The OpenGL® Shading Language

Language Version: 4.30
Document Revision: 6
3-Aug-2012

Editor: John Kessenich, LunarG

Version 1.1 Authors: John Kessenich, Dave Baldwin, Randi Rost
# Table of Contents

1 **Introduction**.................................................................................................................................1  
   1.1 **Acknowledgments**......................................................................................................................2  
   1.2 **Changes**.........................................................................................................................................3  
      1.2.1 **Summary of Changes from Version 4.20**.............................................................................3  
   1.3 **Overview**........................................................................................................................................5  
   1.4 **Error Handling**.............................................................................................................................5  
   1.5 **Typographical Conventions**.......................................................................................................5  
   1.6 **Deprecation**....................................................................................................................................5  

2 **Overview of OpenGL Shading**..........................................................................................................6  
   2.1 **Vertex Processor**..........................................................................................................................6  
   2.2 **Tessellation Control Processor**.....................................................................................................6  
   2.3 **Tessellation Evaluation Processor**................................................................................................7  
   2.4 **Geometry Processor**....................................................................................................................7  
   2.5 **Fragment Processor**.....................................................................................................................7  
   2.6 **Compute Processor**.......................................................................................................................7  

3 **Basics**................................................................................................................................................9  
   3.1 **Character Set**..................................................................................................................................9  
   3.2 **Source Strings**................................................................................................................................9  
   3.3 **Preprocessor**..................................................................................................................................10  
   3.4 **Comments**......................................................................................................................................15  
   3.5 **Tokens**..........................................................................................................................................16  
   3.6 **Keywords**........................................................................................................................................16  
   3.7 **Identifiers**.......................................................................................................................................18  
   3.8 **Definitions**.....................................................................................................................................18  
      3.8.1 **Static Use**.................................................................................................................................19  
      3.8.2 **Uniform and Non-Uniform Control Flow**..............................................................................19  
      3.8.3 **Dynamically Uniform Expressions**.......................................................................................19  

4 **Variables and Types**.........................................................................................................................20  
   4.1 **Basic Types**....................................................................................................................................20  
      4.1.1 **Void**.......................................................................................................................................24  
      4.1.2 **Booleans**...................................................................................................................................24  
      4.1.3 **Integers**....................................................................................................................................24  
      4.1.4 **Floating-Point Variables**.......................................................................................................26  
      4.1.5 **Vectors**....................................................................................................................................27  
      4.1.6 **Matrices**....................................................................................................................................27  
      4.1.7 **Opaque Types**..........................................................................................................................28  
         4.1.7.1 **Samplers**............................................................................................................................28  
         4.1.7.2 **Images**...............................................................................................................................28
8.1 Angle and Trigonometry Functions ................................................................. 129
8.2 Exponential Functions .................................................................................. 131
8.3 Common Functions ...................................................................................... 132
8.4 Floating-Point Pack and Unpack Functions ................................................... 137
8.5 Geometric Functions .................................................................................... 139
8.6 Matrix Functions .......................................................................................... 141
8.7 Vector Relational Functions .......................................................................... 143
8.8 Integer Functions ........................................................................................ 145
8.9 Texture Functions ........................................................................................ 147
  8.9.1 Texture Query Functions ........................................................................ 148
  8.9.2 Texel Lookup Functions ......................................................................... 151
  8.9.3 Texture Gather Functions ....................................................................... 157
  8.9.4 Compatibility Profile Texture Functions ................................................. 160
8.10 Atomic-Counter Functions .......................................................................... 162
8.11 Atomic Memory Functions .......................................................................... 162
8.12 Image Functions ........................................................................................ 163
8.13 Fragment Processing Functions ................................................................... 166
  8.13.1 Derivative Functions ............................................................................ 166
  8.13.2 Interpolation Functions ........................................................................ 168
8.14 Noise Functions ........................................................................................ 169
8.15 Geometry Shader Functions ....................................................................... 170
8.16 Shader Invocation Control Functions ............................................................ 172
8.17 Shader Memory Control Functions .............................................................. 173
9 Shading Language Grammar for Core Profile ................................................. 175
1 Introduction

This document specifies only version 4.30 of the OpenGL Shading Language. It requires __VERSION__ to substitute 430, and requires #version to accept only 430. If #version is declared with a smaller number, the language accepted is a previous version of the shading language, which will be supported depending on the version and type of context in the OpenGL API. See the OpenGL Graphics System Specification, Version 4.3, for details on what language versions are supported.

Previous versions of the OpenGL Shading Language, as well as the OpenGL ES Shading Language, are not strict subsets of the version specified here, particularly with respect to precision, name-hiding rules, and treatment of interface variables. See the specification corresponding to a particular language version for details specific to that version of the language.

All OpenGL Graphics System Specification references in this specification are to version 4.3.
1.1 Acknowledgments

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

Pat Brown, NVIDIA
Jeff Bolz, NVIDIA
Frank Chen
Pierre Boudier, AMD
Piers Daniell, NVIDIA
Chris Dodd, NVIDIA
Nick Haemel, NVIDIA
Jason Green, TransGaming
Brent Insko, Intel
Jon Leech
Bill Licea-Kane, AMD
Daniel Koch, TransGaming
Barthold Lichtenbelt, NVIDIA
Bruce Merry, ARM
Robert Ohannessian
Acorn Pooley, NVIDIA
Christophe Riccio, AMD
Kevin Rogovin
Ian Romanick, Intel
Greg Roth, Nvidia
Graham Sellers, AMD
Dave Shreiner, ARM
Jeremy Sandmel, Apple
Robert Simpson, Qualcomm
Eric Werness, NVIDIA
Mark Young, AMD
1.2 Changes

1.2.1 Summary of Changes from Version 4.20

Note: No features were deprecated between versions 4.20 and 4.30.

Changes

- Add shader storage buffer objects, as per the ARB_shader_storage_buffer_object extension. This includes 1) allowing the last member of a storage buffer block to be an array that does not know its size until render time, and 2) read/write memory shared with the application and other shader invocations. It also adds the std430 layout qualifier for shader storage blocks.
- Allow .length() on all arrays; returning a compile-time constant or not, depending on how the array is sized, as per the ARB_shader_storage_buffer_object extension.
- Be clear that implicit array sizing is only within a stage, not cross stage.
- Array clarifications: 1) All arrays are inherently homogeneous, except for arrays of the new shader storage buffer objects. 2) Arrays of shader storage buffer objects will be dereferenced when the .length() method is used on an unsized array member, so that must a have valid index. 3) Arrays of other objects (uniform blocks) containing implicitly sized arrays will have the same implicit size for all elements of the array.
- Arrays of arrays are now supported, as per the GL_ARB_arrays_of_arrays extension.
- Compute shaders are now supported, as per the GL_ARB_compute_shader extension.
- Added imageSize() built-ins to query the dimensions of an image.
- Define robust out-of-bounds access behavior when enabled, as per the GL_ARB_robust_buffer_access_behavior extension.
- All choice of depth or stencil texturing, for a packed depth-stencil texture, as per the GL_ARB_stencil_texturing extension.
- Allow explicit locations/indexes to be assigned to uniform variables and subroutines, as per the GL_ARB_explicit_uniform_location extension.
- Accept ES GLSL shader #version statements, which will request ES functionality for ES GLSL versions 100 and 300, as per the GL_ARB_ES3_compatibility extension.
- Clarified about 90 errors as being either compile time, link time, or either.
- Clarify and correct scoping rules to what would normally be expected and what was intended. (Function parameters and body nest inside global space. Loop variables and body nest inside loop scope.)
- There are no digraphs (trigraphs were already disallowed).
- Remove the CPP difference that it is a compile-time error to use #if or #elif on expressions containing undefined macro names. This reverts back to following expected CPP behavior.
- Set both gl_MaxFragmentImageUniforms and gl_MaxCombinedImageUniforms to 8.
- Clarify textureSize() for cube map arrays.
• For layout qualifiers,
  ◦ make negative output locations a compile-time error, and
  ◦ make indexes outside the range \([0,1]\) a compile-time error.
• Allow mismatches in interpolation and auxiliary qualification across stages.
• Remove cross-version linking restrictions.
• Add `textureQueryLevels()` built-ins to query the number of mipmap levels, as per the GL_ARB_texture_query_levels extension.
• Make `gl_Layer` and `gl_ViewportIndex` also be inputs to the fragment shader, as per the GL_ARB_fragment_layer_viewport extension.
• Be explicit that compile-time constant expressions evaluating on the host may get a different value than the same expression on the target. Also reinforce that compile-time constant expressions are evaluated as if `precise` and `invariant` were used.
• Add more examples and rules to be more specific about the required behavior of the `precise` qualifier.
• Clarify fragment output variables cannot be double precision.
• Clarify that “floating-point” generally means both single- and double-precision and be more consistent with that.
• Be clear that `early_fragment_tests` is only needed in one fragment-stage compilation unit.
• Allow the new `shared` keyword to be in `layout-qualifier-id`, allowing backward compatibility with the `shared` identifier that was previously used.
• Added overlooked texture function
  
  ```
  float textureOffset(sampler2DArrayShadow sampler, vec4 P, vec2 offset[, float bias]).
  ```
• Add missing type in grammar, `ATOMIC_UINT`, and missing qualifiers `COHERENT`, `VOLATILE`, `RESTRICT`, `READONLY`, and `WRITEONLY`.
• Add missing initializer lists to grammar.
• Fix all API references to refer to sections in the restructured API specification.

Typos

• Fixed typos in the grammar. 1) Add missing `DOUBLECONSTANT` token, 2) remove extraneous “INVARIANT IDENTIFIER” rule, 3) remove extraneous IDENTIFIER in the “SUBROUTINE LEFT_PAREN IDENTIFIER type_name_list RIGHT_PAREN” rule, 4) remove duplicate “PATCH” and “SAMPLE”, and 5) correct lower case “Array” to be “ARRAY”.
• Fix `EndStreamPrimitive()` typo: “input” → “output”.

1 Introduction
1.3 Overview

This document describes The OpenGL Shading Language, version 4.30.

Independent compilation units written in this language are called shaders. A program is a set of shaders that are compiled and linked together, completely creating one or more of the programmable stages of the OpenGL pipeline. All the shaders for a single programmable stage must be within the same program. A complete set of programmable stages can be put into a single program or the stages can be partitioned across multiple programs. The aim of this document is to thoroughly specify the programming language. The OpenGL Graphics System Specification will specify the OpenGL entry points used to manipulate and communicate with programs and shaders.

1.4 Error Handling

Compilers, in general, accept programs that are ill-formed, due to the impossibility of detecting all ill-formed programs. Portability is only ensured for well-formed programs, which this specification describes. Compilers are encouraged to detect ill-formed programs and issue diagnostic messages, but are not required to do so for all cases. Compile-time errors must be returned for lexically or grammatically incorrect shaders. Other errors are reported at compile time or link time as indicated. Code that is “dead” must still be error checked. For example:

```markdown
if (false)     // changing false to true cannot uncover additional errors
statement; // statement must be error checked regardless
```

1.5 Typographical Conventions

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

1.6 Deprecation

Note that while this section is true of GLSL in general, version 4.3 has no deprecated features in it. Previously deprecated features have been moved to the compatibility profile and no new features have been deprecated.

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

The OpenGL Shading Language is actually several closely related languages. These languages are used to create shaders for each of the programmable processors contained in the OpenGL processing pipeline. Currently, these processors are the vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute processors.

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

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

2.1 Vertex Processor

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

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

2.2 Tessellation Control Processor

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

The tessellation control shader is invoked for each vertex of the output patch. Each invocation can read the attributes of any vertex in the input or output patches, but can only write per-vertex attributes for the corresponding output patch vertex. The shader invocations collectively produce a set of per-patch attributes for the output patch. After all tessellation control shader invocations have completed, the output vertices and per-patch attributes are assembled to form a patch to be used by subsequent pipeline stages.

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

2.3 Tessellation Evaluation Processor

The tessellation evaluation processor is a programmable unit that evaluates the position and other attributes of a vertex generated by the tessellation primitive generator, using a patch of incoming vertices and their associated data. Compilation units written in the OpenGL Shading Language to run on this processor are called tessellation evaluation shaders. When a set of tessellation evaluation shaders are successfully compiled and linked, they result in a tessellation evaluation shader executable that runs on the tessellation evaluation processor.

Each invocation of the tessellation evaluation executable computes the position and attributes of a single vertex generated by the tessellation primitive generator. The executable can read the attributes of any vertex in the input patch, plus the tessellation coordinate, which is the relative location of the vertex in the primitive being tessellated. The executable writes the position and other attributes of the vertex.

2.4 Geometry Processor

The geometry processor is a programmable unit that operates on data for incoming vertices for a primitive assembled after vertex processing and outputs a sequence of vertices forming output primitives. Compilation units written in the OpenGL Shading Language to run on this processor are called geometry shaders. When a set of geometry shaders are successfully compiled and linked, they result in a geometry shader executable that runs on the geometry processor.

A single invocation of the geometry shader executable on the geometry processor will operate on a declared input primitive with a fixed number of vertices. This single invocation can emit a variable number of vertices that are assembled into primitives of a declared output primitive type and passed to subsequent pipeline stages.

2.5 Fragment Processor

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

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

2.6 Compute Processor

The compute processor is a programmable unit that operates independently from the other shader processors. Compilation units written in the OpenGL Shading Language to run on this processor are called compute shaders. When a set of compute shaders are successfully compiled and linked, they result in a compute shader executable that runs on the compute processor.
A compute shader has access to many of the same resources as fragment and other shader processors, including textures, buffers, image variables, and atomic counters. It does not have any predefined inputs nor any fixed-function outputs. It is not part of the graphics pipeline and its visible side effects are through changes to images, storage buffers, and atomic counters.

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

3.1 Character Set

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

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

A compile-time error will be given if any other character is used outside a comment.

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

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

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

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

There is no end-of-file character.

3.2 Source Strings

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

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

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

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

### 3.3 Preprocessor

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

The complete list of preprocessor directives is as follows.

```cpp
#  
#define  
#undef  
#if  
#else  
#else  
#error  
#pragma  
#extension  
#version  
#line  
```

The following operators are also available

```cpp
defined  
##  
```

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

The number sign (#) on a line by itself is ignored. Any directive not listed above will cause a diagnostic message and make the implementation treat the shader as ill-formed.
#define and #undef functionality are defined as is standard for C++ preprocessors for macro definitions both with and without macro parameters.

The following predefined macros are available

```
__LINE__
__FILE__
__VERSION__
```

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

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

__VERSION__ will substitute a decimal integer reflecting the version number of the OpenGL shading language. The version of the shading language described in this document will have __VERSION__ substitute the decimal integer 430.

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

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

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator class</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (highest)</td>
<td>parenthetical grouping</td>
<td>( )</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>unary</td>
<td>defined</td>
<td>Right to Left</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ - ~</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>multiplicative</td>
<td>* / %</td>
<td>Left to Right</td>
</tr>
<tr>
<td>4</td>
<td>additive</td>
<td>+ -</td>
<td>Left to Right</td>
</tr>
<tr>
<td>5</td>
<td>bit-wise shift</td>
<td>&lt;&lt; &gt;&gt;=</td>
<td>Left to Right</td>
</tr>
<tr>
<td>6</td>
<td>relational</td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>Left to Right</td>
</tr>
<tr>
<td>7</td>
<td>equality</td>
<td>== !=</td>
<td>Left to Right</td>
</tr>
<tr>
<td>8</td>
<td>bit-wise and</td>
<td>&amp;</td>
<td>Left to Right</td>
</tr>
<tr>
<td>9</td>
<td>bit-wise exclusive or</td>
<td>^</td>
<td>Left to Right</td>
</tr>
<tr>
<td>10</td>
<td>bit-wise inclusive or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>logical and</td>
<td>&amp;&amp;</td>
<td>Left to Right</td>
</tr>
<tr>
<td>12 (lowest)</td>
<td>logical inclusive or</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The defined operator can be used in either of the following ways:

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

Two tokens in a macro can be concatenated into one token using the token pasting (##) operator, as is standard for C++ preprocessors. The result must be a valid single token, which will then be subject to macro expansion. That is, macro expansion happens only after token pasting. There are no other number sign based operators (e.g., no # or #@), nor is there a sizeof operator.

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

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

#error will cause the implementation to put a compile-time diagnostic message into the shader object’s information log (see section 7.12 “Shader and Program Queries” in the OpenGL Graphics System Specification for how to access a shader object’s information log). The message will be the tokens following the #error directive, up to the first new line. The implementation must then consider the shader to be ill-formed.

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

```
#pragma STDGL
```

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

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

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

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

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

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

```
#pragma version number profile opt
```

where number must be a version of the language, following the same convention as __VERSION__ above. The directive “#version 430” is required in any shader that uses version 4.30 of the language. Any number representing a version of the language a compiler does not support will cause a compile-time error to be generated. Version 1.10 of the language does not require shaders to include this directive, and shaders that do not include a #version directive will be treated as targeting version 1.10. Shaders that
specify **#version** 100 will be treated as targeting version 1.00 of the OpenGL ES Shading Language. Shaders that specify **#version** 300 will be treated as targeting version 3.00 of the OpenGL ES Shading Language.

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

```plaintext
core
compatibility
es
```

A *profile* argument can only be used with version 150 or greater. If no profile argument is provided and the version is 150 or greater, the default is **core**. If version 300 is specified, the profile argument is not optional and must be **es**, or a compile-time error results. The Language Specification for the **es** profile is specified in The OpenGL ES Shading Language specification.

Shaders for the **core** or **compatibility** profiles that declare different versions can be linked together. However, **es** profile shaders cannot be linked with non-**es** profile shaders or with **es** profile shaders of a different version, or a link-time error will result. When linking shaders of versions allowed by these rules, remaining link-time errors will be given as per the linking rules in the GLSL version corresponding to the version of the context the shaders are linked under. Shader compile-time errors must still be given strictly based on the version declared (or defaulted to) within each shader.

Unless otherwise specified, this specification is documenting the core profile, and everything specified for the core profile is also available in the compatibility profile. Features specified as belonging specifically to the compatibility profile are not available in the core profile.

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

```plaintext`
#define GL_core_profile 1
```

Implementations providing the **compatibility** profile provide the following macro:

```plaintext`
#define GL_compatibility_profile 1
```

Implementations providing the **es** profile provide the following macro:

```plaintext`
#define GL_es_profile 1
```

The **#version** directive must occur in a shader before anything else, except for comments and white space.
By default, compilers of this language must issue compile-time lexical and grammatical errors for shaders that do not conform to this specification. Any extended behavior must first be enabled. Directives to control the behavior of the compiler with respect to extensions are declared with the \texttt{#extension} directive

\begin{verbatim}
#extension extension_name : behavior
#extension all : behavior
\end{verbatim}

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

<table>
<thead>
<tr>
<th>\texttt{behavior}</th>
<th>Effect</th>
</tr>
</thead>
</table>
| \texttt{require}  | Behave as specified by the extension \texttt{extension\_name}.  
                      Give a compile-time error on the \texttt{#extension} if the extension \texttt{extension\_name} is not supported, or if \texttt{all} is specified. |
| \texttt{enable}   | Behave as specified by the extension \texttt{extension\_name}.  
                      Warn on the \texttt{#extension} if the extension \texttt{extension\_name} is not supported.  
                      Give a compile-time error on the \texttt{#extension} if \texttt{all} is specified. |
| \texttt{warn}     | Behave as specified by the extension \texttt{extension\_name}, except issue warnings on any detectable use of that extension, unless such use is supported by other enabled or required extensions.  
                      If \texttt{all} is specified, then warn on all detectable uses of any extension used.  
                      Warn on the \texttt{#extension} if the extension \texttt{extension\_name} is not supported. |
| \texttt{disable}  | Behave (including issuing errors and warnings) as if the extension \texttt{extension\_name} is not part of the language definition.  
                      If \texttt{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 \texttt{#extension} if the extension \texttt{extension\_name} is not supported. |

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

    #extension all : disable

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

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

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

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

    #line line
    #line line source-string-number

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

3.4 Comments

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

Inside comments, any byte values may be used, except a byte whose value is 0. No errors will be given for the content of comments and no validation on the content of comments need be done.

Removal of new lines by the line-continuation character (\) logically occurs before comments are processed. That is, a single-line comment ending in the line-continuation character (\) includes the next line in the comment.

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

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

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

3.6 Keywords
The following are the language's keywords and (after preprocessing) can only be used as described in this specification, or a compile-time error results:

    attribute const uniform varying buffer shared
    coherent volatile restrict readonly writeonly
    atomic_uint
    layout
    centroid flat smooth noperspective
    patch sample
    break continue do for while switch case default
    if else
    subroutine
    in out inout
    float double int void bool true false
    invariant
    discard return
    mat2 mat3 mat4      dmat2 dmat3 dmat4
    mat2x2 mat2x3 mat2x4 dmat2x2 dmat2x3 dmat2x4
    mat3x2 mat3x3 mat3x4 dmat3x2 dmat3x3 dmat3x4
    mat4x2 mat4x3 mat4x4 dmat4x2 dmat4x3 dmat4x4
    vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4 dvec2 dvec3 dvec4
    uint uvec2 uvec3 uvec4
lowp mediump highp precision
sampler1D sampler2D sampler3D samplerCube
sampler1DShadow sampler2DShadow samplerCubeShadow
sampler1DArray sampler2DArray
sampler1DArrayShadow sampler2DArrayShadow
isampler1D isampler2D isampler3D isamplerCube
isampler1DArray isampler2DArray
usampler1D usampler2D usampler3D usamplerCube
usampler1DArray usampler2DArray
sampler2DRect sampler2DRectShadow isampler2DRect usampler2DRect
samplerBuffer isamplerBuffer usamplerBuffer
sampler2DMS isampler2DMS usampler2DMS
sampler2DMSArray isampler2DMSArray usampler2DMSArray
samplerCubeArray samplerCubeArrayShadow isamplerCubeArray usamplerCubeArray
image1D iimage1D uimage1D
image2D iimage2D uimage2D
image3D iimage3D uimage3D
image2DRect iimage2DRect uimage2DRect
imageCube iimageCube uimageCube
imageBuffer iimageBuffer uimageBuffer
image1DArray iimage1DArray uimage1DArray
image2DArray iimage2DArray uimage2DArray
imageCubeArray iimageCubeArray uimageCubeArray
image2DMS iimage2DMS uimage2DMS
image2DMSArray iimage2DMSArray uimage2DMSArray
struct

The following are the keywords reserved for future use. Using them will result in a compile-time error:

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

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

```
identifier
  nondigit
  identifier nondigit
  identifier digit
nondigit: one of
  _ a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
digit: one of
  0 1 2 3 4 5 6 7 8 9
```

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

3.8 Definitions

Some language rules described below depend on the following definitions.
3.8.1 Static Use
A shader contains a static use of (or static assignment to) a variable \( x \) if, after preprocessing, the shader contains a statement that would read (or write) \( x \), whether or not run-time flow of control will cause that statement to be executed.

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

For example:

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

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

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

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

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

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

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

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

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

The OpenGL Shading Language is type safe. There are some implicit conversions between types. Exactly how and when this can occur is described in section 4.1.10 “Implicit Conversions” and as referenced by other sections in this specification.

4.1 Basic Types

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

Transparent types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>for functions that do not return a value</td>
</tr>
<tr>
<td>bool</td>
<td>a conditional type, taking on values of true or false</td>
</tr>
<tr>
<td>int</td>
<td>a signed integer</td>
</tr>
<tr>
<td>uint</td>
<td>an unsigned integer</td>
</tr>
<tr>
<td>float</td>
<td>a single single-precision floating-point scalar</td>
</tr>
<tr>
<td>double</td>
<td>a single double-precision floating point scalar</td>
</tr>
<tr>
<td>vec2</td>
<td>a two-component single-precision floating-point vector</td>
</tr>
<tr>
<td>vec3</td>
<td>a three-component single-precision floating-point vector</td>
</tr>
<tr>
<td>vec4</td>
<td>a four-component single-precision floating-point vector</td>
</tr>
<tr>
<td>dvec2</td>
<td>a two-component double-precision floating-point vector</td>
</tr>
<tr>
<td>dvec3</td>
<td>a three-component double-precision floating-point vector</td>
</tr>
<tr>
<td>dvec4</td>
<td>a four-component double-precision floating-point vector</td>
</tr>
<tr>
<td>bvec2</td>
<td>a two-component Boolean vector</td>
</tr>
<tr>
<td>bvec3</td>
<td>a three-component Boolean vector</td>
</tr>
<tr>
<td>bvec4</td>
<td>a four-component Boolean vector</td>
</tr>
<tr>
<td>ivec2</td>
<td>a two-component signed integer vector</td>
</tr>
<tr>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ivec3</td>
<td>a three-component signed integer vector</td>
</tr>
<tr>
<td>ivec4</td>
<td>a four-component signed integer vector</td>
</tr>
<tr>
<td>uvec2</td>
<td>a two-component unsigned integer vector</td>
</tr>
<tr>
<td>uvec3</td>
<td>a three-component unsigned integer vector</td>
</tr>
<tr>
<td>uvec4</td>
<td>a four-component unsigned integer vector</td>
</tr>
<tr>
<td>mat2</td>
<td>a $2 \times 2$ single-precision floating-point matrix</td>
</tr>
<tr>
<td>mat3</td>
<td>a $3 \times 3$ single-precision floating-point matrix</td>
</tr>
<tr>
<td>mat4</td>
<td>a $4 \times 4$ single-precision floating-point matrix</td>
</tr>
<tr>
<td>mat2x2</td>
<td>same as a mat2</td>
</tr>
<tr>
<td>mat2x3</td>
<td>a single-precision floating-point matrix with 2 columns and 3 rows</td>
</tr>
<tr>
<td>mat2x4</td>
<td>a single-precision floating-point matrix with 2 columns and 4 rows</td>
</tr>
<tr>
<td>mat3x2</td>
<td>a single-precision floating-point matrix with 3 columns and 2 rows</td>
</tr>
<tr>
<td>mat3x3</td>
<td>same as a mat3</td>
</tr>
<tr>
<td>mat3x4</td>
<td>a single-precision floating-point matrix with 3 columns and 4 rows</td>
</tr>
<tr>
<td>mat4x2</td>
<td>a single-precision floating-point matrix with 4 columns and 2 rows</td>
</tr>
<tr>
<td>mat4x3</td>
<td>a single-precision floating-point matrix with 4 columns and 3 rows</td>
</tr>
<tr>
<td>mat4x4</td>
<td>same as a mat4</td>
</tr>
<tr>
<td>dmat2</td>
<td>a $2 \times 2$ double-precision floating-point matrix</td>
</tr>
<tr>
<td>dmat3</td>
<td>a $3 \times 3$ double-precision floating-point matrix</td>
</tr>
<tr>
<td>dmat4</td>
<td>a $4 \times 4$ double-precision floating-point matrix</td>
</tr>
<tr>
<td>dmat2x2</td>
<td>same as a dmat2</td>
</tr>
<tr>
<td>dmat2x3</td>
<td>a double-precision floating-point matrix with 2 columns and 3 rows</td>
</tr>
<tr>
<td>dmat2x4</td>
<td>a double-precision floating-point matrix with 2 columns and 4 rows</td>
</tr>
<tr>
<td>dmat3x2</td>
<td>a double-precision floating-point matrix with 3 columns and 2 rows</td>
</tr>
<tr>
<td>dmat3x3</td>
<td>same as a dmat3</td>
</tr>
<tr>
<td>dmat3x4</td>
<td>a double-precision floating-point matrix with 3 columns and 4 rows</td>
</tr>
<tr>
<td>dmat4x2</td>
<td>a double-precision floating-point matrix with 4 columns and 2 rows</td>
</tr>
<tr>
<td>dmat4x3</td>
<td>a double-precision floating-point matrix with 4 columns and 3 rows</td>
</tr>
<tr>
<td>dmat4x4</td>
<td>same as a dmat4</td>
</tr>
</tbody>
</table>
### Floating-Point Opaque Types

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

### Signed Integer Opaque Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>isampler1D image1D</td>
<td>a handle for accessing an integer 1D texture</td>
</tr>
</tbody>
</table>
### Variables and Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>isampler2D</td>
<td>a handle for accessing an integer 2D texture</td>
</tr>
<tr>
<td>iimage2D</td>
<td></td>
</tr>
<tr>
<td>isampler3D</td>
<td>a handle for accessing an integer 3D texture</td>
</tr>
<tr>
<td>iimage3D</td>
<td></td>
</tr>
<tr>
<td>isamplerCube</td>
<td>a handle for accessing an integer cube mapped texture</td>
</tr>
<tr>
<td>iimageCube</td>
<td></td>
</tr>
<tr>
<td>isampler2DRect</td>
<td>a handle for accessing an integer 2D rectangle texture</td>
</tr>
<tr>
<td>iimage2DRect</td>
<td></td>
</tr>
<tr>
<td>isampler1DArray</td>
<td>a handle for accessing an integer 1D array texture</td>
</tr>
<tr>
<td>iimage1DArray</td>
<td></td>
</tr>
<tr>
<td>isampler2DArray</td>
<td>a handle for accessing an integer 2D array texture</td>
</tr>
<tr>
<td>iimage2DArray</td>
<td></td>
</tr>
<tr>
<td>isamplerBuffer</td>
<td>a handle for accessing an integer buffer texture</td>
</tr>
<tr>
<td>iimageBuffer</td>
<td></td>
</tr>
<tr>
<td>isampler2DMS</td>
<td>a handle for accessing an integer 2D multi-sample texture</td>
</tr>
<tr>
<td>iimage2DMS</td>
<td></td>
</tr>
<tr>
<td>isampler2DMSArray</td>
<td>a handle for accessing an integer 2D multi-sample array texture</td>
</tr>
<tr>
<td>iimage2DMSArray</td>
<td></td>
</tr>
<tr>
<td>isamplerCubeArray</td>
<td>a handle for accessing an integer cube map array texture</td>
</tr>
<tr>
<td>iimageCubeArray</td>
<td></td>
</tr>
</tbody>
</table>

#### Unsigned Integer Opaque Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_uint</td>
<td>a handle for accessing an unsigned integer atomic counter</td>
</tr>
<tr>
<td>usampler1D</td>
<td>a handle for accessing an unsigned integer 1D texture</td>
</tr>
<tr>
<td>uimage1D</td>
<td></td>
</tr>
<tr>
<td>usampler2D</td>
<td>a handle for accessing an unsigned integer 2D texture</td>
</tr>
<tr>
<td>uimage2D</td>
<td></td>
</tr>
<tr>
<td>usampler3D</td>
<td>a handle for accessing an unsigned integer 3D texture</td>
</tr>
<tr>
<td>uimage3D</td>
<td></td>
</tr>
<tr>
<td>usamplerCube</td>
<td>a handle for accessing an unsigned integer cube mapped texture</td>
</tr>
<tr>
<td>uimageCube</td>
<td></td>
</tr>
<tr>
<td>usampler2DRect</td>
<td>a handle for accessing an unsigned integer rectangle texture</td>
</tr>
<tr>
<td>uimage2DRect</td>
<td></td>
</tr>
<tr>
<td>usampler1DArray</td>
<td>a handle for accessing an unsigned integer 1D array texture</td>
</tr>
<tr>
<td>uimage1DArray</td>
<td></td>
</tr>
<tr>
<td>usampler2DArray</td>
<td>a handle for accessing an unsigned integer 2D array texture</td>
</tr>
<tr>
<td>uimage2DArray</td>
<td></td>
</tr>
</tbody>
</table>
4 Variables and Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>usamplerBuffer</td>
<td>a handle for accessing an unsigned integer buffer texture</td>
</tr>
<tr>
<td>uimageBuffer</td>
<td></td>
</tr>
<tr>
<td>usampler2DMS</td>
<td>a handle for accessing an unsigned integer 2D multi-sample texture</td>
</tr>
<tr>
<td>uimage2DMS</td>
<td></td>
</tr>
<tr>
<td>usampler2DMSArray</td>
<td>a handle for accessing an unsigned integer 2D multi-sample texture array</td>
</tr>
<tr>
<td>uimage2DMSArray</td>
<td></td>
</tr>
<tr>
<td>usamplerCubeArray</td>
<td>a handle for accessing an unsigned integer cube map array texture</td>
</tr>
<tr>
<td>uimageCubeArray</td>
<td></td>
</tr>
</tbody>
</table>

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

There are no pointer types.

4.1.1 Void

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

4.1.2 Booleans

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

```c
bool success; // declare “success” to be a Boolean
bool done = false; // declare and initialize “done”
```

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

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

4.1.3 Integers

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

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

```c
int i, j = 42; // default integer literal type is int
uint k = 3u;   // “u” establishes the type as uint
```
Literal integer constants can be expressed in decimal (base 10), octal (base 8), or hexadecimal (base 16) as follows.

\[
\text{integer-constant} : \\
\quad \text{decimal-constant} \ \text{integer-suffix,} \\
\quad \text{octal-constant} \ \text{integer-suffix,} \\
\quad \text{hexadecimal-constant} \ \text{integer-suffix,}
\]

\[
\text{integer-suffix: one of} \\
\quad \text{u} \ U
\]

\[
\text{decimal-constant} : \\
\quad \text{nonzero-digit} \\
\quad \text{decimal-constant} \ \text{digit}
\]

\[
\text{octal-constant} : \\
\quad 0 \\
\quad \text{octal-constant} \ \text{octal-digit}
\]

\[
\text{hexadecimal-constant} : \\
\quad 0x \ \text{hexadecimal-digit} \\
\quad 0X \ \text{hexadecimal-digit} \\
\quad \text{hexadecimal-constant} \ \text{hexadecimal-digit}
\]

\[
\text{digit} : \\
\quad 0 \\
\quad \text{nonzero-digit}
\]

\[
\text{nonzero-digit: one of} \\
\quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
\]

\[
\text{octal-digit: one of} \\
\quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7
\]

\[
\text{hexadecimal-digit: one of} \\
\quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \\
\quad a \ b \ c \ d \ e \ f \\
\quad A \ B \ C \ D \ E \ F
\]

No white space is allowed between the digits of an integer constant, including after the leading \text{0} or after the leading \text{0x} or \text{0X} of a constant, or before the suffix \text{u} or \text{U}. When tokenizing, the maximal token matching the above will be recognized before a new token is started. When the suffix \text{u} or \text{U} is present, the literal has type \text{uuint}, otherwise the type is \text{uint}. A leading unary minus sign (-) is interpreted as an arithmetic unary negation, not as part of the constant. Hence, literals themselves are always expressed with non-negative syntax, though they could result in a negative value.

It is a compile-time error to provide a literal integer whose bit pattern cannot fit in 32 bits. The bit pattern of the literal is always used unmodified. So a signed literal whose bit pattern includes a set sign bit creates a negative value. For example,
4 Variables and Types

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

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

4.1.4 Floating-Point Variables

Single-precision and double-precision floating point variables are available for use in a variety of scalar calculations. Generally, the term floating-point will refer to both single- and double-precision floating point. Floating-point variables are defined as in the following examples:

```c
float a, b = 1.5; // single-precision floating-point
double c, d = 2.0LF; // double-precision floating-point
```

As an input value to one of the processing units, a single-precision or double-precision floating-point variable is expected to match the corresponding IEEE 754 floating-point definition for precision and dynamic range. Floating-point variables within a shader are also encoded according to the IEEE 754 specification for single-precision floating-point values (logically, not necessarily physically). While encodings are logically IEEE 754, operations (addition, multiplication, etc.) are not necessarily performed as required by IEEE 754. See section 4.7.1 “Range and Precision” for more details on precision and usage of NaNs (Not a Number) and Infs (positive or negative infinities).
Floating-point constants are defined as follows.

\[
\text{floating-constant} : \\
\text{fractional-constant exponent-part}_{opt} \text{ floating-suffix}_{opt} \\
\text{digit-sequence exponent-part}_{opt} \text{ floating-suffix}_{opt}
\]

\[
\text{fractional-constant} : \\
\text{digit-sequence} . \text{digit-sequence} \\
\text{digit-sequence} . \\
. \text{digit-sequence}
\]

\[
\text{exponent-part} : \\
\text{e sign}_{opt} \text{digit-sequence} \\
\text{E sign}_{opt} \text{digit-sequence}
\]

\[
\text{sign} : \text{one of} \\
+ -
\]

\[
\text{digit-sequence} : \\
\text{digit} \\
\text{digit-sequence digit}
\]

\[
\text{floating-suffix}: \text{one of} \\
f F \text{ if LF}
\]

A decimal point (.) is not needed if the exponent part is present. No white space may appear anywhere within a floating-point constant, including before a suffix. When tokenizing, the maximal token matching the above will be recognized before a new token is started. When the suffix "lf" or "LF" is present, the literal has type \text{double}. Otherwise, the literal has type \text{float}. A leading unary minus sign (-) is interpreted as a unary operator and is not part of the floating-point constant.

### 4.1.5 Vectors

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

\[
\text{vec2 texcoord1, texcoord2;} \\
\text{vec3 position;} \\
\text{vec4 myRGBA;} \\
\text{ivec2 textureLookup;} \\
\text{bvec3 less;}
\]

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

### 4.1.6 Matrices

The OpenGL Shading Language has built-in types for \(2\times2\), \(2\times3\), \(2\times4\), \(3\times2\), \(3\times3\), \(3\times4\), \(4\times2\), \(4\times3\), and \(4\times4\) matrices of floating-point numbers. Matrix types beginning with "mat" have single-precision components.
while matrix types beginning with "dmat" have double-precision components. The first number in the type is the number of columns, the second is the number of rows. If there is only one number, the matrix is square. Example matrix declarations:

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

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

### 4.1.7 Opaque Types

The opaque types declare variables that are effectively opaque handles to other objects. These objects are accessed through built-in functions, not through direct reading or writing of the declared variable. They can only be declared as function parameters or `uniform`-qualified variables. Except for array indexing, structure member selection, and parentheses, opaque variables are not allowed to be operands in expressions; such use results in a compile-time error.

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

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

#### 4.1.7.1 Samplers

Sampler types (e.g., `sampler2D`) are opaque types, declared and behaving as described above for opaque types. They do not use memory qualifiers. When aggregated into arrays within a shader, samplers can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.

Sampler variables are handles to one-, two-, and three-dimensional textures, cube maps, depth textures (shadowing), etc., as enumerated in the basic types tables. There are distinct sampler types for each texture target, and for each of float, integer, and unsigned integer data types. Texture accesses are done through built-in texture functions (described in section 8.9 “Texture Functions”) and samplers are used to specify which texture to access and how it is to be filtered.

#### 4.1.7.2 Images

Image types are opaque types, declared and behaving as described above for opaque types. They can be further qualified with memory qualifiers. When aggregated into arrays within a shader, images can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.
4 Variables and Types

Image variables are handles to one-, two-, or three-dimensional images corresponding to all or a portion of a single level of a texture image bound to an image unit. There are distinct image types for each texture target, and for each of float, integer, and unsigned integer data types. Image accesses should use an image type that matches the target of the texture whose level is bound to the image unit, or for non-layered bindings of 3D or array images should use the image type that matches the dimensionality of the layer of the image (i.e., a layer of 3D, 2DArray, Cube, or CubeArray should use image2D, a layer of 1DArray should use image1D, and a layer of 2DMSArray should use image2DMS). If the image target type does not match the bound image in this manner, if the data type does not match the bound image, or if the format layout qualifier does not match the image unit format as described in section 8.25 “Texture Image Loads and Stores” of the OpenGL Specification, the results of image accesses are undefined but cannot include program termination.

Image variables are used in the image load, store, and atomic functions described in section 8.12 "Image Functions" to specify an image to access.

4.1.7.3 Atomic Counters

Atomic counter types (atomic_uint) are opaque handles to counters, declared and behaving as described above for opaque types. The variables they declare specify which counter to access when using the built-in atomic counter functions as described in section 8.10 “Atomic Counter Functions”. They are bound to buffers as described in section 4.4.6.1 “Atomic Counter Layout Qualifiers”. When aggregated into arrays within a shader, atomic counters can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.

4.1.8 Structures

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

```glsl
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).

```glsl
light lightVar2;
```
More formally, structures are declared as follows. However, the complete correct grammar is as given in section 9 “Shading Language Grammar”.

\[
\text{struct-definition} : \\
\quad \text{qualifier} \quad \text{struct} \quad \text{name} \quad \{ \text{member-list} \} \quad \text{declarators};
\]

\[
\text{member-list} :
\quad \text{member-declaration};
\quad \text{member-declaration} \quad \text{member-list};
\]

\[
\text{member-declaration} :
\quad \text{basic-type} \quad \text{declarators};
\]

where \text{name} becomes the user-defined type, and can be used to declare variables to be of this new type. The \text{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 \text{qualifier} only applies to any \text{declarators}, and is not part of the type being defined for \text{name}.

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

Anonymous structures are not supported. Embedded structure definitions are not supported. These result in compile-time errors.

\[
\text{struct S} \{ \text{float f}; \};
\]

\[
\text{struct T} \{
\text{S}; \quad \text{// Error: anonymous structures disallowed}
\text{struct} \{ \text{...}; \}; \quad \text{// Error: embedded structures disallowed}
\text{S s}; \quad \text{// Okay: nested structures with name are allowed}
\}
\]

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

Any restrictions on the usage of a type or qualifier also apply to any structure that contains a member of that type or qualifier. This also applies to structure members that are structures, recursively.

### 4.1.9 Arrays

Variables of the same type can be aggregated into arrays by declaring a name followed by brackets \([\ ]\) enclosing an optional size. When an array size is specified in a declaration, it must be an integral constant expression (see section 4.3.3 “Constant Expressions”) greater than zero. Except for the last declared member of a shader storage block (section 4.3.9 “Interface Blocks”), the size of an array must be declared before it is indexed with anything other than an integral constant expression. The size of any array must be declared before passing it as an argument to a function. Violation of any of these rules result in
Variables and Types

compile-time errors. It is legal to declare an array without a size and then later re-declare the same name as an array of the same type and specify a size. It is a compile-time error to declare an array with a size, and then later (in the same shader) index the same array with an integral constant expression greater than or equal to the declared size. It is also a compile-time error to index an array with a negative constant expression. Arrays declared as formal parameters in a function declaration must specify a size. Undefined behavior results from indexing an array with a non-constant expression that’s greater than or equal to the array’s size or less than 0. Arrays only have a single dimension (a single entry within "[]"), however, arrays of arrays can be declared. All types (basic types, structures, arrays) can be formed into an array.

All arrays are inherently homogeneous; made of elements all having the same type and size, with one exception. The exception is a shader storage block having an unsized array as its last member; an array can be formed from such a shader storage block even if the storage blocks have differing lengths for their last member.

Some examples are:

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

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

```c
float[5]
```

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

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

as a constructor of an array

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

as an unnamed parameter

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

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

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

Arrays can have initializers formed from array constructors:

```c
float a[5] = float[5](3.4, 4.2, 5.0, 5.2, 1.1);
float a[5] = float[](3.4, 4.2, 5.0, 5.2, 1.1); // same thing
```
An array of arrays can be declared as

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

which declares a one-dimensional array of size 3 of one-dimensional arrays of size 2 of vec4s. These following declarations do the same thing:

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

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


The type of `a` needed for both constructors and nameless parameters is “vec4[3][2]”:

```
vec4 b[2] = vec4[2](vec4(0.0), vec4(0.1));
vec4[3][2] a = vec4[3][2](b, b, b); // constructor
void foo(vec4[3][2]); // prototype with unnamed parameter
```

Alternatively, the initializer-list syntax can be used to initialize an array of arrays:

```
vec4 a[3][2] = { vec4[2](vec4(0.0), vec4(1.0)),
                vec4[2](vec4(0.0), vec4(1.0)),
                vec4[2](vec4(0.0), vec4(1.0)) };
```

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

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

However, it is a compile-time error to assign to an implicitly sized array. Note, this is a rare case that initializers and assignments appear to have different semantics.

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

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

This returns a type `int`. If an array has been explicitly sized, the value returned by the length method is a constant expression. If an array has not been explicitly sized and is not the last declared member of a shader storage block, the value returned by the `length` method is not a constant expression and will be determined when a program is linked. If an array has not been explicitly sized and is the last declared member of a shader storage block, the value returned will not be a constant expression and will be determined at run time based on the size of the buffer object providing storage for the block. For such arrays, the value returned by the `length` method will be undefined if the array is contained in an array of
shader storage blocks that is indexed with a non-constant expression less than zero or greater than or equal to the number of blocks in the array.

The \textbf{length} method cannot be called on an array that has not yet been explicitly sized; this results in a compile-time error. This works equally well for arrays of arrays:

\begin{verbatim}
vec4 a[3][2];
a.length()     // this is 3
a[x].length()  // this is 2
\end{verbatim}

When the \textbf{length} method will return a compile-time or link-time constant, the expression in brackets (\(x\) above) will be evaluated and subjected to the rules required for array indexes, but the array will not be dereferenced. Thus, behavior is well defined even if the run-time value of the expression is out of bounds.

If the \textbf{length} method will return a run-time value, then the array will be dereferenced with the value \(x\). If \(x\) is not a compile-time constant and is out of range, an undefined value results.

\begin{verbatim}
// for an array \(b\) containing a member array \(a\):
b[++x].a.length();    // \(b\) is never dereferenced, but "++x" is evaluated

// for an array \(s\) of a shader storage object containing a member array \(a\):
s[x].a.length();      // \(s\) is dereferenced; \(x\) needs to be a valid index
\end{verbatim}

For unsized arrays, only the outermost dimension can be lacking a size. A type that includes an unknown array size cannot be formed into an array until it gets an explicit size, except for shader storage blocks where the only unsized array member is the last member of the block.

In a shader storage block, the last member may be declared without an explicit size. In this case, the effective array size is inferred at run-time from the size of the data store backing the interface block. Such unsized arrays may be indexed with general integer expressions. However, it is a compile-time error to pass them as an argument to a function or index them with a negative constant expression.
### 4.1.10 Implicit Conversions

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

<table>
<thead>
<tr>
<th>Type of expression</th>
<th>Can be implicitly converted to</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>uint</td>
</tr>
<tr>
<td>int uint</td>
<td>float</td>
</tr>
<tr>
<td>int uint float</td>
<td>double</td>
</tr>
<tr>
<td>ivec2</td>
<td>uvec2</td>
</tr>
<tr>
<td>ivec3</td>
<td>uvec3</td>
</tr>
<tr>
<td>ivec4</td>
<td>uvec4</td>
</tr>
<tr>
<td>ivec2 uvec2</td>
<td>vec2</td>
</tr>
<tr>
<td>ivec3 uvec3</td>
<td>vec3</td>
</tr>
<tr>
<td>ivec4 uvec4</td>
<td>vec4</td>
</tr>
<tr>
<td>ivec2 uvec2 vec2</td>
<td>dvec2</td>
</tr>
<tr>
<td>ivec3 uvec3 vec3</td>
<td>dvec3</td>
</tr>
<tr>
<td>ivec4 uvec4 vec4</td>
<td>dvec4</td>
</tr>
<tr>
<td>mat2</td>
<td>dmat2</td>
</tr>
<tr>
<td>mat3</td>
<td>dmat3</td>
</tr>
<tr>
<td>mat4</td>
<td>dmat4</td>
</tr>
<tr>
<td>mat2x3</td>
<td>dmat2x3</td>
</tr>
<tr>
<td>mat2x4</td>
<td>dmat2x4</td>
</tr>
<tr>
<td>mat3x2</td>
<td>dmat3x2</td>
</tr>
<tr>
<td>mat3x4</td>
<td>dmat3x4</td>
</tr>
<tr>
<td>mat4x2</td>
<td>dmat4x2</td>
</tr>
<tr>
<td>mat4x3</td>
<td>dmat4x3</td>
</tr>
</tbody>
</table>
There are no implicit array or structure conversions. For example, an array of \texttt{int} cannot be implicitly converted to an array of \texttt{float}.

When an implicit conversion is done, it is not a re-interpretation of the expression's bit pattern, but a conversion of its value to an equivalent value in the new type. For example, the integer value \texttt{-5} will be converted to the floating-point value \texttt{-5.0}. Integer values having more bits of precision than a single-precision floating-point mantissa will lose precision when converted to \texttt{float}.

When performing implicit conversion for binary operators, there may be multiple data types to which the two operands can be converted. For example, when adding an \texttt{int} value to a \texttt{uint} value, both values can be implicitly converted to \texttt{uint}, \texttt{float}, and \texttt{double}. In such cases, a floating-point type is chosen if either operand has a floating-point type. Otherwise, an unsigned integer type is chosen if either operand has an unsigned integer type. Otherwise, a signed integer type is chosen. If operands can be implicitly converted to multiple data types deriving from the same base data type, the type with the smallest component size is used.

The conversions in the table above are done only as indicated by other sections of this specification.

\subsection{Initializers}

At declaration, an initial value for an aggregate variable may be provided, specified as an equals (\texttt{=}) followed by an initializer. The initializer is either an \texttt{assignment-expression} or a list of initializers enclosed in curly braces. The grammar for the initializer is:

\begin{verbatim}
initializer :
  assignment-expression
  { initializer-list }
  { initializer-list , }

initializer-list :
  initializer
  initializer-list , initializer
\end{verbatim}

The \texttt{assignment-expression} is a normal expression except that a comma (\texttt{,}) outside parentheses is interpreted as the end of the initializer, not as the sequence operator. As explained in more detail below, this allows creation of nested initializers: The aggregate and its initializer must exactly match in terms of nesting, number of components/elements/members present at each level, and types of components/elements/members.

An \texttt{assignment-expression} in an initializer must be either the same type as the object it initializes or be a type that can be converted to the object's type according to section 4.1.10 "Implicit Conversions". Since these include constructors, an aggregate can be initialized by either a constructor or an initializer list; an element in an initializer list can be a constructor.

If an initializer is a list of initializers enclosed in curly braces, the variable being declared must be a vector, a matrix, an array, or a structure.

\begin{verbatim}
int i = { 1 }; // illegal, i is not an aggregate
\end{verbatim}
A list of initializers enclosed in a matching set of curly braces is applied to one aggregate. This may be the variable being declared or an aggregate contained in the variable being declared. Individual initializers from the initializer list are applied to the elements/members of the aggregate, in order.

If the aggregate has a vector type, initializers from the list are applied to the components of the vector, in order, starting with component 0. The number of initializers must match the number of components.

If the aggregate has a matrix type, initializers from the list must be vector initializers and are applied to the columns of the matrix, in order, starting with column 0. The number of initializers must match the number of columns.

If the aggregate has a structure type, initializers from the list are applied to the members of the structure, in the order declared in the structure, starting with the first member. The number of initializers must match the number of members.

Applying these rules, the following matrix declarations are equivalent:

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

All of the following declarations result in a compile-time error.

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

In all cases, the innermost initializer (i.e., not a list of initializers enclosed in curly braces) applied to an object must have the same type as the object being initialized or be a type that can be converted to the object's type according to section 4.1.10 "Implicit Conversions". In the latter case, an implicit conversion will be done on the initializer before the assignment is done.

```cpp
struct {
    float a;
    int   b;
} e = { 1.2, 2 };                     // legal, all types match
```
struct {
    float a;
    int b;
} e = { 1, 3 };               // legal, first initializer is converted

All of the following declarations result in a compile-time error.

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

struct S1 {
    vec4 a;
    vec4 b;
};

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

If an initializer (of either form) is provided for an unsized array, the size of the array is determined by the number of top-level (non-nested) initializers within the initializer. All of the following declarations create arrays explicitly sized with five elements:

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

It is a compile-time error to have too few or too many initializers in an initializer list for the aggregate being initialized. That is, all elements of an array, all members of a structure, all columns of a matrix, and all components of a vector must have exactly one initializer expression present, with no unconsumed initializers.

4.2 Scoping

The scope of a variable is determined by where it is declared. If it is declared outside all function definitions, it has global scope, which starts from where it is declared and persists to the end of the shader it is declared in. If it is declared in a while test or a for statement, then it is scoped to the end of the following sub-statement. If it is declared in an if or else statement, it is scoped to the end of that statement. (See section 6.2 “Selection” and section 6.3 “Iteration” for the location of statements and sub-statements.) Otherwise, if it is declared as a statement within a compound statement, it is scoped to the end of that compound statement. If it is declared as a parameter in a function definition, it is scoped until the end of that function definition. A function's parameter declarations and body together form a single scope nested in the global scope. The if statement’s expression does not allow new variables to be declared, hence does not form a new scope.
Within a declaration, the scope of a name starts immediately after the initializer if present or immediately after the name being declared if not. Several examples:

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

struct S
{
    int x;
};
{
    S S = S(0);   // 'S' is only visible as a struct and constructor
    S;            // 'S' is now visible as a variable
}
int x = x;          // Error if x has not been previously defined.
// If the previous definition of x was in this
// same scope, this causes a redeclaration error.

int f( /* nested scope begins here */ int k)
{
    int k = k + 3; // redeclaration error of the name k
    ...
}

int f(int k)
{
    {
        int k = k + 3; // 2nd k is parameter, initializing nested first k
        int m = k      // use of new k, which is hiding the parameter
    }
}
```

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

```c
for ( /* nested scope begins here */ int i = 0; i < 10; i++) {
    int i; // redeclaration error
}
```
The body of a `do-while` loop introduces a new scope lasting only between the `do` and `while` (not including the while test expression), whether or not the body is simple or compound:

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

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

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

Shared globals are global variables declared with the same name in independently compiled units (shaders) within the same language (i.e., same stage, e.g., vertex) that are linked together when making a single program. (Globals forming the interface between two different shader languages are discussed in other sections.) Shared globals share the same name space, and must be declared with the same type. They will share the same storage. Shared global arrays must have the same base type and the same explicit size. An array implicitly sized in one shader can be explicitly sized by another shader in the same stage. If no shader in a stage has an explicit size for the array, the largest implicit size in that stage is used. There is no cross-stage array sizing. Scalars must have exactly the same type name and type definition. Structures must have the same name, sequence of type names, and type definitions, and member names to be considered the same type. This rule applies recursively for nested or embedded types. If a shared global has multiple initializers, the initializers must all be constant expressions, and they must all have the same value. Otherwise, a link-time error will result. (A shared global having only one initializer does not require that initializer to be a constant expression.)
4.3 Storage Qualifiers

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

<table>
<thead>
<tr>
<th>Storage Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; none: default &gt;</td>
<td>local read/write memory, or an input parameter to a function</td>
</tr>
<tr>
<td><strong>const</strong></td>
<td>a variable whose value cannot be changed</td>
</tr>
<tr>
<td><strong>in</strong></td>
<td>linkage into a shader from a previous stage, variable is copied in</td>
</tr>
<tr>
<td><strong>out</strong></td>
<td>linkage out of a shader to a subsequent stage, variable is copied out</td>
</tr>
<tr>
<td><strong>attribute</strong></td>
<td>compatibility profile only and vertex language only; same as <strong>in</strong> when in a vertex shader</td>
</tr>
<tr>
<td><strong>uniform</strong></td>
<td>value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application</td>
</tr>
<tr>
<td><strong>varying</strong></td>
<td>compatibility profile only and vertex and fragment languages only; same as <strong>out</strong> when in a vertex shader and same as <strong>in</strong> when in a fragment shader</td>
</tr>
<tr>
<td><strong>buffer</strong></td>
<td>value is stored in a buffer object, and can be read or written both by shader invocations and the OpenGL API</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>compute shader only; variable storage is shared across all work items in a local work group</td>
</tr>
</tbody>
</table>

Some input and output qualified variables can be qualified with at most one additional auxiliary storage qualifier:

<table>
<thead>
<tr>
<th>Auxiliary Storage Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>centroid</strong></td>
<td>centroid-based interpolation</td>
</tr>
<tr>
<td><strong>sample</strong></td>
<td>per-sample interpolation</td>
</tr>
<tr>
<td><strong>patch</strong></td>
<td>per-tessellation-patch attributes</td>
</tr>
</tbody>
</table>
Not all combinations of qualification are allowed. Which variable types can have which qualifiers are specifically defined in upcoming sections.

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

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

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

Initializers in global declarations may only be used in declarations of global variables with no storage qualifier, with a `const` qualifier or with a `uniform` qualifier. Global variables without storage qualifiers that are not initialized in their declaration or by the application will not be initialized by OpenGL, but rather will enter `main()` with undefined values.

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

### 4.3.1 Default Storage Qualifier

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

### 4.3.2 Constant Qualifier

Named compile-time constants or read-only variables can be declared using the `const` qualifier. The `const` qualifier can be used with any of the non-void transparent basic data types, as well as with structures and arrays of these. It is a compile-time error to write to a `const` variable outside of its declaration, so they must be initialized when declared. For example,

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

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

Initializers for `const` declarations at global scope must be constant expressions, as defined in section 4.3.3 “Constant Expressions.”

### 4.3.3 Constant Expressions

A `constant expression` is one of

- a literal value (e.g., `5` or `true`)
- a variable declared with the `const` qualifier and an initializer, where the initializer is a constant expression
- an expression formed by an operator on operands that are all constant expressions, including getting an element of a constant array, or a member of a constant structure, or components of a constant vector.
However, the lowest precedence operators of the sequence operator (, ) and the assignment operators (=, +=, ...) are not included in the operators that can create a constant expression.

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

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

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

Constant expressions will be always be evaluated in an invariant way, independent of use of \texttt{invariant} and \texttt{precise} qualification, so as to create the same value in multiple shaders when the same constant expressions appear in those shaders. See section 4.8.1 “The Invariant Qualifier” and section 4.9 “The Precise Qualifier” for more details on how to create invariant expressions. Constant expressions may be evaluated by the compiler's host platform, and are therefore not required to compute the same value that the same expression would evaluate to on the shader execution target. However, the host must use the same or greater precision than the target would use.

### 4.3.4 Input Variables

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

Only the input variables that are statically read need to be written by the previous stage; it is allowed to have superfluous declarations of input variables. This is shown in the following table.

<table>
<thead>
<tr>
<th>Treatment of Mismatched Input Variables</th>
<th>Consuming Shader (input variables)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Declaration</td>
<td>Declared but no Static Use</td>
</tr>
<tr>
<td>Generating Shader (output variables)</td>
<td>No Declaration</td>
<td>Allowed</td>
</tr>
<tr>
<td></td>
<td>Declared but no Static Use</td>
<td>Allowed</td>
</tr>
<tr>
<td></td>
<td>Declared and Static Use</td>
<td>Allowed</td>
</tr>
</tbody>
</table>

Errors are based on static use only. Compilation may generate a warning, but not an error, for any dynamic use the compiler can deduce that might cause consumption of undefined values.
See section 7 “Built-in Variables” for a list of the built-in input names.

Vertex shader input variables (or attributes) receive per-vertex data. They are declared in a vertex shader with the in qualifier. It is a compile-time error to use any auxiliary or interpolation qualifier on a vertex shader input. The values copied in are established by the OpenGL API or through the use of the layout identifier location. Vertex shader inputs can be single- or double-precision floating-point scalars, vectors, and matrices, or signed- and unsigned-integer scalars and vectors. Vertex shader inputs can also form arrays of these types, but not structures.

Example declarations in a vertex shader:

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

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

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

```glsl
in float foo[ ];  // geometry shader input for vertex “out float foo”
```

Each element of such an array corresponds to one vertex of the primitive being processed. Each array can optionally have a size declared. The array size will be set by, (or if provided must be consistent with) the input layout declaration(s) establishing the type of input primitive, as described later in section 4.4.1 “Input Layout Qualifiers”.

Some inputs and outputs are arrayed, meaning that for an interface between two shader stages either the input or output declaration requires an extra level of array indexing for the declarations to match. For example, with the interface between a vertex shader and a geometry shader, vertex shader output variables and geometry shader input variables of the same name must have matching types, except that the geometry shader will have one more array dimension than the vertex shader, to allow for vertex indexing. If such an arrayed interface variable is not declared with the necessary additional input or output array dimension, a link-time error will result.

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

Additionally, tessellation evaluation shaders support per-patch input variables declared with the patch and in qualifiers. Per-patch input variables are filled with the values of per-patch output variables written by
Variables and Types

The tessellation control shader. Per-patch inputs may be declared as one-dimensional arrays, but are not indexed by vertex number. Applying the `patch` qualifier to inputs can only be done in tessellation evaluation shaders. As with other input variables, per-patch inputs must be declared using the same type and qualification as per-patch outputs from the previous (tessellation control) shader stage.

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

Fragment inputs are declared as in the following examples:

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

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

When an interface between shader stages is formed using shaders from two separate program objects, it is not possible to detect mismatches between inputs and outputs when the programs are linked. When there are mismatches between inputs and outputs on such interfaces, the values passed across the interface will be partially or completely undefined. Shaders can ensure matches across such interfaces either by using input and output layout qualifiers (sections 4.4.1 “Input Layout Qualifiers” and 4.4.2 “Output Layout Qualifiers”) or by using identical input and output declarations of blocks or variables. Complete rules for interface matching between programs are found in section 7.4.1 “Shader Interface Matching” of the OpenGL Graphics System Specification.

Compute shaders do not permit user-defined input variables and do not form a formal interface with any other shader stage. See section 7.1 “Built-In Variables” for a description of built-in compute shader input variables. All other input to a compute shader is retrieved explicitly through image loads, texture fetches, loads from uniforms or uniform buffers, or other user supplied code. Redeclaration of built-in input variables in compute shaders is not permitted.
4.3.5 Uniform Variables

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

Example declarations are:

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

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

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

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

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

4.3.6 Output Variables

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

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

Vertex, tessellation evaluation, and geometry output variables output per-vertex data and are declared using the out storage qualifier. Applying patch to an output can only be done in a tessellation control shader. Output variables can only be floating-point scalars, floating-point vectors, matrices, signed or unsigned integers or integer vectors, or arrays or structures of any these.
Individual vertex, tessellation evaluation, and geometry outputs are declared as in the following examples:

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

These can also appear in interface blocks, as described in section 4.3.9 “Interface Blocks”. Interface blocks allow simpler addition of arrays to the interface from vertex to geometry shader. They also allow a fragment shader to have the same input interface as a geometry shader for a given vertex shader.

Tessellation control shader output variables are may be used to output per-vertex and per-patch data. Per-vertex output variables are arrayed (see `arrayed` under 4.3.4 Inputs) and declared using the `out` qualifier without the `patch` qualifier. Per-patch output variables are declared using the `patch` and `out` qualifiers. Per-vertex and per-patch output variables can only be floating-point scalars, floating-point vectors, matrices, signed or unsigned integers or integer vectors, or arrays or structures of any these. Since tessellation control shaders produce an arrayed primitive comprising multiple vertices, each per-vertex output variable (or output block, see interface blocks below) needs to be declared as an array. For example,

```glsl
out float foo[];  // feeds next stage input “in float foo[]”
```

Each element of such an array corresponds to one vertex of the primitive being produced. Each array can optionally have a size declared. The array size will be set by (or if provided must be consistent with) the output layout declaration(s) establishing the number of vertices in the output patch, as described later in section 4.4.2.1 “Tessellation Control Outputs”.  

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

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

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

1. At the beginning of execution.
2. At each synchronization point, unless
   - the value was well-defined after the previous synchronization point and was not written by any
4 Variables and Types

invocation since, or
• the value was written by exactly one shader invocation since the previous synchronization point, or
• the value was written by multiple shader invocations since the previous synchronization point, and the last write performed by all such invocations wrote the same value.

3. When read by a shader invocation, if
• the value was undefined at the previous synchronization point and has not been written by the same shader invocation since, or
• the output variable is written to by any other shader invocation between the previous and next synchronization points, even if that assignment occurs in code following the read.

Fragment outputs output per-fragment data and are declared using the `out` storage qualifier. It is a compile-time error to use auxiliary storage qualifiers or interpolation qualifiers on an output in a fragment shader. Fragment outputs can only be `float`, single-precision floating-point vectors, signed or unsigned integers or integer vectors, or arrays of any of these. It is a compile-time error to declare any double-precision type, matrix, or structure as an output. Fragment outputs are declared as in the following examples:

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

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

4.3.7 Buffer Variables

The `buffer` qualifier is used to declare global variables whose values are stored in the data store of a buffer object bound through the OpenGL API. Buffer variables can be read and written with the underlying storage shared among all active shader invocations. Buffer variable memory reads and writes within a single shader invocation are processed in order. However, the order of reads and writes performed in one invocation relative to those performed by another invocation is largely undefined. Buffer variables may be qualified with memory qualifiers affecting how the underlying memory is accessed, as described in section 4.10 “Memory Qualifiers”.

The `buffer` 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.

Buffer variables may only be declared inside interface blocks (section 4.3.9 “Interface Blocks”), which are then referred to as shader storage blocks. It is a compile-time error to declare buffer variables at global scope (outside a block). Buffer variables cannot have initializers.

There are implementation-dependent limits on the number of shader storage blocks used for each type of shader, the combined number of shader storage blocks used for a program, and the amount of storage required by each individual shader storage block. If any of these limits are exceeded, it will cause a compile-time or link-time error.
If multiple shaders are linked together, then they will share a single global buffer variable name space, including within a language as well as across languages. Hence, the types of buffer variables with the same name must match across all shaders that are linked into a single program.

### 4.3.8 Shared Variables

The `shared` qualifier is used to declare variables that have storage shared between all work items compute shader local work group. Variables declared as `shared` may only be used in compute shaders (see section 2.6 “Compute Processor”). Shared variables are implicitly coherent. That is, writes to shared variables from one shader invocation will eventually be seen by other invocations within the same local work group.

Variables declared as `shared` may not have initializers and their contents are undefined at the beginning of shader execution. Any data written to shared variables will be visible to other shaders executing the same shader within the same local work group. Order of execution with respect to reads and writes to the same shared variable by different invocations of a shader is not defined. In order to achieve ordering with respect to reads and writes to shared variables, memory barriers must be employed using the `barrier()` function (see section 8.16 “Shader Invocation Control Functions”).

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

### 4.3.9 Interface Blocks

Input, output, uniform, and buffer variable declarations can be grouped into named interface blocks to provide coarser granularity backing than is achievable with individual declarations. They can have an optional instance name, used in the shader to reference their members. An output block of one programmable stage is backed by a corresponding input block in the subsequent programmable stage. A uniform block is backed by the application with a buffer object. A block of buffer variables, called a shader storage block, is also backed by the application with a buffer object. It is a compile-time error to have an input block in a vertex shader or an output block in a fragment shader; these uses are reserved for future use.
An interface block is started by an \textbf{in}, \textbf{out}, \textbf{uniform}, or \textbf{buffer} keyword, followed by a block name, followed by an open curly brace (\{\}) as follows:

\begin{verbatim}
interface-block :
  layout-qualifier\_opt interface-qualifier block-name \{ member-list \} instance-name\_opt ;

interface-qualifier :
  in
  out
  uniform
  buffer

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

member-declaration :
  layout-qualifier\_opt qualifiers\_opt type declarators ;

instance-name :
  identifier
  identifier [ ]
  identifier [ integral-constant-expression ]
\end{verbatim}

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

First, an example,

\begin{verbatim}
uniform Transform {
  mat4 ModelViewMatrix;
  mat4 ModelViewProjectionMatrix;
  uniform mat3 NormalMatrix;  // allowed restatement of qualifier
  float Deformation;
};
\end{verbatim}

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

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

\begin{itemize}
  \item initializers are not allowed
  \item opaque types are not allowed
  \item structure definitions cannot be nested inside a block
\end{itemize}

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

If no optional qualifier is used in a member-declaration, the qualification of the variable is just \textbf{in}, \textbf{out}, \textbf{uniform}, or \textbf{buffer} as determined by \textit{interface-qualifier}. If optional qualifiers are used, they can include interpolation qualifiers, auxiliary storage qualifiers, and storage qualifiers and they must declare an input,
output, or uniform variable consistent with the interface qualifier of the block: Input variables, output
variables, uniform variables, and buffer variables can only be in in blocks, out blocks, uniform blocks,
and shader storage blocks, respectively. Repeating the in, out, uniform, or buffer interface qualifier for
a member's storage qualifier is optional. For example,

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

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

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

The block name (block-name) is used to match interfaces: an output block of one pipeline stage will be
matched to an input block with the same name in the subsequent pipeline stage. For uniform blocks, the
application uses the block name to identify the block. Block names have no other use within a shader
beyond interface matching; it is a compile-time error to use a block name at global scope for anything
other than as a block name (e.g., use of a block name for a global variable name or function name is
currently reserved). Matched block names within an interface (as defined above) must match in terms of
having the same number of declarations with the same sequence of types and the same sequence of
member names, as well as having the same member-wise layout qualification (see next section).
Furthermore, if a matching block is declared as an array, then the array sizes must also match (or follow
array matching rules for the interface between a vertex and a geometry shader). Any mismatch will
generate a link-time error. A block name is allowed to have different definitions in different interfaces
within the same shader, allowing, for example, an input block and output block to have the same name.
If an instance name (instance-name) is not used, the names declared inside the block are scoped at the global level and accessed as if they were declared outside the block. If an instance name (instance-name) is used, then it puts all the members inside a scope within its own name space, accessed with the field selector (.) operator (analogously to structures). For example,

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

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

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

out Vertex2 {
    vec4 Color; // API will use “Color”
};
```
4 Variables and Types

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

```cpp
uniform Transform {  // API uses "Transform[2]" to refer to instance 2
    mat4           ModelViewMatrix;
    mat4           ModelViewProjectionMatrix;
    vec4           a[];  // array will get implicitly sized
    float          Deformation;
} transforms[4];
...
... = transforms[2].ModelViewMatrix;  // shader access of instance 2
// API uses "Transform.ModelViewMatrix" to query an offset or other query
transforms[x].a.length(); // same length for 'a' for all x
Transform[x];       // illegal, must use 'transforms'
Transform.a.length(); // illegal, must use 'transforms'
...transforms[2].a[3]...  // if these are the only two dereferences of 'a',
...transforms[3].a[7]...  // then 'a' must be size 8, for all transforms[x]
```

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

When using OpenGL API entry points to identify the name of an individual block in an array of blocks, the name string must include an array index (e.g., Transform[2]). When using OpenGL API entry points to refer to offsets or other characteristics of a block member, an array index must not be specified (e.g., Transform.ModelViewMatrix).

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

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

4.4 Layout Qualifiers

Layout qualifiers can appear in several forms of declaration. They can appear as part of an interface block definition or block member, as shown in the grammar in the previous section. They can also appear with just an interface qualifier (a storage qualifier that is `in`, `out`, or `uniform`) to establish layouts of other declarations made with that interface qualifier:

```
layout-qualifier interface-qualifier
```

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

```
layout-qualifier interface-qualifier declaration
```

Declarations of layouts can only be made at global scope, and only where indicated in the following subsections; their details are specific to what the interface qualifier is, and are discussed individually.
The \textit{layout-qualifier} expands to

\[
\text{layout-qualifier} : \\
\quad \text{layout} ( \text{layout-qualifier-id-list} ) \\
\text{layout-qualifier-id-list} : \\
\quad \text{layout-qualifier-id} \\
\quad \text{layout-qualifier-id} , \text{layout-qualifier-id-list} \\
\text{layout-qualifier-id} : \\
\quad \text{layout-qualifier-name} \\
\quad \text{layout-qualifier-name} = \text{layout-qualifier-value} \\
\quad \text{shared}
\]

The tokens used for \textit{layout-qualifier-name} are identifiers, not keywords, however, the \texttt{shared} keyword is allowed as a \textit{layout-qualifier-id}. Generally, they can be listed in any order. Order-dependent meanings exist only if explicitly called out below. Similarly, these identifiers are not case sensitive, unless explicitly noted otherwise.

More than one layout qualifier may appear in a single declaration. If the same \textit{layout-qualifier-name} occurs in multiple layout qualifiers for the same declaration, the last one overrides the former ones.

### 4.4.1 Input Layout Qualifiers

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

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

\[
\text{layout-qualifier-id} \\
\quad \text{location} = \text{integer-constant}
\]

Only one argument is accepted. For example,

\[
\text{layout(location = 3) in vec4 normal;}
\]

will establish that the shader input \texttt{normal} is assigned to vector location number 3. For vertex shader inputs, the location specifies the number of the generic vertex attribute from which input values are taken. For inputs of all other shader types, the location specifies a vector number that can be used to match against outputs from a previous shader stage, even if that shader is in a different program object.

If a vertex shader input is any scalar or vector type, it will consume a single location. If a non-vertex shader input is a scalar or vector type other than \texttt{dvec3} or \texttt{dvec4}, it will consume a single location, while types \texttt{dvec3} or \texttt{dvec4} will consume two consecutive locations. Inputs of type \texttt{double} and \texttt{dvec2} will consume only a single location, in all stages.

If the declared input is an array of size \(n\) and each element takes \(m\) locations, it will be assigned \(m \times n\) consecutive locations starting with the location specified. For example,

\[
\text{layout(location = 6) in vec4 colors[3]};
\]

will establish that the shader input \texttt{colors} is assigned to vector location numbers 6, 7, and 8.
If the declared input is an $n \times m$ single- or double-precision matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an $n$-element array of $m$-component vectors. For example,

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

will establish that shader input `transforms` is assigned to vector locations 9-16, with `transforms[0]` being assigned to locations 9-12 and `transforms[1]` being assigned to locations 13-16.

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

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

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

Location layout qualifiers may be used on input variables declared as structures, but not on individual members. Location layout qualifiers may not be used on input blocks or input block members. Compile-time errors result if these rules are not followed.

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

A program will fail to link if any two non-vertex shader input variables are assigned to the same location. For vertex shaders, multiple input variables may be assigned to the same location using either layout qualifiers or via the OpenGL API. However, such aliasing is intended only to support vertex shaders where each execution path accesses at most one input per each location. Implementations are permitted, but not required, to generate link-time errors if they detect that every path through the vertex shader executable accesses multiple inputs assigned to any single location. For all shader types, a program will fail to link if explicit location assignments leave the linker unable to find space for other variables without explicit assignments.

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

If a vertex shader input variable with no location assigned in the shader text has a location specified through the OpenGL API, the API-assigned location will be used. Otherwise, such variables will be assigned a location by the linker. See section 11.1.1 “Vertex Attributes” of the OpenGL Graphics System Specification for more details. A link-time error will occur if an input variable is declared in multiple shaders of the same language with conflicting locations.
4.4.1.1 Tessellation Evaluation Inputs

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

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

One subset of these identifiers, **primitive mode**, is used to specify a tessellation primitive mode to be used by the tessellation primitive generator. To specify a primitive mode, the identifier must be one of **triangles**, **quads**, or **isolines**, which specify that the tessellation primitive generator should subdivide a triangle into smaller triangles, a quad into triangles, or a quad into a collection of lines, respectively.

A second subset of these identifiers, **vertex spacing**, is used to specify the spacing used by the tessellation primitive generator when subdividing an edge. To specify vertex spacing, the identifier must be one of the following.

- **equal_spacing** signifying that edges should be divided into a collection of equal-sized segments.
- **fractional_even_spacing** signifying that edges should be divided into an even number of equal-length segments plus two additional shorter "fractional" segments.
- **fractional_odd_spacing** signifying that edges should be divided into an odd number of equal-length segments plus two additional shorter "fractional" segments.

A third subset of these identifiers, **ordering**, specifies whether the tessellation primitive generator produces triangles in clockwise or counter-clockwise order, according to the coordinate system depicted in the OpenGL specification. The ordering identifiers **cw** and **ccw** indicate clockwise and counter-clockwise triangles, respectively. If the tessellation primitive generator does not produce triangles, ordering is ignored.

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

Any or all of these identifiers may be specified one or more times in a single input layout declaration. If primitive mode, vertex spacing, or ordering is declared more than once in the tessellation evaluation shaders of a program, all such declarations must use the same identifier.

At least one tessellation evaluation shader (compilation unit) in a program must declare a primitive mode in its input layout. Declaring vertex spacing, ordering, or point mode identifiers is optional. It is not required that all tessellation evaluation shaders in a program declare a primitive mode. If spacing or vertex ordering declarations are omitted, the tessellation primitive generator will use equal spacing or
counter-clockwise vertex ordering, respectively. If a point mode declaration is omitted, the tessellation primitive generator will produce lines or triangles according to the primitive mode.

### 4.4.1.2 Geometry Shader Inputs

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

```
layout-qualifier-id
  points
  lines
  lines_adjacency
  triangles
  triangles_adjacency
  invocations = integer-constant
```

The identifiers `points`, `lines`, `lines_adjacency`, `triangles`, and `triangles_adjacency` are used to specify the type of input *primitive* accepted by the geometry shader, and only one of these is accepted. At least one geometry shader (compilation unit) in a program must declare this input primitive layout, and all geometry shader input layout declarations in a program must declare the same layout. It is not required that all geometry shaders in a program declare an input primitive layout.

The identifier `invocations` is used to specify the number of times the geometry shader executable is invoked for each input primitive received. Invocation count declarations are optional. If no invocation count is declared in any geometry shader in a program, the geometry shader will be run once for each input primitive. If an invocation count is declared, all such declarations must specify the same count. If a shader specifies an invocation count greater than the implementation-dependent maximum, it will fail to compile.

For example,

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

will establish that all inputs to the geometry shader are triangles and that the geometry shader executable is run six times for each triangle processed.

All geometry shader input unsized array declarations will be sized by an earlier input primitive layout qualifier, when present, as per the following table.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Size of Input Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>points</td>
<td>1</td>
</tr>
<tr>
<td>lines</td>
<td>2</td>
</tr>
<tr>
<td>lines_adjacency</td>
<td>4</td>
</tr>
<tr>
<td>triangles</td>
<td>3</td>
</tr>
<tr>
<td>triangles_adjacency</td>
<td>6</td>
</tr>
</tbody>
</table>
The intrinsically declared input array `gl_in[]` will also be sized by any input primitive-layout declaration. Hence, the expression

```c
gl_in.length()
```

will return the value from the table above.

For inputs declared without an array size, including intrinsically declared inputs (i.e., `gl_in`), a layout must be declared before any use of the method `length` or other any array use that requires the array size to be known.

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

```c
// code sequence within one shader...
in vec4 Color1[];    // legal, size still unknown
in vec4 Color2[2];   // legal, size is 2
in vec4 Color3[3];   // illegal, input sizes are inconsistent
layout(lines) in;    // legal for Color2, input size is 2, matching Color2
in vec4 Color4[3];   // illegal, contradicts layout of lines
layout(lines) in;    // legal, matches other layout() declaration
layout(triangles) in; // illegal, does not match earlier layout() declaration
```

It is a link-time error if not all provided sizes (sized input arrays and layout size) match across all geometry shaders in a program.

### 4.4.1.3 Fragment Shader Inputs

Additional fragment layout qualifier identifiers include the following for `gl_FragCoord`

```c
layout-qualifier-id
    origin_upper_left
    pixel_center_integer
```

By default, `gl_FragCoord` assumes a lower-left origin for window coordinates and assumes pixel centers are located at half-pixel coordinates. For example, the \((x, y)\) location \((0.5, 0.5)\) is returned for the lower-left-most pixel in a window. The origin can be changed by redeclaring `gl_FragCoord` with the `origin_upper_left` identifier, moving the origin of `gl_FragCoord` to the upper left of the window, with \(y\) increasing in value toward the bottom of the window. The values returned can also be shifted by half a pixel in both \(x\) and \(y\) by `pixel_center_integer` so it appears the pixels are centered at whole number pixel offsets. This moves the \((x, y)\) value returned by `gl_FragCoord` of \((0.5, 0.5)\) by default, to \((0.0, 0.0)\) with `pixel_center_integer`.  

57
4 Variables and Types

Redeclarations are done as follows

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

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

If `gl_FragCoord` is redeclared in any fragment shader in a program, it must be redeclared in all the fragment shaders in that program that have a static use `gl_FragCoord`. All redeclarations of `gl_FragCoord` in all fragment shaders in a single program must have the same set of qualifiers. Within any shader, the first redeclarations of `gl_FragCoord` must appear before any use of `gl_FragCoord`. The built-in `gl_FragCoord` is only predeclared in fragment shaders, so redeclaring it in any other shader language results in a compile-time error.

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

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

```glsl
layout-qualifier-id
  early_fragment_tests
```

to request that fragment tests be performed before fragment shader execution, as described in section 15.2.4 “Early Fragment Tests” of the OpenGL Specification.

For example,

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

Specifying this will make per-fragment tests be performed before fragment shader execution. If this is not declared, per-fragment tests will be performed after fragment shader execution. Only one fragment shader (compilation unit) need declare this, though more than one can. If at least one declares this, then it is enabled.

4.4.1.4 Compute Shader Inputs

There are no layout location qualifiers for compute shader inputs.

Layout qualifier identifiers for compute shader inputs are the work-group size qualifiers:

```glsl
layout-qualifier-id
  local_size_x = integer-constant
  local_size_y = integer-constant
  local_size_z = integer-constant
```

Where `local_size_x`, `local_size_y`, and `local_size_z` are used to define the local size of the kernel defined by the compute shader in the first, second, and third dimension, respectively. The default size in each dimension is 1. If a shader does not specify a size for one of the dimensions, that dimension will have a size of 1.
For example, the following declaration in a compute shader

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

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

As another example, the declaration

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

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

If the local size of the shader in any dimension is greater than the maximum size supported by the implementation for that dimension, a compile-time error results. Also, if such a layout qualifier is declared more than once in the same shader, all those declarations must indicate the same local workgroup size; otherwise a compile-time error results. If multiple compute shaders attached to a single program object declare local workgroup size, the declarations must be identical; otherwise a link-time error results.

Furthermore, if a program object contains any compute shaders, at least one must contain an input layout qualifier specifying the local work sizes of the program, or a link-time error will occur.

### 4.4.2 Output Layout Qualifiers

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

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

```glsl
layout-qualifier-id
  location = integer-constant
```

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

```glsl
layout-qualifier-id
  location = integer-constant
  index = integer-constant
```

Each of these qualifiers may appear at most once. If `index` is specified, `location` must also be specified. If `index` is not specified, the value 0 is used. For example, in a fragment shader,

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

will establish that the fragment shader output `color` is assigned to fragment color 3 as the first (index zero) input to the blend equation. And,

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

will establish that the fragment shader output `factor` is assigned to fragment color 3 as the second (index one) input to the blend equation.
For fragment-shader outputs, the location and index specify the color output number and index receiving the values of the output. For outputs of all other shader stages, the location specifies a vector number that can be used to match against inputs in a subsequent shader stage, even if that shader is in a different program object.

If a declared output is a scalar or vector type other than \texttt{dvec3} or \texttt{dvec4}, it will consume a single location. Outputs of type \texttt{dvec3} or \texttt{dvec4} will consume two consecutive locations. Outputs of type \texttt{double} and \texttt{dvec2} will consume only a single location, in all stages.

If the declared output is an array, it will be assigned consecutive locations starting with the location specified. For example,

\begin{verbatim}
layout(location = 2) out vec4 colors[3];
\end{verbatim}

will establish that \textit{colors} is assigned to vector location numbers 2, 3, and 4.

If the declared output is an $n \times m$ single- or double-precision matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned will be the same as for an $n$-element array of $m$-component vectors.

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

Location layout qualifiers may be used on output variables declared as structures, but not on individual members. Location layout qualifiers may not be used on output blocks or output block members. Compile-time errors result if these rules are not followed.

The number of output locations available to a shader is limited. For fragment shaders, the limit is the advertised number of draw buffers. For all other shaders, the limit is implementation-dependent and must be no less than one fourth of the advertised maximum output component count. (Compute shaders have no outputs.) A program will fail to link if any attached shader uses a location greater than or equal to the number of supported locations, unless device-dependent optimizations are able to make the program fit within available hardware resources. Compile-time errors may also be given if at compile time it is known the link will fail. A negative output location will result in a compile-time error. It is also a compile-time error if a fragment shader sets a layout index to less than 0 or greater than 1.

A program will fail to link if any two fragment shader output variables are assigned to the same location and index, or if any two output variables from the same non-fragment shader stage are assigned to the same location. For fragment shader outputs, locations can be assigned using either a layout qualifier or via the OpenGL API. For all shader types, a program will fail to link if explicit location assignments leave the linker unable to find space for other variables without explicit assignments.

If an output variable with no location or index assigned in the shader text has a location specified through the OpenGL API, the API-assigned location will be used. Otherwise, such variables will be assigned a location by the linker. All such assignments will have a color index of zero. See section 15.2 “Shader Execution” of the OpenGL Graphics System Specification for more details. A link-time error will occur if an output variable is declared in multiple shaders of the same language with conflicting location or index values.
For the purposes of determining if a non-fragment output matches an input from a subsequent shader stage, the location layout qualifier (if any) must match.

4.4.2.1 Tessellation Control Outputs

Tessellation control shaders allow output layout qualifiers only on the interface qualifier `out`, not on an output block, block member, or variable declaration. The output layout qualifier identifiers allowed for tessellation control shaders include the vertex-count layout qualifier:

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

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

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

```
gl_out.length()
```

will return the output patch vertex count specified in a previous output layout qualifier. For outputs declared without an array size, including intrinsically declared outputs (i.e., `gl_out[]`), a layout must be must be declared before any use of the method `length()` or other array use requires its size be known.

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

All tessellation control shader layout declarations in a program must specify the same output patch vertex count. There must be at least one layout qualifier specifying an output patch vertex count in any program containing tessellation control shaders; however, such a declaration is not required in all tessellation control shaders.

4.4.2.2 Geometry Outputs

Geometry shaders can have three additional types of output layout identifiers: an output `primitive type`, a maximum output `vertex count`, and per-output `stream` numbers. The primitive type and vertex count identifiers are allowed only on the interface qualifier `out`, not on an output block, block member, or variable declaration. The stream identifier is allowed on the interface qualifier `out`, on output blocks, and on variable declarations.

The layout qualifier identifiers for geometry shader outputs are

```
layout-qualifier-id
  points
  line_strip
  triangle_strip
  max_vertices = integer-constant
  stream = integer-constant
```
The primitive type identifiers `points`, `line_strip`, and `triangle_strip` are used to specify the type of output primitive produced by the geometry shader, and only one of these is accepted. At least one geometry shader (compilation unit) in a program must declare an output primitive type, and all geometry shader output primitive type declarations in a program must declare the same primitive type. It is not required that all geometry shaders in a program declare an output primitive type.

The vertex count identifier `max_vertices` is used to specify the maximum number of vertices the shader will ever emit in a single invocation. At least one geometry shader (compilation unit) in a program must declare a maximum output vertex count, and all geometry shader output vertex count declarations in a program must declare the same count. It is not required that all geometry shaders in a program declare a count.

In this example,

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

all outputs from the geometry shader are triangles and at most 60 vertices will be emitted by the shader. It is an error for the maximum number of vertices to be greater than `gl_MaxGeometryOutputVertices`.

The identifier `stream` is used to specify that a geometry shader output variable or block is associated with a particular vertex stream (numbered beginning with zero). A default stream number may be declared at global scope by qualifying interface qualifier `out` as in this example:

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

The stream number specified in such a declaration replaces any previous default and applies to all subsequent block and variable declarations until a new default is established. The initial default stream number is zero.
Each output block or non-block output variable is associated with a vertex stream. If the block or variable is declared with the stream identifier, it is associated with the specified stream; otherwise, it is associated with the current default stream. A block member may be declared with a stream identifier, but the specified stream must match the stream associated with the containing block. One example:

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

Each vertex emitted by the geometry shader is assigned to a specific stream, and the attributes of the emitted vertex are taken from the set of output blocks and variables assigned to the targeted stream. After each vertex is emitted, the values of all output variables become undefined. Additionally, the output variables associated with each vertex stream may share storage. Writing to an output variable associated with one stream may overwrite output variables associated with any other stream. When emitting each vertex, a geometry shader should write to all outputs associated with the stream to which the vertex will be emitted and to no outputs associated with any other stream.

If a geometry shader output block or variable is declared more than once, all such declarations must associate the variable with the same vertex stream. If any stream declaration specifies a non-existent stream number, the shader will fail to compile.

Built-in geometry shader outputs are always associated with vertex stream zero.

All geometry shader output layout declarations in a program must declare the same layout and same value for `max_vertices`. If geometry shaders are in a program, there must be at least one geometry output layout declaration somewhere in that program, but not all geometry shaders (compilation units) are required to declare it.

### 4.4.2.3 Fragment Outputs

The built-in fragment shader variable `gl_FragDepth` may be redeclared using one of the following layout qualifiers.

```
layout-qualifier-id
    depth_any
    depth_greater
    depth_less
    depth_unchanged
```
For example:

    layout (depth_greater) out float gl_FragDepth;

The layout qualifier for \texttt{gl\_FragDepth} constrains intentions of the final value of \texttt{gl\_FragDepth} written by any shader invocation. GL implementations are allowed to perform optimizations assuming that the depth test fails (or passes) for a given fragment if all values of \texttt{gl\_FragDepth} consistent with the layout qualifier would fail (or pass). If the final value of \texttt{gl\_FragDepth} is inconsistent with its layout qualifier, the result of the depth test for the corresponding fragment is undefined. However, no error will be generated in this case. If the depth test passes and depth writes are enabled, the value written to the depth buffer is always the value of \texttt{gl\_FragDepth}, whether or not it is consistent with the layout qualifier.

By default, \texttt{gl\_FragDepth} is qualified as \texttt{depth\_any}. When the layout qualifier for \texttt{gl\_FragDepth} is \texttt{depth\_any}, the shader compiler will note any assignment to \texttt{gl\_FragDepth} modifying it in an unknown way, and depth testing will always be performed after the shader has executed. When the layout qualifier is \texttt{depth\_greater}, the GL can assume that the final value of \texttt{gl\_FragDepth} is greater than or equal to the fragment's interpolated depth value, as given by the z component of \texttt{gl\_FragCoord}. When the layout qualifier is \texttt{depth\_less}, the GL can assume that any modification of \texttt{gl\_FragDepth} will only decrease its value. When the layout qualifier is \texttt{depth\_unchanged}, the shader compiler will honor any modification to \texttt{gl\_FragDepth}, but the rest of the GL can assume that \texttt{gl\_FragDepth} is not assigned a new value.

Redeclarations of \texttt{gl\_FragDepth} are performed as follows:

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

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

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

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

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

If \texttt{gl\_FragDepth} is redeclared in any fragment shader in a program, it must be redeclared in all fragment shaders in that program that have static assignments to \texttt{gl\_FragDepth}. All redeclarations of \texttt{gl\_FragDepth} in all fragment shaders in a single program must have the same set of qualifiers. Within any shader, the first redeclarations of \texttt{gl\_FragDepth} must appear before any use of \texttt{gl\_FragDepth}. The built-in \texttt{gl\_FragDepth} is only predeclared in fragment shaders, so redeclaring it in any other shader language results in a compile-time error.

### 4.4.3 Uniform Variable Layout Qualifiers

Layout qualifiers can be used for uniform variables and subroutine uniforms. The layout qualifier identifiers for uniform variables and subroutine uniforms are:
The location identifier can be used with default-block uniform variables and subroutine uniforms. The location specifies the location by which the OpenGL API can reference the uniform and update its value. Individual elements of a uniform array are assigned consecutive locations with the first element taking location \textit{location}. No two default-block uniform variables in the program can have the same location, even if they are unused, otherwise a compile-time or link-time error will be generated. No two subroutine uniform variables can have the same location in the same shader stage, otherwise a compile-time or link-time error will be generated. Valid locations for default-block uniform variable locations are in the range of 0 to the implementation-defined maximum number of uniform locations minus one. Valid locations for subroutine uniforms are in the range of 0 to the implementation-defined per-stage maximum number of subroutine uniform locations minus one.

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

### 4.4.4 Subroutine Function Layout Qualifiers

Layout qualifiers can be used for subroutine functions. The layout qualifier identifiers for subroutine functions are:

\[
\text{layout-qualifier-id} \\
\text{index} = \text{integer-constant}
\]

Each subroutine with an index qualifier in the shader must be given a unique index, otherwise a compile- or link-time error will be generated. The indices must be in the range of 0 to the implementation defined maximum number of subroutines minus one. It is recommended, but not required, that the shader assigns a range of tightly packed \textit{index} values starting from zero so that the OpenGL subroutine function enumeration API returns a non-empty name for all active indices.
### 4.4.5 Uniform and Shader Storage Block Layout Qualifiers

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

```
layout-qualifier-id
shared
packed
std140
std430
row_major
column_major
binding = integer-constant
```

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

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

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

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

and for shader storage blocks as

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

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

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

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

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

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

The `shared` qualifier overrides only the `std140`, `std430`, and `packed` qualifiers; other qualifiers are inherited. The compiler/linker will ensure that multiple programs and programmable stages containing this definition will share the same memory layout for this block, as long as all arrays are declared with explicit sizes and all matrices have matching `row_major` and/or `column_major` qualifications (which may...
come from a declaration outside the block definition). This allows use of the same buffer to back the same block definition across different programs.

The packed qualifier overrides only std140, std430, and shared; other qualifiers are inherited. When packed is used, no shareable layout is guaranteed. The compiler and linker can optimize memory use based on what variables actively get used and on other criteria. Offsets must be queried, as there is no other way of guaranteeing where (and which) variables reside within the block. Attempts to share a packed uniform or shader storage block across programs or stages will generally fail. However, implementations may aid application management of packed blocks by using canonical layouts for packed blocks.

The std140 and std430 qualifiers override only the packed, shared, std140, and std430 qualifiers; other qualifiers are inherited. The std430 qualifier is supported only for shader storage blocks; using std430 on a uniform block will result in a compile-time error. The layout is explicitly determined by this, as described in section 7.6.2 “Uniform Blocks” under Standard Uniform Block Layout of the OpenGL Graphics System Specification. Hence, as in shared above, the resulting layout is shareable across programs.

Layout qualifiers on member declarations cannot use the shared, packed, std140, or std430 qualifiers. These can only be used at global scope or on a block declaration, or a compile-time error results.

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

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

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

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

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

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

For example

```cpp
layout(row_major, column_major)
```
results in the qualification being `column_major`. Other examples:

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

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

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

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

### 4.4.6 Opaque-Uniform Layout Qualifiers

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

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

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

```cpp
layout-qualifier-id
    binding = integer-constant
```

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

If the `binding` identifier is used with an array, the first element of the array takes the specified unit and each subsequent element takes the next consecutive unit.

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

A link-time error will result if two compilation units in a program specify different `integer-constant` bindings for the same opaque-uniform name. However, it is not an error to specify a binding on some but not all declarations for the same name, as shown in the examples below.
4 Variables and Types

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

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

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

4.4.6.1 Atomic Counter Layout Qualifiers

The atomic counter qualifiers are

\[\text{layout-qualifier-id}\]
\[\text{binding} = \text{integer-constant}\]
\[\text{offset} = \text{integer-constant}\]

For example,

\[\text{layout (binding = 2, offset = 4) uniform atomic_uint a;}\]

will establish that the opaque handle to the atomic counter \(a\) will be bound to atomic counter buffer binding point 2 at an offset of 4 basic machine units into that buffer. The default offset for binding point 2 will be post incremented by 4 (the size of an atomic counter).

A subsequent atomic counter declaration will inherit the previous (post incremented) offset. For example, a subsequent declaration of

\[\text{layout (binding = 2) uniform atomic_uint bar;}\]

will establish that the atomic counter \(bar\) has a binding to buffer binding point 2 at an offset of 8 basic machine units into that buffer. The offset for binding point 2 will again be post-incremented by 4 (the size of an atomic counter).

When multiple variables are listed in a layout declaration, the effect will be the same as if they were declared one at a time, in order from left to right.

Binding points are not inherited, only offsets. Each binding point tracks its own current default offset for inheritance of subsequent variables using the same binding. The initial state of compilation is that all binding points have an offset of 0. The offset can be set per binding point at global scope (without declaring a variable). For example,

\[\text{layout (binding = 2, offset = 4) uniform atomic_uint;}\]

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

\[\text{layout (binding = 2) uniform atomic_uint bar;} // \text{offset is 4}\]
\[\text{layout (offset = 8) uniform atomic_uint bar;} // \text{error, no default binding}\]

Atomic counters may share the same binding point, but if a binding is shared, their offsets must be either explicitly or implicitly (from inheritance) unique and non overlapping.
Example valid uniform declarations, assuming top of shader:

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

Example of an invalid uniform declaration:

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

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

### 4.4.6.2 Format Layout Qualifiers

Format layout qualifiers can be used on image variable declarations (those declared with a basic type having “image” in its keyword). The format layout qualifier identifiers for image variable declarations are:

```glsl
layout-qualifier-id
  float-image-format-qualifier
  int-image-format-qualifier
  uint-image-format-qualifier
  binding = integer-constant

float-image-format-qualifier
  rgba32f
  rgba16f
  rg32f
  rg16f
  r11f_g11f_b10f
  r32f
  r16f
  rgba16
  rgb10_a2
  rgba8
  rg16
  rg8
  r16
  r8
  rgba16_snorm
  rgba8_snorm
  rg16_snorm
  rg8_snorm
```
A format layout qualifier specifies the image format associated with a declared image variable. Only one format qualifier may be specified for any image variable declaration. For image variables with floating-point component types (keywords starting with “image”), signed integer component types (keywords starting with “iimage”), or unsigned integer component types (keywords starting with “uimage”), the format qualifier used must match the `float-image-format-qualifier`, `int-image-format-qualifier`, or `uint-image-format-qualifier` grammar rules, respectively. It is a compile-time error to declare an image variable where the format qualifier does not match the image variable type.

Any image variable used for image loads or atomic operations must specify a format layout qualifier; it is a compile-time error to pass an image uniform variable or function parameter declared without a format layout qualifier to an image load or atomic function.

The `binding` identifier was described in section 4.4.5 “Uniform and Shader Storage Block Layout Qualifiers”.

Uniforms not qualified with `writeonly` must have a format layout qualifier. Note that an image variable passed to a function for read access cannot be declared as `writeonly` and hence must have been declared with a format layout qualifier.
4.5 Interpolation Qualifiers

Inputs and outputs that could be interpolated can be further qualified by at most one of the following interpolation qualifiers:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>smooth</td>
<td>perspective correct interpolation</td>
</tr>
<tr>
<td>flat</td>
<td>no interpolation</td>
</tr>
<tr>
<td>noperspective</td>
<td>linear interpolation</td>
</tr>
</tbody>
</table>

The presence of and type of interpolation is controlled by the above interpolation qualifiers as well as the auxiliary storage qualifiers centroid and sample. The auxiliary storage qualifier patch is not used for interpolation; it is a compile-time error to use interpolation qualifiers with patch.

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

A variable qualified as smooth will be interpolated in a perspective-correct manner over the primitive being rendered. Interpolation in a perspective correct manner is specified in equation 14.7 in the OpenGL Graphics System Specification, section 14.5 “Line Segments”.

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

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

When multisample rasterization is enabled, centroid and sample may be used to control the location and frequency of the sampling of the qualified fragment shader input. If a fragment shader input is qualified with centroid, a single value may be assigned to that variable for all samples in the pixel, but that value must be interpolated to a location that lies in both the pixel and in the primitive being rendered, including any of the pixel's samples covered by the primitive. Because the location at which the variable is interpolated may be different in neighboring pixels, and derivatives may be computed by computing differences between neighboring pixels, derivatives of centroid-sampled inputs may be less accurate than those for non-centroid interpolated variables. If a fragment shader input is qualified with sample, a separate value must be assigned to that variable for each covered sample in the pixel, and that value must be sampled at the location of the individual sample.

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

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

4.5.1 Redeclaring Built-in Interpolation Variables in the Compatibility Profile

The following predeclared variables can be redeclared with an interpolation qualifier when using the compatibility profile:

Vertex, tessellation control, tessellation evaluation, and geometry languages:

- gl_FrontColor
- gl_BackColor
- gl_FrontSecondaryColor
- gl_BackSecondaryColor

Fragment language:

- gl_Color
- gl_SecondaryColor

For example,

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

Ideally, these are redeclared as part of the redeclaration of an interface block, as described in section 7.1.1 “Compatibility Profile Built-In Language Variables”. However, for the above purpose, they can be redeclared as individual variables at global scope, outside an interface block. A compile-time error will result if a shader has both an interface block redeclaration and a separate redeclaration of a member of that interface block outside the interface block redeclaration.

If gl_Color is redeclared with an interpolation qualifier, then gl_FrontColor and gl_BackColor (if they are written to) must also be redeclared with the same interpolation qualifier, and vice versa. If gl_SecondaryColor is redeclared with an interpolation qualifier, then gl_FrontSecondaryColor and gl_BackSecondaryColor (if they are written to) must also be redeclared with the same interpolation qualifier, and vice versa. This qualifier matching on predeclared variables is only required for variables that are statically used within the shaders in a program.
4.6 Parameter Qualifiers

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

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; none: default &gt;</td>
<td>same is in</td>
</tr>
<tr>
<td>const</td>
<td>for function parameters that cannot be written to</td>
</tr>
<tr>
<td>in</td>
<td>for function parameters passed into a function</td>
</tr>
<tr>
<td>out</td>
<td>for function parameters passed back out of a function, but not initialized for use when passed in</td>
</tr>
<tr>
<td>inout</td>
<td>for function parameters passed both into and out of a function</td>
</tr>
</tbody>
</table>

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

4.7 Precision and Precision Qualifiers

Precision qualifiers are added for code portability with OpenGL ES, not for functionality. They have the same syntax as in OpenGL ES, as described below, but they have no semantic meaning, which includes no effect on the precision used to store or operate on variables.

If an extension adds in the same semantics and functionality in the OpenGL ES 2.0 specification for precision qualifiers, then the extension is allowed to reuse the keywords below for that purpose.

For the purposes of determining if an output from one shader stage matches an input of the next stage, the precision qualifier need not match.

4.7.1 Range and Precision

The precision of stored single- and double-precision floating-point variables is defined by the IEEE 754 standard for 32-bit and 64-bit floating-point numbers. This includes support for NaNs (Not a Number) and Infs (positive or negative infinities).

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

Precisions are expressed in terms of maximum relative error in units of ULP (units in the last place), unless otherwise noted.
For single precision operations, precisions are required as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a + b), (a - b), (a \times b)</td>
<td>Correctly rounded.</td>
</tr>
<tr>
<td>(&lt;), (\leq), (==), (&gt;), (&gt;=)</td>
<td>Correct result.</td>
</tr>
<tr>
<td>(a / b), (1.0 / b)</td>
<td>2.5 ULP for (b) in the range ([2^{-126}, 2^{126}]).</td>
</tr>
<tr>
<td>(a \times b + c)</td>
<td>Correctly rounded single operation or sequence of two correctly rounded operations.</td>
</tr>
<tr>
<td>(\text{fma}())</td>
<td>Inherited from (a \times b + c).</td>
</tr>
<tr>
<td>(\text{pow}(x, y))</td>
<td>Inherited from (\text{exp2}(x \times \text{log2}(y))).</td>
</tr>
<tr>
<td>(\text{exp}(x), \text{exp2}(x))</td>
<td>((3 + 2 \times</td>
</tr>
<tr>
<td>(\text{log}(), \text{log2}())</td>
<td>3 ULP outside the range ([0.5, 2.0]).</td>
</tr>
<tr>
<td></td>
<td>Absolute error (&lt; 2^{-21}) inside the range ([0.5, 2.0]).</td>
</tr>
<tr>
<td>(\text{sqrt}())</td>
<td>Inherited from (1.0 / \text{inversesqrt}()).</td>
</tr>
<tr>
<td>(\text{inversesqrt}())</td>
<td>2 ULP.</td>
</tr>
<tr>
<td>implicit and explicit conversions between types</td>
<td>Correctly rounded.</td>
</tr>
</tbody>
</table>

Built-in functions defined in the specification with an equation built from the above operations inherit the above errors. These include, for example, the geometric functions, the common functions, and many of the matrix functions. Built-in functions not listed above and not defined as equations of the above have undefined precision. These include, for example, the trigonometric functions and determinant.

The precision of double-precision operations is at least that of single precision.

### 4.7.2 Precision Qualifiers

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

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>highp</td>
<td>None.</td>
</tr>
<tr>
<td>mediump</td>
<td>None.</td>
</tr>
<tr>
<td>lowp</td>
<td>None.</td>
</tr>
</tbody>
</table>
4 Variables and Types

For example:

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

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

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

### 4.7.3 Default Precision Qualifiers

The precision statement

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

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

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

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

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

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

```glsl
precision mediump int;
precision highp float;
```
4.7.4 Available Precision Qualifiers

The built-in macro GL_FRAGMENT_PRECISION_HIGH is defined to 1:

```
#define GL_FRAGMENT_PRECISION_HIGH 1
```

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

4.8 Variance and the Invariant Qualifier

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

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

4.8.1 The Invariant Qualifier

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

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

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

or as part of a declaration when a variable is declared

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

Only variables output from a shader (including those that are then input to a subsequent shader) can be candidates for invariance. This includes user-defined output variables and the built-in output variables. As only outputs need be declared with invariant, an output from one shader stage will still match an input of a subsequent stage without the input being declared as invariant.

Input or output instance names on blocks are not used when redeclaring built-in variables.

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

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

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

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

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

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

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

#pragma STDGL invariant(all)

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

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

### 4.8.2 Invariance of Constant Expressions

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

Constant expressions must evaluate to the same result when operated on as already described above for invariant variables, whether or not the `invariant` qualifier is used.

### 4.9 The Precise Qualifier

Some algorithms require floating-point computations to exactly follow the order of operations specified in the source code and to treat all operations consistently, even if the implementation supports optimizations that could produce nearly equivalent results with higher performance. For example, many GL implementations support a "multiply-add" instruction that can compute a floating-point expression such as

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

in two operations instead of three operations; one multiply and one multiply-add instead of two multiplies and one add. The result of a floating-point multiply-add might not always be identical to first doing a multiply yielding a floating-point result and then doing a floating-point add. Hence, in this example, the two multiply operations would not be treated consistently; the two multiplies could effectively appear to have differing precisions.
The key computation that needs to be made consistent appears when tessellating, where intermediate points for subdivision are synthesized in different directions, yet need to yield the same result, as shown in the diagram below.

Without any qualifiers, implementations are permitted to perform such optimizations that effectively modify the order or number of operations used to evaluate an expression, even if those optimizations may produce slightly different results relative to unoptimized code.

The qualifier **precise** will ensure that operations contributing to a variable's value are done in their stated order and are done with operator consistency. Order is determined by operator precedence and parenthesis, as described in section 5.1 “Operators”. Operator consistency means for each particular operator, for example the multiply operator (\(*\)), its operation is always computed with the same precision. Specifically, values computed by compiler-generated code must adhere to the following identities:

1. \(a + b = b + a\)
2. \(a \times b = b \times a\)
3. \(a \times b + c \times d = b \times a + c \times d = d \times c + b \times a = \text{<any other mathematically valid combination>}\)

While the following are prevented:

4. \(a + (b + c)\) is not allowed to become \((a + b) + c\)
5. \(a \times (b \times c)\) is not allowed to become \(a \times (b \times c)\)
6. \(a \times b + c\) is not allowed to become a single operation **fma**(a, b, c)

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

```gl
precise out vec4 position;
```

declares that operations used to produce the value of `position` must be performed in exactly the order specified in the source code and with all operators being treated consistently. As with the invariant qualifier (section 4.8.1 “The Invariant Qualifier”), the precise qualifier may be used to qualify a built-in or previously declared user-defined variable as being precise:

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

This qualifier will affect the evaluation of an r-value in a particular function if and only if the result is eventually consumed in the same function by an l-value qualified as `precise`. Any other expressions within a function are not affected, including return values and output parameters not declared as `precise` but that are eventually consumed outside the function by an variable qualified as `precise`. 
Some examples of the use of `precise`:

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

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

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

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

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

For the purposes of determining if an output from one shader stage matches an input of the next stage, the `precise` qualifier need not match between the input and the output.

All constant expressions are evaluated as if `precise` was present, whether or not it is present. However, as described in section 4.3.3 “Constant Expressions”, there is no requirement that a compile-time constant expression evaluates to the same value as a corresponding non-constant expression.

### 4.10 Memory Qualifiers

Variables declared as image types (the basic opaque types with “image” in their keyword) can be further qualified with a memory qualifier.
Variables declared as image types can qualified with one or more of the following memory qualifiers:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>coherent</td>
<td>memory variable where reads and writes are coherent with reads and writes from other shader invocations</td>
</tr>
<tr>
<td>volatile</td>
<td>memory variable whose underlying value may be changed at any point during shader execution by some source other than the current shader invocation</td>
</tr>
<tr>
<td>restrict</td>
<td>memory variable where use of that variable is the only way to read and write the underlying memory in the relevant shader stage</td>
</tr>
<tr>
<td>readonly</td>
<td>memory variable that can be used to read the underlying memory, but cannot be used to write the underlying memory</td>
</tr>
<tr>
<td>writeonly</td>
<td>memory variable that can be used to write the underlying memory, but cannot be used to read the underlying memory</td>
</tr>
</tbody>
</table>

Memory accesses to image variables declared using the **coherent** qualifier are performed coherently with similar accesses from other shader invocations. In particular, when reading a variable declared as **coherent**, the values returned will reflect the results of previously completed writes performed by other shader invocations. When writing a variable declared as **coherent**, the values written will be reflected in subsequent coherent reads performed by other shader invocations. As described in section 7.11 “Shader Memory Access” of the OpenGL Specification, shader memory reads and writes complete in a largely undefined order. The built-in function **memoryBarrier()** can be used if needed to guarantee the completion and relative ordering of memory accesses performed by a single shader invocation.

When accessing memory using variables not declared as **coherent**, the memory accessed by a shader may be cached by the implementation to service future accesses to the same address. Memory stores may be cached in such a way that the values written might not be visible to other shader invocations accessing the same memory. The implementation may cache the values fetched by memory reads and return the same values to any shader invocation accessing the same memory, even if the underlying memory has been modified since the first memory read. While variables not declared as **coherent** might not be useful for communicating between shader invocations, using non-coherent accesses may result in higher performance.

Memory accesses to image variables declared using the **volatile** qualifier must treat the underlying memory as though it could be read or written at any point during shader execution by some source other than the executing shader invocation. When a volatile variable is read, its value must be re-fetched from the underlying memory, even if the shader invocation performing the read had previously fetched its value from the same memory. When a volatile variable is written, its value must be written to the underlying memory, even if the compiler can conclusively determine that its value will be overwritten by a subsequent write. Since the external source reading or writing a volatile variable may be another shader invocation, variables declared as **volatile** are automatically treated as coherent.

Memory accesses to image variables declared using the **restrict** qualifier may be compiled assuming that the variable used to perform the memory access is the only way to access the underlying memory using the shader stage in question. This allows the compiler to coalesce or reorder loads and stores using **restrict**-qualified image variables in ways that wouldn't be permitted for image variables not so qualified,
4 Variables and Types

because the compiler can assume that the underlying image won't be read or written by other code. Applications are responsible for ensuring that image memory referenced by variables qualified with restrict will not be referenced using other variables in the same scope; otherwise, accesses to restrict-qualified variables will have undefined results.

Memory accesses to image variables declared using the readonly qualifier may only read the underlying memory, which is treated as read-only memory and cannot be written to. It is a compile-time error to pass an image variable qualified with readonly to imageStore() or other built-in functions that modify image memory.

Memory accesses to image variables declared using the writeonly qualifier may only write the underlying memory; the underlying memory cannot be read. It is a compile-time error to pass an image variable qualified with writeonly to imageLoad() or other built-in functions that read image memory. It is a compile-time error to qualify a variable as both readonly and writeonly.

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

Additionally, memory qualifiers may also be used in the declaration of shader storage blocks. When a block declaration is qualified with a memory qualifier, it is as if all of its members were declared with the same memory qualifier. For example, the block declaration

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

is equivalent to

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

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

The values of image variables qualified with coherent, volatile, restrict, readonly, or writeonly may not be passed to functions whose formal parameters lack such qualifiers. (See section 6.1 “Function Definitions” for more detail on function calling.) It is legal to have additional qualifiers on a formal parameter, but not to have fewer.
vec4 funcA(restrict image2D a) { ... }
vec4 funcB(image2D a) { ... }
layout(rgba32f) uniform image2D img1;
layout(rgba32f) coherent uniform image2D img2;

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

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

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

4.11 Order of Qualification

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

## 5.1 Operators

The OpenGL Shading Language has the following operators.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator Class</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (highest)</td>
<td>parenthetical grouping</td>
<td>()</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>array subscript function call and constructor structure field or method selector, swizzle post fix increment and decrement</td>
<td>[ ]</td>
<td>Left to Right</td>
</tr>
<tr>
<td>3</td>
<td>prefix increment and decrement unary</td>
<td>++ -- + - ~ !</td>
<td>Right to Left</td>
</tr>
<tr>
<td>4</td>
<td>multiplicative</td>
<td>* / %</td>
<td>Left to Right</td>
</tr>
<tr>
<td>5</td>
<td>additive</td>
<td>+ -</td>
<td>Left to Right</td>
</tr>
<tr>
<td>6</td>
<td>bit-wise shift</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>Left to Right</td>
</tr>
<tr>
<td>7</td>
<td>relational</td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>Left to Right</td>
</tr>
<tr>
<td>8</td>
<td>equality</td>
<td>== !=</td>
<td>Left to Right</td>
</tr>
<tr>
<td>9</td>
<td>bit-wise and</td>
<td>&amp;</td>
<td>Left to Right</td>
</tr>
<tr>
<td>10</td>
<td>bit-wise exclusive or</td>
<td>^</td>
<td>Left to Right</td>
</tr>
<tr>
<td>11</td>
<td>bit-wise inclusive or</td>
<td></td>
<td>Left to Right</td>
</tr>
<tr>
<td>12</td>
<td>logical and</td>
<td>&amp;&amp;</td>
<td>Left to Right</td>
</tr>
<tr>
<td>13</td>
<td>logical exclusive or</td>
<td>^^</td>
<td>Left to Right</td>
</tr>
<tr>
<td>14</td>
<td>logical inclusive or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>selection</td>
<td>? :</td>
<td>Right to Left</td>
</tr>
<tr>
<td>16</td>
<td>Assignment arithmetic assignments</td>
<td>= += -= *= /= %= &lt;&lt;= &gt;&gt;= &amp;%= ^=</td>
<td>=</td>
</tr>
<tr>
<td>17 (lowest)</td>
<td>sequence</td>
<td>,</td>
<td>Left to Right</td>
</tr>
</tbody>
</table>

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

5.2 Array Operations
These are now described in section 5.7 “Structure and Array Operations”.

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

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

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

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

5.4.1 Conversion and Scalar Constructors
Converting between scalar types is done as the following prototypes indicate:

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

86
When constructors are used to convert any floating-point type to an integer type, the fractional part of the floating-point value is dropped. It is undefined to convert a negative floating-point value to an `uint`.

When a constructor is used to convert any integer or floating-point type to a `bool`, 0 and 0.0 are converted to `false`, and non-zero values are converted to `true`. When a constructor is used to convert a `bool` to any integer or floating-point type, `false` is converted to 0 or 0.0, and `true` is converted to 1 or 1.0.

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

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

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

### 5.4.2 Vector and Matrix Constructors

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

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

If a vector is constructed from multiple scalars, one or more vectors, or one or more matrices, or a mixture of these, the vector's components will be constructed in order from the components of the arguments. The arguments will be consumed left to right, and each argument will have all its components consumed, in order, before any components from the next argument are consumed. Similarly for constructing a matrix from multiple scalars or vectors, or a mixture of these. Matrix components will be constructed and consumed in column major order. In these cases, there must be enough components provided in the arguments to provide an initializer for every component in the constructed value. It is a compile-time 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 a compile-time error to have any other arguments.

If the basic type (`bool`, `int`, `float`, or `double`) of a parameter to a constructor does not match the basic type of the object being constructed, the scalar construction rules (above) are used to convert the parameters.
Some useful vector constructors are as follows:

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

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

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

vec3(vec2, float)         // vec3.x = vec2.x, vec3.y = vec2.y, vec3.z = float
vec3(float, vec2)         // vec3.x = float, vec3.y = vec2.x, vec3.z = vec2.y
vec4(vec3, float)         // vec4.x = vec3.x, vec4.y = vec3.y, vec4.z = vec3.z, vec4.w = float
vec4(float, vec3)         // vec4.x = float, vec4.y = vec3.x, vec4.z = vec3.y, vec4.w = vec3.z
vec4(vec2, vec2)          // vec4.x = vec2.x, vec4.y = vec2.y, vec4.z = vec2.z, vec4.w = float
\end{verbatim}

Some examples of these are:

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
mat2(float)
mat3(float)
mat4(float)
\end{verbatim}

That is, \( result[i][j] \) is set to the float argument for all \( i = j \) and set to 0 for all \( i \neq j \).
To initialize a matrix by specifying vectors or scalars, the components are assigned to the matrix elements in column-major order.

\[
\begin{align*}
\text{mat2} & \left( \begin{array}{c} \text{vec2} \end{array} \right); & \text{one column per argument} \\
\text{mat3} & \left( \begin{array}{c} \text{vec3} \end{array} \right); & \text{one column per argument} \\
\text{mat4} & \left( \begin{array}{c} \text{vec4} \end{array} \right); & \text{one column per argument} \\
\text{mat3x2} & \left( \begin{array}{c} \text{vec2} \end{array} \right); & \text{one column per argument} \\
\text{dmat2} & \left( \begin{array}{c} \text{dvec2} \end{array} \right); \\
\text{dmat3} & \left( \begin{array}{c} \text{dvec3} \end{array} \right); \\
\text{dmat4} & \left( \begin{array}{c} \text{dvec4} \end{array} \right); \\
\text{mat2} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{first column} \\
\text{mat2} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{second column} \\
\text{mat3} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{first column} \\
\text{mat3} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{second column} \\
\text{mat3} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{third column} \\
\text{mat4} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{first column} \\
\text{mat4} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{second column} \\
\text{mat4} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{third column} \\
\text{mat4} & \left( \begin{array}{c} \text{float} \end{array} \right); & \text{fourth column} \\
\text{mat2x3} & \left( \begin{array}{c} \text{vec2} \end{array} \right); & \text{first column} \\
\text{mat2x3} & \left( \begin{array}{c} \text{vec2} \end{array} \right); & \text{second column} \\
\text{dmat2x4} & \left( \begin{array}{c} \text{dvec3} \end{array} \right); & \text{first column} \\
\text{dmat2x4} & \left( \begin{array}{c} \text{dvec3} \end{array} \right); & \text{second column}
\end{align*}
\]

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

\[
\begin{align*}
\text{mat3x3} & \left( \begin{array}{c} \text{mat4x4} \end{array} \right); & \text{takes the upper-left 3x3 of the mat4x4} \\
\text{mat2x3} & \left( \begin{array}{c} \text{mat4x2} \end{array} \right); & \text{takes the upper-left 2x2 of the mat4x4, last row is 0,0} \\
\text{mat4x4} & \left( \begin{array}{c} \text{mat3x3} \end{array} \right); & \text{puts the mat3x3 in the upper-left, sets the lower right component to 1, and the rest to 0}
\end{align*}
\]

### 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:

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

light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));
```
The arguments to the constructor will be used to set the structure's members, in order, using one argument per member. Each argument must be the same type as the member it sets, or be a type that can be converted to the member's type according to section 4.1.10 “Implicit Conversions.”

Structure constructors can be used as initializers or in expressions.

### 5.4.4 Array Constructors

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

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

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

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

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

Arrays of arrays are similarly constructed, but only the outer-most dimension is optional:

```cpp
vec4 b[2] = ...;
vec4[3][2](b, b, b); // constructor
vec4[][2](b, b, b); // constructor, valid, size deduced
vec4[3][](b, b, b); // compile-time error, invalid type constructed
```

### 5.5 Vector and Scalar Components and Length

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

The component names supported are:

<table>
<thead>
<tr>
<th>Component Names</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{x, y, z, w}</td>
<td>Useful when accessing vectors that represent points or normals</td>
</tr>
<tr>
<td>{r, g, b, a}</td>
<td>Useful when accessing vectors that represent colors</td>
</tr>
<tr>
<td>{s, t, p, q}</td>
<td>Useful when accessing vectors that represent texture coordinates</td>
</tr>
</tbody>
</table>
The component names \( x \), \( r \), and \( s \) are, for example, synonyms for the same (first) component in a vector. They are also the names of the only component in a scalar.

Note that the third component of the texture coordinate set, \( r \) in OpenGL, has been renamed \( p \) so as to avoid the confusion with \( r \) (for red) in a color.

Accessing components beyond those declared for the type is a compile-time error so, for example:

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

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

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

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

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

This notation is more concise than the constructor syntax. To form an \( r \)-value, it can be applied to any expression that results in a vector or scalar \( r \)-value. Any resulting vector of any operation must be a valid vector in the language; hence the following results in a compile-time error:

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

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

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

To form an \( l \)-value, swizzling must be applied to an \( l \)-value of vector or scalar type, contain no duplicate components, and it results in an \( l \)-value of scalar or vector type, depending on number of components specified.
Array subscripting syntax can also be applied to vectors (but not to scalars) to provide numeric indexing. So in

```cpp
vec4 pos;
```

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

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

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

sets the constant `L` to 3. The type returned by `length()` on a vector is `int`, and the value returned is a constant expression.

### 5.6 Matrix Components

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

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

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

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

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

sets the constant `L` to 3. The type returned by `length()` on a matrix is `int`, and the value returned is a constant expression.

### 5.7 Structure and Array Operations

The members of a structure and the `length` method of an array are selected using the period (`.`).
In total, only the following operators are allowed to operate on arrays and structures as whole entities:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field selector</td>
<td>.</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>assignment</td>
<td>=</td>
</tr>
<tr>
<td>indexing (arrays only)</td>
<td>[]</td>
</tr>
</tbody>
</table>

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

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

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

Array indices start at zero. Array elements are accessed using an expression whose type is `int` or `uint`. Behavior is undefined if a shader subscripts an array with an index less than 0 or greater than or equal to the size the array was declared with.

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

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

### 5.8 Assignments

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

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

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

Variables that are built-in types, entire structures or arrays, structure members, l-values with the field selector ( . ) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator ( [ ] ) are all l-values. Other binary or unary expressions, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (?:) is also not allowed as an l-value. Using an incorrect expression as an l-value results in a compile-time error.

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

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

where the general expression

\[
\text{lvalue op= expression}
\]

is equivalent to

\[
\text{lvalue = lvalue op expression}
\]

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

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

### 5.9 Expressions

Expressions in the shading language are built from the following:

- Constants of type `bool`, all integer types, all floating-point types, all vector types, and all matrix types.
- Constructors of all types.
- Variable names of all types.
- An array, vector, or matrix expression with the `length` method applied.
- Subscripted array names.
- Function calls that return values.
- Component field selectors and array subscript results.
- Parenthesized expression. Any expression can be parenthesized. Parentheses can be used to group operations. Operations within parentheses are done before operations across parentheses.
The arithmetic binary operators add (+), subtract (-), multiply (*), and divide (/) operate on integer and floating-point scalars, vectors, and matrices. If the fundamental types in the operands do not match, then the conversions from section 4.1.10 “Implicit Conversions” are applied to create matching types. All arithmetic binary operators result in the same fundamental type (signed integer, unsigned integer, single-precision floating point, or double-precision floating point) as the operands they operate on, after operand type conversion. After conversion, the following cases are valid:

- The two operands are scalars. In this case the operation is applied, resulting in a scalar.
- One operand is a scalar, and the other is a vector or matrix. In this case, the scalar operation is applied independently to each component of the vector or matrix.
- The two operands are vectors of the same size. In this case, the operation is done component-wise resulting in the same size vector.
- The operator is add (+), subtract (-), or divide (/), and the operands are matrices with the same number of rows and the same number of columns. In this case, the operation is done component-wise resulting in the same size matrix.
- The operator is multiply (*), where both operands are matrices or one operand is a vector and the other a matrix. A right vector operand is treated as a column vector and a left vector operand as a row vector. In all these cases, it is required that the number of columns of the left operand is equal to the number of rows of the right operand. Then, the multiply (*) operation does a linear algebraic multiply, yielding an object that has the same number of rows as the left operand and the same number of columns as the right operand. Section 5.10 “Vector and Matrix Operations” explains in more detail how vectors and matrices are operated on.

All other cases result in a compile-time error.

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

- The operator modulus (%) operates on signed or unsigned integer scalars or integer vectors. If the fundamental types in the operands do not match, then the conversions from section 4.1.10 “Implicit Conversions” are applied to create matching types. The operands cannot be vectors of differing size; this is a compile time error. If one operand is a scalar and the other vector, then the scalar is applied component-wise to the vector, resulting in the same type as the vector. If both are vectors of the same size, the result is computed component-wise. The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero second operands remain defined. If both operands are non-negative, then the remainder is non-negative. Results are undefined if one or both operands are negative. The operator modulus (%) is not defined for any other data types (non-integer types).

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

• The relational operators greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate only on scalar integer and scalar floating-point expressions. The result is scalar Boolean. Either the operands’ types must match, or the conversions from section 4.1.10 “Implicit Conversions” will be applied to obtain matching types. To do component-wise relational comparisons on vectors, use the built-in functions lessThan, lessThanEqual, greaterThan, and greaterThanEqual.

• The equality operators equal (==), and not equal (!=) operate on all types. They result in a scalar Boolean. If the operand types do not match, then there must be a conversion from section 4.1.10 “Implicit Conversions” applied to one operand that can make them match, in which case this conversion is done. For vectors, matrices, structures, and arrays, all components, members, or elements of one operand must equal the corresponding components, members, or elements in the other operand for the operands to be considered equal. To get a vector of component-wise equality results for vectors, use the built-in functions equal and notEqual.

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

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

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

• The ternary selection operator (?:). It operates on three expressions (exp1 ? exp2 : exp3). This operator evaluates the first expression, which must result in a scalar Boolean. If the result is true, it selects to evaluate the second expression, otherwise it selects to evaluate the third expression. Only one of the second and third expressions is evaluated. The second and third expressions can be any type, as long their types match, or there is a conversion in section 4.1.10 “Implicit Conversions” that can be applied to one of the expressions to make their types match. This resulting matching type is the type of the entire expression.

• The one's complement operator (~). The operand must be of type signed or unsigned integer or integer vector, and the result is the one's complement of its operand; each bit of each component is complemented, including any sign bits.

• The shift operators (<<) and (>>). For both operators, the operands must be signed or unsigned integers or integer vectors. One operand can be signed while the other is unsigned. In all cases, the resulting type will be the same type as the left operand. If the first operand is a scalar, the second operand has to be a scalar as well. If the first operand is a vector, the second operand must be a scalar or a vector, and the result is computed component-wise. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's base type. The value of E1 << E2 is E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1 >> E2 is E1 right-
shifted by E2 bit positions. If E1 is a signed integer, the right-shift will extend the sign bit. If E1 is an
unsigned integer, the right-shift will zero-extend.

- The bitwise operators and (&), exclusive-or (^), and inclusive-or (|). The operands must be of type
  signed or unsigned integers or integer vectors. The operands cannot be vectors of differing size; this is
  a compile-time error. If one operand is a scalar and the other a vector, the scalar is applied
  component-wise to the vector, resulting in the same type as the vector. The fundamental types of the
  operands (signed or unsigned) must match, and will be the resulting fundamental type. For and (&),
  the result is the bitwise-and function of the operands. For exclusive-or (^), the result is the bitwise
  exclusive-or function of the operands. For inclusive-or (|), the result is the bitwise inclusive-or
  function of the operands.

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

5.10 Vector and Matrix Operations

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

```
vec3 v, u;
float f;

v = u + f;
```

will be equivalent to

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

And

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

will be equivalent to

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

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

```
vec3 v, u;
mat3 m;

u = v * m;
```

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

And

u = m * v;

is equivalent to

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

And

mat3 m, n, r;

r = m * n;

is equivalent to

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

and similarly for other sizes of vectors and matrices.

5.11 Out-of-Bounds Accesses

In the subsections described above for array, vector, matrix and structure accesses, any out-of-bounds access produced undefined behavior. However, if robust buffer access is enabled via the OpenGL API, such accesses will be bound within the memory extent of the active program. It will not be possible to access memory from other programs, and accesses will not result in abnormal program termination. Out-of-bounds reads return undefined values, which include values from other variables of the active program or zero. Out-of-bounds writes may be discarded or overwrite other variables of the active program, depending on the value of the computed index and how this relates to the extent of the active program's memory. Applications that require defined behavior for out-of-bounds accesses should range check all computed indices before dereferencing an array.
6 Statements and Structure

The fundamental building blocks of the OpenGL Shading Language are:

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

The overall structure of a shader is as follows

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

global-declaration:
  function-definition
  declaration

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.

```markdown
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:

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

and a function is defined like

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

where `returnType` must be present and include a type. If the type of `returnValue` does not match `returnType`, there must be an implicit conversion in section 4.1.10 “Implicit Conversions” that converts the type of `returnValue` to `returnType`, or a compile-time error will result.

Each of the `typeN` must include a type and can optionally include parameter qualifiers. The formal argument names (`args` above) in the declarations are optional for both the declaration and definition forms.

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

Arrays are allowed as arguments and as the return type. In both cases, the array must be explicitly sized. An array is passed or returned by using just its name, without brackets, and the size of the array must match the size specified in the function’s declaration.

Structures are also allowed as argument types. The return type can also be structure.
See section 9 “Shading Language Grammar” for the definitive reference on the syntax to declare and define functions.

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

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

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

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

Only a precision qualifier is allowed on the return type of a function. Formal parameters can have parameter, precision, and memory qualifiers, but no other qualifiers.

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

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

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

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

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

If implicit conversions can be used to find more than one matching function, a single best-matching function is sought. To determine a best match, the conversions between calling argument and formal parameter types are compared for each function argument and pair of matching functions. After these comparisons are performed, each pair of matching functions are compared. A function declaration A is considered a better match than function declaration B if

- for at least one function argument, the conversion for that argument in A is better than the corresponding conversion in B; and
there is no function argument for which the conversion in $B$ is better than the corresponding conversion in $A$.

If a single function declaration is considered a better match than every other matching function declaration, it will be used. Otherwise, a compile-time semantic error for an ambiguous overloaded function call occurs.

To determine whether the conversion for a single argument in one match is better than that for another match, the following rules are applied, in order:

1. An exact match is better than a match involving any implicit conversion.
2. A match involving an implicit conversion from float to double is better than a match involving any other implicit conversion.
3. A match involving an implicit conversion from either int or uint to float is better than a match involving an implicit conversion from either int or uint to double.

If none of the rules above apply to a particular pair of conversions, neither conversion is considered better than the other.

For the example function prototypes (A), (B), and (C) above, the following examples show how the rules apply to different sets of calling argument types:

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

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

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

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

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

It is a compile-time or link-time error to declare or define a function `main` with any other parameters or return type.
6.1.1 Function Calling Conventions

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

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

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

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

If the function matching described in the previous section required argument type conversions, these conversions are applied at copy-in and copy-out times.

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

When calling a function, expressions that do not evaluate to l-values cannot be passed to parameters declared as `out` or `inout`, or a compile-time error results.
function-prototype :
    precision-qualifier type function-name(parameter-qualifiers precision-qualifier type name
array-specifier, ... )

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

parameter-qualifiers :
    empty
    list of parameter-qualifier

parameter-qualifier :
    const
    in
    out
    inout
    precise
    memory qualifier
    precision qualifier

name :
    empty
    identifier

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

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

Recursion is not allowed, not even statically. Static recursion is present if the static function-call graph of a program contains cycles. This includes all potential function calls through variables declared as subroutine uniform (described below). It is a compile-time or link-time error if a single compilation unit (shader) contains either static recursion or the potential for recursion through subroutine variables.

6.1.2 Subroutines

Subroutines provide a mechanism allowing shaders to be compiled in a manner where the target of one or more function calls can be changed at run-time without requiring any shader recompilation. For example, a single shader may be compiled with support for multiple illumination algorithms to handle different kinds of lights or surface materials. An application using such a shader may switch illumination algorithms by changing the value of its subroutine uniforms. To use subroutines, a subroutine type is declared, one or more functions are associated with that subroutine type, and a subroutine variable of that type is declared. The function currently assigned to the variable function is then called by using function calling syntax replacing a function name with the name of the subroutine variable. Subroutine variables
are uniforms, and are assigned to specific functions only through commands (\texttt{UniformSubroutinesuiv}) in the OpenGL API.

Subroutine types are declared using a statement similar to a function declaration, with the \texttt{subroutine} keyword, as follows:

\begin{verbatim}
subroutine returnType subroutineTypeName(type0 arg0, type1 arg1, ...
\end{verbatim}

As with function declarations, the formal argument names (\texttt{args} above) are optional. Functions are associated with subroutine types of matching declarations by defining the function with the \texttt{subroutine} keyword and a list of subroutine types the function matches:

\begin{verbatim}
subroutine(subroutineTypeName0, ..., subroutineTypeNameN)
returnType functionName(type0 arg0, type1 arg1, ..., typen argn)
\end{verbatim}

It is a compile-time error if arguments and return type don't match between the function and each associated subroutine type.

Functions declared with \texttt{subroutine} must include a body. An overloaded function cannot be declared with \texttt{subroutine}; a program will fail to compile or link if any shader or stage contains two or more functions with the same name if the name is associated with a subroutine type.

Subroutine type variables are required to be \texttt{subroutine uniforms}, and are declared with a specific subroutine type in a subroutine uniform variable declaration:

\begin{verbatim}
subroutine uniform subroutineTypeName subroutineVarName;
\end{verbatim}

Subroutine uniform variables are called the same way functions are called. When a subroutine variable (or an element of a subroutine variable array) is associated with a particular function, all function calls through that variable will call that particular function.

Unlike other uniform variables, subroutine uniform variables are scoped to the shader execution stage the variable is declared in.

Subroutine variables may be declared as explicitly-sized arrays, which can be dynamically indexed at use.

## 6.2 Selection

Conditional control flow in the shading language is done by either \texttt{if, if-else, or switch} statements:

\begin{verbatim}
selection-statement :
  if ( bool-expression ) statement
  if ( bool-expression ) statement else statement
  switch ( init-expression ) { switch-statement-list opt }
\end{verbatim}

Where \texttt{switch-statement-list} is a list of zero or more \texttt{switch-statement} and other statements defined by the language, where \texttt{switch-statement} adds some forms of labels. That is

\begin{verbatim}
switch-statement-list :
  switch-statement
  switch-statement-list switch-statement
\end{verbatim}
switch-statement :
  case constant-expression :
  default :
  statement

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

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

Conditionals can be nested.

The type of the init-expression value in a switch statement must be a scalar int or uint. The type of the constant-expression value in a case label also must be a scalar int or uint. When any pair of these values is tested for "equal value" and the types do not match, an implicit conversion will be done to convert the int to a uint (see section 4.1.10 “Implicit Conversions”) before the compare is done. If a case label has a constant-expression of equal value to init-expression, execution will continue after that label. It is a compile-time error to have two case label constant-expression of equal value. Otherwise, if there is a default label, execution will continue after that label. Otherwise, execution skips the rest of the switch statement. It is a compile-time error to have more than one default. A break statement not nested in a loop or other switch statement (either not nested or nested only in if or if-else statements) will also skip the rest of the switch statement. Fall through labels are allowed, but it is a compile-time error to have no statement between a label and the end of the switch statement. No statements are allowed in a switch statement before the first case statement.

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

6.3 Iteration

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

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

while (condition-expression)
  sub-statement

do
  statement
while (condition-expression)

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

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

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

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

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

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

Loops can be nested.

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

### 6.4 Jumps

These are the jumps:

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

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

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

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

The `discard` keyword is only allowed within fragment shaders. It can be used within a fragment shader to abandon the operation on the current fragment. This keyword causes the fragment to be discarded and no updates to any buffers will occur. Control flow exits the shader, and subsequent implicit or explicit derivatives are undefined when this exit is non-uniform. It would typically be used within a conditional statement, for example:
if (intensity < 0.0)
    discard;

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

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

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

7.1 Built-In Language Variables

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

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

```glsl
// work group dimensions
in    uvec3 gl_NumWorkGroups;
const uvec3 gl_WorkGroupSize;

// work group and invocation IDs
in    uvec3 gl_WorkGroupID;
in    uvec3 gl_LocalInvocationID;

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

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

```glsl
in int   gl_VertexID;
in int   gl_InstanceID;

out gl_PerVertex {
vec4  gl_Position;
float gl_PointSize;
float gl_ClipDistance[];
};
```
In the geometry language, the built-in variables are intrinsically declared as:

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

in int gl_PrimitiveIDIn;
in int gl_InvocationID;

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

out int gl_PrimitiveID;
out int gl_Layer;
out int gl_ViewportIndex;
```
In the tessellation control language, built-in variables are intrinsically declared as:

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

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

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

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

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

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

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

out gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```
In the fragment language, built-in variables are intrinsically declared as:

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

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

Each of the above variables is discussed below.

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

The built-in constant `gl_WorkGroupSize` is a compute-shader constant containing the local work-group size of the shader. The size of the work group in the X, Y, and Z dimensions is stored in the x, y, and z components. The values stored in `gl_WorkGroupSize` match those specified in the required `local_size_x`, `local_size_y`, and `local_size_z` layout qualifiers for the current shader. This value is constant so that it can be used to size arrays of memory that can be shared within the local work group.

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

The built-in variable `gl_LocalInvocationID` is a compute-shader input variable containing the t-dimensional index of the local work group within the global work group that the current invocation is executing in. The possible values for this variable range across the local work group size, i.e., (0,0,0) to `(gl_WorkGroupSize.x - 1, gl_WorkGroupSize.y - 1, gl_WorkGroupSize.z - 1)`.

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

```glsl
gl_GlobalInvocationID =
    gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID;
```
The built-in variable `gl_LocalInvocationIndex` is a compute shader input variable that contains the one-dimensional representation of the `gl_LocalInvocationID`. This is useful for uniquely identifying a unique region of shared memory within the local work group for this invocation to use. This is computed as:

\[
gl_{\text{LocalInvocationIndex}} =
gl_{\text{LocalInvocationID}.z} \times gl_{\text{WorkGroupSize}.x} \times gl_{\text{WorkGroupSize}.y} +
gl_{\text{LocalInvocationID}.y} \times gl_{\text{WorkGroupSize}.x} +
gl_{\text{LocalInvocationID}.x};
\]

The variable `gl_VertexID` is a vertex language input variable that holds an integer index for the vertex, as defined under “Shader Inputs” in section 11.1.3.9 “Shader Inputs” in the OpenGL Graphics System Specification. While the variable `gl_VertexID` is always present, its value is not always defined.

The variable `gl_InstanceID` is a vertex language input variable that holds the instance number of the current primitive in an instanced draw call (see “Shader Inputs” in section 11.1.3.9 “Shader Inputs” in the OpenGL Graphics System Specification). If the current primitive does not come from an instanced draw call, the value of `gl_InstanceID` is zero.

As an output variable, `gl_Position` is intended for writing the homogeneous vertex position. It can be written at any time during shader execution. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations, if present, that operate on primitives after vertex processing has occurred. Its value is undefined after the vertex processing stage if the vertex shader executable does not write `gl_Position`, and it is undefined after geometry processing if the geometry executable calls `EmitVertex()` without having written `gl_Position` since the last `EmitVertex()` (or hasn't written it at all). As an input variable, `gl_Position` reads the output written in the previous shader stage to `gl_Position`.

As an output variable, `gl_PointSize` is intended for a shader to write the size of the point to be rasterized. It is measured in pixels. If `gl_PointSize` is not written to, its value is undefined in subsequent pipe stages. As an input variable, `gl_PointSize` reads the output written in the previous shader stage to `gl_PointSize`.

The variable `gl_ClipDistance` provides the forward compatible mechanism for controlling user clipping. The element `gl_ClipDistance[i]` specifies a clip distance for each plane `i`. A distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the clip plane, and a negative distance means the point is outside the clip plane. The clip distances will be linearly interpolated across the primitive and the portion of the primitive with interpolated distances less than 0 will be clipped.

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

As an output variable, `gl_ClipDistance` provides the place for the shader to write these distances. As an input in all but the fragment language, it reads the values written in the previous shader stage. In the fragment language, `gl_ClipDistance` array contains linearly interpolated values for the vertex values.
written by a shader to the \textit{gl\_ClipDistance} vertex output variable. Only elements in this array that have clipping enabled will have defined values.

The output variable \textit{gl\_PrimitiveID} is available only in the geometry language and provides a single integer that serves as a primitive identifier. This is then available to fragment shaders as the fragment input \textit{gl\_PrimitiveID}, which will select the written primitive ID from the provoking vertex in the primitive being shaded. If a fragment shader using \textit{gl\_PrimitiveID} is active and a geometry shader is also active, the geometry shader must write to \textit{gl\_PrimitiveID} or the fragment shader input \textit{gl\_PrimitiveID} is undefined. See section 11.3.4.5 "Geometry Shader Outputs" of the OpenGL Graphics System Specification for more information.

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

The geometry language input variable \textit{gl\_PrimitiveIDIn} behaves identically to the tessellation control and evaluation language input variable \textit{gl\_PrimitiveID}.

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

The variable \textit{gl\_Layer} is available as an output variable in the geometry language and an input variable in the fragment language. In the geometry language, it is used to select a specific layer (or face and layer of a cube map) of a multi-layer framebuffer attachment. The actual layer used will come from one of the vertices in the primitive being shaded. Which vertex the layer comes from is undefined, so it is best to write the same layer value for all vertices of a primitive. If a shader statically assigns a value to \textit{gl\_Layer}, layered rendering mode is enabled. See section 11.3.4.5 “Geometry Shader Outputs” and section 9.4.9 “Layered Framebuffers” of the OpenGL Graphics System Specification for more information. If a shader statically assigns a value to \textit{gl\_Layer}, and there is an execution path through the shader that does not set \textit{gl\_Layer}, then the value of \textit{gl\_Layer} is undefined for executions of the shader that take that path.
The output variable \( \text{gl\_Layer} \) takes on a special value when used with an array of cube map textures. Instead of only referring to the layer, it is used to select a cube map face and a layer. Setting \( \text{gl\_Layer} \) to the value \( \text{layer}\times6+\text{face} \) will render to face \( \text{face} \) of the cube defined in layer \( \text{layer} \). The face values are defined in Table 9.3 of section 9.4.9 “Layered Framebuffers” of the OpenGL Graphics System Specification, but repeated below for clarity.

<table>
<thead>
<tr>
<th>Face Value</th>
<th>Resulting Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_X</td>
</tr>
<tr>
<td>1</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_X</td>
</tr>
<tr>
<td>2</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_Y</td>
</tr>
<tr>
<td>3</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Y</td>
</tr>
<tr>
<td>4</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_Z</td>
</tr>
<tr>
<td>5</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Z</td>
</tr>
</tbody>
</table>

For example, to render to the positive y cube map face located in the 5th layer of the cube map array, \( \text{gl\_Layer} \) should be set to \( 5\times6+2 \).

The input variable \( \text{gl\_Layer} \) in the fragment language will have the same value that was written to the output variable \( \text{gl\_Layer} \) in the geometry language. If the geometry stage does not dynamically assign a value to \( \text{gl\_Layer} \), the value of \( \text{gl\_Layer} \) in the fragment stage will be undefined. If the geometry stage makes no static assignment to \( \text{gl\_Layer} \), the input \( \text{gl\_Value} \) in the fragment stage will be zero. Otherwise, the fragment stage will read the same value written by the geometry stage, even if that value is out of range. If a fragment shader contains a static access to \( \text{gl\_Layer} \), it will count against the implementation defined limit for the maximum number of inputs to the fragment stage.

The variable \( \text{gl\_ViewportIndex} \) is available as an output variable in the geometry language and an input variable in the fragment language. In the geometry language, it provides the index of the viewport to which the next primitive emitted from the geometry shader should be drawn. Primitives generated by the geometry shader will undergo viewport transformation and scissor testing using the viewport transformation and scissor rectangle selected by the value of \( \text{gl\_ViewportIndex} \). The viewport index used will come from one of the vertices in the primitive being shaded. However, which vertex the viewport index comes from is implementation-dependent, so it is best to use the same viewport index for all vertices of the primitive. If a geometry shader does not assign a value to \( \text{gl\_ViewportIndex} \), viewport transform and scissor rectangle zero will be used. If a geometry shader statically assigns a value to \( \text{gl\_ViewportIndex} \) and there is a path through the shader that does not assign a value to \( \text{gl\_ViewportIndex} \), the value of \( \text{gl\_ViewportIndex} \) is undefined for executions of the shader that take that path. See section 11.3.4.6 “Layer and Viewport Selection” of the OpenGL Graphics System Specification (Core Profile) for more information.

The input variable \( \text{gl\_ViewportIndex} \) in the fragment stage will have the same value that was written to the output variable \( \text{gl\_ViewportIndex} \) in the geometry stage. If the geometry stage does not dynamically assign to \( \text{gl\_ViewportIndex} \), the value of \( \text{gl\_ViewportIndex} \) in the fragment shader will be undefined. If the geometry stage makes no static assignment to \( \text{gl\_ViewportIndex} \), the fragment stage will read zero. Otherwise, the fragment stage will read the same value written by the geometry stage, even if that value is
out of range. If a fragment shader contains a static access to \texttt{gl\_ViewportIndex}, it will count against the implementation defined limit for the maximum number of inputs to the fragment stage.

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

The output variables \texttt{gl\_TessLevelOuter[]} and \texttt{gl\_TessLevelInner[]} are available only in the tessellation control language. The values written to these variables are assigned to the corresponding outer and inner tessellation levels of the output patch. They are used by the tessellation primitive generator to control primitive tessellation and may be read by tessellation evaluation shaders.

The variable \texttt{gl\_TessCoord} is available only in the tessellation evaluation language. It specifies a three-component \((u,v,w)\) vector identifying the position of the vertex being processed by the shader relative to the primitive being tessellated. Its values will obey the properties

\begin{verbatim}
    gl_TessCoord.x == 1.0 - (1.0 - gl_TessCoord.x) // two operations performed
    gl_TessCoord.y == 1.0 - (1.0 - gl_TessCoord.y) // two operations performed
    gl_TessCoord.z == 1.0 - (1.0 - gl_TessCoord.z) // two operations performed
\end{verbatim}

to aid in replicating subdivision computations.

The input variables \texttt{gl\_TessLevelOuter[]} and \texttt{gl\_TessLevelInner[]} are available only in the tessellation evaluation shader. If a tessellation control shader is active, these variables are filled with corresponding outputs written by the tessellation control shader. Otherwise, they are assigned with default tessellation levels specified in section 11.2.3.3 “Tessellation Evaluation Shader Inputs” in the OpenGL Graphics System Specification.

Fragment shaders output values to the OpenGL pipeline using declared \texttt{out} variables, the built-in variables \texttt{gl\_FragDepth} and \texttt{gl\_SampleMask}, unless the \texttt{discard} statement is executed.

The fixed functionality computed depth for a fragment may be obtained by reading \texttt{gl\_FragCoord.z}, described below.

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

If a shader executes the \texttt{discard} keyword, the fragment is discarded, and the values of any user-defined fragment outputs, \texttt{gl\_FragDepth}, and \texttt{gl\_SampleMask} become irrelevant.

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

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

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

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

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

The output array $gl\_SampleMask[]$ sets the sample mask for the fragment being processed. Coverage for the current fragment will become the logical AND of the coverage mask and the output $gl\_SampleMask$. This array must be sized in the fragment shader either implicitly or explicitly to be the same size described above. If the fragment shader statically assigns a value to $gl\_SampleMask$, the sample mask will be undefined for any array elements of any fragment shader invocations that fail to assign a value. If a shader does not statically assign a value to $gl\_SampleMask$, the sample mask has no effect on the processing of a fragment.

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

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

The $gl\_PerVertex$ block can be redeclared in a shader to explicitly indicate what subset of the fixed pipeline interface will be used. This is necessary to establish the interface between multiple programs. For example:

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

This establishes the output interface the shader will use with the subsequent pipeline stage. It must be a subset of the built-in members of $gl\_PerVertex$. 

117
If a built-in interface block is redeclared, it must appear in the shader before any use of any member included in the built-in declaration, or a compile-time error will result. It is also a compile-time error to redeclare the block more than once or to redeclare a built-in block and then use a member from that built-in block that was not included in the redeclaration. Also, if a built-in interface block is redeclared, no member of the built-in declaration can be redeclared outside the block redeclaration. If multiple shaders using members of a built-in block belonging to the same interface are linked together in the same program, they must all redeclare the built-in block in the same way, as described in section 4.3.9 “Interface Blocks” for interface block matching, or a link-time error will result. It will also be a link-time error if some shaders in a program redeclare a specific built-in interface block while another shader in that program does not redeclare that interface block yet still uses a member of that interface block. If a built-in block interface is formed across shaders in different programs, the shaders must all redeclare the built-in block in the same way (as described for a single program), or the values passed along the interface are undefined.

7.1.1 Compatibility Profile Built-In Language Variables

When using the compatibility profile, the GL can provide fixed functionality behavior for the vertex and fragment programmable pipeline stages. For example, mixing a fixed functionality vertex stage with a programmable fragment stage.

The following built-in vertex, tessellation control, tessellation evaluation, and geometry output variables are available to specify inputs for the subsequent programmable shader stage or the fixed functionality fragment stage. A particular one should be written to if any functionality in a corresponding fragment shader or fixed pipeline uses it or state derived from it. Otherwise, behavior is undefined. The following members are added to the output gl_PerVertex block in these languages:

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

The output variable gl_ClipVertex provides a place for vertex and geometry shaders to write the coordinate to be used with the user clipping planes. Writing to gl_ClipDistance is the preferred method for user clipping. It is a compile-time or link-time error for the set of shaders forming a program to statically read or write both gl_ClipVertex and gl_ClipDistance. If neither gl_ClipVertex nor gl_ClipDistance is written, their values are undefined and any clipping against user clip planes is also undefined.

Similarly to what was previously described for the core profile, the gl_PerVertex block can be redeclared in a shader to explicitly include these additional members. For example:
out gl_PerVertex {
    vec4 gl_Position;    // will use gl_Position
    vec4 gl_FrontColor;  // will consume gl_color in the fragment shader
    vec4 gl_BackColor;
    vec4 gl_TexCoord[3]; // 3 elements of gl_TexCoord will be used
};  // no other aspects of the fixed interface will be used

The user must ensure the clip vertex and user clipping planes are defined in the same coordinate space.
User clip planes work properly only under linear transform. It is undefined what happens under non-linear transform.

The output variables gl_FrontColor, glFrontSecondaryColor, gl_BackColor, and glBackSecondaryColor
assign primary and secondary colors for front and back faces of primitives containing the vertex being processed. The output variable gl_TexCoord assigns texture coordinates for the vertex being processed.

For gl_FogFragCoord, the value written will be used as the “c” value in section 16.4 “Fog” of the compatibility profile of the OpenGL Graphics System Specification, by the fixed functionality pipeline. For example, if the z-coordinate of the fragment in eye space is desired as “c”, then that's what the vertex shader executable should write into gl_FogFragCoord.

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

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

(This treatment is a special case for gl_TexCoord[], not a general method for redeclaring members of blocks.) It is a compile-time error to redeclare gl_TexCoord[] at global scope if there is a redeclaration of the corresponding built-in block; only one form of redeclaration is allowed within a shader (and hence within a stage, as block redeclarations must match across all shaders using it).

In the tessellation control, evaluation, and geometry shaders, the outputs of the previous stage described above are also available in the input gl_PerVertex block in these languages.

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

These can be redeclared to establish an explicit pipeline interface, the same way as described above for the output block gl_PerVertex, and the input redeclaration must match the output redeclaration of the previous stage. However, when a built-in interface block with an instance name is redeclared (e.g., gl_in),
the instance name must be included in the redeclaration. It is a compile-time error to not include the built-in instance name or to change its name. For example,

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

Treatment of `gl_TexCoord[]` redeclaration is also identical to that described for the output block `gl_TexCoord[]` redeclaration.

The following fragment input block is also available in a fragment shader when using the compatibility profile:

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

The values in `gl_Color` and `gl_SecondaryColor` will be derived automatically by the system from `gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor` based on which face is visible in the primitive producing the fragment. If fixed functionality is used for vertex processing, then `gl_FogFragCoord` will either be the z-coordinate of the fragment in eye space, or the interpolation of the fog coordinate, as described in section 16.4 “Fog” of the compatibility profile of the OpenGL Graphics System Specification. The `gl_TexCoord[]` values are the interpolated `gl_TexCoord[]` values from a vertex shader or the texture coordinates of any fixed pipeline based vertex functionality.

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

As described above for the input and output `gl_PerVertex` blocks, the `gl_PerFragment` block can be redeclared to create an explicit interface to another program. When matching these interfaces between separate programs, members in the `gl_PerVertex` output block must be declared if and only if the corresponding fragment-shader members generated from them are present in the `gl_PerFragment` input block. These matches are described in detail in section 7.4.1 “Shader Interface Matching” of the OpenGL Graphics System Specification. If they don't match within a program, a link-time error will result. If the mismatch is between two programs, values passed between programs are undefined. Unlike with all other block matching, the order of declaration within `gl_PerFragment` does not have to match across shaders and does not have to correspond with order of declaration in a matching `gl_PerVertex` redeclaration.

The following fragment output variables are available in a fragment shader when using the compatibility profile:

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

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

If a shader statically assigns a value to `gl_FragColor`, it may not assign a value to any element of `gl_FragData`. If a shader statically writes a value to any element of `gl_FragData`, it may not assign a value to `gl_FragColor`. That is, a shader may assign values to either `gl_FragColor` or `gl_FragData`, but not both. Multiple shaders linked together must also consistently write just one of these variables. Similarly, if user-declared output variables are in use (statically assigned to), then the built-in variables `gl_FragColor` and `gl_FragData` may not be assigned to. These incorrect usages all generate compile-time or link-time errors.

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

### 7.2 Compatibility Profile Vertex Shader Built-In Inputs

The following predeclared input names can be used from within a vertex shader to access the current values of OpenGL state when using the compatibility profile.

```glsl
in vec4 gl_Color;
in vec4 gl_SecondaryColor;
in vec3 gl_Normal;
in vec4 gl_Vertex;
in vec4 gl_MultiTexCoord0;
in vec4 gl_MultiTexCoord1;
in vec4 gl_MultiTexCoord2;
in vec4 gl_MultiTexCoord3;
in vec4 gl_MultiTexCoord4;
in vec4 gl_MultiTexCoord5;
in vec4 gl_MultiTexCoord6;
in vec4 gl_MultiTexCoord7;
in float gl_FogCoord;
```
7.3 Built-In Constants

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

///
/// Implementation-dependent constants. The example values below
/// are the minimum values allowed for these maximums.
///
const ivec3 gl_MaxComputeWorkGroupCount = { 65535, 65535, 65535 };  
const ivec3 gl_MaxComputeWorkGroupSize = { 1024, 1024, 64 };  
const int gl_MaxComputeUniformComponents = 1024;  
const int gl_MaxComputeTextureImageUnits = 16;  
const int gl_MaxComputeImageUniforms = 8;  
const int gl_MaxComputeAtomicCounters = 8;  
const int gl_MaxComputeAtomicCounterBuffers = 1;  
const int gl_MaxVertexAttribs = 16;  
const int gl_MaxVertexUniformComponents = 1024;  
const int gl_MaxVaryingComponents = 60;  
const int gl_MaxVertexOutputComponents = 64;  
const int gl_MaxGeometryInputComponents = 64;  
const int gl_MaxGeometryOutputComponents = 128;  
const int gl_MaxFragmentInputComponents = 128;  
const int gl_MaxVertexTextureImageUnits = 16;  
const int gl_MaxTextureImageUnits = 16;  
const int gl_MaxImageUnits = 8;  
const int gl_MaxCombinedImageUnitsAndFragmentOutputs = 8;  
const int gl_MaxImageSamples = 0;  
const int gl_MaxVertexImageUniforms = 0;  
const int gl_MaxTessControlImageUniforms = 0;  
const int gl_MaxTessEvaluationImageUniforms = 0;  
const int gl_MaxGeometryImageUniforms = 0;  
const int gl_MaxFragmentImageUniforms = 8;  
const int gl_MaxCombinedImageUniforms = 8;  
const int gl_MaxFragmentUniformComponents = 1024;  
const int gl_MaxDrawBuffers = 8;  
const int gl_MaxClipDistances = 8;  
const int gl_MaxGeometryTextureImageUnits = 16;  
const int gl_MaxGeometryOutputVertices = 256;  
const int gl_MaxGeometryTotalOutputComponents = 1024;  
const int gl_MaxGeometryUniformComponents = 1024;  
const int gl_MaxGeometryVaryingComponents = 64;
const int gl_MaxTessControlInputComponents = 128;
const int gl_MaxTessControlOutputComponents = 128;
const int gl_MaxTessControlTextureImageUnits = 16;
const int gl_MaxTessControlUniformComponents = 1024;
const int gl_MaxTessControlTotalOutputComponents = 4096;
const int gl_MaxTessEvaluationInputComponents = 128;
const int gl_MaxTessEvaluationOutputComponents = 128;
const int gl_MaxTessEvaluationTextureImageUnits = 16;
const int gl_MaxTessEvaluationUniformComponents = 1024;
const int gl_MaxTessPatchComponents = 120;
const int gl_MaxPatchVertices = 32;
const int gl_MaxTessGenLevel = 64;
const int gl_MaxViewports = 16;
const int gl_MaxVertexUniformVectors = 256;
const int gl_MaxFragmentUniformVectors = 256;
const int gl_MaxVaryingVectors = 15;
const int gl_MaxVertexAtomicCounters = 0;
const int gl_MaxTessControlAtomicCounters = 0;
const int gl_MaxTessEvaluationAtomicCounters = 0;
const int gl_MaxGeometryAtomicCounters = 0;
const int gl_MaxFragmentAtomicCounters = 8;
const int gl_MaxCombinedAtomicCounters = 8;
const int gl_MaxAtomicCounterBindings = 1;
const int gl_MaxVertexAtomicCounterBuffers = 0;
const int gl_MaxTessControlAtomicCounterBuffers = 0;
const int gl_MaxTessEvaluationAtomicCounterBuffers = 0;
const int gl_MaxGeometryAtomicCounterBuffers = 0;
const int gl_MaxFragmentAtomicCounterBuffers = 1;
const int gl_MaxCombinedAtomicCounterBuffers = 1;
const int gl_MaxAtomicCounterBufferSize = 16384;
const int gl_MINProgramTexelOffset = -8;
const int gl_MAXProgramTexelOffset = 7;

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

### 7.3.1 Compatibility Profile Built-In Constants

const int gl_MaxTextureUnits = 2;
const int gl_MaxTextureCoords = 8;
const int gl_MaxClipPlanes = 8;
const int gl_MaxVaryingFloats = 60;
7.4 Built-In Uniform State

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

```c
//
// Depth range in window coordinates,
// section 13.6.1 “Controlling the Viewport” in the
//
// Note: Depth-range state is only for viewport 0.
//
struct gl_DepthRangeParameters {
    float near; // n
    float far;  // f
    float diff; // f - n
};

uniform gl_DepthRangeParameters gl_DepthRange;

uniform int gl_NumSamples;
```

7.4.1 Compatibility Profile State

These variables are present only in the compatibility profile.

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

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

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

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

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

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

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

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

uniform gl_PointParameters gl_Point;

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

uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
struct gl_LightSourceParameters {
    vec4 ambient; // A_cli
    vec4 diffuse; // D_cli
    vec4 specular; // S_cli
    vec4 position; // P_pli
    vec4 halfVector; // Derived: H_i
    vec3 spotDirection; // Sdli
    float spotExponent; // Srl_i
    float spotCutoff; // Crli
        // (range: [0.0,90.0], 180.0)
    float spotCosCutoff; // Derived: cos(Crli)
        // (range: [1.0,0.0],-1.0)
    float constantAttenuation; // K0
    float linearAttenuation; // K1
    float quadraticAttenuation; // K2
};

uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];

struct gl_LightModelParameters {
    vec4 ambient; // Acs
};

uniform gl_LightModelParameters gl_LightModel;

//
// compatibility profile only
//
// Derived state from products of light and material.
//
struct gl_LightModelProducts {
    vec4 sceneColor; // Derived. Ecm + Acm * Acs
};

uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

struct gl_LightProducts {
    vec4 ambient; // Acm * A_cli
    vec4 diffuse; // Dcm * D_cli
    vec4 specular; // Scm * S_cli
};

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

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

uniform gl_FogParameters gl_Fog;
8 Built-in Functions

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

The built-in functions basically fall into three categories:

• They expose some necessary hardware functionality in a convenient way such as accessing a texture map. There is no way in the language for these functions to be emulated by a shader.

• They represent a trivial operation (clamp, mix, etc.) that is very simple for the user to write, but they are very common and may have direct hardware support. It is a very hard problem for the compiler to map expressions to complex assembler instructions.

• They represent an operation graphics hardware is likely to accelerate at some point. The trigonometry functions fall into this category.

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

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

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

When the built-in functions are specified below, where the input arguments (and corresponding output) can be `float`, `vec2`, `vec3`, or `vec4`, `genType` is used as the argument. Where the input arguments (and corresponding output) can be `int`, `ivec2`, `ivec3`, or `ivec4`, `genIType` is used as the argument. Where the input arguments (and corresponding output) can be `uint`, `uvec2`, `uvec3`, or `uvec4`, `genUType` is used as the argument. Where the input arguments (or corresponding output) can be `bool`, `bvec2`, `bvec3`, or `bvec4`, `genBType` is used as the argument. Where the input arguments (and corresponding output) can be `double`, `dvec2`, `dvec3`, `dvec4`, `genDType` is used as the argument. For any specific use of a function, the actual types substituted for `genType`, `genIType`, `genUType`, or `genBType` have to have the same number of components for all arguments and for the return type. Similarly, `mat` is used for any matrix basic type with single-precision components and `dmat` is used for any matrix basic type with double-precision components.
### 8.1 Angle and Trigonometry Functions

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

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genType radians (genType degrees)</code></td>
<td>Converts <code>degrees</code> to radians, i.e., ( \frac{\pi}{180} ) <code>degrees</code></td>
</tr>
<tr>
<td><code>genType degrees (genType radians)</code></td>
<td>Converts <code>radians</code> to degrees, i.e., ( \frac{180}{\pi} ) <code>radians</code></td>
</tr>
<tr>
<td><code>genType sin (genType angle)</code></td>
<td>The standard trigonometric sine function.</td>
</tr>
<tr>
<td><code>genType cos (genType angle)</code></td>
<td>The standard trigonometric cosine function.</td>
</tr>
<tr>
<td><code>genType tan (genType angle)</code></td>
<td>The standard trigonometric tangent.</td>
</tr>
<tr>
<td><code>genType asin (genType x)</code></td>
<td>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 (</td>
</tr>
<tr>
<td><code>genType acos (genType x)</code></td>
<td>Arc cosine. Returns an angle whose cosine is ( x ). The range of values returned by this function is ( [0, \pi] ). Results are undefined if (</td>
</tr>
<tr>
<td><code>genType atan (genType y, genType x)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>genType atan (genType y_over_x)</code></td>
<td>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] ).</td>
</tr>
<tr>
<td>Syntax</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>genType <strong>sinh</strong> (genType x)</td>
<td>Returns the hyperbolic sine function $\frac{e^x - e^{-x}}{2}$</td>
</tr>
<tr>
<td>genType <strong>cosh</strong> (genType x)</td>
<td>Returns the hyperbolic cosine function $\frac{e^x + e^{-x}}{2}$</td>
</tr>
<tr>
<td>genType <strong>tanh</strong> (genType x)</td>
<td>Returns the hyperbolic tangent function $\frac{\sinh(x)}{\cosh(x)}$</td>
</tr>
<tr>
<td>genType <strong>asinh</strong> (genType x)</td>
<td>Arc hyperbolic sine; returns the inverse of <strong>sinh</strong>.</td>
</tr>
<tr>
<td>genType <strong>acosh</strong> (genType x)</td>
<td>Arc hyperbolic cosine; returns the non-negative inverse of <strong>cosh</strong>. Results are undefined if $x &lt; 1$.</td>
</tr>
<tr>
<td>genType <strong>atanh</strong> (genType x)</td>
<td>Arc hyperbolic tangent; returns the inverse of <strong>tanh</strong>. Results are undefined if $</td>
</tr>
</tbody>
</table>
### 8.2 Exponential Functions

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genType <strong>pow</strong> (genType (x), genType (y))</td>
<td>Returns (x) raised to the (y) power, i.e., (x^y)</td>
</tr>
<tr>
<td></td>
<td>Results are undefined if (x &lt; 0).</td>
</tr>
<tr>
<td></td>
<td>Results are undefined if (x = 0) and (y \leq 0).</td>
</tr>
<tr>
<td>genType <strong>exp</strong> (genType (x))</td>
<td>Returns the natural exponentiation of (x), i.e., (e^x).</td>
</tr>
<tr>
<td>genType <strong>log</strong> (genType (x))</td>
<td>Returns the natural logarithm of (x), i.e., returns the value (y) which satisfies the equation (x = e^y).</td>
</tr>
<tr>
<td></td>
<td>Results are undefined if (x \leq 0).</td>
</tr>
<tr>
<td>genType <strong>exp2</strong> (genType (x))</td>
<td>Returns 2 raised to the (x) power, i.e., (2^x).</td>
</tr>
<tr>
<td>genType <strong>log2</strong> (genType (x))</td>
<td>Returns the base 2 logarithm of (x), i.e., returns the value (y) which satisfies the equation (x = 2^y).</td>
</tr>
<tr>
<td></td>
<td>Results are undefined if (x \leq 0).</td>
</tr>
<tr>
<td>genType <strong>sqrt</strong> (genType (x))</td>
<td>Returns (\sqrt{x}).</td>
</tr>
<tr>
<td>genDType <strong>sqrt</strong> (genDType (x))</td>
<td>Results are undefined if (x &lt; 0).</td>
</tr>
<tr>
<td>genType <strong>inversesqrt</strong> (genType (x))</td>
<td>Returns (\frac{1}{\sqrt{x}}).</td>
</tr>
<tr>
<td>genDType <strong>inversesqrt</strong> (genDType (x))</td>
<td>Results are undefined if (x \leq 0).</td>
</tr>
</tbody>
</table>
## 8.3 Common Functions

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genType abs (genType x)</td>
<td>Returns $x$ if $x \geq 0$, otherwise it returns $-x$.</td>
</tr>
<tr>
<td>genIType abs (genIType x)</td>
<td></td>
</tr>
<tr>
<td>genDType abs (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType sign (genType x)</td>
<td>Returns 1.0 if $x &gt; 0$, 0.0 if $x = 0$, or −1.0 if $x &lt; 0$.</td>
</tr>
<tr>
<td>genIType sign (genIType x)</td>
<td></td>
</tr>
<tr>
<td>genDType sign (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType floor (genType x)</td>
<td>Returns a value equal to the nearest integer that is less than or equal to $x$.</td>
</tr>
<tr>
<td>genDType floor (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType trunc (genType x)</td>
<td>Returns a value equal to the nearest integer to $x$ whose absolute value is not larger than the absolute value of $x$.</td>
</tr>
<tr>
<td>genDType trunc (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType round (genType x)</td>
<td>Returns a value equal to the nearest integer to $x$. The fraction 0.5 will round in a direction chosen by the implementation, presumably the direction that is fastest. This includes the possibility that round($x$) returns the same value as roundEven($x$) for all values of $x$.</td>
</tr>
<tr>
<td>genDType round (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType roundEven (genType x)</td>
<td>Returns a value equal to the nearest integer to $x$. A fractional part of 0.5 will round toward the nearest even integer. (Both 3.5 and 4.5 for $x$ will return 4.0.)</td>
</tr>
<tr>
<td>genDType roundEven (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType ceil (genType x)</td>
<td>Returns a value equal to the nearest integer that is greater than or equal to $x$.</td>
</tr>
<tr>
<td>genDType ceil (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType fract (genType x)</td>
<td>Returns $x - \text{floor}(x)$.</td>
</tr>
<tr>
<td>genDType fract (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType mod (genType x, float y)</td>
<td>Modulus. Returns $x - y \times \text{floor}(x/y)$.</td>
</tr>
<tr>
<td>genType mod (genType x, genType y)</td>
<td></td>
</tr>
<tr>
<td>genDType mod (genDType x, double y)</td>
<td></td>
</tr>
<tr>
<td>genDType mod (genDType x, genDType y)</td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genType <code>modf(x, i)</code></td>
<td>Returns the fractional part of <code>x</code> and sets <code>i</code> to the integer part (as a whole number floating-point value). Both the return value and the output parameter will have the same sign as <code>x</code>.</td>
</tr>
<tr>
<td>genDType <code>modf(x, i)</code></td>
<td></td>
</tr>
<tr>
<td>genType <code>min(x, y)</code></td>
<td>Returns <code>y</code> if <code>y &lt; x</code>, otherwise it returns <code>x</code>.</td>
</tr>
<tr>
<td>genType <code>min(x, float y)</code></td>
<td></td>
</tr>
<tr>
<td>genDType <code>min(x, y)</code></td>
<td></td>
</tr>
<tr>
<td>genDType <code>min(x, double y)</code></td>
<td></td>
</tr>
<tr>
<td>genIType <code>min(x, int y)</code></td>
<td></td>
</tr>
<tr>
<td>genUType <code>min(x, uint y)</code></td>
<td></td>
</tr>
<tr>
<td>genType <code>max(x, y)</code></td>
<td>Returns <code>y</code> if <code>x &lt; y</code>, otherwise it returns <code>x</code>.</td>
</tr>
<tr>
<td>genType <code>max(x, float y)</code></td>
<td></td>
</tr>
<tr>
<td>genDType <code>max(x, y)</code></td>
<td></td>
</tr>
<tr>
<td>genDType <code>max(x, double y)</code></td>
<td></td>
</tr>
<tr>
<td>genIType <code>max(x, int y)</code></td>
<td></td>
</tr>
<tr>
<td>genUType <code>max(x, uint y)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| `genType clamp (genType x, genType minVal, genType maxVal)` | Returns `min (max (x, minVal), maxVal)`.
<p>| <code>genType clamp (genType x, float minVal, float maxVal)</code> | Results are undefined if <code>minVal &gt; maxVal</code>. |
| <code>genDType clamp (genDType x, genDType minVal, genDType maxVal)</code> | |
| <code>genDType clamp (genDType x, double minVal, double maxVal)</code> | |
| <code>genIType clamp (genIType x, genIType minVal, genIType maxVal)</code> | |
| <code>genIType clamp (genIType x, int minVal, int maxVal)</code> | |
| <code>genUType clamp (genUType x, genUType minVal, genUType maxVal)</code> | |
| <code>genUType clamp (genUType x, uint minVal, uint maxVal)</code> | |
| <code>genType mix (genType x, genType y, genType a)</code> | Returns the linear blend of <code>x</code> and <code>y</code>, i.e., <code>x*(1-a)+y*a</code> |
| <code>genType mix (genType x, genType y, float a)</code> | |
| <code>genDType mix (genDType x, genDType y, genDType a)</code> | |
| <code>genDType mix (genDType x, genDType y, double a)</code> | |</p>
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genType <code>mix</code> (genType x,</td>
<td>Selects which vector each returned component comes from. For a component of <code>a</code> that is <code>false</code>, the corresponding component of <code>x</code> is returned. For a component of <code>a</code> that is <code>true</code>, the corresponding component of <code>y</code> is returned. Components of <code>x</code> and <code>y</code> that are not selected are allowed to be invalid floating-point values and will have no effect on the results. Thus, this provides different functionality than, for example, genType <code>mix</code>(genType x, genType y, genType(a)) where <code>a</code> is a Boolean vector.</td>
</tr>
<tr>
<td>genDType <code>mix</code> (genDType x,</td>
<td></td>
</tr>
<tr>
<td>genUType <code>floatBitsToInt</code> (genType value)</td>
<td></td>
</tr>
<tr>
<td>genUType <code>floatBitsToUint</code> (genType value)</td>
<td></td>
</tr>
<tr>
<td>genType <code>step</code> (genType edge, genType x)</td>
<td>Returns 0.0 if <code>x &lt; edge</code>, otherwise it returns 1.0.</td>
</tr>
<tr>
<td>genType <code>step</code> (float edge, genType x)</td>
<td></td>
</tr>
<tr>
<td>genDType <code>step</code> (genDType edge, genDType x)</td>
<td></td>
</tr>
<tr>
<td>genDType <code>step</code> (double edge, genDType x)</td>
<td></td>
</tr>
<tr>
<td>genType <code>smoothstep</code> (genType edge0, genType edge1, genType x)</td>
<td>Returns 0.0 if <code>x &lt;= edge0</code> and 1.0 if <code>x &gt;= edge1</code> and performs smooth Hermite interpolation between 0 and 1 when <code>edge0 &lt; x &lt; edge1</code>. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to: genType t; t = clamp ((x – edge0) / (edge1 – edge0), 0, 1); return t * t * (3 – 2 * t); (And similarly for doubles.) Results are undefined if <code>edge0 &gt;= edge1</code>.</td>
</tr>
<tr>
<td>genType <code>smoothstep</code> (float edge0, float edge1, genType x)</td>
<td></td>
</tr>
<tr>
<td>genDType <code>smoothstep</code> (genDType edge0, genDType edge1, genDType x)</td>
<td></td>
</tr>
<tr>
<td>genDType <code>smoothstep</code> (double edge0, double edge1, genDType x)</td>
<td></td>
</tr>
<tr>
<td>genBType <code>isnan</code> (genType x)</td>
<td>Returns <code>true</code> if <code>x</code> holds a NaN. Returns <code>false</code> otherwise. Always returns <code>false</code> if NaNs are not implemented.</td>
</tr>
<tr>
<td>genBType <code>isnan</code> (genDType x)</td>
<td></td>
</tr>
<tr>
<td>genBType <code>isinf</code> (genType x)</td>
<td>Returns <code>true</code> if <code>x</code> holds a positive infinity or negative infinity. Returns <code>false</code> otherwise.</td>
</tr>
<tr>
<td>genBType <code>isinf</code> (genDType x)</td>
<td></td>
</tr>
</tbody>
</table>
## 8 Built-in Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genType intBitsToFloat (genIType value)</code></td>
<td>Returns a <code>float</code> value corresponding to a signed or unsigned integer encoding of a <code>float</code>. If a NaN is passed in, it will not signal, and the resulting value is unspecified. If an Inf is passed in, the resulting value is the corresponding Inf.</td>
</tr>
<tr>
<td><code>genType uintBitsToFloat (genUType value)</code></td>
<td></td>
</tr>
<tr>
<td><code>genType fma (genType a, genType b, genType c)</code></td>
<td>Computes and returns $a \times b + c$. In uses where the return value is eventually consumed by a variable declared as <code>precise</code>:</td>
</tr>
<tr>
<td></td>
<td>• <code>fma()</code> is considered a single operation, whereas the expression “$a \times b + c$” consumed by a variable declared <code>precise</code> is considered two operations.</td>
</tr>
<tr>
<td></td>
<td>• The precision of <code>fma()</code> can differ from the precision of the expression “$a \times b + c$”.</td>
</tr>
<tr>
<td></td>
<td>• <code>fma()</code> will be computed with the same precision as any other <code>fma()</code> consumed by a <code>precise</code> variable, giving invariant results for the same input values of $a$, $b$, and $c$.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, in the absence of <code>precise</code> consumption, there are no special constraints on the number of operations or difference in precision between <code>fma()</code> and the expression “$a \times b + c$”.</td>
</tr>
<tr>
<td><code>genType frexp (genType x, out genIType exp)</code></td>
<td>Splits $x$ into a floating-point significand in the range $[0.5, 1.0)$ and an integral exponent of two, such that: [ x = \text{significand} \cdot 2^{\exp} ] ]</td>
</tr>
<tr>
<td><code>genDType frexp (genDType x, out genIType exp)</code></td>
<td>The significand is returned by the function and the exponent is returned in the parameter <code>exp</code>. For a floating-point value of zero, the significant and exponent are both zero. For a floating-point value that is an infinity or is not a number, the results are undefined.</td>
</tr>
<tr>
<td><code>genType ldexp (genType x, in genIType exp)</code></td>
<td>Builds a floating-point number from $x$ and the corresponding integral exponent of two in <code>exp</code>, returning: $\text{significand} \cdot 2^{\exp}$. If this product is too large to be represented in the floating-point type, the result is undefined.</td>
</tr>
<tr>
<td><code>genDType ldexp (genDType x, in genIType exp)</code></td>
<td></td>
</tr>
</tbody>
</table>
## 8.4 Floating-Point Pack and Unpack Functions

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| uint `packUnorm2x16` (vec2 `v`) | First, converts each component of the normalized floating-point value `v` into 8- or 16-bit integer values. Then, the results are packed into the returned 32-bit unsigned integer. The conversion for component `c` of `v` to fixed point is done as follows:  
`packUnorm2x16`: `round(clamp(c, 0, +1) * 65535.0)`  
`packSnorm2x16`: `round(clamp(c, -1, +1) * 32767.0)`  
`packUnorm4x8`: `round(clamp(c, 0, +1) * 255.0)`  
`packSnorm4x8`: `round(clamp(c, -1, +1) * 127.0)`  
The first component of the vector will be written to the least significant bits of the output; the last component will be written to the most significant bits. |
| vec2 `unpackUnorm2x16` (uint `p`) | First, unpacks a single 32-bit unsigned integer `p` into a pair of 16-bit unsigned integers, four 8-bit unsigned integers, or four 8-bit signed integers. Then, each component is converted to a normalized floating-point value to generate the returned two- or four-component vector. The conversion for unpacked fixed-point value `f` to floating point is done as follows:  
`unpackUnorm2x16`: `f / 65535.0`  
`unpackSnorm2x16`: `clamp(f / 32767.0, -1, +1)`  
`unpackUnorm4x8`: `f / 255.0`  
`unpackSnorm4x8`: `clamp(f / 127.0, -1, +1)`  
The first component of the returned vector will be extracted from the least significant bits of the input; the last component will be extracted from the most significant bits. |
<p>| <code>packSnorm2x16</code> (vec2 <code>v</code>) |  |
| <code>packUnorm4x8</code> (vec4 <code>v</code>) |  |
| <code>packSnorm4x8</code> (vec4 <code>v</code>) |  |</p>
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double <code>packDouble2x32 (uvec2 v)</code></td>
<td>Returns a double-precision value obtained by packing the components of <code>v</code> into a 64-bit value. If an IEEE 754 Inf or NaN is created, it will not signal, and the resulting floating-point value is unspecified. Otherwise, the bit-level representation of <code>v</code> is preserved. The first vector component specifies the 32 least significant bits; the second component specifies the 32 most significant bits.</td>
</tr>
<tr>
<td><code>uvec2 unpackDouble2x32 (double v)</code></td>
<td>Returns a two-component unsigned integer vector representation of <code>v</code>. The bit-level representation of <code>v</code> is preserved. The first component of the vector contains the 32 least significant bits of the double; the second component consists the 32 most significant bits.</td>
</tr>
<tr>
<td><code>uint packHalf2x16 (vec2 v)</code></td>
<td>Returns an unsigned integer obtained by converting the components of a two-component floating-point vector to the 16-bit floating-point representation found in the OpenGL Specification, and then packing these two 16-bit integers into a 32-bit unsigned integer. The first vector component specifies the 16 least-significant bits of the result; the second component specifies the 16 most-significant bits.</td>
</tr>
<tr>
<td><code>vec2 unpackHalf2x16 (uint v)</code></td>
<td>Returns a two-component floating-point vector with components obtained by unpacking a 32-bit unsigned integer into a pair of 16-bit values, interpreting those values as 16-bit floating-point numbers according to the OpenGL Specification, and converting them to 32-bit floating-point values. The first component of the vector is obtained from the 16 least-significant bits of <code>v</code>; the second component is obtained from the 16 most-significant bits of <code>v</code>.</td>
</tr>
</tbody>
</table>
### 8.5 Geometric Functions

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float <code>length (genType x)</code></td>
<td>Returns the length of vector x, i.e., ( \sqrt{x[0]^2 + x[1]^2 + ...} )</td>
</tr>
<tr>
<td>double <code>length (genDType x)</code></td>
<td></td>
</tr>
<tr>
<td>float <code>distance (genType p0, genType p1)</code></td>
<td>Returns the distance between ( p0 ) and ( p1 ), i.e., <code>length (p0 − p1)</code></td>
</tr>
<tr>
<td>double <code>distance (genDType p0, genDType p1)</code></td>
<td></td>
</tr>
<tr>
<td>float <code>dot (genType x, genType y)</code></td>
<td>Returns the dot product of ( x ) and ( y ), i.e., ( x[0] \cdot y[0] + x[1] \cdot y[1] + ... )</td>
</tr>
<tr>
<td>double <code>dot (genDType x, genDType y)</code></td>
<td></td>
</tr>
</tbody>
</table>
| vec3 `cross (vec3 x, vec3 y)` | Returns the cross product of \( x \) and \( y \), i.e., \[
\begin{bmatrix}
    x[2] \cdot y[0] - y[2] \cdot x[0] \\
    x[0] \cdot y[1] - y[0] \cdot x[1]
\end{bmatrix}
\] |
| dvec3 `cross (dvec3 x, dvec3 y)` | |
| genType `normalize (genType x)` | Returns a vector in the same direction as \( x \) but with a length of 1. |
| genDType `normalize (genDType x)` | |
| `compatibility profile only vec4 ftransform ()` | Available only when using the compatibility profile. For core OpenGL, use `invariant`. For vertex shaders only. This function will ensure that the incoming vertex value will be transformed in a way that produces exactly the same result as would be produced by OpenGL’s fixed functionality transform. It is intended to be used to compute `gl_Position`, e.g., `gl_Position = ftransform()` |
| `compatibility profile only genType ftransform ()` | |
| `compatibility profile only genDType ftransform ()` | |

For vertex shaders only. This function will ensure that the incoming vertex value will be transformed in a way that produces exactly the same result as would be produced by OpenGL’s fixed functionality transform. It is intended to be used to compute `gl_Position`, e.g.,

```glsl
gl_Position = ftransform();
```

This function should be used, for example, when an application is rendering the same geometry in separate passes, and one pass uses the fixed functionality path to render and another pass uses programmable shaders.
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genType <strong>faceforward</strong> (genType ( N ), genType ( I ), genType ( N_{\text{ref}} ))</td>
<td>If ( \text{dot}(N_{\text{ref}}, I) ) &lt; 0 return ( N ), otherwise return (-N).</td>
</tr>
<tr>
<td>genDType <strong>faceforward</strong> (genDType ( N ), genDType ( I ), genDType ( N_{\text{ref}} ))</td>
<td></td>
</tr>
<tr>
<td>genType <strong>reflect</strong> (genType ( I ), genType ( N ))</td>
<td>For the incident vector ( I ) and surface orientation ( N ), returns the reflection direction: ( I - 2 \times \text{dot}(N, I) \times N ) ( N ) must already be normalized in order to achieve the desired result.</td>
</tr>
<tr>
<td>genDType <strong>reflect</strong> (genDType ( I ), genDType ( N ))</td>
<td></td>
</tr>
<tr>
<td>genType <strong>refract</strong> (genType ( I ), genType ( N ), float ( \eta ))</td>
<td>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 \times \eta \times (1.0 - \text{dot}(N, I) \times \text{dot}(N, I)) ) if ( k &lt; 0.0 ) return genType(0.0) // or genDType(0.0) else return ( \eta \times I - (\eta \times \text{dot}(N, I) + \text{sqrt}(k)) \times N ) The input parameters for the incident vector ( I ) and the surface normal ( N ) must already be normalized to get the desired results.</td>
</tr>
<tr>
<td>genDType <strong>refract</strong> (genDType ( I ), genDType ( N ), float ( \eta ))</td>
<td></td>
</tr>
</tbody>
</table>
8.6 Matrix Functions

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mat matrixCompMult (mat x, mat y)</td>
<td>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 (*).</td>
</tr>
<tr>
<td>mat2 outerProduct (vec2 c, vec2 r)</td>
<td>Treats the first parameter c as a column vector (matrix with one column) and the second parameter r as a row vector (matrix with one row) and does a linear algebraic matrix multiply c * r, yielding a matrix whose number of rows is the number of components in c and whose number of columns is the number of components in r.</td>
</tr>
<tr>
<td>mat3 outerProduct (vec3 c, vec3 r)</td>
<td></td>
</tr>
<tr>
<td>mat4 outerProduct (vec4 c, vec4 r)</td>
<td></td>
</tr>
<tr>
<td>mat2x3 outerProduct (vec3 c, vec2 r)</td>
<td></td>
</tr>
<tr>
<td>mat3x2 outerProduct (vec2 c, vec3 r)</td>
<td></td>
</tr>
<tr>
<td>mat2x4 outerProduct (vec4 c, vec2 r)</td>
<td></td>
</tr>
<tr>
<td>mat4x2 outerProduct (vec2 c, vec4 r)</td>
<td></td>
</tr>
<tr>
<td>mat3x4 outerProduct (vec4 c, vec3 r)</td>
<td></td>
</tr>
<tr>
<td>mat4x3 outerProduct (vec3 c, vec4 r)</td>
<td></td>
</tr>
<tr>
<td>mat2 transpose (mat2 m)</td>
<td>Returns a matrix that is the transpose of m. The input matrix m is not modified.</td>
</tr>
<tr>
<td>mat3 transpose (mat3 m)</td>
<td></td>
</tr>
<tr>
<td>mat4 transpose (mat4 m)</td>
<td></td>
</tr>
<tr>
<td>mat2x3 transpose (mat3x2 m)</td>
<td></td>
</tr>
<tr>
<td>mat3x2 transpose (mat2x3 m)</td>
<td></td>
</tr>
<tr>
<td>mat2x4 transpose (mat4x2 m)</td>
<td></td>
</tr>
<tr>
<td>mat4x2 transpose (mat2x4 m)</td>
<td></td>
</tr>
<tr>
<td>mat3x4 transpose (mat4x3 m)</td>
<td></td>
</tr>
<tr>
<td>mat4x3 transpose (mat3x4 m)</td>
<td></td>
</tr>
<tr>
<td>float determinant (mat2 m)</td>
<td>Returns the determinant of m.</td>
</tr>
<tr>
<td>float determinant (mat3 m)</td>
<td></td>
</tr>
<tr>
<td>float determinant (mat4 m)</td>
<td></td>
</tr>
</tbody>
</table>
## 8 Built-in Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mat2 inverse (mat2 m)</code></td>
<td>Returns a matrix that is the inverse of <code>m</code>. The input matrix <code>m</code> is not modified. The values in the returned matrix are undefined if <code>m</code> is singular or poorly-conditioned (nearly singular).</td>
</tr>
<tr>
<td><code>mat3 inverse (mat3 m)</code></td>
<td></td>
</tr>
<tr>
<td><code>mat4 inverse (mat4 m)</code></td>
<td></td>
</tr>
</tbody>
</table>
8.7 Vector Relational Functions

Relational and equality operators (<, <=, >, >=, ==, !) are defined to operate on scalars and produce scalar Boolean results. For vector results, use the following built-in functions. Below, the following placeholders are used for the listed specific types:

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Specific Types Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>bvec</td>
<td>bvec2, bvec3, bvec4</td>
</tr>
<tr>
<td>ivec</td>
<td>ivec2, ivec3, ivec4</td>
</tr>
<tr>
<td>uvec</td>
<td>uvec2, uvec3, uvec4</td>
</tr>
<tr>
<td>vec</td>
<td>vec2, vec3, vec4, dvec2, dvec3, dvec4</td>
</tr>
</tbody>
</table>

In all cases, the sizes of all the input and return vectors for any particular call must match.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bvec lessThan (vec x, vec y)</td>
<td>Returns the component-wise compare of ( x &lt; y ).</td>
</tr>
<tr>
<td>bvec lessThan (ivec x, ivec y)</td>
<td></td>
</tr>
<tr>
<td>bvec lessThan (uvec x, uvec y)</td>
<td></td>
</tr>
<tr>
<td>bvec lessThanEqual (vec x, vec y)</td>
<td>Returns the component-wise compare of ( x \leq y ).</td>
</tr>
<tr>
<td>bvec lessThanEqual (ivec x, ivec y)</td>
<td></td>
</tr>
<tr>
<td>bvec lessThanEqual (uvec x, uvec y)</td>
<td></td>
</tr>
<tr>
<td>bvec greaterThan (vec x, vec y)</td>
<td>Returns the component-wise compare of ( x &gt; y ).</td>
</tr>
<tr>
<td>bvec greaterThan (ivec x, ivec y)</td>
<td></td>
</tr>
<tr>
<td>bvec greaterThan (uvec x, uvec y)</td>
<td></td>
</tr>
<tr>
<td>bvec greaterThanEqual (vec x, vec y)</td>
<td>Returns the component-wise compare of ( x \geq y ).</td>
</tr>
<tr>
<td>bvec greaterThanEqual (ivec x, ivec y)</td>
<td></td>
</tr>
<tr>
<td>bvec greaterThanEqual (uvec x, uvec y)</td>
<td></td>
</tr>
<tr>
<td>bvec equal (vec x, vec y)</td>
<td>Returns the component-wise compare of ( x = y ).</td>
</tr>
<tr>
<td>bvec equal (ivec x, ivec y)</td>
<td></td>
</tr>
<tr>
<td>bvec equal (uvec x, uvec y)</td>
<td></td>
</tr>
<tr>
<td>bvec equal (bvec x, bvec y)</td>
<td></td>
</tr>
<tr>
<td>bvec notEqual (vec x, vec y)</td>
<td>Returns the component-wise compare of ( x \neq y ).</td>
</tr>
<tr>
<td>bvec notEqual (ivec x, ivec y)</td>
<td></td>
</tr>
<tr>
<td>bvec notEqual (uvec x, uvec y)</td>
<td></td>
</tr>
<tr>
<td>bvec notEqual (bvec x, bvec y)</td>
<td></td>
</tr>
<tr>
<td>bool any (bvec x)</td>
<td>Returns true if any component of ( x ) is true.</td>
</tr>
</tbody>
</table>
## 8 Built-in Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool <code>all (bvec x)</code></td>
<td>Returns true only if all components of <code>x</code> are <code>true</code>.</td>
</tr>
<tr>
<td>bvec <code>not (bvec x)</code></td>
<td>Returns the component-wise logical complement of <code>x</code>.</td>
</tr>
</tbody>
</table>
8.8 Integer Functions

These all operate component-wise. The description is per component. The notation \([a, b]\) means the set of bits from bit-number \(a\) through bit-number \(b\), inclusive. The lowest-order bit is bit 0. “Bit number” will always refer to counting up from the lowest-order bit as bit 0.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genUType uaddCarry (genUType x, genUType y, out genUType carry)</code></td>
<td>Adds 32-bit unsigned integer (x) and (y), returning the sum modulo (2^{32}). The value (carry) is set to 0 if the sum was less than (2^{32}), or to 1 otherwise.</td>
</tr>
<tr>
<td><code>genUType usubBorrow (genUType x, genUType y, out genUType borrow)</code></td>
<td>Subtracts the 32-bit unsigned integer (y) from (x), returning the difference if non-negative, or (2^{32}) plus the difference otherwise. The value (borrow) is set to 0 if (x \geq y), or to 1 otherwise.</td>
</tr>
<tr>
<td><code>void umulExtended (genUType x, genUType y, out genUType msh, out genUType lsb)</code></td>
<td>Multiplies 32-bit integers (x) and (y), producing a 64-bit result. The 32 least-significant bits are returned in (lsb). The 32 most-significant bits are returned in (msh).</td>
</tr>
<tr>
<td><code>void imulExtended (genIType x, genIType y, out genIType msh, out genIType lsb)</code></td>
<td>Multiplies 32-bit integers (x) and (y), producing a 64-bit result. The 32 least-significant bits are returned in (lsb). The 32 most-significant bits are returned in (msh).</td>
</tr>
<tr>
<td><code>genIType bitfieldExtract (genIType value, int offset, int bits)</code>, <code>genUType bitfieldExtract (genUType value, int offset, int bits)</code></td>
<td>Extracts bits ([offset, offset + bits - 1]) from (value), returning them in the least significant bits of the result. For unsigned data types, the most significant bits of the result will be set to zero. For signed data types, the most significant bits will be set to the value of bit (offset + bits - 1). If (bits) is zero, the result will be zero. The result will be undefined if (offset) or (bits) is negative, or if the sum of (offset) and (bits) is greater than the number of bits used to store the operand.</td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Returns the insertion of the bits least-significant bits of `insert` into `base`. The result will have bits `[offset, offset + bits - 1]` taken from bits `{0, bits – 1}` of `insert`, and all other bits taken directly from the corresponding bits of `base`. If `bits` is zero, the result will simply be `base`. The result will be undefined if `offset` or `bits` is negative, or if the sum of `offset` and `bits` is greater than the number of bits used to store the operand. | genIType `bitfieldInsert` (genIType `base`, genIType `insert`, int `offset`, int `bits`)  
 genUType `bitfieldInsert` (genUType `base`, genUType `insert`, int `offset`, int `bits`) |
| Returns the reversal of the bits of `value`. The bit numbered `n` of the result will be taken from bit `(bits - 1) - n of value`, where `bits` is the total number of bits used to represent `value`. | genIType `bitfieldReverse` (genIType `value`)  
 genUType `bitfieldReverse` (genUType `value`) |
| Returns the number of bits set to 1 in the binary representation of `value`. | genIType `bitCount` (genIType `value`)  
 genIType `bitCount` (genUType `value`) |
| Returns the bit number of the least significant bit set to 1 in the binary representation of `value`. If `value` is zero, `-1` will be returned. | genIType `findLSB` (genIType `value`)  
 genIType `findLSB` (genUType `value`) |
| Returns the bit number of the most significant bit in the binary representation of `value`. For positive integers, the result will be the bit number of the most significant bit set to 1. For negative integers, the result will be the bit number of the most significant bit set to 0. For a `value` of zero or negative one, `-1` will be returned. | genIType `findMSB` (genIType `value`)  
 genIType `findMSB` (genUType `value`) |
8.9 Texture Functions

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

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

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

<table>
<thead>
<tr>
<th>Internal Texture Format</th>
<th>Floating Point Sampler Types</th>
<th>Signed Integer Sampler Types</th>
<th>Unsigned Integer Sampler Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normalized Integer</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signed Integer</td>
<td></td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Unsigned Integer</td>
<td></td>
<td></td>
<td>Supported</td>
</tr>
</tbody>
</table>

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

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

For shadow forms (the sampler parameter is a shadow-type), a depth comparison lookup on the depth texture bound to $\textbf{sampler}$ is done as described in section 8.22 “Texture Comparison Modes” of the OpenGL Graphics System Specification. See the table below for which component specifies $D_{op}$. The texture bound to $\textbf{sampler}$ must be a depth texture, or results are undefined. If a non-shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned on, then results are undefined. If a shadow texture call is made to a sampler that represents a depth texture with depth comparisons turned off, then results are undefined. If a shadow texture call is made to a sampler that does not represent a depth texture, then results are undefined.
In all functions below, the $bias$ parameter is optional for fragment shaders. The $bias$ parameter is not accepted in any other shaders. For a fragment shader, if $bias$ is present, it is added to the implicit level of detail prior to performing the texture access operation. No $bias$ or $lod$ parameters for rectangle textures, multi-sample textures, or texture buffers are supported because mipmap maps are not allowed for these types of textures.

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

Some texture functions (non-“Lod” and non-“Grad” versions) may require implicit derivatives. Implicit derivatives are undefined within non-uniform control flow and for non-fragment-shader texture fetches.

For Cube forms, the direction of $P$ is used to select which face to do a 2-dimensional texture lookup in, as described in section 8.13 “Cube Map Texture Selection” in the OpenGL Graphics System Specification.

For Array forms, the array layer used will be

$$\text{max}(0, \text{min}(d - 1, \text{floor}(\text{layer} + 0.5)))$$

where $d$ is the depth of the texture array and $\text{layer}$ comes from the component indicated in the tables below.

For depth/stencil textures, the sampler type should match the component being accessed as set through the OpenGL API. When the depth/stencil texture mode is set to DEPTH_COMPONENT, a floating-point sampler type should be used. When the depth/stencil texture mode is set to STENCIL_INDEX, an unsigned integer sampler type should be used. Doing a texture lookup with an unsupported combination will return undefined values.

### 8.9.1 Texture Query Functions

The $\text{textureSize}$ functions query the dimensions of a specific texture level for a sampler.

The $\text{textureQueryLod}$ functions are available only in a fragment shader. They take the components of $P$ and compute the level of detail information that the texture pipe would use to access that texture through a normal texture lookup. The level of detail $\lambda'$ (equation 3.18 in the OpenGL Graphics System Specification) is obtained after any LOD bias, but prior to clamping to [$\text{TEXTURE_MIN_LOD}$, $\text{TEXTURE_MAX_LOD}$]. The mipmap array(s) that would be accessed are also computed. If a single level of detail would be accessed, the level-of-detail number relative to the base level is returned. If multiple levels of detail would be accessed, a floating-point number between the two levels is returned, with the fractional part equal to the fractional part of the computed and clamped level of detail.
The algorithm used is given by the following pseudo-code:

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

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

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

The value `maxAccessibleLevel` is the level number of the smallest accessible level of the mipmap array (the value \(q\) in section 8.14.3 “Mipmapping” of the OpenGL Graphics System Specification) minus the base level.

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int <code>textureSize</code> (gsampler1D <code>sampler</code>, int <code>lod</code>)</td>
<td>Returns the dimensions of level <code>lod</code> (if present) for the texture bound to <code>sampler</code>, as described in section 11.1.3.4 “Texture Queries” of the OpenGL Graphics System Specification. The components in the return value are filled in, in order, with the width, height, depth of the texture. For the array forms, the last component of the return value is the number of layers in the texture array, or the number of cubes in the texture cube map array.</td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (gsampler2D <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec3 <code>textureSize</code> (gsampler3D <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (gsamplerCube <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>int <code>textureSize</code> (gsampler1DShadow <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (sampler2DShadow <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (samplerCubeShadow <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec3 <code>textureSize</code> (gsamplerCubeArray <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec3 <code>textureSize</code> (samplerCubeArrayShadow <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (gsampler2DRect <code>sampler</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (gsampler2DRectShadow <code>sampler</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (gsampler1DArray <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec3 <code>textureSize</code> (gsampler1DArrayShadow <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (sampler1DArrayShadow <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec3 <code>textureSize</code> (sampler2DArray <code>sampler</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>int <code>textureSize</code> (gsamplerBuffer <code>sampler</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec2 <code>textureSize</code> (gsampler2DMS <code>sampler</code>)</td>
<td></td>
</tr>
<tr>
<td>ivec3 <code>textureSize</code> (gsampler2DMSArray <code>sampler</code>)</td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vec2 textureQueryLod(gsampler1D sampler, float P)</code></td>
<td>Returns the mipmap array(s) that would be accessed in the x component of the return value.</td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler2D sampler, vec2 P)</code></td>
<td>Returns the computed level of detail relative to the base level in the y component of the return value.</td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler3D sampler, vec3 P)</code></td>
<td>If called on an incomplete texture, the results are undefined.</td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsamplerCube sampler, vec3 P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler1DArray sampler, float P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler2DArray sampler, vec2 P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsamplerCubeArray sampler, vec3 P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler1DShadow sampler, float P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler2DShadow sampler, vec2 P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsamplerCubeShadow sampler, vec3 P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler1DArrayShadow sampler, float P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsampler2DArrayShadow sampler, vec2 P)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec2 textureQueryLod(gsamplerCubeArrayShadow sampler, vec3 P)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler1D sampler)</code></td>
<td>Returns the number of mipmap levels accessible in the texture associated with <code>sampler</code>, as defined in the OpenGL Specification.</td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler2D sampler)</code></td>
<td>The value zero will be returned if no texture or an incomplete texture is associated with <code>sampler</code>.</td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler3D sampler)</code></td>
<td>Available in all shader stages.</td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsamplerCube sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler1DArray sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler2DArray sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsamplerCubeArray sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler1DShadow sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler2DShadow sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsamplerCubeShadow sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler1DArrayShadow sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsampler2DArrayShadow sampler)</code></td>
<td></td>
</tr>
<tr>
<td><code>int textureQueryLevels(gsamplerCubeArrayShadow sampler)</code></td>
<td></td>
</tr>
</tbody>
</table>
## 8.9.2 Texel Lookup Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gvec4 <code>texture</code> (gsampler1D <code>sampler</code>, float <code>P</code> [, float <code>bias</code>] )</td>
<td>Use the texture coordinate <code>P</code> to do a texture lookup in the texture currently bound to <code>sampler</code>.</td>
</tr>
<tr>
<td>gvec4 <code>texture</code> (gsampler2D <code>sampler</code>, vec2 <code>P</code> [, float <code>bias</code>] )</td>
<td>For shadow forms: When <code>compare</code> is present, it is used as $D_{of}$ and the array layer comes from <code>P.w</code>.</td>
</tr>
<tr>
<td>gvec4 <code>texture</code> (gsampler3D <code>sampler</code>, vec3 <code>P</code> [, float <code>bias</code>] )</td>
<td>When <code>compare</code> is not present, the last component of <code>P</code> is used as $D_{of}$ and the array layer comes</td>
</tr>
<tr>
<td>gvec4 <code>texture</code> (gsamplerCube <code>sampler</code>, vec3 <code>P</code> [, float <code>bias</code>] )</td>
<td>from the second to last component of <code>P</code>. (The second component of <code>P</code> is unused for 1D shadow lookups.)</td>
</tr>
<tr>
<td>float <code>texture</code> (sampler1DShadow <code>sampler</code>, vec3 <code>P</code> [, float <code>bias</code>] )</td>
<td>For non-shadow forms: the array layer comes from the last component of <code>P</code>.</td>
</tr>
<tr>
<td>float <code>texture</code> (sampler2DShadow <code>sampler</code>, vec3 <code>P</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>float <code>texture</code> (samplerCubeShadow <code>sampler</code>, vec4 <code>P</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texture</code> (gsampler1DArray <code>sampler</code>, vec2 <code>P</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texture</code> (gsampler2DArray <code>sampler</code>, vec3 <code>P</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texture</code> (gsamplerCubeArray <code>sampler</code>, vec4 <code>P</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td><code>textureLod</code> (gsampler1D <code>sampler</code>, float <code>P</code>, float <code>lod</code>)</td>
<td>Do a texture lookup with projection. The texture coordinates consumed from <code>P</code>, not including the last</td>
</tr>
<tr>
<td><code>textureLod</code> (gsampler2D <code>sampler</code>, vec2 <code>P</code>, float <code>lod</code>)</td>
<td>component of <code>P</code>, are divided by the last component of <code>P</code>. The resulting 3rd component of <code>P</code> in the</td>
</tr>
<tr>
<td><code>textureLod</code> (gsampler3D <code>sampler</code>, vec3 <code>P</code>, float <code>lod</code>)</td>
<td>shadow forms is used as $D_{of}$. After these values are computed, texture lookup proceeds as in <code>texture</code>.</td>
</tr>
<tr>
<td><code>textureLod</code> (gsamplerCube <code>sampler</code>, vec3 <code>P</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td><code>textureLod</code> (sampler1DShadow <code>sampler</code>, vec3 <code>P</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td><code>textureLod</code> (sampler2DShadow <code>sampler</code>, vec3 <code>P</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td><code>textureLod</code> (samplerCubeShadow <code>sampler</code>, vec4 <code>P</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td><code>textureLod</code> (gsampler1DArray <code>sampler</code>, vec2 <code>P</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td><code>textureLod</code> (gsampler2DArray <code>sampler</code>, vec3 <code>P</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td><code>textureLod</code> (gsamplerCubeArray <code>sampler</code>, vec4 <code>P</code>, float <code>lod</code>)</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\frac{\partial u}{\partial x} &= 0 & \frac{\partial v}{\partial x} &= 0 & \frac{\partial w}{\partial x} &= 0 \\
\frac{\partial u}{\partial y} &= 0 & \frac{\partial v}{\partial y} &= 0 & \frac{\partial w}{\partial y} &= 0
\end{align*}
\]
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gvec4 <code>textureOffset</code> (gsampler1D <code>sampler</code>, float <code>P</code>, int <code>offset</code> [, float <code>bias</code>] )</td>
<td>Do a texture lookup as in <code>texture</code> but with <code>offset</code> added to the ((u,v,w)) texel coordinates before looking up each texel. The offset value must be a constant expression. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by <code>gl_MinProgramTexelOffset</code> and <code>gl_MaxProgramTexelOffset</code>, respectively. Note that <code>offset</code> does not apply to the layer coordinate for texture arrays. This is explained in detail in section 8.14.2 “Coordinate Wrapping and Texel Selection” of the OpenGL Graphics System Specification, where <code>offset</code> is ((\delta_u, \delta_v, \delta_w)). Note that texel offsets are also not supported for cube maps.</td>
</tr>
<tr>
<td>gvec4 <code>textureOffset</code> (gsampler2D <code>sampler</code>, vec2 <code>P</code>, ivec2 <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>textureOffset</code> (gsampler3D <code>sampler</code>, vec3 <code>P</code>, ivec3 <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>textureOffset</code> (gsampler2DRect <code>sampler</code>, vec2 <code>P</code>, ivec2 <code>offset</code> )</td>
<td></td>
</tr>
<tr>
<td>float <code>textureOffset</code> (sampler2DRectShadow <code>sampler</code>, vec3 <code>P</code>, ivec2 <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>float <code>textureOffset</code> (sampler1DShadow <code>sampler</code>, vec3 <code>P</code>, int <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>float <code>textureOffset</code> (sampler2DShadow <code>sampler</code>, vec3 <code>P</code>, ivec2 <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>textureOffset</code> (gsampler1DArray <code>sampler</code>, vec2 <code>P</code>, int <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>textureOffset</code> (gsampler2DArray <code>sampler</code>, vec3 <code>P</code>, ivec2 <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>float <code>textureOffset</code> (sampler1DArrayShadow <code>sampler</code>, vec3 <code>P</code>, int <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>float <code>textureOffset</code> (sampler2DArrayShadow <code>sampler</code>, vec4 <code>P</code>, vec2 <code>offset</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler1D <code>sampler</code>, int <code>P</code>, int <code>lod</code>)</td>
<td>Use integer texture coordinate <code>P</code> to lookup a single texel from <code>sampler</code>. The array layer comes from the last component of <code>P</code> for the array forms. The level-of-detail <code>lod</code> (if present) is as described in sections 11.1.3.2 “Texel Fetches” and 8.14.1 “Scale Factor and Level of Detail” of the OpenGL Graphics System Specification.</td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler2D <code>sampler</code>, ivec2 <code>P</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler3D <code>sampler</code>, ivec3 <code>P</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler2DRect <code>sampler</code>, ivec2 <code>P</code>)</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler1DArray <code>sampler</code>, ivec2 <code>P</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler2DArray <code>sampler</code>, ivec3 <code>P</code>, int <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsamplerBuffer <code>sampler</code>, int <code>P</code>)</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler2DMS <code>sampler</code>, ivec2 <code>P</code>, int <code>sample</code>)</td>
<td></td>
</tr>
<tr>
<td>gvec4 <code>texelFetch</code> (gsampler2DMSArray <code>sampler</code>, ivec3 <code>P</code>, int <code>sample</code>)</td>
<td></td>
</tr>
<tr>
<td>Syntax</td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td><code>gvec4 texelFetchOffset (gsampler1D sampler, int P, int lod, int offset)</code></td>
<td>Fetch a single texel as in <code>texelFetch offset</code> by <code>offset</code> as described in <code>textureOffset</code>.</td>
</tr>
<tr>
<td><code>gvec4 texelFetchOffset (gsampler2D sampler, ivec2 P, int lod, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 texelFetchOffset (gsampler3D sampler, ivec3 P, int lod, ivec3 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 texelFetchOffset (gsampler2DRect sampler, ivec2 P, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 texelFetchOffset (gsampler1DArray sampler, ivec2 P, int lod, int offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 texelFetchOffset (gsampler2DArray sampler, ivec3 P, int lod, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjOffset (gsampler1D sampler, vec2 P, int offset [, float bias])</code></td>
<td>Do a projective texture lookup as described in <code>textureProj offset</code> by <code>offset</code> as described in <code>textureOffset</code>.</td>
</tr>
<tr>
<td><code>gvec4 textureProjOffset (gsampler1D sampler, vec4 P, int offset [, float bias])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjOffset (gsampler2D sampler, vec3 P, ivec2 offset [, float bias])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjOffset (gsampler2D sampler, vec4 P, ivec2 offset [, float bias])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjOffset (gsampler3D sampler, vec4 P, ivec3 offset [, float bias])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjOffset (gsampler2DRect sampler, vec3 P, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjOffset (gsampler2DRect sampler, vec4 P, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureProjOffset (sampler2DRectShadow sampler, vec4 P, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureProjOffset (sampler1DShadow sampler, vec4 P, int offset [, float bias])</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureProjOffset (sampler2DShadow sampler, vec4 P, ivec2 offset [, float bias])</code></td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gvec4 textureLodOffset (gsampler1D sampler, float P, float lod, int offset)</code></td>
<td>Do an offset texture lookup with explicit LOD. See <code>textureLod</code> and <code>textureOffset</code>.</td>
</tr>
<tr>
<td><code>gvec4 textureLodOffset (gsampler2D sampler, vec2 P, float lod, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureLodOffset (gsampler3D sampler, vec3 P, float lod, ivec3 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureLodOffset (sampler1DShadow sampler, vec3 P, float lod, ivec3 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureLodOffset (sampler2DShadow sampler, vec3 P, float lod, int offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureLodOffset (gsampler1DArray sampler, vec2 P, float lod, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureLodOffset (gsampler2DArray sampler, vec3 P, float lod, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureLodOffset (sampler1DArrayShadow sampler, vec3 P, float lod, int offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLod (gsampler1D sampler, vec2 P, float lod)</code></td>
<td>Do a projective texture lookup with explicit LOD. See <code>textureProj</code> and <code>textureLod</code>.</td>
</tr>
<tr>
<td><code>gvec4 textureProjLod (gsampler1D sampler, vec4 P, float lod)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLod (gsampler2D sampler, vec3 P, float lod)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLod (gsampler2D sampler, vec4 P, float lod)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLod (gsampler3D sampler, vec4 P, float lod)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureProjLod (sampler1DShadow sampler, vec4 P, float lod)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureProjLod (sampler2DShadow sampler, vec4 P, float lod)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLodOffset (gsampler1D sampler, vec2 P, float lod, int offset)</code></td>
<td>Do an offset projective texture lookup with explicit LOD. See <code>textureProj</code>, <code>textureLod</code>, and <code>textureOffset</code>.</td>
</tr>
<tr>
<td><code>gvec4 textureProjLodOffset (gsampler1D sampler, vec4 P, float lod, int offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLodOffset (gsampler2D sampler, vec3 P, float lod, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLodOffset (gsampler2D sampler, vec4 P, float lod, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureProjLodOffset (gsampler3D sampler, vec4 P, float lod, ivec3 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureProjLodOffset (sampler1DShadow sampler, vec4 P, float lod, int offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>float textureProjLodOffset (sampler2DShadow sampler, vec4 P, float lod, ivec2 offset)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do a texture lookup as in <code>texture</code> but with explicit gradients. The partial derivatives of ( P ) are with respect to window ( x ) and window ( y ). Set</td>
<td></td>
</tr>
</tbody>
</table>
| \[
\frac{\partial s}{\partial x} = \begin{cases} 
\frac{\partial P}{\partial x} & \text{for a 1D texture} \\
\frac{\partial P.s}{\partial x} & \text{otherwise}
\end{cases}
\] |
| \[
\frac{\partial s}{\partial y} = \begin{cases} 
\frac{\partial P}{\partial y} & \text{for a 1D texture} \\
\frac{\partial P.s}{\partial y} & \text{otherwise}
\end{cases}
\] |
| \[
\frac{\partial t}{\partial x} = \begin{cases} 
0.0 & \text{for a 1D texture} \\
\frac{\partial P.t}{\partial x} & \text{otherwise}
\end{cases}
\] |
| \[
\frac{\partial t}{\partial y} = \begin{cases} 
0.0 & \text{for a 1D texture} \\
\frac{\partial P.t}{\partial y} & \text{otherwise}
\end{cases}
\] |
| \[
\frac{\partial r}{\partial x} = \begin{cases} 
0.0 & \text{for 1D or 2D} \\
\frac{\partial P.r}{\partial x} & \text{cube, other}
\end{cases}
\] |
| \[
\frac{\partial r}{\partial y} = \begin{cases} 
0.0 & \text{for 1D or 2D} \\
\frac{\partial P.r}{\partial y} & \text{cube, other}
\end{cases}
\] |

For the cube version, the partial derivatives of \( P \) are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face.
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gvec4 textureGradOffset (gsampler1D sampler, float P, float dPdx, float dPdy, int offset)</td>
<td>Do a texture lookup with both explicit gradient and offset, as described in <code>textureGrad</code> and <code>textureOffset</code>.</td>
</tr>
<tr>
<td>gvec4 textureGradOffset (gsampler2D sampler, vec2 P, vec2 dPdx, vec2 dPdy, ivec2 offset)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureGradOffset (gsampler3D sampler, vec3 P, vec3 dPdx, vec3 dPdy, ivec3 offset)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureGradOffset (gsampler2DRect sampler, vec2 P, vec2 dPdx, vec2 dPdy, ivec2 offset)</td>
<td></td>
</tr>
<tr>
<td>float textureGradOffset (sampler2DRectShadow sampler, vec3 P, vec2 dPdx, vec2 dPdy, ivec2 offset)</td>
<td></td>
</tr>
<tr>
<td>float textureGradOffset (sampler1DShadow sampler, vec3 P, float dPdx, vec2 dPdy, ivec2 offset)</td>
<td></td>
</tr>
<tr>
<td>float textureGradOffset (sampler2DShadow sampler, vec3 P, float dPdx, float dPdy, int offset)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureGradOffset (gsampler1DArray sampler, vec2 P, float dPdx, float dPdy, int offset)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureGradOffset (gsampler2DArray sampler, vec3 P, vec2 dPdx, vec2 dPdy, ivec2 offset)</td>
<td></td>
</tr>
<tr>
<td>float textureGradOffset (sampler1DArrayShadow sampler, vec3 P, float dPdx, vec2 dPdy, ivec2 offset)</td>
<td></td>
</tr>
<tr>
<td>float textureGradOffset (sampler2DArrayShadow sampler, vec4 P, vec2 dPdx, vec2 dPdy, ivec2 offset)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureProjGrad (gsampler1D sampler, vec2 P, float dPdx, float dPdy)</td>
<td>Do a texture lookup both projectively, as described in <code>textureProj</code>, and with explicit gradient as described in <code>textureGrad</code>. The partial derivatives <code>dPdx</code> and <code>dPdy</code> are assumed to be already projected.</td>
</tr>
<tr>
<td>gvec4 textureProjGrad (gsampler1D sampler, vec4 P, float dPdx, float dPdy)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureProjGrad (gsampler2D sampler, vec3 P, vec2 dPdx, vec2 dPdy)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureProjGrad (gsampler2D sampler, vec4 P, vec2 dPdx, vec2 dPdy)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureProjGrad (gsampler3D sampler, vec4 P, vec3 dPdx, vec2 dPdy)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureProjGrad (gsampler2DRect sampler, vec3 P, vec2 dPdx, vec2 dPdy)</td>
<td></td>
</tr>
<tr>
<td>gvec4 textureProjGrad (gsampler2DRect sampler, vec4 P, vec2 dPdx, vec2 dPdy)</td>
<td></td>
</tr>
<tr>
<td>float textureProjGrad (sampler2DRectShadow sampler, vec4 P, vec2 dPdx, vec2 dPdy)</td>
<td></td>
</tr>
<tr>
<td>float textureProjGrad (sampler1DShadow sampler, vec4 P, float dPdx, float dPdy)</td>
<td></td>
</tr>
<tr>
<td>float textureProjGrad (sampler2DShadow sampler, vec4 P, vec2 dPdx, vec2 dPdy)</td>
<td></td>
</tr>
</tbody>
</table>
8.9.3 Texture Gather Functions

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

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gvec4 textureGather(gsampler2D sampler, vec2 P [, int comp])</code></td>
<td>Returns the value vec4(Sample_i0_j1(P, base).comp, Sample_i1_j1(P, base).comp, Sample_i1_j0(P, base).comp, Sample_i0_j0(P, base).comp)</td>
</tr>
<tr>
<td><code>gvec4 textureGather(gsampler2DArray sampler, vec3 P [, int comp])</code></td>
<td>If specified, the value of <code>comp</code> must be a constant integer expression with a value of 0, 1, 2, or 3, identifying the x, y, z, or w post-swizzled component of the four-component vector lookup result for each texel, respectively. If <code>comp</code> is not specified, it is treated as 0, selecting the x component of each texel to generate the result.</td>
</tr>
<tr>
<td><code>gvec4 textureGather(gsamplerCube sampler, vec3 P [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureGather(gsamplerCubeArray sampler, vec4 P [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureGather(gsampler2DRect sampler, vec2 P [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGather(sampler2DShadow sampler, vec2 P, float refZ)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGather(sampler2DArrayShadow sampler, vec3 P, float refZ)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGather(samplerCubeShadow sampler, vec3 P, float refZ)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGather(samplerCubeArrayShadow sampler, vec4 P, float refZ)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGather(sampler2DRectShadow sampler, vec2 P, float refZ)</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureGatherOffset(gsampler2D sampler, vec2 P, ivec2 offset [, int comp])</code></td>
<td>Perform a texture gather operation as in <code>textureGather</code> by <code>offset</code> as described in <code>textureOffset</code> except that the <code>offset</code> can be variable (non constant) and the implementation-dependent minimum and maximum offset values are given by <code>MIN_PROGRAM_TEXTURE_GATHER_OFFSET</code> and <code>MAX_PROGRAM_TEXTURE_GATHER_OFFSET</code>, respectively.</td>
</tr>
<tr>
<td><code>gvec4 textureGatherOffset(gsampler2DArray sampler, vec3 P, ivec2 offset [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureGatherOffset(gsampler2DRect sampler, vec2 P, ivec2 offset [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGatherOffset(sampler2DShadow sampler, vec2 P, float refZ, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGatherOffset(sampler2DArrayShadow sampler, vec3 P, float refZ, ivec2 offset)</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGatherOffset(sampler2DRectShadow sampler, vec2 P, float refZ, ivec2 offset)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gvec4 textureGatherOffsets(</code></td>
<td><strong>Operate identically to</strong> <code>textureGatherOffset</code> <strong>except that</strong> <code>offsets</code> <strong>is used to determine the location of the four texels to sample. Each of the four texels is obtained by applying the corresponding offset in</strong> <code>offsets</code> <strong>as a</strong> <code>(u, v)</code> <strong>coordinate offset to</strong> <code>P</code>, <strong>identifying the four-texel LINEAR footprint, and then selecting the texel</strong> <code>i,j</code> <strong>of that footprint. The specified values in</strong> <code>offsets</code> <strong>must be set with constant integral expressions.</strong></td>
</tr>
<tr>
<td><code>gsampler2D sampler, vec2 P, ivec2 offsets[4] [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureGatherOffsets(</code></td>
<td></td>
</tr>
<tr>
<td><code>gsampler2DArray sampler, vec3 P, ivec2 offsets[4] [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>gvec4 textureGatherOffsets(</code></td>
<td></td>
</tr>
<tr>
<td><code>gsampler2DRect sampler, vec3 P, ivec2 offsets[4] [, int comp])</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGatherOffsets(</code></td>
<td></td>
</tr>
<tr>
<td><code>sampler2DShadow sampler, vec2 P, float refZ, ivec2 offsets[4])</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGatherOffsets(</code></td>
<td></td>
</tr>
<tr>
<td><code>sampler2DArrayShadow sampler, vec3 P, float refZ, ivec2 offsets[4])</code></td>
<td></td>
</tr>
<tr>
<td><code>vec4 textureGatherOffsets(</code></td>
<td></td>
</tr>
<tr>
<td><code>sampler2DRectShadow sampler, vec2 P, float refZ, ivec2 offsets[4])</code></td>
<td></td>
</tr>
</tbody>
</table>
### 8.9.4 Compatibility Profile Texture Functions

The following texture functions are only in the compatibility profile.

<table>
<thead>
<tr>
<th>Syntax (deprecated)</th>
<th>Description (deprecated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec4 <code>texture1D</code> (sampler1D <code>sampler</code>, float <code>coord</code> [, float <code>bias</code>] )</td>
<td>See corresponding signature above without “1D” in the name.</td>
</tr>
<tr>
<td>vec4 <code>texture1DProj</code> (sampler1D <code>sampler</code>, vec2 <code>coord</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture1DProj</code> (sampler1D <code>sampler</code>, vec4 <code>coord</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture1DLod</code> (sampler1D <code>sampler</code>, float <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture1DProjLod</code> (sampler1D <code>sampler</code>, vec2 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture1DProjLod</code> (sampler1D <code>sampler</code>, vec4 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture2D</code> (sampler2D <code>sampler</code>, vec2 <code>coord</code> [, float <code>bias</code>] )</td>
<td>See corresponding signature above without “2D” in the name.</td>
</tr>
<tr>
<td>vec4 <code>texture2DProj</code> (sampler2D <code>sampler</code>, vec3 <code>coord</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture2DProj</code> (sampler2D <code>sampler</code>, vec4 <code>coord</code> [, float <code>bias</code>] )</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture2DLod</code> (sampler2D <code>sampler</code>, vec2 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture2DProjLod</code> (sampler2D <code>sampler</code>, vec2 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture2DProjLod</code> (sampler2D <code>sampler</code>, vec4 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture3D</code> (sampler3D <code>sampler</code>, vec3 <code>coord</code> [, float <code>bias</code>] )</td>
<td>See corresponding signature above without “3D” in the name.</td>
</tr>
<tr>
<td>vec4 <code>texture3DProj</code> (sampler3D <code>sampler</code>, vec4 <code>coord</code> [, float <code>bias</code>] )</td>
<td>Use the texture coordinate <code>coord</code> to do a texture lookup in the 3D texture currently bound to <code>sampler</code>. For the projective (“Proj”) versions, the texture coordinate is divided by <code>coord.q</code>.</td>
</tr>
<tr>
<td>vec4 <code>texture3DLod</code> (sampler3D <code>sampler</code>, vec3 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>texture3DProjLod</code> (sampler3D <code>sampler</code>, vec4 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>vec4 <code>textureCube</code> (samplerCube <code>sampler</code>, vec3 <code>coord</code> [, float <code>bias</code>] )</td>
<td>See corresponding signature above without “Cube” in the name.</td>
</tr>
<tr>
<td>vec4 <code>textureCubeLod</code> (samplerCube <code>sampler</code>, vec3 <code>coord</code>, float <code>lod</code>)</td>
<td></td>
</tr>
<tr>
<td>Syntax (deprecated)</td>
<td>Description (deprecated)</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>vec4 <strong>shadow1D</strong> (sampler1DShadow sampler, vec3 coord [, float bias] )</td>
<td>Same functionality as the “<strong>texture</strong>” based names above with the same signature.</td>
</tr>
<tr>
<td>vec4 <strong>shadow2D</strong> (sampler2DShadow sampler, vec3 coord [, float bias] )</td>
<td></td>
</tr>
<tr>
<td>vec4 <strong>shadow1DProj</strong> (sampler1DShadow sampler, vec4 coord [, float bias] )</td>
<td></td>
</tr>
<tr>
<td>vec4 <strong>shadow2DProj</strong> (sampler2DShadow sampler, vec4 coord [, float bias] )</td>
<td></td>
</tr>
<tr>
<td>vec4 <strong>shadow1DLod</strong> (sampler1DShadow sampler, vec4 coord, float lod)</td>
<td></td>
</tr>
<tr>
<td>vec4 <strong>shadow2DLod</strong> (sampler2DShadow sampler, vec3 coord, float lod)</td>
<td></td>
</tr>
<tr>
<td>vec4 <strong>shadow1DProjLod</strong> (sampler1DShadow sampler, vec4 coord, float lod)</td>
<td></td>
</tr>
<tr>
<td>vec4 <strong>shadow2DProjLod</strong> (sampler2DShadow sampler, vec4 coord, float lod)</td>
<td></td>
</tr>
</tbody>
</table>
8.10 Atomic-Counter Functions

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

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

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

The underlying counter is a 32-bit unsigned integer. Increments and decrements at the limit of the range will wrap to \([0, 2^{32}-1]\).

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint atomicCounterIncrement (atomic_uint </code>c<code>)</code></td>
<td>Atomically 1. increments the counter for <code>c</code>, and 2. returns its value prior to the increment operation. These two steps are done atomically with respect to the atomic counter functions in this table.</td>
</tr>
<tr>
<td><code>uint atomicCounterDecrement (atomic_uint </code>c<code>)</code></td>
<td>Atomically 1. decrements the counter for <code>c</code>, and 2. returns the value resulting from the decrement operation. These two steps are done atomically with respect to the atomic counter functions in this table.</td>
</tr>
<tr>
<td><code>uint atomicCounter (atomic_uint </code>c<code>)</code></td>
<td>Returns the counter value for <code>c</code>.</td>
</tr>
</tbody>
</table>

8.11 Atomic Memory Functions

Atomic memory functions perform atomic operations on an individual signed or unsigned integer stored in buffer-object or shared-variable storage. All of the atomic memory operations read a value from memory, compute a new value using one of the operations described below, write the new value to memory, and return the original value read. The contents of the memory being updated by the atomic operation are guaranteed not to be modified by any other assignment or atomic memory function in any shader invocation between the time the original value is read and the time the new value is written.
Atomic memory functions are supported only for a limited set of variables. A shader will fail to compile if the value passed to the `mem` argument of an atomic memory function does not correspond to a buffer or shared variable. It is acceptable to pass an element of an array or a single component of a vector to the `mem` argument of an atomic memory function, as long as the underlying array or vector is a buffer or shared variable.

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint atomicAdd (inout uint mem, uint data)</code></td>
<td>Computes a new value by adding the value of <code>data</code> to the contents <code>mem</code>.</td>
</tr>
<tr>
<td><code>int atomicAdd (inout int mem, int data)</code></td>
<td>Computes a new value by taking the minimum of the value of <code>data</code> and the contents of <code>mem</code>.</td>
</tr>
<tr>
<td><code>uint atomicMin (inout uint mem, uint data)</code></td>
<td>Computes a new value by taking the maximum of the value of <code>data</code> and the contents of <code>mem</code>.</td>
</tr>
<tr>
<td><code>int atomicMin (inout int mem, int data)</code></td>
<td>Computes a new value by performing a bit-wise and of the value of <code>data</code> and the contents of <code>mem</code>.</td>
</tr>
<tr>
<td><code>uint atomicMax (inout uint mem, uint data)</code></td>
<td>Computes a new value by performing a bit-wise or of the value of <code>data</code> and the contents of <code>mem</code>.</td>
</tr>
<tr>
<td><code>int atomicMax (inout int mem, int data)</code></td>
<td>Computes a new value by performing a bit-wise exclusive or of the value of <code>data</code> and the contents of <code>mem</code>.</td>
</tr>
<tr>
<td><code>uint atomicAnd (inout uint mem, uint data)</code></td>
<td>Computes a new value by simply copying the value of <code>data</code>.</td>
</tr>
<tr>
<td><code>int atomicAnd (inout int mem, int data)</code></td>
<td>Computes the value of <code>compare</code> and the contents of <code>mem</code>. If the values are equal, the new value is given by <code>data</code>; otherwise, it is taken from the original contents of <code>mem</code>.</td>
</tr>
</tbody>
</table>

### 8.12 Image Functions

Variables using one of the image basic types may be used by the built-in shader image memory functions defined in this section to read and write individual texels of a texture. Each image variable references an image unit, which has a texture image attached.

When image memory functions below access memory, an individual texel in the image is identified using an `(i)`, `(i, j)`, or `(i, j, k)` coordinate corresponding to the values of `P`. For `image2DMS` and `image2DMSArray` variables (and the corresponding int/unsigned int types) corresponding to multi-sample textures, each texel may have multiple samples and an individual sample is identified using the...
integer sample parameter. The coordinates and sample number are used to select an individual texel in the manner described in section 8.25 “Texture Image Loads and Stores” of the OpenGL specification.

Loads and stores support float, integer, and unsigned integer types. The data types below starting “gimage” serve as placeholders meaning types starting either “image”, “iimage”, or “uimage” in the same way as gvec or gsampler in earlier sections.

The IMAGE_PARAMS in the prototypes below is a placeholder representing 33 separate functions, each for a different type of image variable. The IMAGE_PARAMS placeholder is replaced by one of the following parameter lists:

- gimage1D image, int P
- gimage2D image, ivec2 P
- gimage3D image, ivec3 P
- gimage2DRect image, ivec2 P
- gimageCube image, ivec3 P
- gimageBuffer image, int P
- gimage1DArray image, ivec2 P
- gimage2DArray image, ivec3 P
- gimageCubeArray image, ivec3 P
- gimage2DMS image, ivec2 P, int sample
- gimage2DMSArray image, ivec3 P, int sample

where each of the line represents one of three different image variable types, and image, P, and sample specify the individual texel to operate on. The method for identifying the individual texel operated on from image, P, and sample, and the method for reading and writing the texel are specified in section 8.25 “Texture Image Loads and Stores” of the OpenGL specification.

The atomic functions perform atomic operations on individual texels or samples of an image variable. Atomic memory operations read a value from the selected texel, compute a new value using one of the operations described below, write the new value to the selected texel, and return the original value read. The contents of the texel being updated by the atomic operation are guaranteed not to be modified by any other image store or atomic function between the time the original value is read and the time the new value is written.

Atomic memory operations are supported on only a subset of all image variable types; image must be either:

- a signed integer image variable (type starts “iimage”) and a format qualifier of r32i, used with a data argument of type int, or
- an unsigned image variable (type starts “uimage”) and a format qualifier of r32ui, used with a data argument of type uint.
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| int `imageSize` (gimage1D image)  
ivec2 `imageSize` (gimage2D image)  
ivec3 `imageSize` (gimage3D image)  
ivec2 `imageSize` (gimageCube image)  
ivec3 `imageSize` (gimageCubeArray image)  
ivec2 `imageSize` (gimageRect image)  
ivec2 `imageSize` (gimage1DArray image)  
ivec3 `imageSize` (gimage2DArray image)  
int `imageSize` (gimageBuffer image)  
ivec2 `imageSize` (gimage2DMS image)  
ivec3 `imageSize` (gimage2DMSArray image) | Returns the dimensions of the image or images bound to `image`. For arrayed images, the last component of the return value will hold the size of the array. Cube images only return the dimensions of one face, and number of cubes in the cube map array, if arrayed. |
| `gvec4 imageLoad` (readonly `IMAGE_PARAMS`) | Loads the texel at the coordinate `P` from the image unit `image` (in `IMAGE_PARAMS`). For multi-sample loads, the sample number is given by `sample`. When `image`, `P`, `sample` identify a valid texel, the bits used to represent the selected texel in memory are converted to a `vec4`, `ivec4`, or `uvec4` in the manner described in section 8.25 “Texture Image Loads and Stores” of the OpenGL Specification and returned. |
| void `imageStore` (writeonly `IMAGE_PARAMS`,  
gvec4 `data`) | Stores `data` into the texel at the coordinate `P` from the image specified by `image`. For multi-sample stores, the sample number is given by `sample`. When `image`, `P`, and `sample` identify a valid texel, the bits used to represent `data` are converted to the format of the image unit in the manner described in section 8.25 “Texture Image Loads and Stores” of the OpenGL Specification and stored to the specified texel. |
| `uint imageAtomicAdd` (`IMAGE_PARAMS`,  
uint `data`)  
`int imageAtomicAdd` (`IMAGE_PARAMS`,  
int `data`) | Computes a new value by adding the value of `data` to the contents of the selected texel. |
| `uint imageAtomicMin` (`IMAGE_PARAMS`,  
uint `data`)  
`int imageAtomicMin` (`IMAGE_PARAMS`,  
int `data`) | Computes a new value by taking the minimum of the value of `data` and the contents of the selected texel. |
| `uint imageAtomicMax` (`IMAGE_PARAMS`,  
uint `data`)  
`int imageAtomicMax` (`IMAGE_PARAMS`,  
int `data`) | Computes a new value by taking the maximum of the value `data` and the contents of the selected texel. |
8 Built-in Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint <code>imageAtomicAnd</code> <em>(IMAGE_PARAMS, uint data)</em></td>
<td>Computes a new value by performing a bit-wise and of the value of <em>data</em> and the contents of the selected texel.</td>
</tr>
<tr>
<td>int <code>imageAtomicAnd</code> <em>(IMAGE_PARAMS, int data)</em></td>
<td></td>
</tr>
<tr>
<td>uint <code>imageAtomicOr</code> <em>(IMAGE_PARAMS, uint data)</em></td>
<td>Computes a new value by performing a bit-wise or of the value of <em>data</em> and the contents of the selected texel.</td>
</tr>
<tr>
<td>int <code>imageAtomicOr</code> <em>(IMAGE_PARAMS, int data)</em></td>
<td></td>
</tr>
<tr>
<td>uint <code>imageAtomicXor</code> <em>(IMAGE_PARAMS, uint data)</em></td>
<td>Computes a new value by performing a bit-wise exclusive or of the value of <em>data</em> and the contents of the selected texel.</td>
</tr>
<tr>
<td>int <code>imageAtomicXor</code> <em>(IMAGE_PARAMS, int data)</em></td>
<td></td>
</tr>
<tr>
<td>uint <code>imageAtomicExchange</code> <em>(IMAGE_PARAMS, uint data)</em></td>
<td>Computes a new value by simply copying the value of <em>data</em>.</td>
</tr>
<tr>
<td>int <code>imageAtomicExchange</code> <em>(IMAGE_PARAMS, int data)</em></td>
<td></td>
</tr>
<tr>
<td>uint <code>imageAtomicCompSwap</code> <em>(IMAGE_PARAMS, uint compare, uint data)</em></td>
<td>Compares the value of <em>compare</em> and the contents of the selected texel. If the values are equal, the new value is given by <em>data</em>; otherwise, it is taken from the original value loaded from the texel.</td>
</tr>
<tr>
<td>int <code>imageAtomicCompSwap</code> <em>(IMAGE_PARAMS, int compare, int data)</em></td>
<td></td>
</tr>
</tbody>
</table>

8.13 Fragment Processing Functions

Fragment processing functions are only available in fragment shaders.

8.13.1 Derivative Functions

Derivatives may be computationally expensive and/or numerically unstable. Therefore, an OpenGL implementation may approximate the true derivatives by using a fast but not entirely accurate derivative computation. Derivatives are undefined within non-uniform control flow.

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

Forward differencing:

\[
F(x + dx) - F(x) \sim dFdx(x) \cdot dx \quad 1a
\]

\[
dFdx(x) \sim \frac{F(x + dx) - F(x)}{dx} \quad 1b
\]
Backward differencing:

\[ F(x - dx) - F(x) \sim -dFdx(x) \cdot dx \]  
\[ dFdx(x) \sim \frac{F(x) - F(x - dx)}{dx} \]

With single-sample rasterization, \( dx \leqslant 1.0 \) in equations 1b and 2b. For multi-sample rasterization, \( dx < 2.0 \) in equations 1b and 2b.

\( dFdy \) is approximated similarly, with \( y \) replacing \( x \).

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

1. The method may use piecewise linear approximations. Such linear approximations imply that higher order derivatives, \( dFdx(dFdx(x)) \) and above, are undefined.

2. The method may assume that the function evaluated is continuous. Therefore derivatives within non-uniform control flow are undefined.

3. The method may differ per fragment, subject to the constraint that the method may vary by window coordinates, not screen coordinates. The invariance requirement described in section 14.2 “Invariance” of the OpenGL Graphics System Specification, is relaxed for derivative calculations, because the method may be a function of fragment location.

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

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

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

6. Derivatives of constant arguments should be 0.

In some implementations, varying degrees of derivative accuracy may be obtained by providing GL hints (section 21.4 “Hints” of the OpenGL Graphics System Specification), allowing a user to make an image quality versus speed trade off.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genType dFdx (genType p)</td>
<td>Returns the derivative in x using local differencing for the input argument p.</td>
</tr>
</tbody>
</table>
8 Built-in Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| genType \( \text{dFdy} \) \( \text{genType } p \) | Returns the derivative in y using local differencing for the input argument \( p \).

These two functions are commonly used to estimate the filter width used to anti-alias procedural textures. We are assuming that the expression is being evaluated in parallel on a SIMD array so that at any given point in time the value of the function is known at the grid points represented by the SIMD array. Local differencing between SIMD array elements can therefore be used to derive \( \text{dFdx} \), \( \text{dFdy} \), etc.

| genType \text{fwidth} \( \text{genType } p \) | Returns the sum of the absolute derivative in x and y using local differencing for the input argument \( p \), i.e., \[ \text{abs}(\text{dFdx}(p)) + \text{abs}(\text{dFdy}(p)) \]; |

8.13.2 Interpolation Functions

Built-in interpolation functions are available to compute an interpolated value of a fragment shader input variable at a shader-specified \((x, y)\) location. A separate \((x, y)\) location may be used for each invocation of the built-in function, and those locations may differ from the default \((x, y)\) location used to produce the default value of the input.

For all of the interpolation functions, \text{interpolant} must be an input variable or an element of an input variable declared as an array. Component selection operators (e.g., \( .xy \)) may not be used when specifying \text{interpolant}. If \text{interpolant} is declared with a \text{flat} or \text{centroid} qualifier, the qualifier will have no effect on the interpolated value. If \text{interpolant} is declared with the \text{noperspective} qualifier, the interpolated value will be computed without perspective correction.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float \text{interpolateAtCentroid} ( \text{float } \text{interpolant} )</td>
<td>Return the value of the input \text{interpolant} sampled at a location inside the both the pixel and the primitive being processed. The value obtained would be the same value assigned to the input variable if declared with the \text{centroid} qualifier.</td>
</tr>
<tr>
<td>vec2 \text{interpolateAtCentroid} ( \text{vec2 } \text{interpolant} )</td>
<td></td>
</tr>
<tr>
<td>vec3 \text{interpolateAtCentroid} ( \text{vec3 } \text{interpolant} )</td>
<td></td>
</tr>
<tr>
<td>vec4 \text{interpolateAtCentroid} ( \text{vec4 } \text{interpolant} )</td>
<td></td>
</tr>
<tr>
<td>float \text{interpolateAtSample} ( \text{float } \text{interpolant} ), ( \text{int } \text{sample} )</td>
<td>Return the value of the input \text{interpolant} variable at the location of sample number \text{sample}. If multisample buffers are not available, the input variable will be evaluated at the center of the pixel. If sample \text{sample} does not exist, the position used to interpolate the input variable is undefined.</td>
</tr>
<tr>
<td>vec2 \text{interpolateAtSample} ( \text{vec2 } \text{interpolant} ), ( \text{int } \text{sample} )</td>
<td></td>
</tr>
<tr>
<td>vec3 \text{interpolateAtSample} ( \text{vec3 } \text{interpolant} ), ( \text{int } \text{sample} )</td>
<td></td>
</tr>
<tr>
<td>vec4 \text{interpolateAtSample} ( \text{vec4 } \text{interpolant} ), ( \text{int } \text{sample} )</td>
<td></td>
</tr>
</tbody>
</table>
8 Built-in Functions

Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float interpolateAtOffset (float interpolant, vec2 offset)</td>
<td>Return the value of the input interpolant variable sampled at an offset from the center of the pixel specified by offset. The two floating-point components of offset, give the offset in pixels in the x and y directions, respectively. An offset of (0, 0) identifies the center of the pixel. The range and granularity of offsets supported by this function is implementation-dependent.</td>
</tr>
<tr>
<td>vec2 interpolateAtOffset (vec2 interpolant, vec2 offset)</td>
<td></td>
</tr>
<tr>
<td>vec3 interpolateAtOffset (vec3 interpolant, vec2 offset)</td>
<td></td>
</tr>
<tr>
<td>vec4 interpolateAtOffset (vec4 interpolant, vec2 offset)</td>
<td></td>
</tr>
</tbody>
</table>

8.14 Noise Functions

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

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float noise1 (genType x)</td>
<td>Returns a 1D noise value based on the input value x.</td>
</tr>
<tr>
<td>vec2 noise2 (genType x)</td>
<td>Returns a 2D noise value based on the input value x.</td>
</tr>
<tr>
<td>vec3 noise3 (genType x)</td>
<td>Returns a 3D noise value based on the input value x.</td>
</tr>
<tr>
<td>vec4 noise4 (genType x)</td>
<td>Returns a 4D noise value based on the input value x.</td>
</tr>
</tbody>
</table>

8.15 Geometry Shader Functions

These functions are only available in geometry shaders. They are described in more depth following the table.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void EmitStreamVertex (int stream)</td>
<td>Emit the current values of output variables to the current output primitive on stream <em>stream</em>. The argument to <em>stream</em> must be a constant integral expression. On return from this call, the values of all output variables are undefined. Can only be used if multiple output streams are supported.</td>
</tr>
<tr>
<td>void EndStreamPrimitive (int stream)</td>
<td>Completes the current output primitive on stream <em>stream</em> and starts a new one. The argument to <em>stream</em> must be a constant integral expression. No vertex is emitted. Can only be used if multiple output streams are supported.</td>
</tr>
<tr>
<td>void EmitVertex ()</td>
<td>Emit the current values of output variables to the current output primitive. On return from this call, the values of output variables are undefined. When multiple output streams are supported, this is equivalent to calling EmitStreamVertex(0).</td>
</tr>
<tr>
<td>void EndPrimitive ()</td>
<td>Completes the current output primitive and starts a new one. No vertex is emitted. When multiple output streams are supported, this is equivalent to calling EndStreamPrimitive(0).</td>
</tr>
</tbody>
</table>

The function `EmitStreamVertex()` specifies that a vertex is completed. A vertex is added to the current output primitive in vertex stream *stream* using the current values of all output variables associated with
stream. These include \texttt{gl_PointSize}, \texttt{gl_ClipDistance}, \texttt{gl_Layer}, \texttt{gl_Position}, \texttt{gl_PrimitiveID} and \texttt{gl_ViewportIndex}. The values of all output variables for all output streams are undefined after a call to \texttt{EmitStreamVertex()}. If a geometry shader invocation has emitted more vertices than permitted by the output layout qualifier \texttt{max_vertices}, the results of calling \texttt{EmitStreamVertex()} are undefined.

The function \texttt{EndStreamPrimitive()} specifies that the current output primitive for vertex stream \texttt{stream} is completed and a new output primitive (of the same type) will started by any subsequent \texttt{EmitStreamVertex()}. This function does not emit a vertex. If the output layout is declared to be “points”, calling \texttt{EndStreamPrimitive()} is optional.

A geometry shader starts with an output primitive containing no vertices for each stream. When a geometry shader terminates, the current output primitive for each stream is automatically completed. It is not necessary to call \texttt{EndStreamPrimitive()} if the geometry shader writes only a single primitive.

Multiple output streams are supported only if the output primitive type is declared to be \texttt{points}. A program will fail to link if it contains a geometry shader calling \texttt{EmitStreamVertex()} or \texttt{EndStreamPrimitive()} if its output primitive type is not \texttt{points}. 


8.16 Shader Invocation Control Functions

The shader invocation control function is available only in tessellation control shaders and compute shaders. It is used to control the relative execution order of multiple shader invocations used to process a patch (in the case of tessellation control shaders) or a local work group (in the case of compute shaders), which are otherwise executed with an undefined relative order.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void <code>barrier()</code></td>
<td>For any given static instance of <code>barrier()</code>, all tessellation control shader invocations for a single input patch must enter it before any will be allowed to continue beyond it, or all invocations for a single work group must enter it before any will continue beyond it.</td>
</tr>
</tbody>
</table>

The function `barrier()` provides a partially defined order of execution between shader invocations. This ensures that values written by one invocation prior to a given static instance of `barrier()` can be safely read by other invocations after their call to the same static instance `barrier()`. Because invocations may execute in undefined order between these barrier calls, the values of a per-vertex or per-patch output variable or shared variables for compute shaders will be undefined in a number of cases enumerated in section 4.3.6 “Output Variables” (for tessellation control shaders) and section 4.3.8 "Shared Variables" (for compute shaders).

The `barrier()` function may only be placed inside the function `main()` of the tessellation control shader and may not be called within any control flow. Barriers are also disallowed after a return statement in the function `main()`. Any such misplaced barriers result in a compile-time error.
8 Built-in Functions

8.17 Shader Memory Control Functions

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void memoryBarrier ()</td>
<td>Control the ordering of memory transactions issued by a single shader invocation.</td>
</tr>
<tr>
<td>void memoryBarrierAtomicCounter ()</td>
<td>Control the ordering of accesses to atomic-counter variables issued by a single shader invocation.</td>
</tr>
<tr>
<td>void memoryBarrierBuffer ()</td>
<td>Control the ordering of memory transactions to buffer variables issued within a single shader invocation.</td>
</tr>
<tr>
<td>void memoryBarrierShared ()</td>
<td>Control the ordering of memory transactions to shared variables issued within a single shader invocation. Only available in compute shaders.</td>
</tr>
<tr>
<td>void memoryBarrierImage ()</td>
<td>Control the ordering of memory transactions to images issued within a single shader invocation.</td>
</tr>
<tr>
<td>void groupMemoryBarrier ()</td>
<td>Control the ordering of all memory transactions issued within a single shader invocation, as viewed by other invocations in the same work group. Only available in compute shaders.</td>
</tr>
</tbody>
</table>

The memory barrier built-in functions can be used to order reads and writes to variables stored in memory accessible to other shader invocations. When called, these functions will wait for the completion of all reads and writes previously performed by the caller that access selected variable types, and then return with no other effect. The built-in functions memoryBarrierAtomicCounter(), memoryBarrierBuffer(), memoryBarrierImage(), and memoryBarrierShared() wait for the completion of accesses to atomic counter, buffer, image, and shared variables, respectively. The built-in functions memoryBarrier() and groupMemoryBarrier() wait for the completion of accesses to all of the above variable types. The functions memoryBarrierShared() and groupMemoryBarrier() are available only in compute shaders; the other functions are available in all shader types.
When these functions return, the results of any memory stores performed using coherent variables performed prior to the call will be visible to any future coherent access to the same memory performed by any other shader invocation. In particular, the values written this way in one shader stage are guaranteed to be visible to coherent memory accesses performed by shader invocations in subsequent stages when those invocations were triggered by the execution of the original shader invocation (e.g., fragment shader invocations for a primitive resulting from a particular geometry shader invocation).

Additionally, memory barrier functions order stores performed by the calling invocation, as observed by other shader invocations. Without memory barriers, if one shader invocation performs two stores to coherent variables, a second shader invocation might see the values written by the second store prior to seeing those written by the first. However, if the first shader invocation calls a memory barrier function between the two stores, selected other shader invocations will never see the results of the second store before seeing those of the first. When using the function `groupMemoryBarrier()`, this ordering guarantee applies only to other shader invocations in the same compute shader work group; all other memory barrier functions provide the guarantee to all other shader invocations. No memory barrier is required to guarantee the order of memory stores as observed by the invocation performing the stores; an invocation reading from a variable that it previously wrote will always see the most recently written value unless another shader invocation also wrote to the same memory.
The grammar is fed from the output of lexical analysis. The tokens returned from lexical analysis are:

- **CONST** BOOL FLOAT DOUBLE INT UINT
- **BREAK** CONTINUE **DO** ELSE **FOR** IF **DISCARD** RETURN **SWITCH** CASE **DEFAULT** **SUBROUTINE**
- **BVEC2** BVEC3 BVEC4 **IVEC2** IVEC3 IVEC4 **UVEC2** UVEC3 UVEC4 **VEC2** VEC3 VEC4
- **MAT2** MAT3 MAT4 CENTROID IN OUT INOUT
- **UNIFORM** PATCH SAMPLE BUFFER SHARED
- **COHERENT** VOLATILE **RESTRICT** READONLY **WRITEONLY**
- **DVEC2** DVEC3 DVEC4 **DMAT2** DMAT3 DMAT4
- **NOPERSPECTIVE** FLAT SMOOTH LAYOUT
- **MAT2X2** MAT2X3 MAT2X4
- **MAT3X2** MAT3X3 MAT3X4
- **MAT4X2** MAT4X3 MAT4X4
- **DMAT2X2** DMAT2X3 DMAT2X4
- **DMAT3X2** DMAT3X3 DMAT3X4
- **DMAT4X2** DMAT4X3 DMAT4X4
- **ATOMIC_UINT**
- **SAMPLER1D** **SAMPLER2D** **SAMPLER3D** **SAMPLERCUBE** **SAMPLER1DARRAY** **SAMPLER2DARRAY** **SAMPLER1DARRAYSHADOW** **SAMPLER2DARRAYSHADOW**
- **SAMPLERCUBESHADOW** **SAMPLER1DARRAY** **SAMPLER2DARRAY** **SAMPLER1DARRAYSHADOW**
- **SAMPLERBUFFER** **SAMPLERBUFFER**
- **SAMPLER2DRECT** **SAMPLER2DRECTSHADOW** **ISAMPLER2DRECT** **USAMPLER2DRECT**
- **SAMPLERCUBEARRAY** **SAMPLERCUBEARRAYSHADOW**
- **SAMPLERCUBEARRAY** **SAMPLERCUBEARRAY**
- **SAMPLER2DMS** **SAMPLER2DMS**
- **SAMPLER2DMSARRAY** **SAMPLER2DMSARRAY**
- **IMAGE1D** **IMAGE1D** **IMAGE1D** **IMAGE2D** **IMAGE2D**
- **UIMAGE2D** **IMAGE2D** **IMAGE3D** **IMAGE3D** **IMAGE2D**
- **IMAGE2DRECT** **IMAGE2DRECT** **IMAGE2DRECT**
- **IMAGECUBE** **IMAGECUBE** **IMAGECUBE**
- **IMAGEBUFFER** **IMAGEBUFFER**
- **IMAGE1DARRAY** **IMAGE1DARRAY** **IMAGE1DARRAY**
- **IMAGE2DARRAY** **IMAGE2DARRAY** **IMAGE2DARRAY**
- **IMAGE2DARRAY** **IMAGE2DARRAY**
- **IMAGE2DARRAY** **IMAGE2DARRAY** **IMAGE2DARRAY**
- **IMAGE2DARRAY** **IMAGE2DARRAY** **IMAGE2DARRAY**
- **IMAGE2DMS** **IMAGE2DMS** **IMAGE2DMS**
- **IMAGE2DMSARRAY** **IMAGE2DMSARRAY** **IMAGE2DMSARRAY**

175
The following describes the grammar for the OpenGL Shading Language in terms of the above tokens.

variable_identifier:
  IDENTIFIER

primary_expression:
  variable_identifier
  INTCONSTANT
  UINTCONSTANT
  FLOATCONSTANT
  BOOLCONSTANT
  DOUBLECONSTANT
  LEFT_PAREN expression RIGHT_PAREN

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

integer_expression:
  expression
function_call:
  function_call_or_method

function_call_or_method:
  function_call_generic

function_call_generic:
  function_call_header_with_parameters RIGHT_PAREN
  function_call_header_no_parameters RIGHT_PAREN

function_call_header_no_parameters:
  function_call_header VOID
  function_call_header

function_call_header_with_parameters:
  function_call_header_assignment_expression
  function_call_header_with_parameters COMMA assignment_expression

function_call_header:
  function_identifier LEFT_PAREN

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

function_identifier:
  typeSpecifier
  postfix_expression

unary_expression:
  postfix_expression
  INC_OP unary_expression
  DEC_OP unary_expression
  unary_operator unary_expression

// Grammar Note: No traditional style type casts.

unary_operator:
  PLUS
DASH
BANG
TILDE

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

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

additive_expression:
    multiplicative_expression
    additive_expression PLUS multiplicative_expression
    additive_expression DASH multiplicative_expression

shift_expression:
    additive_expression
    shift_expression LEFT_OP additive_expression
    shift_expression RIGHT_OP additive_expression

relational_expression:
    shift_expression
    relational_expression LEFT_ANGLE shift_expression
    relational_expression RIGHT_ANGLE shift_expression
    relational_expression LE_OP shift_expression
    relational_expression GE_OP shift_expression

equality_expression:
    relational_expression
    equality_expression EQ_OP relational_expression
    equality_expression NE_OP relational_expression

and_expression:
    equality_expression
    and_expression AMPERSAND equality_expression

exclusive_or_expression:
    and_expression
exclusive_or_expression CARET and_expression

inclusive_or_expression:
  exclusive_or_expression
  inclusive_or_expression VERTICAL_BAR exclusive_or_expression

logical_and_expression:
  inclusive_or_expression
  logical_and_expression AND_OP inclusive_or_expression

logical_xor_expression:
  logical_and_expression
  logical_xor_expression XOR_OP logical_and_expression

logical_or_expression:
  logical_xor_expression
  logical_or_expression OR_OP logical_xor_expression

conditional_expression:
  logical_or_expression
  logical_or_expression QUESTION expression COLON assignment_expression

assignment_expression:
  conditional_expression
  unary_expression assignment_operator assignment_expression

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

equation: 179


assignment_expression
expression COMMA assignment_expression

constant_expression:
conditional_expression

declaration:
  function_prototype SEMICOLON
  init_declarator_list SEMICOLON
  PRECISION precision_qualifier type_specifier SEMICOLON
  type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE SEMICOLON
  type_qualifier IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
  IDENTIFIER SEMICOLON
  type_qualifier SEMICOLON
  type_qualifier IDENTIFIER SEMICOLON
  type_qualifier IDENTIFIER identifier_list SEMICOLON

identifier_list:
  COMMA IDENTIFIER
  identifier_list COMMA IDENTIFIER

function_prototype:
  function_declarator RIGHT_PAREN

function_declarator:
  function_header
  function_header_with_parameters

function_header_with_parameters:
  function_header parameter_declaration
  function_header_with_parameters COMMA parameter_declaration

function_header:
  fully_specified_type IDENTIFIER LEFT_PAREN

parameter_declarator:
  type_specifier IDENTIFIER
  type_specifier IDENTIFIER arraySpecifier
parameter_declaration:
    type_qualifier parameter_declarator
    parameter_declarator
    type_qualifier parameter_type_specifier
    parameter_type_specifier

parameter_type_specifier:
    type_specifier

init_declarator_list:
    single_declaration
    init_declarator_list COMMA IDENTIFIER
    init_declarator_list COMMA IDENTIFIER array_specifier
    init_declarator_list COMMA IDENTIFIER array_specifier EQUAL initializer
    init_declarator_list COMMA IDENTIFIER EQUAL initializer

single_declaration:
    fully_specified_type
    fully_specified_type IDENTIFIER
    fully_specified_type IDENTIFIER array_specifier
    fully_specified_type IDENTIFIER array_specifier EQUAL initializer
    fully_specified_type IDENTIFIER EQUAL initializer

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

fully_specified_type:
    type_specifier
    type_qualifier type_specifier

invariant_qualifier:
    INVARIANT

interpolation_qualifier:
    SMOOTH
    FLAT
    NOPERSPECTIVE
layout_qualifier:
  LAYOUT LEFT_PAREN layout_qualifier_id_list RIGHT_PAREN

layout_qualifier_id_list:
  layout_qualifier_id
  layout_qualifier_id_list COMMA layout_qualifier_id

layout_qualifier_id:
  IDENTIFIER
  IDENTIFIER EQUAL INTCONSTANT

precise_qualifier:
  PRECISE

type_qualifier:
  single_type_qualifier
  type_qualifier single_type_qualifier

single_type_qualifier:
  storage_qualifier
  layout_qualifier
  precision_qualifier
  interpolation_qualifier
  invariant_qualifier
  precise_qualifier

storage_qualifier:
  CONST
  INOUT
  IN
  OUT
  CENTROID
  PATCH
  SAMPLE
  UNIFORM
  BUFFER
  SHARED
  COHERENT
  VOLATILE
RESTRICT
READONLY
WRITEONLY
SUBROUTINE
SUBROUTINE LEFT_PAREN type_name_list RIGHT_PAREN

type_name_list:
   TYPE_NAME
   type_name_list COMMA TYPE_NAME

type_specifier:
   type_specifier_nonarray
   type_specifier_nonarray array_specifier

array_specifier:
   LEFT_BRACKET RIGHT_BRACKET
   LEFT_BRACKET constant_expression RIGHT_BRACKET
   array_specifier LEFT_BRACKET RIGHT_BRACKET
   array_specifier LEFT_BRACKET constant_expression RIGHT_BRACKET

type_specifier_nonarray:
   VOID
   FLOAT
   DOUBLE
   INT
   UINT
   BOOL
   VEC2
   VEC3
   VEC4
   DVEC2
   DVEC3
   DVEC4
   BVEC2
   BVEC3
   BVEC4
   IVEC2
IVEC3
IVEC4
UVEC2
UVEC3
UVEC4
MAT2
MAT3
MAT4
MAT2X2
MAT2X3
MAT2X4
MAT3X2
MAT3X3
MAT3X4
MAT4X2
MAT4X3
MAT4X4
DMAT2
DMAT3
DMAT4
DMAT2X2
DMAT2X3
DMAT2X4
DMAT3X2
DMAT3X3
DMAT3X4
DMAT4X2
DMAT4X3
DMAT4X4
ATOMIC_UINT
SAMPLER1D
SAMPLER2D
SAMPLER3D
SAMPLERCUBE
SAMPLER1DSHADOW
SAMPLER2DSHADOW
SAMPLERCUBESHADOW
SAMPLER1DARRAY
SAMPLER2DARRAY
SAMPLER1DARRAYSHADOW
SAMPLER2DARRAYSHADOW
SAMPLERCUBEARRAY
SAMPLERCUBEARRAYSHADOW
ISAMPLER1D
ISAMPLER2D
ISAMPLER3D
ISAMPLERCUBE
ISAMPLER1DARRAY
ISAMPLER2DARRAY
ISAMPLERCUBEARRAY
USAMPLER1D
USAMPLER2D
USAMPLER3D
USAMPLERCUBE
USAMPLER1DARRAY
USAMPLER2DARRAY
USAMPLERCUBEARRAY
SAMPLER2DRECT
SAMPLER2DRECTSHADOW
ISAMPLER2DRECT
USAMPLER2DRECT
SAMPLERBUFFER
ISAMPLERBUFFER
USAMPLERBUFFER
SAMPLER2DMS
ISAMPLER2DMS
USAMPLER2DMS
SAMPLER2DMSARRAY
ISAMPLER2DMSARRAY
USAMPLER2DMSARRAY
IMAGE1D
IMAGE1D
UIMAGE1D
IMAGE2D
IIMAGE2D
UIMAGE2D
IMAGE3D
IIIMAGE3D
UIMAGE3D
IMAGE2DRECT
IIIMAGE2DRECT
UIMAGE2DRECT
IMAGECUBE
IIIMAGECUBE
UIMAGECUBE
IMAGEBUFFER
IIIMAGEBUFFER
UIMAGEBUFFER
IMAGE1DARRAY
IIIMAGE1DARRAY
UIMAGE1DARRAY
IMAGE2DARRAY
IIIMAGE2DARRAY
UIMAGE2DARRAY
IMAGECUBEARRAY
IIIMAGECUBEARRAY
UIMAGECUBEARRAY
IMAGE2DMS
IIIMAGE2DMS
UIMAGE2DMS
IMAGE2DMSARRAY
IIIMAGE2DMSARRAY
UIMAGE2DMSARRAY
struct_specifier
TYPE_NAME
precision_qualifier:
HIGH_PRECISION
MEDIUM_PRECISION
LOW_PRECISION
struct_specifier:
STRUCT IDENTIFIER LEFT_BRACE struct_declaration_list RIGHT_BRACE
STRUCT LEFT_BRACE struct_declaration_list RIGHT_BRACE
struct_declaration_list:
    struct_declaration
    struct_declaration_list struct_declaration

struct_declaration:
    type_specifier struct_declarator_list SEMICOLON
    type_qualifier type_specifier struct_declarator_list SEMICOLON

struct_declarator_list:
    struct_declarator
    struct_declarator_list COMMA struct_declarator

struct_declarator:
    IDENTIFIER
    IDENTIFIER array_specifier

initializer:
    assignment_expression
    LEFT_BRACE initializer_list RIGHT_BRACE
    LEFT_BRACE initializer_list COMMA RIGHT_BRACE

initializer_list:
    initializer
    initializer_list COMMA initializer

declaration_statement:
    declaration

statement:
    compound_statement
    simple_statement

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

simple_statement:
    declaration_statement
    expression_statement
    selection_statement
    switch_statement
case_label
iteration_statement
jump_statement

compound_statement:
    LEFT_BRACE RIGHT_BRACE
    LEFT_BRACE statement_list RIGHT_BRACE

statement_no_new_scope:
    compound_statement_no_new_scope
    simple_statement

compound_statement_no_new_scope:
    LEFT_BRACE RIGHT_BRACE
    LEFT_BRACE statement_list RIGHT_BRACE

statement_list:
    statement
    statement_list statement

expression_statement:
    SEMICOLON
    expression SEMICOLON

selection_statement:
    IF LEFT_PAREN expression RIGHT_PAREN selection_rest_statement

selection_rest_statement:
    statement ELSE statement
    statement

condition:
    expression
    fully_specified_type IDENTIFIER EQUAL initializer

switch_statement:
    SWITCH LEFT_PAREN expression RIGHT_PAREN LEFT_BRACE switch_statement_list
    RIGHT_BRACE

switch_statement_list:
    /* nothing */
statement_list

case_label:
  CASE expression COLON
  DEFAULT COLON

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

for_init_statement:
  expression_statement
  declaration_statement

conditionopt:
  condition
  /* empty */

for_rest_statement:
  conditionopt SEMICOLON
  conditionopt SEMICOLON expression

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

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

translation_unit:
  external_declaration
  translation_unit external_declaration

external_declaration:
  function_definition
  declaration
function_definition:
  function_prototype compound_statement_no_new_scope