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

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

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

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

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

The initial state of the compiler is as if the directive

```
#extension all : disable
```

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

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

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

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

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

where *line* and *source-string-number* are constant integer expressions. After processing this directive (including its new-line), the implementation will behave as if it is compiling at line number *line+1* and source string number *source-string-number*. Subsequent source strings will be numbered sequentially, until another **#line** directive overrides that numbering.

3.4 Comments

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

3.5 Tokens

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

token:
- *keyword*
- *identifier*
- *integer-constant*
- *floating-constant*
- *operator*

3.6 Keywords

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

attribute const uniform varying

break continue do for while

if else

in out inout

float int void bool true false

lowp mediump highp precision invariant

discard return

mat2 mat3 mat4

vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4

~~sampler1D~~ sampler2D sampler3D samplerCube

~~sampler1DShadow sampler2DShadow~~

struct

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

asm

class union enum typedef template this packed

goto switch default

inline noinline volatile public static extern external interface flat

long short double half fixed unsigned superp

input output

hvec2 hvec3 hvec4 dvec2 dvec3 dvec4 fvec2 fvec3 fvec4

sampler1D

sampler1DShadow sampler2DShadow

sampler2DRect sampler3DRect sampler2DRectShadow

sizeof cast

namespace using

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

The keyword **sampler3D** is only available if the extension OES_texture_3D is enabled, e.g.

```
#extension GL_OES_texture_3D : enable
```

3.7 Identifiers

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

identifier
nondigit
identifier nondigit
identifier digit

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

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

Identifiers starting with “gl_” are reserved for use by OpenGL, and may not be declared in a shader as either a variable or a function.

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.

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
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 integer vector
ivec3	a three component integer vector
ivec4	a four component integer vector
mat2	a 2×2 floating-point matrix
mat3	a 3×3 floating-point matrix
mat4	a 4×4 floating-point matrix
~~**sampler1D**~~	~~a handle for accessing a 1D texture~~

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
~~**sampler1DShadow**~~	~~a handle for accessing a 1D depth texture with comparison~~
~~**sampler2DShadow**~~	~~a handle for accessing a 2D depth texture with comparison~~

The type **sampler3D** is only available if the extension name string GL_OES_texture_3D is enabled. 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.

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 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 (=) can be any 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

Integers are mainly supported as a programming aid. At the hardware level, real integers would aid efficient implementation of loops and array indices, and referencing texture units. However, there is no requirement that integers in the language map to an integer type in hardware. It is not expected that underlying hardware has full support for a wide range of integer operations. ~~Because of their intended (limited) purpose, integers are limited to 16 bits of precision, plus a sign representation in both the vertex and fragment languages.~~ An OpenGL ES Shading Language implementation may convert integers to floats to operate on them. ~~An implementation is allowed to use more than 16 bits of precision to manipulate integers.~~ Hence, there is no portable wrapping behavior. ~~Shaders that overflow the 16 bits of precision may not be portable.~~

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

```
int i, j = 42;
```

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

integer-constant :
    decimal-constant
    octal-constant
    hexadecimal-constant

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. A leading unary minus sign (-) is interpreted as an arithmetic unary negation, not as part of the constant. There are no letter suffixes.

4.1.4 Floats

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

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

~~As an input value to one of the processing units, a floating-point variable is expected to match the IEEE single precision floating-point definition for precision and dynamic range. It is not required that the precision of internal processing match the IEEE floating-point specification for floating-point operations, but the guidelines for precision established by the OpenGL 1.4 specification must be met. Similarly,~~ ~~t~~Treatment of conditions such as divide by 0 may lead to an unspecified result, but in no case should such a condition lead to the interruption or termination of processing.

Floating-point constants are defined as follows.

floating-constant :
fractional-constant exponent-part$_{opt}$
digit-sequence exponent-part

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

A decimal point (.) is not needed if the exponent part is present.

No white space is allowed between the characters in a floating point constant. This includes the decimal point, 'e' and the sign of the exponent. A leading '-' is interpreted as a unary operator and is not part of the constant.

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, or Booleans. Floating-point vector variables can be used to store a variety of things that are very useful in computer graphics: colors, normals, positions, texture coordinates, texture lookup results and the like. Boolean vectors can be used for component-wise comparisons of numeric vectors. Defining vectors as part of the shading language allows for direct mapping of vector operations on graphics hardware that is capable of doing vector processing. In general, applications will be able to take better advantage of the parallelism in graphics hardware by doing computations on vectors rather than on scalar values. Some examples of vector declaration are:

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

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

4.1.6 Matrices

Matrices are another useful data type in computer graphics, and the OpenGL ES Shading Language defines support for 2×2, 3×3, and 4×4 matrices of floating point numbers. Matrices are read from and written to in column major order. Example matrix declarations:

```
mat2 mat2D;
mat3 optMatrix;
mat4 view, projection;
```

Initialization of matrix values is done with constructors (described in Section 5.4 “Constructors”).

4.1.7 Samplers

Sampler types (e.g. **sampler2D**) are effectively opaque handles to textures. They are used with the built-in texture functions (described in Section 8.7 “Texture Lookup Functions”) to specify which texture to access. They can only be declared as function parameters or uniforms (see Section 4.3.5 “Uniform”). Samplers are not allowed to be operands in expressions nor can they be assigned into. As uniforms, they are initialized with the OpenGL API. As function parameters, only samplers may be passed to samplers of matching type. This enables consistency checking between shader texture accesses and OpenGL texture state before a shader is run.

4.1.8 Structures

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

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

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

```
light lightVar2;
```

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

struct-definition :
qualifiers$_{opt}$ **struct** *name*$_{opt}$ **{** *member-list* **}** *declarators*$_{opt}$ *;*

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

member-declaration :
basic-type declarators;
embedded-struct-definition

embedded-struct-definition:
struct *name*$_{opt}$ **{** *member-list* **}** *declarator;*

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 and types, with the same scoping rules.~~ A struct declaration creates a type name and a constructor name. The type name is added to the same name space as other variables and types where it hides all previously defined variables and types with the same name. The constructor name is added to the name space used by functions where it hides any and all previously defined constructors and functions with the same name. Any subsequent variable or function declaration with the same name hides both the type name and the constructor name. A previously hidden function can only be made visible by redeclaring the function. The scoping rules for structs are otherwise the same as for variables. The optional *qualifier*s only apply~~applies~~ to any *declarators*, and ~~is~~ are 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 ~~do~~ not contain any other qualifiers. Bit fields are not supported. ~~Nor do they contain any bit fields.~~ Member types must be either already defined (there are no forward references), or defined in-place by embedding another struct definition. Member declarations cannot contain initializers. Member declarators can contain arrays. Such arrays must have a size specified, and the size must be an integral constant expression that's greater than zero (see Section 4.3.3 “Integral 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; so embedded structures must have a declarator. A name given to an embedded struct is scoped at the same level as the struct it is embedded in.

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

4.1.9 Arrays

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

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

There is no mechanism for initializing arrays at declaration time from within a shader.

Reading from or ~~W~~writing to an array with an index that is less than zero or greater than or equal to the array's size results in undefined behavior. It is platform dependent how bounded this undefined behavior may be. It is possible that it leads to instability of the underlying system or corruption of memory. However, a particular platform may bound the behavior such that this is not the case.

4.2 Scoping

The scope of a variable is determined by where it is declared. If it is declared outside all function definitions, it has global scope, which starts from where it is declared and persists to the end of the shader it is declared in. If it is declared in a **while** test or a **for** statement, then it is scoped to the end of the following sub-statement. Otherwise, if it is declared as a statement within a compound statement, it is scoped to the end of that compound statement. If it is declared as a parameter in a function definition, it is scoped until the end of that function definition. A function body has a scope nested inside the function's definition. The **if** statement's expression does not allow new variables to be declared, hence does not form a new scope.

A variable declared as an empty array can be re-declared as an array of the same base type. Otherwise, within one compilation unit, a variable with the same name cannot be re-declared in the same scope. However, a nested scope can override an outer scope's declaration of a particular variable name. Declarations in a nested scope provide separate storage from the storage associated with an overridden name. There is no way to access the overridden name.

All variables in the same scope share the same name space. Functions names are always identifiable as function names based on context, and they have their own name space.

Shared globals are global variables declared with the same name in independently compiled units (shaders) of the same language (vertex or fragment) that are linked together to make a single program. Shared globals share the same name space, and must be declared with the same type. They will share the same storage. Shared global arrays must have the same base type and the same size. Scalars must have exactly the same type name and type definition. Structures must have the same name, sequence of type names, and type definitions, and field names to be considered the same type. This rule applies recursively for nested or embedded types. All initializers for a shared global must have the same value, or a link error will result.

There are two name spaces. The first is used for variables and types, the second is used for functions and constructors.

Outside of all function definitions, there are two scoping levels. The outermost is reserved for built-in functions. Within that is the region of global scope.

Functions may only be defined within the global scope.

Within each scope, a function, variable or type may be defined at most once. Functions and unsized arrays may be declared multiple times.

Within each scoping level, a name may be used either

A) in a variable declaration or a function declaration or both

or

B) as a struct name

Declaring a struct adds the struct name to the variable/type namespace and a constructor to the function namespace. All previous (i.e. visible) function declarations with that name are hidden.

A function may be declared within a scope. The effect of this is to hide any visible struct declarations and associated constructors with the same name. It also hides any functions with the same name that are declared in an enclosing scope. The function declaration makes visible the function with the signature specified. It does not make visible any other functions with the same name but different signatures.

Redeclaring a function does not have any effect on any previous function declarations within the same scope.

Following the rules above, variables may be declared within each scope. The declaration hides any struct declaration (including constructors) with the same name. Redeclaring variables creates a new instance with its own storage.

The only situation where a hidden declaration can be made visible is where a function declaration is used to expose a previously hidden function declaration.

Function declarations hide other functions that have the same name but different signatures e.g.

```
void f(int);
void f(float);
void g()
{
  void f(int);
  f(1);         // OK.
  f(1.0);       // Error. This is now hidden by f(int).
}
```

4.3 ~~Type~~Storage Qualifiers

Variable declarations may have ~~one or more~~a storage qualifier~~s~~, 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
attribute	linkage between a vertex shader and OpenGL for per-vertex data
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application
varying	linkage between a vertex shader and a fragment shader for interpolated data
~~**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~~

~~Global variables can only use the qualifiers **const**, **attribute**, **uniform**, or **varying**. Only one may be specified.~~

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

Function parameters can only use ~~the **in, out, inout,** or~~ **const** storage qualifier~~s~~. Parameter qualifiers are discussed in more detail 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 to its next run (to communicate between fragments or between vertices) do not exist. This would prevent parallel execution of the same shader on multiple vertices or fragments.

Declarations of globals without a storage qualifier, or with just the **const** qualifier, may include initializers, in which case they will be initialized before the first line of *main()* is executed. Such initializers must have constant type. Global variables without storage qualifiers that are not initialized in their declaration or by the application will not be initialized by OpenGL, but rather will enter *main()* with undefined values.

4.3.1 Default Storage Qualifier~~s~~

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

4.3.2 Const

Named compile-time constants can be declared using the **const** qualifier. Any variables qualified as constant are read-only variables for that shader. Declaring variables as constant allows more descriptive shaders than using hard-wired numerical constants. The **const** qualifier can be used with any of the basic data types. It is an error to write to a **const** variable outside of its declaration, so they must be initialized when declared. For example,

```
const vec3 zAxis = vec3 (0.0, 0.0, 1.0);
```

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

Initializers for **const** declarations must be formed from literal values, other **const** variables (not including function call parameters), or expressions of these.

Constructors may be used in such expressions, but function calls may not.

4.3.3 Integral Constant Expressions

An integral constant expression can be one of

- a literal integer value
- a global or local scalar integer variable qualified as **const**, not including function parameters qualified as **const**
- an expression whose operands are integral constant expressions, including constructors, but excluding function calls.

4.3.4 Attribute

The **attribute** qualifier is used to declare variables that are passed to a vertex shader from OpenGL on a per-vertex basis. It is an error to declare an attribute variable in any type of shader other than a vertex shader. Attribute variables are read-only as far as the vertex shader is concerned. Values for attribute variables are passed to a vertex shader through the OpenGL vertex API or as part of a vertex array. They convey vertex attributes to the vertex shader and are expected to change on every vertex shader run. The attribute qualifier can be used only with the data types **float**, **vec2**, **vec3**, **vec4**, **mat2**, **mat3**, and **mat4**. Attribute variables cannot be declared as arrays or structures.

Example declarations:

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

~~All the standard OpenGL vertex attributes have built-in variable names to allow easy integration between user programs and OpenGL vertex functions. See Section 7 " Built-in Variables" for a list of the built-in attribute names.~~

It is expected that graphics hardware will have a small number of fixed locations for passing vertex attributes. Therefore, the OpenGL ES Shading language defines each non-matrix attribute variable as having space for up to four floating-point values (i.e., a vec4). There is an implementation dependent limit on the number of attribute variables that can be used and if this is exceeded it will cause a link error. (Declared attribute variables that are not used do not count against this limit.) A float attribute counts the same amount against this limit as a vec4, so applications may want to consider packing groups of four unrelated float attributes together into a vec4 to better utilize the capabilities of the underlying hardware. A mat4 attribute will use up the equivalent of 4 vec4 attribute variable locations, a mat3 will use up the equivalent of 3 attribute variable locations, and a mat2 will use up 2 attribute variable locations. How this space is utilized by the matrices is hidden by the implementation through the API and language.

Attribute variables are required to have global scope, and must be declared outside of function bodies, before their first use.

4.3.5 Uniform

The **uniform** qualifier is used to declare global variables whose values are the same across the entire primitive being processed. All **uniform** variables are read-only and are initialized either directly by an application via API commands, or indirectly by OpenGL.

An example declaration is:

```
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.

If multiple shaders are linked together, then they will share a single global uniform name space. Hence, types of uniforms with the same name must match across all shaders that are linked into a single executable.

4.3.6 Varying

Varying variables provide the interface between the vertex shader, the fragment shader, and the fixed functionality between them. The vertex shader will compute values per vertex (such as color, texture coordinates, etc.) and write them to variables declared with the **varying** qualifier. A vertex shader may also read **varying** variables, getting back the same values it has written. Reading a **varying** variable in a vertex shader returns undefined values if it is read before being written.

By definition, varying variables are set per vertex and are interpolated in a perspective-correct manner over the primitive being rendered. If single-sampling, the interpolated value is for the fragment center. If multi-sampling, the interpolated value can be anywhere within the pixel, including the fragment center or one of the fragment samples.

A fragment shader may read from varying variables and the value read will be the interpolated value, as a function of the fragment's position within the primitive. A fragment shader can not write to a varying variable.

The type of varying variables with the same name declared in both the vertex and fragments shaders must match, otherwise the link command will fail. Only those varying variables used (i.e. read) in the fragment shader must be written to by the vertex shader; declaring superfluous varying variables in the vertex shader is permissible.

Varying variables are declared as in the following example:

```
varying vec3 normal;
```

The **varying** qualifier can be used only with the data types **float**, **vec2**, **vec3**, **vec4**, **mat2**, **mat3**, and **mat4**, or arrays of these. Structures cannot be **varying**.

~~If no vertex shader is active, the fixed functionality pipeline of OpenGL will compute values for the built-in varying variables that will be consumed by the fragment shader. Similarly, if no fragment shader is active, the vertex shader is responsible for computing and writing to the varying variables that are needed for OpenGL's fixed functionality fragment pipeline.~~

Varying variables are required to have global scope, and must be declared outside of function bodies, before their first use.

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 range and precision used to store or represent floating point and integer variables depends on the source of the value (varying, uniform, texture look-up, etc.), whether it's a vertex or a fragment shader, and other details in the underlying implementation. Minimum storage requirements are declared through use of *precision qualifiers*. Typically, the precision of operations must preserve the storage precisions of the variables involved. The only exceptions allowed are for a small number of computationally intensive built-in functions, e.g. **atan**(), which may return results at less than the declared precisions.

It is strongly advised that the vertex language provide a floating point range and precision matching that of an IEEE single precision floating point number, or better. It is required that the vertex language provide floating point variables whose range is at least (-2^{62}, 2^{62}) and whose precision is at least one part in 65536. This is stated in more detail in the following tables.

The vertex language must provide an integer precision of at least 16 bits, plus a sign bit.

It is useful, but not required, for the fragment language to provide the same floating point range and precision as is required for the vertex shader. It is required that the fragment language provide floating point variables whose range is at least (-16384, +16384) and whose precision of at least one part in 1024. This is described in more detail in the following tables.

The fragment language must provide an integer precision of at least 10 bits, plus a sign bit.

The actual ranges and precisions provided by an implementation can be queried through the API. See the OpenGL ES 2.0 specification for details on how to do this.

4.5.2 Precision Qualifiers

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

Qualifier	Meaning
highp	Satisfies the minimum requirements for the vertex language described above. Optional in the fragment language.
mediump	Satisfies the minimum requirements above for the fragment language. Its range and precision has to be greater than or the same as provided by **lowp** and less than or the same as provided by **highp.**
lowp	Range and precision that can be less than **mediump,** but still intended to represent all color values for any color channel.

For example:

```
lowp float color;
varying mediump vec2 Coord;
lowp ivec2 foo(lowp mat3);
highp mat4 m;
```

Precision qualifiers declare a minimum range and precision that the underlying implementation must use when storing these variables. Implementations may use greater range and precision than requested, but not less. The amount of increased range and precision used to implement a particular precision qualifier can depend on the variable, the operations involving the variable, and other implementation dependent details.

The required minimum ranges and precisions for precision qualifiers are

Qualifier	Floating Point Range	Floating Point Magnitude Range	Floating Point Precision	Integer Range
highp	$(-2^{62}, 2^{62})$	$(2^{-62}, 2^{62})$	Relative: 2^{-16}	$(-2^{16}, 2^{16})$
mediump	$(-2^{14}, 2^{14})$	$(2^{-14}, 2^{14})$	Relative: 2^{-10}	$(-2^{10}, 2^{10})$
lowp	$(-2, 2)$	$(2^{-8}, 2)$	Absolute: 2^{-8}	$(-2^{8}, 2^{8})$

where **Floating Point Magnitude Range** is the range of magnitudes of non-zero values. For **Floating Point Precision**, relative means the precision for any value measured relative to that value, for all non-zero values. For all precision levels, zero must be represented exactly.

If an implementation cannot provide the declared precision for storage of a variable in a shader, it must result in a compilation or link error.

Integer ranges must be such that they can be accurately represented by the corresponding floating point value of the same precision qualifier. That is, a **highp int** can be represented by a **highp float**, a **mediump int** can be represented by a **mediump float**, and a **lowp int** can be represented by a **lowp float**.

The vertex language requires uses of **lowp**, **mediump**, and **highp** to all compile and link without error. The fragment language requires uses of **lowp** and **mediump** to compile without error. Support for **highp** is optional.

The actual range and precision provided by an implementation can be queried through the API.

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. Some operands (e.g. literal constants) might not have a precision qualifier, in which case 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~~ if an entire expression is composed only of operands with no precision qualifier, and the result is not assigned or passed as an argument, ~~it is possible that the compiler will evaluate the entire expression at compile time with compiler host precision, but, in general, the precision used to evaluate such an expression is undefined.~~ then it is evaluated at the default precision 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 p);
f(3.3);                     // 3.3 will be passed in at highp precision
if (2.0/3.0 > 0.6)          // evaluated with ~~undefined~~ default precision
```

When the result of a floating point operation is larger (smaller) than what the required precision can store, the result can be either the maximum (minimum) value that that precision can represent, or a representation of infinity (negative infinity). It cannot result in, for example, wrapping behavior, or generation of a NaN, or an exception condition. Similarly, if the result is closer to zero than what the resulting precision can store, the result should be zero or a representation of a (correctly signed) infinitesimal value.

Integer overflow behavior is undefined. It is possible that it wraps, or that it does not.

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

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

4.5.3 Default Precision Qualifiers

The **precision** statement

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

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

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

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

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

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

```
precision mediump int;
```

The fragment language has no default precision qualifier for floating point types.

4.5.4 Available Precision Qualifiers

The built-in macro GL_FRAGMENT_PRECISION_HIGH is defined to one on systems supporting **highp** precision in the fragment language

```
#define GL_FRAGMENT_PRECISION_HIGH 1
```

and is not defined on systems not supporting **highp** precision in the fragment language. When defined, this macro is available in both the vertex and fragment languages.

4.6 Variance and the Invariant Qualifier

In this section, *variance* refers to the possibility of getting different values from the same expression in different shaders. For example, say two vertex shaders 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.*~~

In general, such variance between shaders can occur. To prevent variance, variables can be declared to be invariant, either individually or with a global setting.

4.6.1 The Invariant Qualifier

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

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

varying mediump vec3 Color;
invariant Color;         // make existing Color be invariant
```

or as part of a declaration when a variable is declared

```
invariant varying mediump vec3 Color;
```

The invariant qualifier must appear before any storage qualifiers (**varying**) when combined with a declaration. Only variables that are output from a shader can be 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 in two shaders, the following must also be true:

- The output variable is declared as invariant in both shaders.
- The same values must be input to all shader input variables consumed by expressions and flow control contributing to the value assigned to the output variable.
- The texture formats, texel values, and texture filtering are set the same way for any texture function calls contributing to the value of the output variable.

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

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

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

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

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

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

4.6.2 Invariance Within Shaders

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. This behaviour is specifically prohibited.

All values are invariant within an individual shader.

There is no invariance rule for values generated by different expressions, even if those expressions are identical.

Example:

```
precision mediump;
vec4 col;
vec2 m = ...;
vec2 n = ...;
vec2 a = m + n;
vec2 b = m + n; // a and b are not guaranteed to be exactly equal
col = texture2D(tex, a);
...
col = texture2D(tex, a);   // the value of a is unchanged from when it was
                           // created.
```

4.7 Order of Qualification

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

invariant-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. Those marked reserved are illegal.

Precedence	Operator Class	Operators	Associativity
1 (highest)	parenthetical grouping	**( )**	NA
2	array subscript function call and constructor structure field selector, swizzler post fix increment and decrement	**[ ]** **( )** **.** **++ --**	Left to Right
3	prefix increment and decrement unary (tilde is reserved)	**++ --** **+ - ~ !**	Right to Left
4	multiplicative (modulus reserved)	*** / %**	Left to Right
5	additive	**+ -**	Left to Right
6	bit-wise shift (reserved)	**<< >>**	Left to Right
7	relational	**< > <= >=**	Left to Right
8	equality	**== !=**	Left to Right
9	bit-wise and (reserved)	**&**	Left to Right
10	bit-wise exclusive or (reserved)	**^**	Left to Right
11	bit-wise inclusive or (reserved)	**|**	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 (modulus, shift, and bit-wise are reserved)	**=** **+= -=** ***= /=** **%= <<= >>=** **&= ^= |=**	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 Subscripting

Array elements are accessed using the array subscript operator (**[ ]**). This is the only operator that operates on arrays. 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 an integer.

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.

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 basic-type keyword or structure name, to make values of the desired type for use in an initializer or an expression. (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.

There is no fixed list of constructor prototypes. Constructors are not built-in functions. Syntactically, all lexically correct parameter lists are valid. Semantically, the number of parameters must be of sufficient size and correct type to perform the initialization. It is an error to include so many arguments to a constructor 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 an integer value to a float
bool(float)  // converts a float value to a Boolean
bool(int)    // converts an integer value to a Boolean
```

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

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

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 there are non-scalar parameters, and/or multiple scalar parameters, they will be assigned in order, from left to right, to the components of the constructed value. In this case, there must be enough components provided in the parameters to provide an initializer for every component in the constructed value. If more components are provided in the last used argument to a constructor than are needed to initialize the constructed value, the left most components of that argument are used, and the remaining ones are ignored. It is an error to provide extra arguments beyond this last used argument. Matrices will be constructed in column major order. It is an error to construct matrices from other matrices. This is reserved for future use.

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

Some useful vector constructors are as follows:

```
vec3(float)  // initializes each component of with the float
vec4(ivec4)  // makes a vec4 with component-wise conversion

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)
```

To initialize a matrix by specifying vectors, or by all 4, 9, or 16 floats for mat2, mat3 and mat4 respectively. The floats are assigned to elements in column major order.

```
mat2(vec2, vec2);
mat3(vec3, vec3, vec3);
mat4(vec4, vec4, vec4, vec4);

mat2(float, float,
     float, float);

mat3(float, float, float,
     float, float, float,
     float, float, float);

mat4(float, float, float, float,
     float, float, float, float,
     float, float, float, float,
     float, float, float, float);
```

A wide range of other possibilities exist, as long as enough components are present to initialize the matrix. However, construction of a matrix from other matrices is currently reserved for future use.

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 must be in the same order and of the same type as they were declared in the structure.

Structure constructors can be used as initializers or in expressions.

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 a texture, *r* in OpenGL, has been renamed *p* so as to avoid the confusion with *r* (for red) in a color.

Accessing components beyond those declared for the vector type is an error so, for example:

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

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

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

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

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

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

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

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

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

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

```
vec4    pos;
```

pos[2] refers to the third element of pos and is equivalent to *pos.z*. This allows variable indexing into a vector, as well as a generic way of accessing components. Any integer expression can be used as the subscript. The first component is at index zero. Behavior is undefined if the index is greater than or equal to the size of the vector.

5.6 Matrix Components

The components of a matrix can be accessed using array subscripting syntax. Applying a single subscript to a matrix treats the matrix as an array of column vectors, and selects a single column, whose type is a vector of the same size as the matrix. The leftmost column is column 0. A second subscript would then operate on the column 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 (e.g., component [3][3] of a mat3).

5.7 Structures and Fields

As with vector components and swizzling, the fields of a structure are also selected using the period (**.**).

In total, the following operators are allowed to operate on a structure:

structure field selector	**.**
equality	**==** **!=**
assignment	**=**

The equality and assignment operators are only valid if the two operands' types are 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.

5.8 Assignments

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

```
lvalue = expression
```

The assignment operator stores the value of *expression* into *lvalue*. It will compile only if *expression* and *lvalue* have the same type. All desired type-conversions must be specified explicitly via a constructor. L-values must be writable. Variables that are built-in types, entire structures, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, and l-values dereferenced with the array subscript operator (**[ ]**) are all l-values. Other binary or unary expressions, non-dereferenced arrays, 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. Other assignment operators are

- The arithmetic assignments add into (+=), subtract from (-=), multiply into (*=), and divide into (/=). The expression

  ```
  lvalue op= expression
  ```

  is equivalent to

  ```
  lvalue = lvalue op expression
  ```

  and the l-value and expression must satisfy the semantic requirements of both *op* and equals (=).
- The assignments modulus into (**%=**), left shift by (<<=), right shift by (>>=), inclusive or into (|=), and exclusive or into (^=) are reserved for future use.

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, float,** all vector types, and all matrix types.
- Constructors of all types.
- Variable names of all types, except array names not followed by a subscript.
- Subscripted array names.
- Function calls that return values.
- Component field selectors and array subscript results.
- Parenthesized expression. 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 typed expressions (including vectors and matrices). The two operands must be the same type, or one can be a scalar float and the other a float vector or matrix, or one can be a scalar integer and the other an integer vector. Additionally, for multiply (*****), one can be a vector and the other a matrix with the same dimensional size of the vector. These result in the same fundamental type (integer or float) as the expressions they operate on. If one operand is scalar and the other is a vector or matrix, the scalar is applied component-wise to the vector or matrix, resulting in the same type as the vector or matrix. Dividing by zero does not cause an exception but does result in an unspecified value. Multiply (*****) applied to two vectors yields a component-wise multiply. Multiply (*****) applied to two matrices yields a linear algebraic matrix multiply, not a component-wise multiply. Multiply of a matrix and a vector yields a linear algebraic transform. Use the built-in functions **dot, cross, and matrixCompMult** to get, respectively, vector dot product, vector cross product, and matrix component-wise multiplication.
- The operator modulus (**%**) is reserved for future use.
- The arithmetic unary operators negate (-), post- and pre-increment and decrement (**--** and ++) operate on integer or floating-point values (including vectors and matrices). 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 operands' types must match. To do component-wise comparisons on vectors, use the built-in functions **lessThan, lessThanEqual, greaterThan,** and **greaterThanEqual.**
- The equality operators **equal (==**), and not equal (**!=**) operate on all types except arrays. They result in a scalar Boolean. For vectors, matrices, and structures, all components of the operands must be equal for the operands to be considered equal. To get 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 must be the same type, but can be of any type other than an array. The resulting type is the same as the type of the second and third expressions.
- Operators and (**&**), or (|), exclusive or (^), not (~), right-shift (>>), left-shift (<<). These operators are reserved for future use.

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

When the operands are of a different type they must fit into one of the following rules:

- one of the arguments is a float (i.e. a scalar), in which case the result is as if the scalar value was replicated into a vector or matrix before being applied.
- the left argument is a floating-point vector and the right is a matrix with a compatible dimension in which case the * operator will do a row vector matrix multiplication.
- the left argument is a matrix and the right is a floating-point vector with a compatible dimension in which case the * operator will do a column vector matrix multiplication.

5.10 Vector and Matrix Operations

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

```
vec3 v, u;
float f;

v = u + f;
```

will be equivalent to

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

And

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

will be equivalent to

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

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

```
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

```
mat 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 vectors and matrices of size 2 and 4.

All unary operations work component-wise on their operands. For binary arithmetic operations, if the two operands are the same type, then the operation is done component-wise and produces a result that is the same type as the operands. If one operand is a scalar float and the other operand is a vector or matrix, then the operation proceeds as if the scalar value was replicated to form a matching vector or matrix operand.

6 Statements and Structure

The fundamental building blocks of the OpenGL ES Shading Language are:

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

The overall structure of a shader is as follows

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

global-declaration:
function-definition
declaration

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

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

statement-list:
statement
statement-list statement

statement:
compound-statement
simple-statement

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

compound-statement:
{ statement-list }

simple-statement:
declaration-statement
expression-statement
selection-statement
iteration-statement
jump-statement

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

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

Declarations and expressions have already been discussed.

6.1 Function Definitions

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

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

and a function is defined like

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

where *returnType* must be present and include a type. Each of the *typeN* must include a type and can optionally include a parameter qualifier, ~~the qualifier **in**, **out**, **inout**, and/or~~ **const**, and a precision qualifier.

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

Arrays are allowed as arguments, but not as the return type. When arrays are declared as formal parameters, their size must be included. An array is passed to a function by using the array name without any subscripting or brackets, and the size of the array argument passed in must match the size specified in the formal parameter declaration.

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

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

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

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

Functions that return no value must be declared as **void**. Functions that accept no input arguments need not use **void** in the argument list because prototypes (or definitions) are required and therefore there is no ambiguity when an empty argument list "( )" is declared. The idiom "(**void**)" as a parameter list is provided for convenience.

Function names can be overloaded. This allows the same function name to be used for multiple functions, as long as the argument list types differ. If functions' names and argument types match, then their return type and parameter qualifiers must also match. No qualifiers are included when checking if types match, function signature matching is based on parameter type only. Overloading is used heavily in the built-in functions. When overloaded functions (or indeed any functions) are resolved, an exact match for the function's signature is sought. This includes exact match of array size as well. No promotion or demotion of the return type or input argument types is done. All expected combination of inputs and outputs must be defined as separate functions.

For example, the built-in dot product function has the following prototypes:

```
float dot (float x, float y);
float dot (vec2 x, vec2 y);
float dot (vec3 x, vec3 y);
float dot (vec4 x, vec4 y);
```

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

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

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

The function *main* can contain uses of **return**. See Section 6.4 "Jumps" for more details.

The declaration of a function named main with any other signature is an error.

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. At call time, input arguments are evaluated in order, from left to right. However, the order in which output parameters are copied back to the caller is undefined. 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**.

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**.

Only precision qualifiers are ~~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, structure name, *or structure definition*

const-qualifier :
    empty
    const

parameter-qualifier :
    empty
    in
    out
    inout

name :
    empty
    identifier

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

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

Behavior is undefined if recursion is used. Recursion means that the static call graph of the program contains cycles. ~~Recursion means having any function appearing more than once at any one time in the run-time stack of function calls. That is, a function may not call itself either directly or indirectly. Compilers may give diagnostic messages when this is detectable at compile time, but not all such cases can be detected at compile time.~~

6.2 Selection

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

```
if (bool-expression)
    true-statement
```

or

```
if (bool-expression)
    true-statement
else
    false-statement
```

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

6.3 Iteration

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

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

while (condition-expression)
    sub-statement

do
    statement
while (condition-expression)
```

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

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

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

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

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

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

Loops can be nested.

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

6.4 Jumps

These are the jumps:

jump_statement:

continue;

break;

return;

return *expression***;**

discard; // in the fragment shader language only

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

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

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

The **discard** keyword is only allowed within fragment shaders. It can be used within a fragment shader to abandon the operation on the current fragment. This keyword causes the fragment to be discarded and no updates to any buffers will occur. It would typically be used within a conditional statement, for example:

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

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

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

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

7 Built-in Variables

7.1 Vertex Shader Special Variables

Some OpenGL operations still continue to occur in fixed functionality in between the vertex processor and the fragment processor. Other OpenGL operations continue to occur in fixed functionality after the fragment processor. Shaders communicate with the fixed functionality of OpenGL through the use of built-in variables.

The variable *gl_Position* is available only in the vertex language and is intended for writing the homogeneous vertex position. All executions of a well-formed vertex shader must write a value into this variable. It can be written at any time during shader execution. It may also be read back by the shader after being written. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations that operate on primitives after vertex processing has occurred. Compilers may generate a diagnostic message if they detect *gl_Position* is not written, or read before being written, but not all such cases are detectable. Results are undefined if a vertex shader is executed and does not write *gl_Position.*

The variable *gl_PointSize* is available only in the vertex language and is intended for a vertex shader to write the size of the point to be rasterized. It is measured in pixels.

~~The variable *gl_ClipVertex* is available only in the vertex language and provides a place for vertex shaders to write the coordinate to be used with the user clipping planes. The user must ensure the clip vertex and user clipping planes are defined in the same coordinate space. User clip planes work properly only under linear transform. It is undefined what happens under non-linear transform.~~

These built-in vertex shader variables for communicating with fixed functionality are intrinsically declared with the following types:

```
highp   vec4  gl_Position;    // must be written to
mediump float gl_PointSize;   // may be written to
vec4  gl_ClipVertex;  // may be written to
```

If ~~*gl_PointSize* or *gl_ClipVertex*~~ any of these variables are not written to, their values are undefined. ~~Any of these variables~~They can be read back by the shader after writing to them, to retrieve what was written. Reading them before writing them results in undefined behavior. If they are written more than once, it is the last value written that is consumed by the subsequent operations.

These built-in variables have global scope.

7.2 Fragment Shader Special Variables

The output of the fragment shader is processed by the fixed function operations at the back end of the OpenGL pipeline. Fragment shaders output values to the OpenGL pipeline using the built-in variables *gl_FragColor*~~,~~ and *gl_FragData*,~~ *and gl_FragDepth,*~~ unless the **discard** keyword is executed.

These variables may be written to more than once within a fragment shader. If so, the last value assigned is the one used in the subsequent fixed function pipeline. The values written to these variables may be read back after writing them. Reading from these variables before writing to them results in an undefined value. ~~The fixed functionality computed depth for a fragment may be obtained by reading *gl_FragCoord.z*, described below.~~

Writing to *gl_FragColor* specifies the fragment color that will be used by the subsequent fixed functionality pipeline. If subsequent fixed functionality consumes fragment color and an execution of a fragment shader does not write a value to *gl_FragColor* then the fragment color consumed is undefined.

If the frame buffer is configured as a color index buffer then behavior is undefined when using a fragment shader.

~~Writing to *gl_FragDepth* will establish the depth value for the fragment being processed. If depth buffering is enabled, and a shader does not write *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 a shader statically contains a write to *gl_FragDepth*, then it is responsible for always writing it.~~

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

The variable *gl_FragData* is an array. Writing to *gl_FragData[n]* specifies the fragment data that will be used by the subsequent fixed functionality pipeline for data *n*. If subsequent fixed functionality consumes fragment data and an execution of a fragment shader does not write a value to it, then the fragment data consumed is undefined.

If a shader statically assigns a value to *gl_FragColor*, it may not assign a value to any element of *gl_FragData*. If a shader statically writes a value to any element of *gl_FragData*, it may not assign a value to *gl_FragColor*. That is, a shader may assign values to either *gl_FragColor* or *gl_FragData*, but not both.

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

The variable *gl_FragCoord* is available as a read-only variable from within fragment shaders and it holds the window relative coordinates x, y, z, and 1/w values for the fragment. 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 a shader contained no writes to gl_FragDepth~~will be used for the fragment's depth~~. This is useful for invariance if a shader conditionally computes gl_FragDepth but otherwise wants the fixed functionality fragment depth.~~

The fragment shader has access to the read-only 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 the vertex shader.

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

```
mediump vec4  gl_FragCoord;
        bool  gl_FrontFacing;
mediump vec4  gl_FragColor;
mediump vec4  gl_FragData[gl_MaxDrawBuffers];
~~float gl_FragDepth;~~
```

However, they do not behave like variables with no storage qualifier; their behavior is as described above. These built-in variables have global scope.

7.3 Vertex Shader Built-In Attributes

There are no built-in attribute names in OpenGL ES.

~~The following attribute names are built into the OpenGL vertex language and can be used from within a vertex shader to access the current values of attributes declared by OpenGL. All page numbers and notations are references to the OpenGL 1.4 specification.~~

```
~~//~~
~~// Vertex Attributes, p. 19.~~
~~//~~
~~attribute float gl_FogCoord;~~
~~attribute vec4  gl_MultiTexCoord0;~~
~~attribute vec4  gl_MultiTexCoord1;~~
~~attribute vec4  gl_MultiTexCoord2;~~
~~attribute vec4  gl_MultiTexCoord3;~~
~~attribute vec4  gl_MultiTexCoord4;~~
~~attribute vec4  gl_MultiTexCoord5;~~
~~attribute vec4  gl_MultiTexCoord6;~~
~~attribute vec4  gl_MultiTexCoord7;~~
~~attribute vec4  gl_Color;~~
~~attribute vec4  gl_SecondaryColor;~~
~~attribute vec3  gl_Normal;~~
~~attribute vec4  gl_Vertex;~~
```

7.4 Built-In Constants

The following built-in constants are provided to vertex and fragment shaders.

```
//
// Implementation dependent constants.  The example values below
// are the minimum values allowed for these maximums.
//
const int  gl_MaxLights = 8;                    // GL 1.0
const int  gl_MaxClipPlanes = 6;                // GL 1.0
const int  gl_MaxTextureUnits = 2;              // GL 1.3
const int  gl_MaxTextureCoords = 2;             // ARB_fragment_program
const mediump int gl_MaxVertexAttribs = 816;
const mediump int gl_MaxVertexUniformComponents = 384512;
const mediump int gl_MaxVaryingFloats = 32;
const mediump int gl_MaxVertexTextureImageUnits = 0;
const mediump int gl_MaxCombinedTextureImageUnits = 2;
const mediump int gl_MaxTextureImageUnits = 2;
const mediump int gl_MaxFragmentUniformComponents = 64;
const mediump int gl_MaxDrawBuffers = 1;
```

7.5 Built-In Uniform State

As an aid to accessing OpenGL processing state, the following uniform variables are built into the OpenGL ES Shading Language. All page numbers and notations are references to the 1.42.0 specification. If an implementation does not support **highp** precision in the fragment language, and state is listed as **highp**, then that state will only be available as **mediump** in the fragment language.

```
//
// Matrix state. p. 31, 32, 37, 39, 40.
//
uniform mat4  gl_ModelViewMatrix;
uniform mat4  gl_ProjectionMatrix;
uniform mat4  gl_ModelViewProjectionMatrix;
uniform mat4  gl_TextureMatrix[gl_MaxTextureCoords];

//
// Derived matrix state that provides inverse and transposed versions
// of the matrices above.  Poorly conditioned matrices may result
// in unpredictable values in their inverse forms.
//
uniform mat3  gl_NormalMatrix; // transpose of the inverse of the
                               // upper leftmost 3x3 of gl_ModelViewMatrix
```

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

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

uniform mat4  gl_ModelViewMatrixInverseTranspose;
uniform mat4  gl_ProjectionMatrixInverseTranspose;
uniform mat4  gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4  gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];


//
// Normal scaling p. 39.
//
uniform float gl_NormalScale;

//
// Depth range in window coordinates, p. 4533
//
struct gl_DepthRangeParameters {
    highp float near;        // n
    highp float far;         // f
    highp float diff;        // f - n
};
uniform gl_DepthRangeParameters gl_DepthRange;

//
// Clip planes p. 42.
//
uniform vec4  gl_ClipPlane[gl_MaxClipPlanes];

//
// Point Size, p. 66, 67.
//
struct gl_PointParameters {
    float size;
    float sizeMin;
    float sizeMax;
    float fadeThresholdSize;
    float distanceConstantAttenuation;
    float distanceLinearAttenuation;
    float distanceQuadraticAttenuation;
};

uniform gl_PointParameters gl_Point;
```

```
//
// Material State p. 50, 55.
//
struct gl_MaterialParameters {
    vec4  emission;    // Ecm
    vec4  ambient;     // Acm
    vec4  diffuse;     // Dcm
    vec4  specular;    // Scm
    float shininess;   // Srm
};
uniform gl_MaterialParameters  gl_FrontMaterial;
uniform gl_MaterialParameters  gl_BackMaterial;

//
// Light State p 50, 53, 55.
//

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

uniform gl_LightSourceParameters  gl_LightSource[gl_MaxLights];

struct gl_LightModelParameters {
    vec4  ambient;          // Acs
};

uniform gl_LightModelParameters  gl_LightModel;

/*
// Derived state from products of light and material.
//

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

```
uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

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

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


//
// Texture Environment and Generation, p. 152, p. 40-42.
//
uniform vec4  gl_TextureEnvColor[gl_MaxTextureImageUnits];
uniform vec4  gl_EyePlaneS[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneT[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneR[gl_MaxTextureCoords];
uniform vec4  gl_EyePlaneQ[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneS[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneT[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneR[gl_MaxTextureCoords];
uniform vec4  gl_ObjectPlaneQ[gl_MaxTextureCoords];

//
// Fog p. 161
//
struct gl_FogParameters {
    vec4 color;
    float density;
    float start;
    float end;
    float scale;    // Derived:    1.0 / (end - start)
};

uniform gl_FogParameters gl_Fog;
```

7.6 Varying Variables

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

~~The following built-in varying variables are available to write to in a vertex shader. A particular one should be written to if any functionality in a corresponding fragment shader or fixed pipeline uses it or state derived from it. Otherwise, behavior is undefined.~~

```
varying vec4  gl_FrontColor;
varying vec4  gl_BackColor;
varying vec4  gl_FrontSecondaryColor;
varying vec4  gl_BackSecondaryColor;
varying vec4  gl_TexCoord[];  // at most will be gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

~~For *gl_FogFragCoord*, the value written will be used as the "e" value on page 160 of the OpenGL 1.4 Specification by the fixed functionality pipeline. For example, if the z-coordinate of the fragment in eye space is desired as "e", then that's what the vertex shader should write into *gl_FogFragCoord*.~~

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

~~The following varying variables are available to read from in a fragment shader. The *gl_Color* and *gl_SecondaryColor* names are the same names as attributes passed to the vertex shader. However, there is no name confliet, because attributes are visible only in vertex shaders and the following are only visible in a fragment shader.~~

```
varying vec4  gl_Color;
varying vec4  gl_SecondaryColor;
varying vec4  gl_TexCoord[];  // at most will be gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

~~The values in *gl_Color* and *gl_SecondaryColor* will be derived automatically by the system from *gl_FrontColor*, *gl_BackColor*, *gl_FrontSecondaryColor*, and *gl_BackSecondaryColor* based on which face is visible. If fixed functionality is used for vertex processing, then *gl_FogFragCoord* will either be the z-coordinate of the fragment in eye space, or the interpolation of the fog coordinate, as described in section 3.10 of the OpenGL 1.4 Specification. The *gl_TexCoord[]* values are the interpolated *gl_TexCoord[]* values from a vertex shader or the texture coordinates of any fixed pipeline based vertex functionality.~~

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

The following varying variables are available to read from in a fragment shader:

```
varying mediump vec2 gl_PointCoord;
```

The values in *gl_PointCoord* are two-dimensional coordinates indicating where within a point primitive the current fragment is located. They range from 0.0 to 1.0 across the point. This is described in more detail in Section 3.3.1 Basic Point Rasterization of version 2.0 of the OpenGL Specification, where point sprites are discussed. If the current primitive is not a point, then the values read from *gl_PointCoord* are undefined.

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).

User code can replace built-in functions with their own if they choose, by simply re-declaring and defining the same name and argument list.

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. For any specific use of a function, the actual type has to be the same for all arguments and for the return type. Similarly for *mat,* which can be a **mat2, mat3,** or **mat4**.

Precision qualifiers for parameters and return values are not shown. The precision qualification of built-in function formal parameters is irrelevant. A call to a built-in function 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.

8.1 Angle and Trigonometry Functions

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

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

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

8.2 Exponential Functions

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

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

8.3 Common Functions

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

Syntax	Description
genType **abs** (genType *x*)	Returns *x* if *x* >= 0, otherwise it returns –*x*.
genType **sign** (genType *x*)	Returns 1.0 if *x* > 0, 0.0 if *x* = 0, or –1.0 if *x* < 0

Syntax	Description
genType **floor** (genType *x*)	Returns a value equal to the nearest integer that is less than or equal to *x*
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*)
genType **mod** (genType *x*, float *y*)	Modulus. Returns *x* – *y* * **floor** (*x*/*y*)
genType **mod** (genType *x*, genType *y*)	Modulus. Returns *x* – *y* * **floor** (*x*/*y*)
genType **min** (genType *x*, genType *y*) genType **min** (genType *x*, float *y)*	Returns *y* if *y* < *x*, otherwise it returns *x*
genType **max** (genType *x*, genType *y*) genType **max** (genType *x*, float *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*)	Returns **min** (**max** (*x*, *minVal*), *maxVal*) Results are undefined if *minVal* > *maxVal.* ~~Note that colors and depths written by fragment shaders will be clamped by the implementation after the fragment shader runs.~~
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 **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.*

8.4 Geometric Functions

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

Syntax	Description
float **length** (genType *x*)	Returns the length of vector *x*, i.e., $\sqrt{x[0]^2+x[1]^2+...}$
float **distance** (genType *p0*, genType *p1*)	Returns the distance between *p0* and *p1*, i.e. **length** (*p0* – *p1*)
float **dot** (genType *x*, genType *y*)	Returns the dot product of *x* and *y*, i.e., $x[0]\cdot y[0]+x[1]\cdot y[1]+...$
vec3 **cross** (vec3 *x*, vec3 *y*)	Returns the cross product of x and y, i.e. $\begin{bmatrix} x[1]\cdot y[2]-y[1]\cdot x[2] \\ x[2]\cdot y[0]-y[2]\cdot x[0] \\ x[0]\cdot y[1]-y[0]\cdot x[1] \end{bmatrix}$
genType **normalize** (genType *x*)	Returns a vector in the same direction as *x* but with a length of 1.
~~vec4 **ftransform**()~~	~~For vertex shaders only. This function will ensure that the incoming vertex value will be transformed in a way that produces exactly the same result as would be produced by OpenGL's fixed functionality transform. It is intended to be used to compute gl_Position, e.g.,~~ ~~gl_Position = **ftransform**()~~ ~~This function should be used, for example, when an application is rendering the same geometry in separate passes, and one pass uses the fixed functionality path to render and another pass uses programmable shaders.~~
genType **faceforward**(genType *N*, genType *I*, genType *Nref*)	If **dot**(*Nref*, *I*) < 0 return *N*, otherwise return –*N*.
genType **reflect** (genType *I*, genType *N*)	For the incident vector *I* and surface orientation *N*, returns the reflection direction: *I* – 2 * **dot**(*N*, *I*) * *N* *N* must already be normalized in order to achieve the

Syntax	Description
	desired result.

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

8.5 Matrix Functions

Syntax	Description
mat **matrixCompMult** (mat *x*, mat *y*)	Multiply matrix *x* by matrix *y* component-wise, i.e., result[i][j] is the scalar product of *x*[i][j] and *y*[i][j]. Note: to get linear algebraic matrix multiplication, use the multiply operator (*****).

8.6 Vector Relational Functions

Relational and equality operators (**<, <=, >, >=, ==, !=**) are defined (or reserved) 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**, 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)	Returns the component-wise compare of *x* < *y*.
bvec **lessThanEqual**(vec x, vec y) bvec **lessThanEqual**(ivec x, ivec y)	Returns the component-wise compare of *x* <= *y*.

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

8.7 Texture Lookup Functions

Texture lookup functions are available to both vertex and fragment shaders. However, level of detail is not computed by fixed functionality for vertex shaders, so there are some differences in operation between vertex and fragment texture lookups. The functions in the table below provide access to textures through samplers, as set up through the OpenGL API. Texture properties such as size, pixel format, number of dimensions, filtering method, number of mip-map levels, depth comparison, and so on are also defined by OpenGL API calls. Such properties are taken into account as the texture is accessed via the built-in functions defined below.

~~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 calculated level of detail prior to performing the texture access operation. If the *bias* parameter is not provided, then the implementation automatically selects level of detail: 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.

The built-ins suffixed with "**Lod**" are allowed only in a vertex shader. For the "**Lod**" functions, *lod* is directly used as the level of detail.

Syntax	Description
~~vec4 **texture1D** (sampler1D *sampler*, float *coord* [, float *bias*])~~ ~~vec4 **texture1DProj** (sampler1D *sampler*, vec2 *coord* [, float *bias*])~~ ~~vec4 **texture1DProj** (sampler1D *sampler*, vec4 *coord* [, float *bias*])~~ ~~vec4 **texture1DLod** (sampler1D *sampler*, float *coord*, float *lod*)~~ ~~vec4 **texture1DProjLod** (sampler1D *sampler*, vec2 *coord*, float *lod*)~~ ~~vec4 **texture1DProjLod** (sampler1D *sampler*, vec4 *coord*, float *lod*)~~	~~Use the texture coordinate *coord* to do a texture lookup in the 1D texture currently bound to *sampler*. For the projective ("**Proj**") versions, the texture coordinate *coord.s* is divided by the last component of *coord*.~~
vec4 **texture2D** (sampler2D *sampler*, vec2 *coord* [, float *bias*]) vec4 **texture2DProj** (sampler2D *sampler*, vec3 *coord* [, float *bias*]) vec4 **texture2DProj** (sampler2D *sampler*, vec4 *coord* [, float *bias*]) vec4 **texture2DLod** (sampler2D *sampler*, vec2 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*,	Use the texture coordinate *coord* to do a texture lookup in the 2D texture currently bound to *sampler*. For the projective ("**Proj**") versions, the texture coordinate (*coord.s, coord.t*) is divided by the last component of *coord*. The third component of coord is ignored for the vec4 coord variant.

Syntax	Description
vec3 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*, vec4 *coord*, float *lod*)	
vec4 **texture3D** (sampler3D *sampler*, vec3 *coord* [, float *bias*]) vec4 **texture3DProj** (sampler3D *sampler*, vec4 *coord* [, float *bias*]) vec4 **texture3DLod** (sampler3D *sampler*, vec3 *coord*, float *lod*) vec4 **texture3DProjLod** (sampler3D *sampler*, vec4 *coord*, float *lod*)	Use the texture coordinate *coord* to do a texture lookup in the 3D texture currently bound to *sampler*. For the projective ("**Proj**") versions, the texture coordinate is divided by *coord.q*. These functions are only available if the extension name string GL_OES_texture_3D is enabled.
vec4 **textureCube** (samplerCube *sampler*, vec3 *coord* [, float *bias*]) vec4 **textureCubeLod** (samplerCube *sampler*, vec3 *coord*, float *lod*)	Use the texture coordinate *coord* to do a texture lookup in the cube map texture currently bound to *sampler*. The direction of *coord* is used to select which face to do a 2-dimensional texture lookup in, as described in section 3.8.6 in version ~~1.4~~2.0 of the OpenGL specification.
~~*vec4* **shadow1D** *(sampler1DShadow sampler, vec3 coord [, float bias])*~~ ~~*vec4* **shadow2D** *(sampler2DShadow sampler, vec3 coord [, float bias])*~~ ~~*vec4* **shadow1DProj** *(sampler1DShadow sampler, vec4 coord [, float bias])*~~ ~~*vec4* **shadow2DProj** *(sampler2DShadow sampler, vec4 coord [, float bias])*~~ ~~*vec4* **shadow1DLod** *(sampler1DShadow sampler, vec3 coord, float lod)*~~ ~~*vec4* **shadow2DLod** *(sampler2DShadow sampler, vec3 coord, float lod)*~~ ~~*vec4* **shadow1DProjLod***(sampler1DShadow sampler, vec4 coord, float lod)*~~ ~~*vec4* **shadow2DProjLod***(sampler2DShadow sampler, vec4 coord, float lod)*~~	~~Use texture coordinate *coord* to do a depth comparison lookup on the depth texture bound to *sampler*, as described in section 3.8.14 of version 1.4 of the OpenGL specification. The 3rd component of *coord* (*coord.p*) is used as the R value. The texture bound to *sampler* must be a depth texture, or results are undefined. For the projective ("**Proj**") version of each built-in, the texture coordinate is divide by *coord.q*, giving a depth value R of *coord.p/coord.q*. The second component of *coord* is ignored for the "**1D**" variants.~~

8.8 Fragment Processing Functions

Fragment processing functions are only available in ~~shaders intended for use on the fragment processor~~the fragment language. The built-in derivative functions **dFdx**, **dFdy**, and **fwidth** are optional, and must be enabled by

```
#extension GL_OES_standard_derivatives : enable
```

before being used. Derivatives may be computationally expensive and/or numerically unstable. Therefore, an OpenGL implementation may approximate the true derivatives by using a fast but not entirely accurate derivative computation.

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

Forward differencing:

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

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

Backward differencing:

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

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

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

dFdy is approximated similarly, with y replacing x.

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

1. The method may use piecewise linear approximations. Such linear approximations imply that higher order derivatives, **dFdx**(**dFdx**(x)) and above, are undefined.
2. The method may assume that the function evaluated is continuous. Therefore derivatives within the body of a non-uniform conditional are undefined.
3. The method may differ per fragment, subject to the constraint that the method may vary by window coordinates, not screen coordinates. The invariance requirement described in section 3.1 of the OpenGL ~~1.4~~2.0 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.

In some implementations, varying degrees of derivative accuracy may be obtained by providing GL hints (section 5.6 of the OpenGL ~~1.4~~2.0 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*));

8.9 Noise Functions

The built-in noise functions **noise1, noise2, noise3,** and **noise4** are optional, and must be enabled by

```
#extension GL_OES_standard_noise : enable
```

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

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

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

9 Shading Language Grammar

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

```
ATTRIBUTE CONST BOOL FLOAT INT
BREAK CONTINUE DO ELSE FOR IF DISCARD RETURN
BVEC2 BVEC3 BVEC4 IVEC2 IVEC3 IVEC4 VEC2 VEC3 VEC4
MAT2 MAT3 MAT4 IN OUT INOUT UNIFORM VARYING
~~SAMPLER1D~~ SAMPLER2D SAMPLER3D SAMPLERCUBE ~~SAMPLER1DSHADOW SAMPLER2DSHADOW~~
STRUCT VOID WHILE

IDENTIFIER TYPE_NAME FLOATCONSTANT INTCONSTANT 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
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_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

function_identifier:
constructor_identifier
IDENTIFIER

// Grammar Note: Constructors look like functions, but lexical analysis recognized most of them as // keywords.

constructor_identifier:
FLOAT
INT
BOOL
VEC2
VEC3
VEC4

BVEC2
BVEC3
BVEC4
IVEC2
IVEC3
IVEC4
MAT2
MAT3
MAT4
TYPE_NAME

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 // reserved

// 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 // reserved

additive_expression:
multiplicative_expression
additive_expression PLUS multiplicative_expression
additive_expression DASH multiplicative_expression

shift_expression:

additive_expression
shift_expression LEFT_OP additive_expression // reserved
shift_expression RIGHT_OP additive_expression // reserved

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 // reserved

exclusive_or_expression:
and_expression
exclusive_or_expression CARET and_expression // reserved

inclusive_or_expression:
exclusive_or_expression
inclusive_or_expression VERTICAL_BAR exclusive_or_expression // reserved

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 ~~conditional~~assignment_expression

assignment_expression:
conditional_expression
unary_expression assignment_operator assignment_expression

assignment_operator:
EQUAL
MUL_ASSIGN
DIV_ASSIGN
MOD_ASSIGN // reserved
ADD_ASSIGN
SUB_ASSIGN
LEFT_ASSIGN // reserved
RIGHT_ASSIGN // reserved
AND_ASSIGN // reserved
XOR_ASSIGN // reserved
OR_ASSIGN // reserved

expression:
assignment_expression
expression COMMA assignment_expression

constant_expression:
conditional_expression

declaration:
function_prototype SEMICOLON
init_declarator_list SEMICOLON
PRECISION precision_qualifier type_specifier_no_prec SEMICOLON

function_prototype:
function_declarator RIGHT_PAREN

function_declarator:
function_header
function_header_with_parameters

function_header_with_parameters:
function_header parameter_declaration
function_header_with_parameters COMMA parameter_declaration

function_header:
fully_specified_type IDENTIFIER LEFT_PAREN

parameter_declarator:
type_specifier IDENTIFIER
type_specifier IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

parameter_declaration:
type_qualifier parameter_qualifier parameter_declarator
parameter_qualifier parameter_declarator
type_qualifier parameter_qualifier parameter_type_specifier
parameter_qualifier parameter_type_specifier

parameter_qualifier:
/ empty */*
IN
OUT
INOUT

parameter_type_specifier:
type_specifier
type_specifier LEFT_BRACKET constant_expression RIGHT_BRACKET

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

single_declaration:
fully_specified_type
fully_specified_type IDENTIFIER
fully_specified_type IDENTIFIER LEFT_BRACKET RIGHT_BRACKET
fully_specified_type IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

fully_specified_type IDENTIFIER EQUAL initializer
INVARIANT IDENTIFIER // Vertex only.

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

fully_specified_type:
type_specifier
type_qualifier type_specifier

type_qualifier:
CONST
ATTRIBUTE // Vertex only.
VARYING
INVARIANT VARYING
UNIFORM

type_specifier:
type_specifier_no_prec
precision_qualifier type_specifier_no_prec

type_specifier_no_prec:
VOID
FLOAT
INT
BOOL
VEC2
VEC3
VEC4
BVEC2
BVEC3
BVEC4
IVEC2
IVEC3
IVEC4
MAT2
MAT3
MAT4
~~*SAMPLER1D*~~

SAMPLER2D
SAMPLER3D
SAMPLERCUBE
~~*SAMPLER1DSHADOW*~~
~~*SAMPLER2DSHADOW*~~
struct_specifier
TYPE_NAME

precision_qualifier:
HIGH_PRECISION
MEDIUM_PRECISION
LOW_PRECISION

struct_specifier:
STRUCT IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
STRUCT LEFT_BRACE struct_declaration_list RIGHT_BRACE

struct_declaration_list:
struct_declaration
struct_declaration_list struct_declaration

struct_declaration:
type_specifier struct_declarator_list SEMICOLON

struct_declarator_list:
struct_declarator
struct_declarator_list COMMA struct_declarator

struct_declarator:
IDENTIFIER
IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

initializer:
assignment_expression

declaration_statement:
declaration

statement:
compound_statement
simple_statement

// Grammar Note: No labeled statements; 'goto' is not supported.

simple_statement:
declaration_statement
expression_statement
selection_statement
iteration_statement
jump_statement

compound_statement:
LEFT_BRACE RIGHT_BRACE
LEFT_BRACE statement_list RIGHT_BRACE

statement_no_new_scope:
compound_statement_no_new_scope
simple_statement

compound_statement_no_new_scope:
LEFT_BRACE RIGHT_BRACE
LEFT_BRACE statement_list RIGHT_BRACE

statement_list:
statement
statement_list statement

expression_statement:
SEMICOLON
expression SEMICOLON

selection_statement:
IF LEFT_PAREN expression RIGHT_PAREN selection_rest_statement

selection_rest_statement:
statement ELSE statement
statement

// Grammar Note: No 'switch'. Switch statements not supported.

condition:

expression
fully_specified_type IDENTIFIER EQUAL initializer

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

for_init_statement:
expression_statement
declaration_statement

conditionopt:
condition
/ empty */*

for_rest_statement:
conditionopt SEMICOLON
conditionopt SEMICOLON expression

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

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

translation_unit:
external_declaration
translation_unit external_declaration

external_declaration:
function_definition
declaration

function_definition:
function_prototype compound_statement_no_new_scope

10 Appendix A: Standard Extensions

10.1 Standard Noise Language Extension

Name

OES_standard_noise

Name Strings

GL_OES_standard_noise

Contributors

Standard core OpenGL.

Contact

John Kessenich (johnk 'at' 3dlabs.com)

Notice

Status

Complete.

Version

Date: July 7, 2005
Revision: 0.90

Number

N/A

Dependencies

OpenGL ES 2.0 is required.

Overview

The standard noise functions and semantics from OpenGL version 2.0 are optional for OpenGL ES 2.0. When this extension is enabled, it makes available these standard functions with the semantics already documented in the OpenGL ES 2.0 Shading Language Specification.

Using this extension requires it to be enabled, e.g.

```
#extension GL_OES_standard_noise : enable
```

Issues

1) Should we use an exact algorithmic description of noise (e.g. a Perlin algorithm), instead of a functional description? This would aid portability in getting the same results on different implementations.

Yes, this is desirable, but not possible until such a description is freely available to include in the specification.

New Keywords

None.

New Built-in Functions

noise1()
noise2()
noise3()
noise4()

New Macro Definitions

#define GL_OES_standard_noise 1

Additions to Chapter 8:

Already documented, see main specification, section 8.9 Noise Functions, in the OpenGL ES 2.0 Shading Language Specification.

Revision History

7/7/2005 Created.

10.2 Standard Derivatives Extension

Name

OES_standard_derivatives

Name Strings

GL_OES_standard_derivatives

Contributors

Standard core OpenGL.

Contact

John Kessenich (johnk 'at' 3dlabs.com)

Notice

Status

Complete.

Version

Date: July 7, 2005
Revision: 0.90

Number

N/A

Dependencies

OpenGL ES 2.0 is required.

Overview

The standard derivatives functions and semantics from OpenGL version 2.0 are optional for OpenGL ES 2.0. When this extension is enabled, it makes available these standard functions with the semantics already documented in the OpenGL ES 2.0 Shading Language Specification.

Using this extension in a shader requires it to be enabled, e.g.

```
#extension GL_OES_standard_derivatives : enable
```

Issues

None.

New Keywords

None.

New Built-in Functions

dFdx()
dFdy()
fwidth()

New Macro Definitions

#define GL_OES_standard_derivatives 1

Additions to Chapter 8:

Already documented, see main specification, section 8.8 Fragment Processing Functions, in the OpenGL ES 2.0 Shading Language Specification.

Revision History

7/7/2005 Created.