The OpenGL® ES Shading Language

Language Version: 1.00
Document Revision: 17
12 May, 2009
Editor: Robert J. Simpson

(Editor, version 1.00, revisions 1-11: John Kessenich)

Version 1.00 Revision 17 (12 May, 2009)

Table of Contents

1 Introduction....1
1.1 Change History....1
1.2 Overview....6
1.3 Error Handling....6
1.4 Typographical Conventions....7
2 Overview of OpenGL ES Shading....8
2.1 Vertex Processor....8
2.2 Fragment Processor....8
3 Basics....9
3.1 Character Set....9
3.2 Source Strings....9
3.3 Logical Phases of compilation....10
3.4 Preprocessor....11
3.5 Comments....15
3.6 Tokens....16
3.7 Keywords....16
3.8 Identifiers....17
4 Variables and Types....18
4.1 Basic Types....18
4.1.1 Void....19
4.1.2 Booleans....19
4.1.3 Integers....19
4.1.4 Floats....21
4.1.5 Vectors....21
4.1.6 Matrices....22
4.1.7 Samplers....22
4.1.8 Structures....22
4.1.9 Arrays....24
4.2 Scoping....25
4.2.1 Definition of Terms....25
4.2.2 Types of Scope....25
4.2.3 Redeclaring Variables....26
4.2.4 Shared Globals....26
4.2.5 Global Scope....27
4.2.6 Name spaces and Hiding....27
4.2.7 Redeclarations and Redefinitions Within the Same Scope....27

4.3 Storage Qualifiers....29
4.3.1 Default Storage Qualifier....29
4.3.2 Constant Qualifier....29
4.3.3 Attribute....30
4.3.4 Uniform....30
4.3.5 Varying....31
4.4 Parameter Qualifiers....32
4.5 Precision and Precision Qualifiers....32
4.5.1 Range and Precision....32
4.5.2 Precision Qualifiers....33
4.5.3 Default Precision Qualifiers....35
4.5.4 Available Precision Qualifiers....36
4.6 Variance and the Invariant Qualifier....36
4.6.1 The Invariant Qualifier....37
4.6.2 Invariance Within Shaders....38
4.6.3 Invariance of Constant Expressions....39
4.6.4 Invariance and Linkage....39
4.7 Order of Qualification....39
5 Operators and Expressions....40
5.1 Operators....40
5.2 Array Subscripting....41
5.3 Function Calls....41
5.4 Constructors....41
5.4.1 Conversion and Scalar Constructors....41
5.4.2 Vector and Matrix Constructors....42
5.4.3 Structure Constructors....43
5.5 Vector Components....44
5.6 Matrix Components....45
5.7 Structures and Fields....46
5.8 Assignments....46
5.9 Expressions....47
5.10 Constant Expressions....49
5.11 Vector and Matrix Operations....50
5.12 Precisions of operations....51
5.13 Evaluation of expressions....51
6 Statements and Structure....52
6.1 Function Definitions....53

6.1.1 Function Calling Conventions....54
6.2 Selection....56
6.3 Iteration....56
6.4 Jumps....57
7 Built-in Variables....59
7.1 Vertex Shader Special Variables....59
7.2 Fragment Shader Special Variables....60
7.3 Vertex Shader Built-In Attributes....61
7.4 Built-In Constants....61
7.5 Built-In Uniform State....62
8 Built-in Functions....63
8.1 Angle and Trigonometry Functions....64
8.2 Exponential Functions....65
8.3 Common Functions....66
8.4 Geometric Functions....68
8.5 Matrix Functions....69
8.6 Vector Relational Functions....70
8.7 Texture Lookup Functions....71
9 Shading Language Grammar....73
10 Issues....84
10.1 Vertex Shader Precision....84
10.2 Fragment Shader Precision....84
10.3 Precision Qualifiers....84
10.4 Function and Variable Name Spaces....88
10.5 Local Function Declarations and Function Hiding....88
10.6 Overloading main()....88
10.7 Error Reporting....88
10.8 Structure Declarations....89
10.9 Embedded Structure Definitions....89
10.10 Redefining Built-in Functions....90
10.11 Constant Expressions....90
10.12 Varying Linkage....91
10.13 gl_Position....91
10.14 Pre-processor....91
10.15 Phases of Compilation....92
10.16 Maximum Number of Varyings....92
10.17 Unsized Array Declarations....93

10.18 Invariance....94
10.19 Invariance Within a shader....95
10.20 While-loop Declarations....96
10.21 Cross Linking Between Shaders....96
10.22 Visibility of Declarations....96
10.23 Language Version....97
10.24 Samplers....97
10.25 Dynamic Indexing....97
10.26 Maximum Number of Texture Units....98
10.27 On-target Error Reporting....98
10.28 Rounding of Integer Division....98
10.29 Undefined Return Values....98
10.30 Precisions of Operations....99
10.31 Compiler Transforms....100
10.32 Expansion of Function-like Macros in the Preprocessor....100
10.33 Should Extension Macros be Globally Defined?....100
10.34 Increasing the Minimum Requirements....101
11 Errors....103
11.1 Preprocessor Errors....103
11.2 Lexer/Parser Errors....103
11.3 Semantic Errors....103
11.4 Linker
....105
12 Normative References....106
13 Acknowledgements....107
Appendix A: Limitations for ES 2.0....108
1 Overview....108
2 Length of Shader Executable....108
3 Usage of Temporary Variables....108
4 Control Flow....108
5 Indexing of Arrays, Vectors and Matrices....109
6 Texture Accesses....110
7 Counting of Varyings and Uniforms....111
8 Shader Parameters....113

1 Introduction

The OpenGL ES Shading Language (also known as GLSL ES or ESSL) is based on the OpenGL Shading Language (GLSL) version 1.20. This document restates the relevant parts of the GLSL specification and so is self-contained in this respect. However GLSL ES is also based on C++ (see section 12: Normative References) and this reference must be used in conjunction with this document.

1.1 Change History

Changes from Revision 16 of the OpenGL ES Shading Language specification:

- Corrected grammar for statements with scope
- Clarified that scalars cannot be treated as single-component vectors.
- Extended minimum requirements for array indexing.
- Corrected behavior of #line directive.

Changes from Revision 15 of the OpenGL ES Shading Language specification:

- Behavior of logical operators in the preprocessor.
- Precision of arithmetic operations.
- Rules for compiler transforms of arithmetic expressions.
- Extension macros are defined before #extension.
- Support of high precision types in the fragment shader is an option, not an extension.

Changes from Revision 14 of the OpenGL ES Shading Language specification:

- Clarify void can be used to specify an empty actual parameter list.
- Assignments return r-values and not l-values.
- The term *shader* refers to the set of compilation units running on either the vertex or fragment processor.
- Undefined operands in preprocessor expressions do not default to '0'
- Returning from a function with a non-void return type but without executing a return statement causes an undefined value to be returned.
- Clarified scoping of variables declared within if statements.

Changes from Revision 13 of the OpenGL ES Shading Language specification:

- Clarified the order of evaluation of function parameters.
- Restrictions on the use of types and qualifiers also apply to structures that contain them.

- Unsubscripted arrays are not allowed as parameters to constructors.
- Not writing to an **out** parameter in a function leaves that parameter undefined when the call returns.
- Added references section.
- gl_MaxVaryingFloats=32 changed to gl_MaxVaryingVectors=8.
- gl_MaxCombinedTextureImageUnits and gl_MaxTextureImageUnits changed from 2 to 8.
- Clarified that one function prototype is allowed for each function definition.

Changes from Revision 12 of the OpenGL ES shading Language specification:

- Added ES 2.0 minimum requirements section.
- Matrix constructors are allowed to take matrix parameters (unification with desktop GLSL).
- Precision qualifiers allowed on samplers.
- Invariant qualifier allowed on varying inputs to the fragment shader. Use must match use on varying outputs from vertex shader. Clarified how built-in special variables can be qualified.
- Structure names are visible at the end of the *struct_specifier*.
- Clarify uniforms and varyings cannot have initializers.
- Move all extensions to separate extension specification.
- Clarified integer precision.
- Changed gl_MaxVertexUniformComponents to gl_MaxVertexUniformVectors.
- Changed gl_MaxFragmentUniformComponents to gl_MaxFragmentUniformVectors.
- Removed invariance within a shader (global flag means invariant within a shader as well).
- Moved built-ins back to outer level scope.
- Correction: **lowp int** cannot be represented by **lowp float**
- Increased vertex uniforms from 384 to 512 to take into account the inefficiencies of the packing algorithm.
- Changed __VERSION__ to 100.
- The only shared globals in ES are uniforms. The precisions of varyings do not need to match.
- Defined gl_MaxVaryingFloats in terms of a packing algorithm.
- Defined gl_MaxUniformComponents in terms of the packing algorithm.
- Removed unsized array declarations.
- Defined compilation stages. Preprocessor runs after preprocessing tokens are generated.
- Removed embedded structure definitions.
- Prohibit cross linking between vertex and fragment shader/compilation unit.
- **void** cannot be used to declare a variable or structure element.

- Fixed function varyings count as part of the total when they are referenced by the fragment shader.
- Clarification of anonymous structures.
- ES allows only two compilation units, one for the vertex shader, one for the fragment shader.
- Static recursion must be detected by the compiler.
- Constant expressions must be invariant.
- Clarify when the compiler or linker must report errors.
- Unsubscripted arrays are l-value expressions but can only be used as actual parameters or within parentheses.
- Constant expression can contain built-in functions.
- Constant expression can be an element of a vector or matrix.
- Clarify that arrays cannot be declared constant since there is no method to initialize them.
- Removed structure definitions from formal parameters.
- Removed the dual name space.
- Clarify allowing repeated declarations but disallow repeated definitions within the same scope.
- Disallow local function declarations.
- Remove the ability to redefine built-in functions (Simplification. Function not useful for ES).

Changes from Revision 11 of the OpenGL ES Shading Language specification:

- Added list of errors generated by the compiler.
- 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 must 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. **clamp()** and **smoothstep()** domain descriptions are improved.

1.2 Overview

This document describes *The OpenGL ES Shading Language.*

The OpenGL ES pipeline contains a programmable vertex stage and a programmable fragment stage. The remaining stages are referred to as *fixed function* and the application has only limited control over their behavior. The set of compilation units for each programmable stage form a *shader*. OpenGL ES 2.0 only supports a single compilation unit per shader.

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

Implementations of GLSL ES 2.0 are further restricted as described in Appendix A.

1.3 Error Handling

Compilers, in general, accept programs that are ill-formed, due to the impossibility of detecting all ill-formed programs. Portability is only ensured for well-formed programs, which this specification describes. Compilers are encouraged to detect ill-formed programs and issue diagnostic messages, but are not required to do so for all cases. Either the compiler or the linker is required to reject lexically or grammatically incorrect shaders. Some semantic errors must also be detected and reported as indicated in the specification.

Where the specification uses the terms ***required***, ***must***/***must not***, ***does***/***does not***, ***disallowed*** or ***not supported***, the compiler or linker is required to detect and report any violations. Similarly when a condition or situation is an **error**, it must be reported. Where the specification uses the terms ***should***/***should not*** or ***undefined behavior*** there is no such requirement but compilers are encouraged to report possible violations.

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

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

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

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

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

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

1.4 Typographical Conventions

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

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 ES 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 ES state used by the shader is automatically tracked and made available to shaders. This automatic state tracking mechanism allows the application to use OpenGL ES 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. Source code that is compiled and run on this processor forms a *vertex shader*.

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.

2.2 Fragment Processor

The *fragment processor* is a programmable unit that operates on fragment values and their associated data. Source code that is compiled and run on this processor forms a *fragment shader*.

A fragment shader cannot change a fragment's 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 ES state and the OpenGL ES command that caused the fragments to be generated.

3 Basics

3.1 Character Set

The source character set used for the OpenGL ES shading languages is a subset of ASCII (see section 12: "Normative References"). 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.

The line continuation character (\) is not part of the language.

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 compilation unit is an array of strings of characters from the character set. A single compilation unit 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 the strings are concatenated. Vertex and fragment shaders each consist of a single compilation unit. A single vertex shader and a single fragment shader are linked together to form a single program.

Diagnostic messages returned from compilation 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, each shader consists of a single compilation unit. The architecture of the system and this specification are designed to link together multiple compilation units for each shader, but this will not be supported until a future version of the specification.

3.3 Logical Phases of compilation

The compilation process is based on a subset of the c++ standard (see section 12: Normative References). The compilation units for the vertex and fragment processor are processed separately before being linked together in the final stage of compilation. The logical phases of compilation are:

1. Source strings are concatenated.
2. The source string is converted into a sequence of preprocessing tokens. These tokens include preprocessing numbers, identifiers and preprocessing operations. Comments are each replaced by one space character. Line breaks are retained.
3. The preprocessor is run. Directives are executed and macro expansion is performed.
4. Preprocessing tokens are converted into tokens.
5. White space and line breaks are discarded.
6. The syntax is analyzed according to the GLSL ES grammar.
7. The result is checked according to the semantic rules of the language.
8. The vertex and fragment shaders are linked together. Any varyings not used in both the vertex and fragment shaders may be discarded.
9. The binary is generated.

3.4 Preprocessor

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

The complete list of preprocessor directives is as follows.

```
#
#define
#undef

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

#error
#pragma

#extension
#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 for C++, 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 100.

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 for C++ except for the following:

- Expressions following **#if** and **#elif** are restricted to expressions operating on literal integer constants, plus identifiers consumed by the **defined** operator.
- Undefined identifiers not consumed by the **defined** operator do not default to '0'. Use of such identifiers causes an error.
- Character constants are not supported.

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 in the preprocessor match those standard in the C++ preprocessor with the following exceptions:

- The 2nd operand in a logical and ('&&') operation is evaluated if and only if the 1st operand evaluates to non-zero.
- The 2nd operand in a logical or ('||') operation is evaluated if and only if the 1st operand evaluates to zero.

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

Preprocessor expressions will be evaluated at compile time.

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

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

```
#pragma STDGL
```

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

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

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

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

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

Each compilation unit should declare the version of the language it is written to using the #**version** directive:

```
#version number
```

where *number* must be 100 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 100 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 100 of the language does not require compilation units to include this directive, and compilation units that do not include a **#version** directive will be treated as targeting version 100.

The **#version** directive must occur in a compilation unit 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 compilation units 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. The 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 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 compilation unit will have to contain the necessary extension directive.

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

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

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

#line must have, after macro substitution, one of the following 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 the following line has line number *line* and starts with source string number *source-string-number*. Subsequent source strings will be numbered sequentially, until another **#line** directive overrides that numbering.

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

3.5 Comments

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

3.6 Tokens

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

token:
keyword
identifier
integer-constant
floating-constant
operator

3.7 Keywords

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

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

sampler2D samplerCube

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 sampler3D

sampler1DShadow sampler2DShadow

sampler2DRect sampler3DRect sampler2DRectShadow

sizeof cast

namespace using

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

3.8 Identifiers

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

identifier

nondigit

identifier nondigit

identifier digit

nondigit: one of

_ a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

Identifiers starting with “gl_” are reserved for use by OpenGL ES. No user-defined identifiers may begin with “gl_”.

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 or for an empty parameter list
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
sampler2D	a handle for accessing a 2D texture
samplerCube	a handle for accessing a cube mapped texture

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

There are no pointer types.

4.1.1 Void

The **void** type may only be used as a function return type or as an empty formal or actual parameter list.

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. An OpenGL ES Shading Language implementation may convert integers to floats to operate on them. Hence, there is no portable wrapping behavior.

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;
```

Treatment of conditions such as divide by 0 may lead to an unspecified result, but must not 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. A leading unary minus sign (-) 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”). Except for parameters to texture lookup functions, array indexing, structure field selection, and parentheses, samplers are not allowed to be operands in expressions. Samplers cannot be treated as l-values and cannot be used as **out** or **inout** function parameters. These restrictions also apply to any structures that contain sampler types. As uniforms, they are initialized with the OpenGL ES API. As function parameters, only samplers may be passed to samplers of matching type. This enables consistency checking between shader texture accesses and OpenGL ES texture state before a shader is run.

4.1.8 Structures

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

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

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

```
light lightVar2;
```

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

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

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

member-declaration :
basic-type declarators;

where *name* becomes the user-defined type, and can be used to declare variables to be of this new type. The name shares the same name space as other variables, types and functions. All previously visible variables, types, constructors or functions with that name are hidden. The optional *qualifiers* only apply to any *declarators*, and 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 not contain any other qualifiers. Bit fields are not supported. Member types must already be defined (there are no forward references and no embedded definitions). 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 structure declarators (member declaration whose type is a structure but has no declarator) are not supported.

```
struct S
{
  int x;
};

struct T
{
  S;          // Error: anonymous structures are disallowed.
  int y;
};
```

Embedded structure definitions are not supported:

Example 1:

```
struct S
{
  struct T   // error: embedded structure definition is not supported.
  {
    int a;
  } t;
  int b:
};
```

Example 2:

```
struct T
{
  int a;
};
struct S
{
  T t;       // ok.
  int b;
};
```

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

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

4.1.9 Arrays

Variables of the same type can be aggregated into arrays by declaring a name followed by brackets (**[ ]**) enclosing a size. The array size must be an integral constant expression (see Section 4.3.3 “Integral Constant Expressions”) greater than zero. It is illegal to index an array with an integral constant expression greater than or equal to its 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. 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];

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

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

Reading from or writing to an array with a non-constant 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 declaration determines where the declaration is visible. GLSL ES uses a system of statically nested scopes. This allows names to be redefined within a shader.

4.2.1 Definition of Terms

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

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

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

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

4.2.2 Types of Scope

The scope of a name 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 compilation unit 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 *statement-no-new-scope*. 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.

Representing the if construct as:

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

a variable declared in the if-statement is scoped to the end of the if-statement. A variable declared in the else-statement is scoped to the end of the else-statement. This applies both when these statements are simple statements and when they are compound statements. The if-expression does not allow new variables to be declared, hence does not form a new scope.

A variable declaration is visible immediately following the initializer if present, otherwise immediately following the identifier:

E.g.

```
int x = 1;
{
  int x = 2 /* 2nd x visible here  */, y = x;  // y is initialized to 2
}
```

E.g.

```
int x=1;
{
  int x = 2, y = x; // y is initialized to '2'
  int z = z;        // error if z not previously defined.
}
{
  int x = x;        // x is initialized to '1'
}
```

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

E.g.

```
struct S
{
  int x;
  int y;
};

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

A function declaration is visible at the end of the function prototype.

Note that the scoping rules for GLSL ES and C++ are not identical.

4.2.3 Redeclaring Variables

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.

4.2.4 Shared Globals

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

Shared globals must have the same name, storage and precision qualifiers. They must have the same equivalent type according to the following rules:

Shared global arrays must have the same precision, base type and 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.

4.2.5 Global Scope

Outside of all function definitions, there are two levels of scoping. Built-in functions are implicitly defined in the outermost scope. The inner scope is known as the global scope. User defined functions may only be defined within the global scope.

4.2.6 Name spaces and Hiding

Within each scope, there is a single name space. Declaring a variable adds a variable name to the name space. Declaring a function adds a function name to the name space. Declaring a structure adds a structure name and a constructor name to the name space.

All variable and structure declarations hide all declarations with the same name in outer scopes.

There is no mechanism for accessing hidden declarations.

Functions can only be declared within the global scope. Local function declarations (i.e. within a nested scope) are disallowed. Functions (including built-in functions) may not be redefined. User defined functions may overload built-in functions.

With the exception of uniform declarations, vertex and fragment shaders have separate name spaces. Functions and global variables declared in a vertex shader cannot be referenced by a fragment shader and *vice versa*. Uniforms have a single name space. Uniforms declared with the same name must have matching types and precisions.

4.2.7 Redeclarations and Redefinitions Within the Same Scope

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

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

A particular variable, structure or function declaration may occur at most once within a scope with the exception that a single function prototype plus the corresponding function definition are allowed.

The determination of equivalence of two declarations depends on the type of declaration. For functions, the whole function signature must be considered (see section 6.1). For variables (including arrays) and structures only the names must match.

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

Examples of combinations that are allowed:

1.
```
void f(int) {...}
void f(float) {...} // function overloading allowed
```
2.
```
void f(int);
void f(int) {...}   // single definition allowed
```

Examples of combinations that are disallowed:

1.
```
void f(int) {...}
void f(int) {...}   // Error: repeated definition
```
2.
```
void f(int);
void f(int);       // Error: repeated function prototype
```
3.
```
void f(int);
struct f {int x;};  // Error: type 'f' conflicts with function 'f'
```
4.
```
struct f {int x;};
int f;              // Error: conflicts with the type 'f'
```
5.
```
int a[3];
int a[3];           // Error: repeated array definition
```
6.
```
int x;
int x;              // Error: repeated variable definition
```

4.3 Storage Qualifiers

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

Qualifier	Meaning
< none: default >	local read/write memory, or an input parameter to a function
const	a compile-time constant, or a function parameter that is read-only
attribute	linkage between a vertex shader and OpenGL ES for per-vertex data
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL ES, and the application
varying	linkage between a vertex shader and a fragment shader for interpolated data

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

Function parameters can only use **const** storage qualifier. 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 be a constant expression. Global variables without storage qualifiers that are not initialized in their declaration or by the application will not be initialized by OpenGL ES, but rather will enter *main()* with undefined values. Uniforms, attributes and varyings may not have initializers.

4.3.1 Default Storage Qualifier

If no qualifier is present on a global variable, then the variable has no linkage to the application or compilation units 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 Constant Qualifier

Named compile-time constants can be declared using the **const** qualifier. Any variables qualified as constant are read-only variables for that compilation unit. 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 a *constant expression*.

Arrays and structures containing arrays may not be declared constant since they cannot be initialized.

See 5.10 Constant Expressions.

4.3.3 Attribute

The **attribute** qualifier is used to declare variables that are passed to a vertex shader from OpenGL ES 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 ES 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;
```

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.

4.3.4 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 ES.

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 statically used do not count against this limit. *Static use* means the shader contains a reference to the variable after preprocessing. The number of user-defined uniform variables and the number of built-in uniform variables that are statically used within a shader are added together to determine whether available uniform storage has been exceeded.

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

4.3.5 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 fragment shaders must match, otherwise the link command will fail. The precision of varying variables does not need to match. Only those varying variables statically used (i.e. read) in the fragment shader must be declared in the vertex shader; declaring superfluous varying variables in the vertex shader is permissible.

The following table summarizes the rules for matching varyings in the vertex and fragment shaders:

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

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

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**.

Varying variables are required to have global scope.

4.4 Parameter Qualifiers

Parameters can have these qualifiers.

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

Parameter qualifiers are discussed in more detail in 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 is 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. For integer types, all integer values within the specified range must be represented.

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

For high and medium precisions, 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.** However, **lowp int** cannot be represented by a **lowp float**;

The vertex language requires any uses of **lowp**, **mediump** and **highp** to compile and link without error. The fragment language requires any 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.

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

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

For example, consider the statements.

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

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 must be zero or a representation of a (correctly signed) infinitesimal value.

Integer overflow results in an undefined value. 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.

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 **int** or **float** or any of the sampler types, and the *precision-qualifier* can be **lowp**, **mediump**, or **highp**. Any other types or qualifiers will result in an error. If *type* is **float**, the directive applies to non-precision-qualified floating point type (scalar, vector, and matrix) declarations. If *type* is **int**, the directive applies to 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;
precision lowp sampler2D;
precision lowp samplerCube;
```

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

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

The fragment language has no default precision qualifier for floating point types. Hence for **float**, floating point vector and matrix variable declarations, either the declaration must include a precision qualifier or the default float precision must have been previously declared.

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. The **highp** qualifier is an optional feature in the fragment language and is not enabled by **#extension**.

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. To prevent variance, variables can be declared to be *invariant*, either individually or with a global setting.

4.6.1 The Invariant Qualifier

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

```
invariant gl_Position;   // make 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;
```

These are the only situations where the invariant qualifier may be used. The invariant qualifier must appear before any storage qualifiers (**varying**) when combined with a declaration.

Only the following variables may be declared as invariant:

- Built-in special variables output from the vertex shader
- Varying variables output from the vertex shader
- Built-in special variables input to the fragment shader
- Varying variables input to the fragment shader
- Built-in special variables output from the fragment shader.

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

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 compilation unit. 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.

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

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

Example 1:

```
precision mediump;
vec4 col;
vec2 a = ...
...
col = texture2D(tex, a);  // a has a value a1
...
col = texture2D(tex, a);  // a has a value a2 where possibly a1 ≠ a2
```

To enforce invariance in this example use:

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

Example 2:

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

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

4.6.3 Invariance of Constant Expressions

Invariance must be guaranteed for constant expressions. A particular constant expression must evaluate to the same result if it appears again in the same shader or a different shader. This includes the same expression appearing in both a vertex and fragment shader or the same expression appearing in different vertex or fragment shaders.

Constant expressions must evaluate to the same result when all the following are true:

- The inputs to the expression have the same value.
- The operations performed are the same and are applied in the same order.
- The operations are performed at the same precision.

4.6.4 Invariance and Linkage

The invariance of varyings that are declared in both the vertex and fragment shaders must match.
For the built-in special variables, gl_FragCoord can only be declared invariant if and only if gl_Position is declared invariant. Similarly gl_PointCoord can only be declared invariant if and only if gl_PointSize is declared invariant. It is an error to declare gl_FrontFacing as invariant. The invariance of gl_FrontFacing is the same as the invariance of gl_Position.

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 (remainder 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 (remainder, 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;
```

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. Unsubscripted arrays and structures containing arrays are not allowed as parameters to constructors. Arguments are evaluated left to right. 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 a vector is constructed from multiple scalars, vectors, or matrices, or a mixture of these, the vectors' components will be constructed in order from the components of the arguments. The arguments will be consumed left to right, and each argument will have all its components consumed, in order, before any components from the next argument are consumed. Similarly for constructing a matrix from multiple scalars or vectors, or a mixture of these. Matrix components will be constructed and consumed in column major order. In these cases, there must be enough components provided in the arguments to provide an initializer for every component in the constructed value. It is an error to provide extra arguments beyond this last used argument.

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

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

Some useful vector constructors are as follows:

```
vec3(float)  // initializes each component of 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.

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

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

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

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

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

No more than 4 components can be selected.

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

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

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

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

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

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

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

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

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

```
vec4    pos;
```

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

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

5.6 Matrix Components

The components of a matrix can be accessed using array subscripting syntax. Applying a single subscript to a matrix treats the matrix as an array of column vectors, and selects a single column, whose type is a vector of the same size as the 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
```

The behavior when accessing a component outside the bounds of a matrix are the same as those for vectors and arrays. The compiler must generate an error if the index expression is a constant expression but the behavior is undefined for non-constant expression. (e.g., component [3][3] of a mat3 must generate a compile-time error).

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. The assignment and equality operators are not defined for structures that contain arrays or sampler types.

```
Struct S {int x;};
S a;
{
  struct S {int x;};
  S b;
  a = b; // error: type mismatch
}
```

5.8 Assignments

Assignments of values to variable names are performed using the assignment operator (=):

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

The lvalue-expression evaluates to an l-value. The assignment operator stores the value of the rvalue-expression into the l-value and returns an r-value with the type and precision of the lvalue-expression. The lvalue-expression and rvalue-expression must have the same type. All desired type-conversions must be specified explicitly via a constructor. 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, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[ ]**) are all l-values. Other unary, binary and ternary expressions are not l-values. This includes assignments, function names, swizzles with repeated fields and constants.

Array variables are l-values and may be passed to parameters declared as **out** or **inout**. However, they may not be used as the target of an assignment. Similarly, structures containing arrays may be passed to parameters declared as **out** or **inout** but may not be used as the target of an assignment.

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-expression op= expression
  ```

  is equivalent to

  ```
  lvalue = lvalue op expression
  ```

  where *lvalue* is the object returned by *lvalue-expression. lvalue-expression* is evaluated only once.The l-value and expression must satisfy the semantic requirements of both *op* and equals (=).

- The assignments remainder 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.
- Unsubscripted array names. Unsubscripted arrays may only be used as actual parameters in function calls or within parentheses. The only operator defined for arrays is the subscript operator (**[ ]**).
- Function calls that return values.
- Component field selectors and array subscript results.
- Parenthesized expression. Any expression can be parenthesized. Parentheses can be used to group operations. Operations within parentheses are done before operations across parentheses.

- The arithmetic binary operators add (+), subtract (**-**), multiply (*****), and divide (/) operate on integer and floating-point 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 remainder (**%**) 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, structures containing arrays, sampler types and structures containing sampler types 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.

Except where stated otherwise, the order of evaluation of operands in an expression is undefined.

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 scalar (e.g. a float), 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 Constant Expressions

A *constant expression* is one of

- a literal value (e.g., 5 or **true**)
- a global or local variable qualified as **const** excluding function parameters
- an expression formed by an operator on operands that are constant expressions, including getting an element of a constant vector or a constant matrix, or a field of a constant structure
- a constructor whose arguments are all constant expressions
- a built-in function call whose arguments are all constant expressions, with the exception of the texture lookup functions.

The following may not be used in constant expressions:

- User-defined functions
- Uniforms, attributes and varyings.

Array variables cannot be constant expressions since constants must be initialized at the point of declaration and there is no mechanism to initialize arrays.

Constant expressions must be invariant.

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

5.11 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.

5.12 Precisions of operations

The precisions of operations are implementation dependent.

5.13 Evaluation of expressions

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

- Addition and multiplication are assumed to be associative.
- Multiplication may be replaced by repeated addition
- Division may be replaced by reciprocal and multiplication:
- Values may be represented by a higher precision than that specified. Within the constraints of invariance (where applicable), the precision used may vary.
- Integer values may be represented by floating point values. Operations on these values may be performed by the corresponding floating point operation.

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 compilation unit is as follows

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

global-declaration:
function-definition
declaration

That is, a compilation unit 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, **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 a structure if the structure does not contain an array.

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
```

Function declarations must specify a return type. This type may be **void**. If the return type is not **void**, any return statements within the function definition must include a return-expression and the type of the expression must match the return type. If the return type of the function is specified as **void**, any such return statements must not include a return-expression. 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. Function signature matching is based on parameter type only, no qualifiers are used. 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.

The function *main* is used as the entry point to a shader. Both the vertex shader and fragment shader must contain a function named *main*. This function takes no arguments, returns no value, and must be declared as type **void:**

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

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

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.

To control what parameters are copied in and/or out through a function definition or declaration:

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

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

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

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

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

If a function does not write to an **out** parameter, the value of the actual parameter is undefined when the function returns.

If a function is declared with a non-void return type and the function returns without executing a return statement, the value returned is undefined.

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

Structure declarations are not permitted as part of parameter declarations or return type declarations.

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.

Static and dynamic recursion is not allowed. Static recursion is present if the static function call graph of the program contains cycles.

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 (for-init-statement; condition(opt); expression)
    statement-no-new-scope

while (condition)
    statement-no-new-scope

do
    statement
while (expression)
```

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

The **for** loop first evaluates the *for-init-statement*, then the *condition*. If the *condition* evaluates to true, then the body of the loop is executed. An empty *condition* evaluates to true. After the body is executed, a **for** loop will then evaluate the *expression*, and then loop back to evaluate the *condition*, repeating until the *condition* evaluates to false. The loop is then exited, skipping its body and skipping its *expression.* Variables modified by the *expression* maintain their value after the loop is exited, provided they are still in scope. Variables declared in *for-init-statement* or *condition* are only in scope until the end of the *statement-no-new-scope* of the **for** loop.

The **while** loop first evaluates the *condition*. If true, then the body is executed. This is then repeated, until the *condition* evaluates to false, exiting the loop and skipping its body. Variables declared in the *condition* are only in scope until the end of the *statement-no-new-scope* of the while loop.

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

Expressions for *condition* must evaluate to a Boolean.

Both the *condition* and the *for-init-statement* can declare and initialize a variable, except in the **do-while** loop, which cannot declare a variable in its *expression.* The variable’s scope lasts only until the end of the *statement* or *statement-no-new-scope* 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

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

In OpenGL ES, built-in special variables form part of the output from the vertex shader, part of the input to the fragment shader and the output from the fragment shader. Unlike user-defined varying variables, the built-in special variables do not have a strict one-to-one correspondence between the vertex language and the fragment language. Instead, two sets are provided, one for each language.

7.1 Vertex Shader Special 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 should 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. The value of *gl_Position* is 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.

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

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

If any of these variables are not written to, their values are undefined. 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 values being returned. 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 ES pipeline. Fragment shaders output values to the OpenGL ES pipeline using the built-in variables *gl_FragColor* and *gl_FragData,* unless the **discard** keyword is executed.

It is not a requirement for the fragment shader to write to either *gl_FragColor* or *gl_FragData*. There are many algorithms, such as shadow volumes, that include rendering passes where a color value is not written.

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.

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.

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.

(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.)

If a shader executes the **discard** keyword, the fragment is discarded, and the values of *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 *will be used for the fragment's 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 fragment shader has access to the read-only built-in variable *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.

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];
mediump vec2  gl_PointCoord;
```

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.

7.4 Built-In Constants

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

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

const mediump int gl_MaxVertexAttribs = 8;
const mediump int gl_MaxVertexUniformVectors = 128;
const mediump int gl_MaxVaryingVectors = 8;
const mediump int gl_MaxVertexTextureImageUnits = 0;
const mediump int gl_MaxCombinedTextureImageUnits = 8;
const mediump int gl_MaxTextureImageUnits = 8;
const mediump int gl_MaxFragmentUniformVectors = 16;
const mediump int gl_MaxDrawBuffers = 1;
```

7.5 Built-In Uniform State

As an aid to accessing OpenGL ES processing state, the following uniform variables are built into the OpenGL ES Shading Language. All notations are references to the 2.0 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.

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

8 Built-in Functions

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

The built-in functions basically fall into three categories:

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

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

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

User code can overload the built-in functions but cannot redefine them.

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. For the texture functions, the precision of the return type matches the precision of the sampler type.

```
uniform lowp sampler2D sampler;
highp vec2 coord;
...
lowp vec4 col = texture2D (sampler, coord); // texture2D returns lowp
```

The precision qualification of other built-in function formal parameters is irrelevant. A call to these built-in functions 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
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 (modulo). 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.*
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. $x \cdot (1-a) + y \cdot a$

Syntax	Description
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.
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 desired result.

Syntax	Description

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*.
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 ES API. Texture properties such as size, pixel format, number of dimensions, filtering method, number of mip-map levels, depth comparison, and so on are also defined by OpenGL ES API calls. Such properties are taken into account as the texture is accessed via the built-in functions defined below.

Functions containing the bias parameter are available only in the 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 **texture2D** (sampler2D *sampler*, vec2 *coord*) vec4 **texture2D** (sampler2D *sampler*, vec2 *coord*, float *bias*) vec4 **texture2DProj** (sampler2D *sampler*, vec3 *coord*) vec4 **texture2DProj** (sampler2D *sampler*, vec3 *coord*, float *bias*) vec4 **texture2DProj** (sampler2D *sampler*, vec4 *coord*) vec4 **texture2DProj** (sampler2D *sampler*, vec4 *coord*, float *bias*) vec4 **texture2DLod** (sampler2D *sampler*, vec2 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*, vec3 *coord*, float *lod*) vec4 **texture2DProjLod** (sampler2D *sampler*, vec4 *coord*, float *lod*)	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
vec4 **textureCube** (samplerCube *sampler*, vec3 *coord*) 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 2.0 of the OpenGL specification.

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
 SAMPLER2D SAMPLERCUBE
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 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 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 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

SAMPLER2D
SAMPLERCUBE
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_no_new_scope:
compound_statement_with_scope
simple_statement

simple_statement:
declaration_statement
expression_statement
selection_statement
iteration_statement
jump_statement

compound_statement_with_scope:
LEFT_BRACE RIGHT_BRACE
LEFT_BRACE statement_list RIGHT_BRACE

statement_with_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_no_new_scope
statement_list statement_no_new_scope

expression_statement:
SEMICOLON
expression SEMICOLON

selection_statement:
IF LEFT_PAREN expression RIGHT_PAREN selection_rest_statement

selection_rest_statement:
statement_with_scope ELSE statement_with_scope
statement_with_scope

condition:
expression
fully_specified_type IDENTIFIER EQUAL initializer

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

for_init_statement:
expression_statement
declaration_statement

conditionopt:
condition
/ empty */*

for_rest_statement:
conditionopt SEMICOLON
conditionopt SEMICOLON expression

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

translation_unit:
    external_declaration
    translation_unit external_declaration

external_declaration:
    function_definition
    declaration

function_definition:
    function_prototype compound_statement_no_new_scope

10 Issues

10.1 Vertex Shader Precision

Should the minimum precision be (the equivalent of) float 24 or float 32?

RESOLUTION: The minimum precision is float 24. Implementations will be encouraged to use float 32.

10.2 Fragment Shader Precision

Should the minimum precision be (the equivalent of) float 16 or float 24?

Float 24 is required to do most cases of dependent texture lookup.

Float 24 is considered too expensive for low-end hardware.

Even float 16 is expensive when doing simple color blend operations. Availability of a lower precision would be useful.

It is clear this standard must address the needs of different market segments. It is also an advantage to be able to specify when a reduced precision value can be used.

RESOLUTION: There will be 3 precisions available, roughly corresponding to float32, float16 and int10. The highest precision will be optional in the fragment shader.

10.3 Precision Qualifiers

Should the precisions be specified as float16, float24 etc.? This would help portability. It implies different types rather than hints. It will require all implementations to use the same or similar algorithms and reduces the scope for innovation.

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

Do integers have precision qualifiers? It is assumed that integers will be mapped to floating point hardware in many implementations. This would only allow 10-bit integers for **mediump float**.

RESOLUTION: Yes, integers have precision qualifiers. The precision of integers will be specified so that they can be implemented using the corresponding floating point precision.

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

Option: The standard should specify either wrapping or clamping.

RESOLUTION: This should be undefined. It is too costly to implement. The specification should clearly state that any wrapping or clamping behavior cannot be relied upon.

Are precision qualifiers available in the vertex shader?

RESOLUTION: Yes. It could be useful for some implementations and keeps the languages consistent.

Should there be a default precision? It would make sense to specify the default vertex precision as highp as that it what is currently specified. There is no agreement on what the default precision for the fragment side should be.

RESOLUTION: **highp** for the vertex shader, no default precision for the fragment shader.

Should there be a precision equivalent to float32? This is already supported by desktop implementations.

RESOLUTION: Reserve **superp** for possible inclusion in a future version of GLSL ES.

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

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

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

RESOLUTION: **highp** must mean that at least high precision is used and similarly for **mediump** and **lowp** so precision qualifiers are more than just hints. As far as the language is concerned it doesn't affect the behavior so they can either be considered as hints or as different types with implicit type conversion. In any case, implementations are free to calculate everything at high precision or greater.

Should precisions be considered when resolving function calls?

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

How should precisions be propagated in an expression?

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

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

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

What if there is no precision in an expression?

Option 1: Leave this as undefined.

Option 2: Use the default precision.

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

Do precision qualifiers for uniforms need to match?

Option 1: Yes.

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

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

Option 2: No.

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

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

RESOLUTION: Yes, precision qualifiers for uniforms must match.

Do precision qualifiers for varyings need to match?

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

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

RESOLUTION Precision qualifiers for varyings do not need to match.

Is **highp** in the fragment shader an extension or an option? Does it need to be enabled with **#extension**?

The macro can be used within the shader to select high or medium precision if shader portability is an issue. Requiring the use of **#extension** seems unnecessary.

RESOLUTION: Fragment shader **highp** is an option and is not enabled with **#extension**.

lowp int

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

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

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

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

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

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

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

Precision of built-in texture functions.

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

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

What should the default precision of sampler types be?

RESOLUTION: The default precision of all sampler types should be **lowp**.

10.4 Function and Variable Name Spaces

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

RESOLUTION: Functions and variables share the same name space.

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

1. Redeclaring a function. A function prototype is a declaration but not a definition. A function definition is both a declaration and a definition. Consequently a function prototype and a function definition (of the same function) within the same scope qualifies as redeclaration.
2. Declaring a name as a function and then redeclaring it as a structure.
3. Declaring a name as a variable and then redeclaring it as a structure.

RESOLUTION: For a particular function, a single prototype and a single definition is permitted. All other cases of multiple declarations are disallowed.

10.5 Local Function Declarations and Function Hiding

Should local functions hide all functions of the same same? Yes, if we keep local function declarations. The only use for local function declarations in GLSL ES is to unhide functions that have been hidden by variable or structure declarations. This is not a compelling reason to keep them.

RESOLUTION: Remove local function declarations.

10.6 Overloading main()

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

RESOLUTION: No. The main function cannot be overloaded.

10.7 Error Reporting

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

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

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

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

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

Should static recursion be detected?

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

10.8 Structure Declarations

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

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

Should structure declarations be allowed in function parameters?

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

10.9 Embedded Structure Definitions

Should embedded structure definitions be allowed?

e.g.

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

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

Option 1: Disallow embedded structure definitions.

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

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

RESOLUTION: Remove embedded structure definitions.

10.10 Redefining Built-in Functions

If a built-in function is redefined, should the new function hide all built-in functions with the same name? Yes, that follows the c++ rules. The built-in functions are then in an 'outer' scope and the global scope is nested within this.

Is this feature useful? Built-in functions are likely to be efficiently mapped to the hardware. User-defined functions may not be as efficient but may be able to offer greater precision (e.g. for the trig functions). The application may then want access to both the original and new function. Some user-defined functions would benefit from access to the original function. Once the new function has been declared, the original function is hidden so both these use cases are impossible with the current specification.

Option 1: Remove the ability to redefine built-in functions. Allow them to be overloaded. This creates a subtle incompatibility with the desktop:

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

Option 2: Remove the ability to redefine or overload functions.

RESOLUTION: Built-in functions reside in an outer scope, distinct from the global scope. However, redefining functions is disallowed. The overloading of built-in functions is allowed.

10.11 Constant Expressions

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

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

Option 1: Yes, this useful.

Option 2: No. If the built-in function is redefined, then ensuring it is constant becomes very difficult. It also requires the evaluation of user-defined functions at compile time.

Option 3: Yes but not if the function is redefined. This allows the common use cases but introduces non-orthogonality.

Option 4: Yes and allow user-defined functions to be defined as **const**.

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

10.12 Varying Linkage

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

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

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

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

RESOLUTION: No.

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

10.13 gl_Position

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

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

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

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

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

10.14 Pre-processor

Is the preprocessor necessary? The extension mechanism relies on the preprocessor so this would need to be replaced. The **#define, #ifdef**, **#ifndef**, **#elsif** and **#endif** constructs are commonly used for managing different versions and for include guards. Macros, especially parameterized macros are considered less useful since these should be replaced by constants and functions. They are still commonly used in C programs so there may be similar cases in GLSL ES although the argument for including them is not strong. Other parts of the preprocessor have already been removed.

RESOLUTION: Keep the basic preprocessor.

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

RESOLUTION: No, this is not required since macro definitions are not expected to be complex.

10.15 Phases of Compilation

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

The cases where the result is different are not common.

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

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

RESOLUTION: Follow c++ rules.

10.16 Maximum Number of Varyings

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

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

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

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

Should attributes and uniforms follow this rule?

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

RESOLUTION: Uniforms should follow this rule but the number of vertex uniforms should be increased to 128 4-vectors to allow for the inefficiencies of the packing algorithm

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

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

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

RESOLUTION: gl_FragCoord is included in the count of varyings.

How should **mat2** varyings be packed?

Option 1: Pack them as 2x2.

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

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

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

RESOLUTION: Option 3

Should **mat3** take 3 whole rows?

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

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

Should **vec3** take a whole row?

RESOLUTION: No.

Should gl_MaxVertexUniformsComponents be changed to reflect the packing rules?

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

10.17 Unsized Array Declarations

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

```
varying vec4 gl_TexCoord[];
...
gl_FragColor = texture2D (tex, gl_TexCoord[0].xy);
```

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

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

RESOLUTION: Remove unsized array declarations.

10.18 Invariance

How should invariance between shaders be handled?

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

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

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

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

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

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

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

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

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

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

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

Option 2: Specify that they must match.

RESOLUTION: Invariance qualifiers for varying declarations must match.

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

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

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

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

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

10.19 Invariance Within a shader

How should invariance within a shader be specified?

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

Option 1: Prohibit this behavior.

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

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

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

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

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

RESOLUTION: Yes, constant expressions must be invariant.

10.20 While-loop Declarations

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

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

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

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

10.21 Cross Linking Between Shaders

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

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

10.22 Visibility of Declarations

At what point should a declaration take effect?

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

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

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

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

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

10.23 Language Version

This version of the language is based on version 1.10 of the desktop GLSL. However it includes a number of features that are in version 1.20 but not 1.10. Should __VERSION__ return 110 or 120 or something else?

Option 1: It should return 110. GL_ES is defined so there is no ambiguity with the desktop version

Option 2: It should return 115 to indicate that it is midway between desktop versions 1.10 and 1.20. No, this is wrong, ES simplifies the language in many areas and adds features in others.

Option 3: The languages are independent enough that there should be an independent numbering scheme. __VERSION__ should return 100.

RESOLUTION: __VERSION__ should return 100

10.24 Samplers

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

Current specification:

```
uniform sampler2D sampler[8];
int index = f(...);
vec4 tex = texture2D(sampler[index], xy); // allowed
```

Using assignment of sampler types:

```
uniform sampler2D s;
s = g(...);
vec4 tex = texture2D(s, xy); //  not allowed
```

RESOLUTION: Keep current specification. Support for dynamic indexing of arrays of samplers is not mandated for ES2.0 (specified in the limitations section).

10.25 Dynamic Indexing

Dynamic indexing of arrays, vectors and matrices is not directly supported by some implementations. Program transforms exist but the performance is reduced.

Should dynamic indexing of arrays be removed from the specification?

RESOLUTION: Keep dynamic indexing of uniforms (for skinning). Remove for temps (in the limitations section).

Should dynamic indexing of vectors and matrices be removed from the specification?

RESOLUTION: Keep in main specification. Support is not mandated.

10.26 Maximum Number of Texture Units

The minimum number of texture units that must be supported in the fragment shader is currently 2 as defined by gl_MaxTextureImageUnits = 2. Is this too low for ES 2.0?

Option 1: Yes, the number of texturing units is the limiting factor for fragment shaders. The number of texture units was increased from 1 to 2 going from ES 1.0 to ES 1.1 so it should be doubled for ES 2.0 (i.e. 4).

Option 2: Increase to 8.

RESOLUTION: Increase to 8.

10.27 On-target Error Reporting

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

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

10.28 Rounding of Integer Division

Should the rounding mode be specified for integer division?

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

(a / b) * b + a % b = a (a and b are integers)

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

Since the remainder operator is not part of GLSL ES 1.00, it is not necessary to specify the rounding mode.

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

10.29 Undefined Return Values

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

Example:

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

...

int a = f();
```

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

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

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

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

E.g.

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

This becomes impossible to determine in the presence of loops.

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

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

10.30 Precisions of Operations

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

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

9 / 3 = 9.0 * (1.0 / 3.0) = 9.0 * 0.333 = 2.997 = 2 (integer)

RESOLUTION: Given the wide range of current implementations, it is not feasible to standardize these precisions. The precisions of operations are therefore implementation-dependent.

10.31 Compiler Transforms

What compiler transforms should be allowed?

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

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

10.32 Expansion of Function-like Macros in the Preprocessor

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

RESOLUTION: Follow the C++ specification.

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

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

RESOLUTION: Leave as undefined behavior.

10.33 Should Extension Macros be Globally Defined?

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

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

Option 1: Globally defined

```
#ifdef GL_OES_standard_derivatives
  #extension GL_OES_standard_derivatives : enable
  ...
#endif
```

Option 2: Defined as part of #extension

```
#extension GL_OES_standard_derivatives : enable // warning if not available
#ifdef GL_OES_standard_derivatives
  ...
#endif
```

RESOLUTION: The macros are defined globally.

10.34 Increasing the Minimum Requirements

Since version 1.00 was released, it has become apparent that all implementations support more than the minimum requirements. Some implementations go far beyond the minima.
The minimum requirements does not include being able to index an array with the loop index which is a very useful feature.

How should features that are part of the language specification but beyond the minima be exposed?

Option 1: Do nothing. Applications can test for the presence of a particular feature by attempting to compile it. If it is not supported, the compile will fail.
This is considered unsatisfactory because there is no incentive for new designs to support the optional features.

Option 2: Define an extension. Applications can test for the extension but absence of the extension does not necessarily imply that the features are unsupported. Extensions would be appropriate if there were more than one set of optional features that needed to be exposed.

Option 3: Define a new language version e.g. 1.01. Shaders will be required to include #version 101 and compilers that do not support the extended features will be required to fail compilation. The version directive will also cause existing compilers to fail compilation even if they support the extended feature set.

Option 4: Define a new language version 1.01 but shaders are not required to include #version 101. Compilers that support the extended feature set may return __VERSION__ = 101. Compiler that do not support the extended feature set must continue to return __VERSION__ = 100 and must fail on #version 101.
This allows existing implementations that already support the 1.01 features to continue to support 1.01 shaders, providing the shaders do not declare #version 101.

Option 5: Define a new language version but still call it version 1.00.
In general, new functionality cannot be added to an existing published specification. However, in this case, the functionality is already part of the specification. The functionality, which is currently specified as optional, is being made mandatory.

RESOLUTION: Option 5: A new revision of the current version (version 1.00) will be published.

Which of the minimum requirements should be relaxed?

Option 1: Only indexing into arrays (and matrices and vectors) with constant expressions, loop indices or a combination. All known implementations already support this.

Option 2: Indexing into arrays with any expression. Some implementations already support this but it is less common.

Option 3: Define more than one extension. The first would expose the universally supported features and subsequent extensions would expose the less commonly supported features.

RESOLUTION: Only the universally supported features should be exposed.

Does the indexing apply to vectors and matrices as well as arrays?
RESOLUTION: Yes

Should a limit on the number of iterations be specified?

RESOLUTION: No

Should extended control flow be exposed at the same time? As with indexing, most implementations support more than the minima.
RESOLUTION: No, only the indexing is exposed.

Are integer divisions allowed within index expressions? Integer divisions may not yield the correct result (see issue 30).

Are floating point operations allowed within index expressions?

11 Errors

This section lists errors that must be detected by the compiler and/or linker. The error string returned is implementation-dependent.

11.1 Preprocessor Errors

P0001: Preprocessor syntax error

P0002: #error

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

P0004: High Precision not supported

P0005: #version must be the 1st directive/statement in a program

P0006: #line has wrong parameters

P0007: Language version not supported

11.2 Lexer/Parser Errors

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

L0001: Syntax error

The parser also detects the following errors:

L0002: Undefined identifier.

L0003: Use of reserved keywords

11.3 Semantic Errors

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

S0002: Array parameter must be an integer

S0003: **if** parameter must be a boolean

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

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

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

S0007: Wrong arguments for constructor.

S0008: Argument unused in constructor

S0009: Too few arguments for constructor

S0011: Arguments in wrong order for structure constructor.

S0012: Expression must be a constant expression.

S0013: Initializer for constant variable must be a constant expression.

S0015: Expression must be an integral constant expression.

S0017: Array size must be greater than zero.

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

S0021: Indexing an array with a negative integral constant expression.

S0022: Redefinition of variable in same scope.

S0023: Redefinition of function in same scope.

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

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

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

S0027: Target of assignment is not an l-value.

S0028: Precision used with type other than integer, floating point or sampler type.

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

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

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

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

S0034: Variable cannot be declared invariant.

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

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

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

S0039: Function declared void but return statement has an argument.

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

S0041: Function return type is an array.

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

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

S0044: Declaring an attribute outside of a vertex shader.

S0045: Declaring an attribute inside a function.

S0046: Declaring a uniform inside a function.

S0047: Declaring a varying inside a function.

S0048: Illegal data type for varying.

S0049: Illegal data type for attribute (can only use **float**, **vec2**, **vec3**, **vec4**, **mat2**, **mat3**, and **mat4**).

S0050: Initializer for attribute.

S0051: Initializer for varying.

S0052: Initializer for uniform.

11.4 Linker

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

L0004: Too many attribute values.

L0005: Too many uniform values.

L0006: Too many varyings.

L0007: Fragment shader uses a varying that has not been declared in the vertex shader.

L0008: Type mismatch between varyings.

L0009: Missing main function for shader.

12 Normative References

1. International Standard ISO/IEC 14882:1998(E). Programming Languages – C++
2. International Standard ISO/IEC 646:1991. Information technology - ISO 7-bit coded character set for information interchange
3. The OpenGL® Graphics System: A Specification (Version 2.0 - October 22, 2004)

13 Acknowledgements

This specification contains many contributions from discussions with members of the Khronos OpenGL ES group, including:

Jon Leech

Aaftab Munshi, Apple
Jeremy Sandmel, Apple

Nathan Charles, 3Dlabs,
Jeremy Hayes, 3Dlabs,
Jon Kennedy, 3Dlabs
Nick Murphy, 3Dlabs

Bill Marshall, Alt Software
John Jarvis, Alt Software
John Boal, Alt Software

Mario Blazevic, ARM
Aske Simon Christensen, ARM
Sean Ellis, ARM
Erik Faye-Lund, ARM
Frode Heggelund, ARM
Rune Holm, ARM
Jørn Nystad, ARM
Borgar Ljosland, ARM
Remi Pedersen, ARM
Ed Plowman, ARM,
Justin Radeka, ARM
David Schreiner, ARM
Edvard Sørgård, ARM
Gareth Vaughan, ARM

Pierre Boudier, ATI
Gordon Grigor, ATI
Chris Grimm, ATI
Bill Licea-Kane, ATI
Robert Simpson, ATI

Eben Upton, Broadcom
Roger Nixon, Broadcom

Christopher MacPherson, Codeplay
Tarik Rahman, Codeplay
Wan Mahamood, Codeplay,

Eisaku Ohbuchi, DMP
Eric Fausett, DMP
Yoshihiko Kuwahara, DMP
Keisuke Kirii, DMP
Yukitaka Takemuta, DMP
Max Kazakov, DMP

Jacob Ström, Ericsson,

Hoon-Young Cho, ETRI
Minhong Yun, ETRI
Young Seok Kim, ETRI

Jung Chol Kwon, Ex3d
Mike Khim, Ex3d

Christian Laforte, Feeling Software

Brian Murray, Freescale

Tero Sarkkinen, Futuremark
Timo Suoranta, Futuremark

Anthony Tai, GiQuilla
Mike Cai, GiQuilla,

Avi Shapira, Graphic Remedy
Yaki Tebeka, Graphic Remedy

Mark Callow, Hi Corp,

Bruno Schwander, Hooked Wireless

Petri Kero, Hybrid Graphics
Tuomas Lukka, Hybrid Graphics
Ville Miettinen, Hybrid Graphics
Jasin Bushnaief, Hybrid Graphics

Kristoff Beets, Imagination Technologies
Mark Butler, Imagination Technologies
Graham Connor, Imagination Technologies
John Howson, Imagination Technologies
Ben Bowman, Imagination Technologies
James McCarthy, Imagination Technologies
Mohit Mehta, Imagination Technologies
Nicolas Thibieroz Imagination Technologies

John Kessenich Intel

Yong Moo Kim, LG Electronics
Woo Sedo Kim, LG Electronics

Hiroyasu Negishi, Mitsubishi
Yoshiyuki Kato, Mitsubishi

Hwanyong Lee, MTIS
Youngwook Oh, MTIS

Jani Vaarala, Nokia
Kari Pulli, Nokia
Tero Pihlajakoski, Nokia
Jarkko Kemppainen, Nokia
Koichi Mori,Nokia
Joonas Itäranta, Nokia
Mika Rytkonen, Nokia
Tomi Aarnio, Nokia

Cass Everitt, Nvidia
Daniel Horowitz, Nvidia
Barthold Lichtenbelt, Nvidia
Tom McReynolds, Nvidia
Gary King, Nvidia
Ross Thompson, Nvidia
Neil Trevett, Nvidia
Chris Wynn, Nvidia,

Andrew Ennons, OpenText

Andy Methley, Panasonic
Akira Uesaki, Panasonic
Katzutaka Nishio, Panasonic,

Angus Dorbie Qualcomm
Aleksandra Krstic, Qualcomm,

Eli Ben-Ami, Samsung
Jitaek Lim, Samsung
Kee Chang Lee, Samsung
Natan Linder, Samsung
Rami Mayer, Samsung,

Michael Antonov, Scaleform

Steve Lee, SGI
Thomas Tannert, SGI

Steve Lee, SIS

Remi Arnaud, Sony
Axel Mamode, Sony
Robin Green, Sony,

Steve Hill, STMicroelectronics

Dan Rice, Sun

Robert Palmer, Symbian
Lane Roberts, Symbian
Stefan von Cavallar, Symbian

Marko Lukat, Tao
Phil Huxley, Tao

Tommy Asano, Takumi
Toshio Nishidai, Takumi

Gerry Hamel, Texas Instruments
Fred Noraz, Texas Instruments
Stephen Wilkinson, Texas Instruments
Tom Olson, Texas Instruments

Hans-Martin Will, Vincent.

Appendix A: Limitations for ES 2.0

1 Overview

OpenGL ES 2.0 implementations are not required to support the full GLSL ES 1.00 specification. This section lists the features which are not fully supported in ES 2.0. Features not listed in this section must be supported in their entirety.

Within the GLSL ES specification, implementations are permitted to implement features beyond the minima described in this section, without the use of an extension.

2 Length of Shader Executable

This is defined by the conformance tests.

3 Usage of Temporary Variables

The maximum number of variables is defined by the conformance tests.

4 Control Flow

In general, control flow is limited to forward branching and to loops where the maximum number of iterations can easily be determined at compile time.

Forward branching is allowed, both for constant and non-constant conditions. Therefore **if-then**, **if-then-else, break and continue** statements are permitted.

Backward branching is only permitted for constant iteration loops as defined below.

In the following section, loop indices are defined as all the non-constant variables appearing in the branch condition expression in a loop.

for loops are supported but with the following restrictions:

- There is one loop index.
- The loop index has type **int** or **float**.
- The for statement has the form:

  for (*init-declaration* **;** *condition* **;** *expression* **)** *statement*

- init-declaration has the form:

  type-specifier identifier = constant-expression

  Consequently the loop variable cannot be a global variable.

- *condition* has the form

  loop_index relational_operator constant_expression

  where *relational_operator* is one of: **>** **>=** **<** **<=** **==** or **!=**

- *for_header* has one of the following forms:

  loop_index++
  loop_index--
  loop_index += *constant_expression*
  loop_index -= *constant_expression*

- Within the body of the loop, the loop index is not statically assigned to nor is it used as the argument to a function **out** or **inout** parameter.

Support for **while** and **do-while** is not mandated.

5 Indexing of Arrays, Vectors and Matrices

Definition:
constant-index-expressions are a superset of *constant-expressions*. Constant-index-expressions can include loop indices as defined in Appendix A section 4.

The following are constant-index-expressions:

- Constant expressions
- Loop indices as defined in section 4
- Expressions composed of both of the above

When used as an index, a *constant-index-expression* must have integral type.

Uniforms (excluding samplers)

In the vertex shader, support for all forms of array indexing is mandated. In the fragment shader, support for indexing is only mandated for *constant-index-expressions.*

Samplers

GLSL ES 1.00 supports both arrays of samplers and arrays of structures which contain samplers. In both these cases, for ES 2.0, support for indexing with a *constant-index-expression* is mandated but support for indexing with other values is not mandated.

Attributes

Support for indexing of matrices and vectors with *constant-index-expressions* is mandated.

Support for indexing of matrices and vectors with other values is not mandated.

Attribute arrays are disallowed by the specification.

Varyings

Support for indexing with a *constant-index-expression* is mandated.

Support for indexing with other values is not mandated.

Variables

Support for indexing with a *constant-index-expression* is mandated.

Support for indexing with other values is not mandated.

Constants

Support for indexing of matrices and vectors with *constant-index-expression* is mandated.

Support for indexing of matrices and vectors with other values is not mandated.

Constant arrays are disallowed by the specification.

Summary

The following array indexing functionality must be supported:

	Vertex Shaders	**Fragment Shaders**
Uniforms	Any integer	constant-index-expression
Attributes (vectors and matrices)	constant-index-expression	Not applicable
Varyings	constant-index-expression	constant-index-expression
Samplers	constant-index-expression	constant-index-expression
Variables	constant-index-expression	constant-index-expression
Constants (vectors and matrices)	constant-index-expression	constant-index-expression

6 Texture Accesses

Accessing mip-mapped textures within the body of a non-uniform conditional block gives an undefined value. A non-uniform conditional block is a block whose execution cannot be determined at compile time.

7 Counting of Varyings and Uniforms

GLSL ES 1.00 specifies the storage available for varying variables in terms of an array of 4-vectors. Similarly for uniform variables. The assumption is that variables will be packed into these arrays without wasting space. This places significant burden on implementations since optimal packing is computationally intensive. Implementations may have more internal resources than exposed to the application and so avoid the need to perform packing but this is also considered an expensive solution.

ES 2.0 therefore relaxes the requirements for packing by specifying a simpler algorithm that may be used. This algorithm specifies a minimum requirement for when a set of variables must be supported by an implementation. The implementation is allowed to support more than the minimum and so may use a more efficient algorithm and/or may support more registers than the virtual target machine.

In all cases, failing resource allocation for variables must result in an error.

The resource allocation of variables must succeed for all cases where the following packing algorithm succeeds:

- The target architecture consists of a grid of registers, 8 rows by 4 columns for varying variables and 128 rows by 4 columns for uniform variables. Each register can contain a float value.
- Variables are packed into the registers one at a time so that they each occupy a contiguous sub-rectangle. No splitting of variables is permitted.
- The orientation of variables is fixed. Vectors always occupy registers in a single row. Elements of an array must be in different rows. E.g. vec4 will always occupy one row; float[8] will occupy one column. Since it is not permitted to split a variable, large arrays e.g.. for varyings, float[16] will always fail with this algorithm.
- Variables consume only the minimum space required with the exception that mat2 occupies 2 complete rows. This is to allow implementations more flexibility in how variables are stored.
- Arrays of size N are assumed to take N times the size of the base type.
- Variables are packed in the following order:
    1. Arrays of mat4 and mat4
    2. Arrays of mat2 and mat2 (since they occupy full rows)
    3. Arrays of vec4 and vec4
    4. Arrays of mat3 and mat3
    5. Arrays of vec3 and vec3
    6. Arrays of vec2 and vec2
    7. Arrays of float and float
- For each of the above types, the arrays are processed in order of size, largest first. Arrays of size 1 and the base type are considered equivalent. In the case of varyings, the first type to be packed (successfully) is mat4[2] followed by mat4, mat2[2], mat2, vec4[8], ve4[7],...vec4[1], vec4, mat3[2], mat3 and so on. The last variables to be packed will be float (and float[1]).

- For 2,3 and 4 component variables packing is started using the 1st column of the 1st row. Variables are then allocated to successive rows, aligning them to the 1st column.
- For 2 component variables, when there are no spare rows, the strategy is switched to using the highest numbered row and the lowest numbered column where the variable will fit. (In practice, this means they will be aligned to the x or z component.) Packing of any further 3 or 4 component variables will fail at this point.
- 1 component variables (i.e. floats and arrays of floats) have their own packing rule. They are packed in order of size, largest first. Each variable is placed in the column that leaves the least amount of space in the column and aligned to the lowest available rows within that column. During this phase of packing, space will be available in up to 4 columns. The space within each column is always contiguous.
- If at any time the packing of a variable fails, the compiler or linker must report an error.

Example: pack the following types:

```
varying vec4 a;     // top left
varying mat3 b;     // align to left, lowest numbered rows
varying vec2 c[3];  // align to left, lowest numbered rows
varying vec2 d[2];  // Cannot align to left so align to z column, highest
                    // numbered rows
varying vec2 e;     // Align to left, lowest numbered rows.
varying float f[3]  // Column with minimum space
varying float g[2]; // Column with minimum space (choice of 2, either one
                    // can be used)
varying float h;    // Column with minimum space
```

In this example, the varyings happen to be listed in the order in which they are packed. Packing is independent of the order of declaration.

	x	y	z	w
0	**a**	**a**	**a**	**a**
1	**b**	**b**	**b**	**f**
2	**b**	**b**	**b**	**f**
3	**b**	**b**	**b**	**f**
4	**c**	**c**	**g**	**h**
5	**c**	**c**	**g**	
6	**c**	**c**	**d**	**d**
7	**e**	**e**	**d**	**d**

Some varyings e.g. mat4[8] will be too large to fit. These always fail with this algorithm.

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

Only varyings statically used in both shaders are counted.

When calculating the number of uniform variables used, any literal constants present in the shader source after preprocessing are included when calculating the storage requirements. Multiple instances of identical constants should count multiple times.

Part of the storage may be reserved by an implementation for its own use e.g. for computation of transcendental functions. This reduces the number of uniforms available to the shader. The size of this reduction is undefined but should be minimized.

8 Shader Parameters

The following are the minimum values that must be supported by an ES 2.0 implementation:

```
const mediump int gl_MaxVertexAttribs = 8;
const mediump int gl_MaxVertexUniformVectors = 128;
const mediump int gl_MaxVaryingVectors = 8;
const mediump int gl_MaxVertexTextureImageUnits = 0;
const mediump int gl_MaxCombinedTextureImageUnits = 8;
const mediump int gl_MaxTextureImageUnits = 8;
const mediump int gl_MaxFragmentUniformVectors = 16;
const mediump int gl_MaxDrawBuffers = 1;
```