Features of the OpenGL Shading Language

by John Kessenich
3Dlabs Inc. Ltd.

03-May-2005

In this white paper, we present the language features of the OpenGL Shading Language. We start with a simple example of a working pair of vertex and fragment shaders to show their basic structure and interfaces. Each aspect of the language will then be discussed in turn.

The OpenGL Shading Language syntax comes from the C family of programming languages. Tokens, identifiers, semicolons, nesting with curly braces, control-flow, and many keywords look like C. Both comment styles **// ...** and **/* ... */** are accepted. Much is also different though, and all important differences from C will be discussed.

Each shader example is presented as it might appear in a file or onscreen. However, as explained in the OpenGL 2.0 specification, the OpenGL API passes shaders as strings, not files, as OpenGL does not consider shaders file based.

1.1 Example Shader Pair

A program will typically contain two shaders: one vertex shader and one fragment shader. More than one shader of each type can be present, but there must be exactly one function **main** between all the fragment shaders and exactly one function **main** between all the vertex shaders. Frequently, it's easiest to just have one shader of each type.

The following is a simple pair of vertex and fragment shaders that can smoothly express a surface's temperature with color. The range of temperatures and their colors are parameterized. First, we'll show the vertex shader. It will be executed once for each vertex.

```
// uniform qualified variables are changed at most once per primitive
uniform float CoolestTemp;
uniform float TempRange;

// attribute qualified variables are typically changed per vertex
attribute float VertexTemp;

// varying qualified variables communicate from the vertex shader to
// the fragment shader
varying float Temperature;

void main()
{
    // compute a temperature to be interpolated per fragment,
    // in the range [0.0, 1.0]
    Temperature = (VertexTemp - CoolestTemp) /  TempRange;

    /*
       The vertex position written in the application using
       glVertex() can be read from the built-in variable
```

```
       gl_Vertex. Use this value and the current model
       view transformation matrix to tell the rasterizer where
       this vertex is.
    */
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

That's it for the vertex shader. Primitive assembly will follow the preceding vertex processing, providing the rasterizer with enough information to create fragments. The rasterizer interpolates the *Temperature* values written per vertex to create values per fragment. Each fragment is then delivered to a single execution of the fragment shader, as follows:

```
// uniform qualified variables are changed at most once per primitive
// by the application, and vec3 declares a vector of three
// floating-point numbers
uniform vec3 CoolestColor;
uniform vec3 HottestColor;

// Temperature contains the now interpolated per-fragment
// value of temperature set by the vertex shader
varying float Temperature;

void main()
{
    // get a color between coolest and hottest colors, using
    // the mix() built-in function
    vec3 color = mix(CoolestColor, HottestColor, Temperature);

    // make a vector of 4 floating-point numbers by appending an
    // alpha of 1.0, and set this fragment's color
    gl_FragColor = vec4(color, 1.0);
}
```

Both shaders receive user-defined state from the application through the declared **uniform** qualified variables. The vertex shader gets information associated with each vertex through the **attribute** qualified variable. Information is passed from the vertex shader to the fragment shader through **varying** qualified variables, whose declarations must match between the vertex and fragment shaders. The fixed functionality located between the vertex and fragment processors will interpolate the per-vertex values written to this varying variable. When the fragment shader reads this same varying variable, it reads the value interpolated for the location of the fragment being processed.

Shaders interact with the fixed functionality OpenGL pipeline by writing built-in variables. OpenGL prefixes built-in variables with "gl_". In the preceding examples, writing to *gl_Position* tells the OpenGL pipeline where the transformed vertices are located, and writing to *gl_FragColor* tells the OpenGL pipeline what color to attach to a fragment.

Execution of the preceding shaders occurs multiple times to process a single primitive, once per vertex for the vertex shader and once per fragment for the fragment shader. Many such executions of the same shader can happen in parallel. In general, there is no direct tie or ordering between shader executions. Information can be communicated neither from vertex to vertex, nor from fragment to fragment.

1.2 Data Types

We saw vectors of floating-point numbers in the example in the previous section. Many other built-in data types are available to ease the expression of graphical operations. Booleans, integers, matrices, vectors of other types, structures, and arrays are all included. Each is discussed in the following sections. Notably missing are string and character types, as there is little use for them in processing vertex and fragment data.

1.2.1 Scalars

The scalar types available are

float	declares a single floating-point number
int	declares a single integer number
bool	declares a single Boolean number

These are used to declare variables, as is familiar from C/C++.

```
float f;
float g, h = 2.4;
int NumTextures = 4;
bool skipProcessing;
```

Unlike the original C, you must provide the type name, as there are no default types. As in C++, declarations may appear when needed, not just after an open curly brace (**{**).

Literal floating-point numbers are also specified as in C, except there are no suffixes to specify precision, as there is only one floating-point type.

```
3.14159
3.
0.2
.609
1.5e10
0.4E-4
etc.
```

In general, floating-point values and operations act as they do in C.

Integers are not the same as in C. There is no requirement that they appear to be backed in hardware by a fixed-width integer register. Consequently, wrapping behavior, when arithmetic would overflow or underflow a fixed-width integer register, is undefined. Bit-wise operations like left-shift (<<) and bit-wise and (**&**) are also not supported.

What can be said about integers? They are guaranteed to have at least 16 bits of precision; they can be positive, negative, or zero; and integer arithmetic that stays within this range will give the expected results. Note that the precision truly is 16 bits plus the sign of the value—that is, a full range of [-65535, 65535] or greater.

Literal integers can be given as decimal values, octal values, or hexadecimal values, as in C.

```
42    // a literal decimal integer
052   // a literal octal integer
0x2A  // a literal hexadecimal integer
```

Again, there are no suffixes to specify precision, as there is only one integer type. Integers are useful as sizes of structures or arrays and as loop counters. Graphical types, such as color or position, are best expressed in floating-point variables within a shader.

Boolean variables are as **bool** in C++. They can have only one of two values: true or false. Literal Boolean constants **true** and **false** are provided. Relational operators like less-than (<) and logical operators like logical-and (**&&**) always result in Boolean type. Flow-control constructs like **if-else** will accept only Boolean-typed expressions. In these regards, the OpenGL Shading Language is more restrictive than C++.

1.2.2 Vectors

Vectors of **float, int,** or **bool** are built-in basic types. They can have two, three, or four components and are named as follows:

vec2	Vector of two floating-point numbers
vec3	Vector of three floating-point numbers

vec4	Vector of four floating-point numbers
ivec2	Vector of two integers
ivec3	Vector of three integers
ivec4	Vector of four integers
bvec2	Vector of two Booleans
bvec3	Vector of three Booleans
bvec4	Vector of four Booleans

Built-in vectors are quite useful. They conveniently store and manipulate colors, positions, texture coordinates, and so on. Built-in variables and built-in functions make heavy use of these types. Also, special operations are supported. Finally, hardware is likely to have vector-processing capabilities that mirror vector expressions in shaders.

Note that the language does not distinguish between a color vector and a position vector or other uses of a floating-point vector. These are all just floating-point vectors from the language's perspective.

Special features of vectors include component access that can be done either through field selection (as with structures) or as array accesses. For example, if *position* is a **vec3**, it can be considered as the vector (*x*, *y*, *z*), and *position.x* will select the first component of the vector.

In all, the following names are available for selecting components of vectors:

x, y, z, w	Treat a vector as a position or direction
r, g, b, a	Treat a vector as a color
s, t, p, q	Treat a vector as a texture coordinate

There is no explicit way of stating a vector is a color, a position, a coordinate, and so on. Rather, these component selection names are provided simply for readability in a shader. The only compile-time checking done is that the vector is large enough to provide a specified component. Also, if multiple components are selected (swizzling, discussed in Section 1.7.2), all the components are from the same family.

Vectors can also be indexed as a zero-based array to obtain components. For instance, *position[2]* returns the third component of *position*. Variable indices are allowed, making it possible to loop over the components of a vector. Multiplication takes on special meaning when operating on a vector, as linear algebraic multiplies with matrices are understood. Swizzling, indexing, and other operations are discussed in detail in section Section 1.7.

1.2.3 Matrices

Built-in types are available for matrices of floating-point numbers. There are 2 × 2, 3 × 3, and 4 × 4 sizes.

mat2	2 × 2 matrix of floating-point numbers
mat3	3 × 3 matrix of floating-point numbers
mat4	4 × 4 matrix of floating-point numbers

These are useful for storing linear transforms or other data. They are treated semantically as matrices, particularly when a vector and a matrix are multiplied together, in which case the proper linear-algebraic computation is performed. When relevant, matrices are organized in column-major order, as is the tradition in OpenGL.

You may access a matrix as an array of column vectors—that is, if *transform* is a **mat4**, *transform[2]* is the third column of *transform.* The resulting type of *transform[2]* is **vec4**. Column 0 is the first column. Because *transform[2]* is a vector and you can also treat vectors as arrays, *transform[3][1]* is the second component of the vector forming the fourth column of *transform*. Hence, it ends up looking as if *transform* is a two-dimensional array. Just remember that the first index selects the column, not the row, and the second index selects the row.

1.2.4 Samplers

Texture lookups require some indication as to what texture and/or texture unit will do the lookup. The OpenGL Shading Language doesn't really care about the underlying implementation of texture units or other forms of organizing texture lookup hardware. Hence, it provides a simple opaque handle to encapsulate what to look up. These handles are called SAMPLERS. The sampler types available are

sampler1D	Accesses a one-dimensional texture
sampler2D	Accesses a two-dimensional texture
sampler3D	Accesses a three-dimensional texture
samplerCube	Accesses a cube-mapped texture
sampler1DShadow	Accesses a one-dimensional depth texture with comparison
sampler2DShadow	Accesses a two-dimensional depth texture with comparison

When the application initializes a sampler, the OpenGL implementation stores into it whatever information is needed to communicate the texture to be accessed. Shaders cannot themselves initialize samplers. They can only receive them from the application, through a **uniform** qualified sampler, or pass them on to user or built-in functions. As a function parameter, a sampler cannot be modified, so there is no way for a shader to change a sampler's value.

For example, a sampler could be declared as

```
uniform sampler2D Grass;
```

(Uniform qualifiers are discussed in more detail in Section 1.5.)

This variable can then be passed into a corresponding texture lookup function to access a texture:

```
vec4 color = texture2D(Grass, coord);
```

where *coord* is a **vec2** holding the two-dimensional position used to index the grass texture, and *color* is the result of doing the texture lookup. Together, the compiler and the OpenGL API will validate that *Grass* really references a two-dimensional texture and that *Grass* is only passed into two-dimensional texture lookups.

Shaders may not manipulate sampler values. For example, the expression `Grass + 1` is not allowed. If a shader wants to combine multiple textures procedurally, an array of samplers can be used as shown here:

```
const int NumTextures = 4;
uniform sampler2D textures[NumTextures];
```

These can be processed in a loop:

```
for (int i = 0; i < NumTextures; ++i)
    … = texture2D(textures[i], …);
```

The idiom `Grass + 1` could then become something like `textures[GrassIndex + 1]`, which is a valid way of manipulating the sampler to be used.

1.2.5 Structures

The OpenGL Shading Language provides user-defined structures similar to C. For example,

```
struct light
{
    vec3 position;
    vec3 color;
};
```

As in C++, the name of the structure is the name of this new user-defined type. No **typedef** is needed. In fact, the **typedef** keyword is still reserved, as there is not yet a need for it. A variable of type *light* from the preceding example is simply declared as

```
light ceilingLight;
```

Most other aspects of structures mirror C. They can be embedded and nested. Embedded structure type names have the same scope as the structure they are declared in. However, embedded structures cannot be anonymous. Structure members can also be arrays. Finally, each level of structure has its own name space for its member's names, as is familiar.

Bit-fields (the capability to declare an integer with a specified number of bits) are not supported.

Currently, structures are the only user-definable type. The keywords **union**, **enum**, and **class** are reserved for possible future use.

1.2.6 Arrays

Arrays of any type can be created. The declaration

```
vec4 points[10];
```

creates an array of ten **vec4**, indexed starting with zero. There are no pointers; the only way to declare an array is with square brackets. Declaring an array as a function parameter also requires square brackets and a size, as currently array arguments are passed as if the whole array is a single object, not as if the argument is a pointer.

Arrays, unless they are function parameters, do not have to be declared with a size. A declaration like

```
vec4 points[];
```

is allowed, as long as *either* of the following two cases is true:

1. Before the array is referenced, it is declared again with a size, with the same type as the first declaration. For example,

```
vec4 points[];    // points is an array of unknown size
vec4 points[10];  // points is now an array of size 10
```

This cannot be followed by another declaration:

```
vec4 points[];    // points is an array of unknown size
vec4 points[10];  // points is now an array of size 10
vec4 points[20];  // this is illegal
vec4 points[];    // this is also illegal
```

2. All indices that statically reference the array are compile-time constants. In this case, the compiler will make the array large enough to hold the largest index it sees used. For example,

```
vec4 points[];         // points is an array of unknown size
points[2] = vec4(1.0); // points is now an array of size 3
points[7] = vec4(2.0); // points is now an array of size 8
```

In this case, at runtime the array will only have one size, determined by the largest index the compiler sees. Such automatically sized arrays cannot be passed as function arguments.

This feature is quite useful for handling the built-in array of texture coordinates. Internally, this array is declared as

```
varying vec4 gl_TexCoord[];
```

If a program uses only compile-time constant indices of 0 and 1, the array will be implicitly sized as *gl_TexCoord[2]*. If a shader uses a nonconstant variable to index the array, that shader will have to explicitly declare the array with the desired size. Of course, keeping the size to a minimum is important, especially for varying variables, as they are likely a limited hardware resource.

Multiple shaders sharing the same array must declare it with the same size. The linker will verify this.

1.2.7 Void

The type **void** is provided for declaring a function that returns no value. For example, the function **main** returns no value and must be declared as type **void**.

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

Other than for functions that return nothing, the **void** type is not useful.

1.2.8 Declarations and Scope

Variables are declared quite similarly to C++. They can be declared where needed and have scope as in C++. For example,

```
float f;

f = 3.0;

vec4 u, v;

for (int i = 0; i < 10; ++i)
    v = f * u + v;
```

The scope of a variable declared in a **for** statement ends at the end of the loop's substatement. However, variables may not be declared in an **if** statement. This simplifies implementation of scoping across the **else** sub-statement, with little practical cost.

As in C, variable names are case sensitive, must start with a letter or underscore (_), and contain only letters, numbers, and underscores (_). User-defined variables cannot start with the string "gl_", as those names are reserved for future use by OpenGL. Names containing consecutive underscores (__) are also reserved.

1.2.9 Type Matching and Promotion

The OpenGL Shading Language is strict with type matching. In general, types being assigned must match, argument types passed into functions must match formal parameter declarations, and types being operated on must match the requirements of the operator. There are no automatic promotions from one type to another. This may occasionally make a shader have an extra explicit conversion. However, it also simplifies the language, preventing some forms of obfuscated code and some classes of defects. For example, there are no ambiguities in which overloaded function should be chosen for a given function call.

1.3 Initializers and Constructors

A shader variable may be initialized when it is declared. As is familiar from C, the following example initializes *b* at declaration time and leaves *a* and *c* undefined:

```
float a, b = 3.0, c;
```

Constant qualified variables have to be initialized.

```
const int Size = 4;  // initializer is required
```

Attribute, uniform, and varying variables *cannot* be initialized when declared.

```
attribute float Temperature;  // no initializer allowed,
                              // the vertex API sets this

uniform int Size;             // no initializer allowed,
                              // the uniform setting API sets this

varying float density;        // no initializer allowed, the vertex
                              // shader must programmatically set this
```

To initialize aggregate types, at either declaration time or elsewhere, CONSTRUCTORS are used. There is no initializer using the brace syntax "{…}" from C, only constructors. Syntactically, constructors look like function calls that have a type name where the function name would go—for example, to initialize a **vec4** with the values (1.0, 2.0, 3.0, 4.0) use

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
```

Or, because constructor syntax is the same whether it's in an initializer or not,

```
vec4 v;
v = vec4(1.0, 2.0, 3.0, 4.0);
```

There are constructors for all the built-in types (except samplers) as well as for structures. Some examples:

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
ivec2 c = ivec2(3, 4);
vec3 color = vec3(0.2, 0.5, 0.8);
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
struct light
{
    vec4 position;
    struct tLightColor
    {
        vec3 color;
        float intensity;
    } lightColor;
} light1 = light(v, tLightColor(color, 0.9));
```

For matrices, the components are filled in column-major order. The variable *m* from the previous example is the matrix

$$\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$$

So far, we've only shown constructors taking one argument for each component being constructed. Built-in constructors for vectors can also take a single argument, which is replicated into each component.

```
vec3 v = vec3(0.6);
```

is equivalent to

```
vec3 v = vec3(0.6, 0.6, 0.6);
```

This is true only for vectors. Structure constructors must receive one argument per member being constructed. Matrix constructors also have a single argument form, but in this case it initializes just the diagonal of the matrix. The remaining components are initialized to 0.0.

```
mat2 m = mat2(1.0);  // makes a 2x2 identity matrix
```

is equivalent to

```
mat2 m = mat2(1.0, 0.0, 0.0, 1.0);    // makes a 2x2 identity matrix
```

Constructors can also have vectors and matrices as arguments. However, constructing matrices from other matrices is reserved for future definition.

```
vec4 v = vec4(1.0);
vec2 u = vec2(v);  // the first two components of v initialize u
mat2 m = mat2(v);
vec2 t = vec2(1.0, 2.0, 3.0);  // allowed; the 3.0 is ignored
```

Matrix components will be read out of arguments in column major order and filled in column major order.

Extra components in a constructor argument are silently ignored. Normally, this is useful for shrinking a value, like eliminating *alpha* from a color or *w* from a position. It is an error to have completely unused arguments passed to a constructor.

1.4 Type Conversions

Explicit type conversions are performed with constructors. For example,

```
float f = 2.3;
bool b = bool(f);
```

will set b to **true**. This is useful for flow-control constructs, like **if**, which require Boolean values. Boolean constructors convert non-zero numeric values to **true** and zero numeric values to false.

The OpenGL Shading Language does not provide C-style typecast syntax, which can be ambiguous as to whether a value is converted to a different type or simply reinterpreted as a different type. In fact, there is no way of reinterpreting a value as a different type in the OpenGL Shading Language. There are no pointers, no type unions, no implicit type changes, and no reinterpret casts. Instead, constructors are used to perform conversions. The arguments to a constructor are converted to the type they are constructing. Hence, the following are allowed:

```
float f = float(3);  // convert integer 3 to floating-point 3.0
float g = float(b);  // convert Boolean b to floating point
vec4 v = vec4(2);    // all components of v are set to 2.0
```

When converting from a Boolean, **true** is converted to 1 or 1.0, and **false** is converted to a zero.

1.5 Qualifiers and Interface to a Shader

Qualifiers prefix both variables and formal function parameters. The qualifiers used to modify formal function parameters (**const, in, out,** and **inout**) are discussed in Section 1.6.2. This section will focus on the other qualifiers, most of which form the interfaces of the shaders to their outside world. The following is the complete list of qualifiers (used outside of formal function parameters).

attribute	For frequently changing information, from the application to a vertex shader
uniform	For infrequently changing information, from the application to either a vertex shader or a fragment shader
varying	For interpolated information passed from a vertex shader to a fragment shader
const	To declare nonwritable, compile time constant variables, as in C

Getting information into and out of a shader is quite different from more typical programming environments. Information is transferred to and from a shader by reading and writing built-in variables and user-defined **attribute**, **uniform,** and **varying** variables. The most common built-in variables were shown in the example at the beginning of this paper. They are *gl_Position* for output of the homogeneous coordinates of the vertex position and *gl_FragColor* for output of the fragment's color from a fragment shader. The complete set of built-in variables is provided in the OpenGL Shading Language specification document. Examples of **attribute**, **uniform,** and **varying** qualified variables have been seen briefly in the opening example for getting other information into and out of shaders. Each is discussed in this section.

Variables qualified as **attribute, uniform**, or **varying** must be declared at global scope. This is sensible as they are visible outside of shaders and, for a single program, they all share a single name space.

Qualifiers are always specified before the type of a variable, and because there is no default type, the form of a qualified variable declaration will always include a type.

```
attribute float Temperature;
const int NumLights = 3;
uniform vec4 LightPosition[NumLights];
varying float LightIntensity;
```

1.5.1 Attribute Qualifiers

Attribute qualified variables (or attributes) are provided for an application to pass frequently modified data into a vertex shader. They can be changed as often as once per vertex, either directly or indirectly by the application. There are built-in attributes, like *gl_Vertex* and *gl_Normal*, for reading traditional OpenGL state, and there are user-defined attributes, which the coder can name.

Attributes are limited to floating-point scalars, floating-point vectors, and matrices. Attributes declared as integers or booleans are not allowed, nor are attributes declared as structures or arrays. This is, in part, a result of encouraging high-performance frequent changing of attributes in hardware implementations of the OpenGL system. Attributes cannot be modified by a shader.

Attributes cannot be declared in fragment shaders.

1.5.2 Uniform Qualifiers

Uniform qualified variables (or uniforms), like attributes, are set only outside a shader and are intended for data that changes less frequently. They can be changed at most once per primitive. All data types and arrays of all data types are supported for uniform qualified variables. All the vertex and fragment shaders forming a single program share a single global name space for uniforms. Hence, uniforms of the same name in a vertex and fragment program will be the same uniform variable.

Uniforms cannot be written to in a shader. This is sensible because an array of processors may be sharing the same resources to hold uniforms and other language semantics break down if uniforms could be modified.

Recall that unless a sampler (e.g., **sampler2D**) is a function parameter, the **uniform** qualifier must be used when declaring it. This is because samplers are opaque, and making them uniforms allows the system to validate that the application initializes a sampler with a texture and texture unit consistent with its use in the shader.

1.5.3 Varying Qualifiers

Varying qualified variables (or varyings) are the only way a vertex shader can communicate results to a fragment shader. Such variables form the dynamic interface between vertex and fragment shaders. The intention is that for a particular attribute of a drawing primitive, each vertex might have a different value and that these values need to be interpolated across the fragments in the primitive. The vertex shader writes the per-vertex values into a varying variable, and when the fragment shader reads from this variable, it gets back a value interpolated between the vertices. If some attribute were to be the same across a large primitive, not requiring interpolation, the vertex shader need not communicate it to the fragment shader at all. Instead, the application could pass this value directly to the fragment shader through a uniform qualified variable.

The exception to using varying variables only for interpolated values is for any value the application will change often, either per triangle or per some small set of triangles or vertices. These values may be faster to pass as **attribute** variables and forwarded on as **varying** variables because changing **uniform** values frequently may impact performance.

The automatic interpolation of **varying** qualified variables is done in a perspective correct manner. This is necessary no matter what type of data is being interpolated. Otherwise, such values would not change smoothly across edges introduced for surface subdivision. The non-perspective correct interpolated result would be continuous, but its derivative would not be, and this can be quite visible.

A **varying** qualified variable is written in a vertex shader and read in a fragment shader. It is illegal for a fragment shader to write to a varying variable. However, the vertex shader may read a varying variable, getting back what it has just written. Reading a varying qualified variable before writing it returns an undefined value.

1.5.4 Constant Qualifiers

Variables qualified as **const** (except for formal function parameters) are compile-time constants and are not visible outside the shader that declares them. There is support for both scalar and nonscalar constants. Structure fields may not be qualified with **const**, but structure variables can be declared as **const** and initialized with a structure constructor. Initializers for **const** declarations must be formed from literal values, other **const** qualified variables (not including function call parameters), or expressions of these.

Some examples:

```
const int numIterations = 10;
const float pi = 3.14159;
const vec2 v = vec2(1.0, 2.0);
const vec3 u = vec3(v, pi);
const struct light
{
    vec3 position;
    vec3 color;
} fixedLight = light(vec3(1.0, 0.5, 0.5), vec3(0.8, 0.8, 0.5));
```

All the preceding variables are compile-time constants. The compiler may propagate and fold constants at compile time, using the precision of the processor executing the compiler, and need not allocate any runtime resources to **const** qualified variables.

1.5.5 Absent Qualifier

If no qualifier is specified when declaring a variable (not a function parameter), the variable can be both read and written by the shader. Nonqualified variables declared at global scope can be shared between shaders of the same type that are linked in the same program. Vertex shaders and fragment shaders each have their own separate global name space for nonqualified globals. However, nonqualified user-defined variables are not visible outside of a program. That privilege is reserved for variables qualified as **attribute** or **uniform** and for built-in variables representing OpenGL state.

Unqualified variables have a lifetime limited to a single run of a shader. There is also no concept corresponding to a **static** variable in a C function that would allow a variable to be set to a value and have its shader retain that value from one execution to the next. This is made difficult by the parallel processing nature of the execution environment, where multiple instantiations run in parallel, sharing much of the same memory. In general, writable variables must have unique instances per processor executing a shader and therefore cannot be shared.

Because unqualified global variables have a different name space for vertex shaders than for fragment shaders, it is not possible to share information through such variables between vertex and fragment shaders. Read-only variables can

be shared if declared as **uniform**, and variables written by a vertex shader can be read by the fragment shader only through the **varying** mechanism.

1.6 Flow Control

Flow control is very much like C++. The entry point into a shader is the function **main**. A program containing both vertex and fragment shaders will have two functions named **main**, one for entering a vertex shader to process each vertex, and one to enter a fragment shader to process each fragment. Before **main** is entered, any initializers for global variable declarations will be executed.

Looping can be done with **for**, **while**, and **do-while**, just as in C++. Variables can be declared in **for** and **while** statements, and their scope lasts until the end of their substatements. The keywords **break** and **continue** also exist and behave as in C.

Selection can be done with **if** and **if-else**, just as in C++, with the exception that a variable cannot be declared in the **if** statement. Selection using the selection operator (**?:**) is also available, with the extra constraint that the second and third operands must have exactly the same type.

The type of the expression provided to an **if** statement or a **while** statement, or to terminate a **for** statement, must be a scalar Boolean. As in C, the right-hand operand to logical-and (**&&**) is not evaluated (or at least appears not to be evaluated) if the left-hand operand evaluates to false, and the right-hand operand to logical-or (||) is not evaluated if the left-hand operand evaluates to true. Similarly, only one of the second or third operands in the selection operator (**:?**) will be evaluated. A logical exclusive or (^^) is also provided, for which both sides are always evaluated.

A special branch, **discard**, can prevent a fragment from updating the frame buffer. When a fragment shader executes the **discard** keyword, the fragment being processed is marked to be discarded. An implementation might or might not continue executing the shader, but it is guaranteed that there will be no effect on the frame buffer.

A **goto** keyword or equivalent is not available, nor are labels. Switching with **switch** is also not provided.

1.6.1 Functions

Function calls operate much as in C++. Function names can be overloaded by parameter type but not solely by return type. Either a function definition (body) or declaration must be in scope before calling a function. Parameter types are always checked. This means an empty parameter list **()** in a function declaration is not ambiguous, as in C, but rather explicitly means the function accepts no arguments. Also, parameters have to have exact matches as no automatic promotions are done, so selection of overloaded functions is quite straightforward.

Exiting from a function with **return** operates the same as in C++. Functions returning nonvoid types must return values, whose type must exactly match the return type of the function.

Functions may not be called recursively, either directly or indirectly.

1.6.2 Calling Conventions

The OpenGL Shading Language uses call by value-return as its calling convention. The call by value part is familiar from C: Parameters qualified as input parameters will be copied into the function and not passed as a reference. Because there are no pointers, a function need not worry about its parameters being aliases of some other memory. The return part of call by value-return means parameters qualified as output parameters will be returned to the caller by being copied back out from the called function to the caller when the function returns.

To indicate which parameters are copied when, prefix them with the qualifier keywords **in, out**, or **inout**. For something that is just copied into the function but not returned, use **in**. The **in** qualifier is also implied when no qualifiers are specified. To say a parameter is not to be copied in but is to be set and copied back on return, use the qualifier **out**. To say a parameter is copied both in and out, use the qualifier **inout**.

in	Copy in but don't copy back out; still writable within the function
out	Only copy out; readable, but undefined at entry to function
inout	Copy in and copy out

The **const** qualifier can also be applied to function parameters. Here, it does not mean the variable is a compile-time constant, but rather that the function is not allowed to write it. Note that an ordinary, nonqualified input-only parameter can be written to, it just won't be copied back to the caller. Hence, there is a distinction between a parameter qualified as **const in** and one qualified only as **in** (or with no qualifier). Of course, **out** and **inout** parameters cannot be declared as **const**.

Some examples:

```
void ComputeCoord(in vec3 normal,  // Parameter 'normal' is copied in,
                                   // can be written to, but will not be
                                   // copied back out.
                  vec3 tangent,    // Same behavior as if "in" was used.
                  inout vec3 coord)// Copied in and copied back out.
```

Or,

```
vec3 ComputeCoord(const vec3 normal,// normal cannot be written to
                  vec3 tangent,
                  in vec3 coord)    //the function will return the result
```

The following are not legal:

```
void ComputeCoord(const out vec3 normal, //not legal; can't write normal
                  const inout vec3 tang, //not legal; can't write tang
                  in out vec3 coord)     //not legal; use inout
```

Structures and arrays may also be passed as arguments to a function. Keep in mind though that these data types are passed by value and there are no pointers, so it is possible to cause some large copies to occur at function call time. Array parameters must be declared with their size, and only arrays of matching type and size can be passed to an array parameter. The return type of a function is not allowed to be an array.

Functions can either return a value or return nothing. If a function returns nothing, it must be declared as type **void**. If a function returns a value, the type can be any type except an array. However, structures can be returned, and structures can contain arrays.

1.6.3 Built-in Functions

There is a large set of built-in functions available. These are documented in full in the OpenGL Shading Language specification.

A shader can override these functions, providing its own definition. To override a function, provide a prototype or definition that is in scope at the time of call time. The compiler or linker then looks for a user-defined version of the function to resolve that call. For example, one of the built-in sine functions is declared as

```
float sin(float x);
```

If a shader wants to experiment with performance/accuracy trade-offs in a sine function or specialize it for a particular domain, it can override them and do so.

```
float sin(float x)
{
    return <.. some function of x..>
```

```
}

void main()
{
    // call the sin function above, not the built-in sin function
    float s = sin(x);
}
```

This is similar to the standard language linking techniques of using libraries of functions and to having more locally scoped function definitions satisfy references before the library is checked. If the definition is in a different shader, just make sure a prototype is visible before calling the function. Otherwise, the built-in version will be used.

1.7 Operations

Table 3.1 includes the operators, in order of precedence, available in the OpenGL Shading Language. The precedence and associativity is consistent with C.

Table 1.1 Operators, in order of precedence

Operator	Description
[ ]	Index
.	Member selection and swizzle
++ --	Postfix increment/decrement
++ --	Prefix increment/decrement
- !	Unary negation and logical not
* /	Multiply and divide
+ -	Add and subtract
< > <= >=	Relational
== !=	Equality
&&	Logical and
^^	Logical exclusive or
||	Logical inclusive or
?:	Selection
= += -= *= /=	Assignment
,	Sequence

1.7.1 Indexing

Vectors, matrices, and arrays can be indexed using the index operator (**[ ]**). All indexing is zero based; the first element is at index 0. Indexing an array operates just as in C.

Indexing a vector returns scalar components. This allows giving components numerical names of 0, 1, …, and also provides variable selection of vector components, should that be needed. For example,

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
float f = v[2];  // f takes the value 3.0
```

Here, *v[2]* will be the floating-point scalar 3.0, which is then assigned into *f*.

Indexing a matrix returns columns of the matrix as vectors. For example,

```
mat4 m = mat4(3.0);  // initializes the diagonal to all 3.0
vec4 v;
v = m[1];   // places the vector (0.0, 3.0, 0.0, 0.0) into v
```

Here, the second column of *m*, *m[1]* is treated as a vector that is copied into *v*.

Behavior is undefined if an array, vector, or matrix is accessed with an index that's less than zero or greater than or equal to the size of the object.

1.7.2 Swizzling

The normal structure-member selector (**.**) is also used to SWIZZLE components of a vector—that is, components can be selected and/or rearranged by listing their names after the swizzle operator (**.**). Examples:

```
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 component names can be out of order to rearrange the components or they can be replicated to duplicate the components:

```
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)
```

At most, four component names can be listed in a swizzle; otherwise, they would result in a nonexistent type. The rules for swizzling are slightly different for R-VALUES (expressions that are read from) and L-VALUES (expressions that say where to write to). R-values can have any combination and repetition of components. L-values must not have any repetition. For example:

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

For R-values, this syntax can be used on any expression whose resultant type is a vector. For example, getting a two-component vector from a texture lookup can be done as

```
vec2 v = texture1D(sampler, coord).xy;
```

where the built-in function **texture1D** returns a **vec4**.

1.7.3 Component-wise Operation

With a few important exceptions, when an operator is applied to a vector, it behaves as if it were applied to each component of the vector, independently.

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;
```

If a binary operation operates on a vector and a scalar, the scalar is applied to each component of the vector. If two vectors are operated on, their sizes must match.

Exceptions are multiplication of a vector times a matrix and a matrix times a matrix, which perform standard linear-algebraic multiplies, not component-wise multiplies.

Increment and decrement operators (**++** and **--**) and unary negation (**-**) behave as in C. When applied to a vector or matrix, they increment or decrement each component. They operate on integer and floating-point-based types.

Arithmetic operators of addition (**+**), subtraction (**-**), multiplication (*****), and division (/) behave as in C, or component-wise, with the previously described exception of using linear-algebraic multiplication on vectors and matrices:

```
vec4 v, u;
mat4 m;
v * u;  // This is a component-wise multiply
v * m;  // This is a linear-algebraic row-vector times matrix multiply
m * v;  // This is a linear-algebraic matrix times column-vector multiply
m * m;  // This is a linear-algebraic matrix times matrix multiply
```

All other operations are performed component by component.

Logical not (**!**), logical and (**&&**), logical or (||), and logical inclusive or (^^) operate only on expressions that are typed as scalar Booleans, and they result in a Boolean. These cannot operate on vectors. There is a built-in function, **not**, to compute the component-wise logical not of a vector of Booleans.

Relational operations (**<, >, <=,** and >=) operate only on floating-point and integer scalars and result in a scalar Boolean. There are built-in functions, for instance **lessThanEqual,** to compute a Boolean vector result of component-wise comparisons of two vectors.

The equality operators (**==** and **!=**) operate on all types except arrays. They compare every component or structure member across the operands. This results in a scalar Boolean, indicating whether the two operands were equal. For two operands to be equal, their types must match, and each of their components or members must be equal. To get a component-wise vector result, call the built-in functions **equal** and **notEqual**.

Scalar Booleans are produced by the operators equal (==), not equal (**!=**), relational (**<, >, <=,** and >=), and logical not (**!**) because flow-control constructs (**if**, **for,** etc.) require a scalar Boolean. If built-in functions like **equal** are called to compute a vector of Booleans, such a vector can be turned into a scalar Boolean using the built-in functions **any** or **all**. For example, to do something if any component of a vector is less than the corresponding component of another vector, say

```
vec4 u, v;
```

```
...
if (any(lessThan(u, v)))
    ...
```

Assignment (=) requires exact type match between the left- and right-hand side. Any type, except for arrays, can be assigned. Other assignment operators (**+=, -=, *=,** and /=) are similar to C but must make semantic sense when expanded out

`a *= b` ⇒ `a = a * b`

where the expression *a* * *b* must be semantically valid, and the type of the expression *a* * *b* must be the same as the type of *a*. The other assignment operators behave similarly.

The ternary selection operator (**?:**) 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 will appear to be evaluated. The second and third expressions must be the same type, but they can be of any type other than an array. The resulting type is the same as the type of the second and third expressions.

The sequence operator (**,**) 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.

1.8 Preprocessor

The preprocessor is much like that in C. Support for

```
#define
#undef
#if
#ifdef
#ifndef
#else
#elif
#endif
```

as well as the `defined` operator are exactly as in standard C. This includes macros with arguments and macro expansion. Built-in macros are

```
__LINE__
__FILE__
__VERSION__
```

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

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

`__VERSION__` substitutes 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 110.

Macro names containing two consecutive underscores (__) are reserved for future use as predefined macro names, as are all macro names prefixed with "`GL_`"

There is also the usual support for

```
#error message
#line
#pragma
```

`#error` puts *message* into the shader's information log. The compiler then proceeds as if a semantic error has been encountered.

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

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

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

#pragma is implementation dependent. Tokens following #**pragma** are not subject to preprocessor macro expansion. If an implementation does not recognize the tokens specified by the pragma, the pragma will be ignored. However, 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 the OpenGL Shading Language. No implementation may use a pragma whose first token is `STDGL`.

Use the optimize pragma

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

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

The debug pragma

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

enables compiling and annotating a shader with debug information so it can be used with a debugger. The debug pragma can occur only outside function definitions. By default, debug is set to off.

Shaders should declare the version of the language to which they are written using

```
#version number
```

If the targeted version of the language is the version that was approved in conjunction with OpenGL 2.0, then a value of 110 should be used for *number*. Any value less than 110 will cause an error to be generated. Any value greater than the latest version of the language supported by the compiler will also cause an error to be generated. Version 110 of the language does not require shaders to include this directive, and shaders without this directive are assumed to target version 110 of the OpenGL Shading Language. This directive, when present, must occur in a shader before anything else except comments and white space.

By default, compilers must issue compile time syntactic, grammatical, and semantic errors for shaders that do not conform to the OpenGL Shading Language specification. Any extended behavior must first be enabled through a preprocessor directive. The behavior of the compiler with respect to extensions is declared using the `#extension` directive:

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

`extension_name` is the name of an extension. The token `all` means that the specified behavior should apply to all extensions supported by the compiler. The possible values for behavior and their corresponding effects are shown in Table 1.2.

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

The initial state of the compiler is as if the directive

```
#extension all : behavior
```

was issued, telling the compiler that all error and warning reporting must be done according to the non-extended version of the OpenGL Shading Language that is being targeted (i.e., all extensions will be ignored).

Table 1.2 Permitted values for the *behavior* expression in the `#extension` directive

Value of *behavior*	Effect
`require`	Behave as specified by the extension `extension_name`. The compiler will report an error on the `#extension` directive if the specified extension is not supported, or if the token `all` is used instead of an extension name.
`enable`	Behave as specified by the extension `extension_name`. The compiler will provide a warning on the `#extension` directive if the specified extension is not supported, and it will report an error if the token `all` is used instead of an extension name.
`warn`	Behave as specified by the extension `extension_name`, except to cause the compiler to issue warnings on any detectable use of the specified extension that is not supported by other enabled or required extensions. If `all` is specified, then the compiler will warn on all detectable uses of any extension used.
`disable`	Behave (including errors and warnings) as if the extension specified by `extension_name` is not part of the language definition. If `all` is specified, then behavior must revert back to that of the non-extended version of the language that is being targeted. The compiler will provide a warning if the specified extension is not supported.

Each extension to the language can define its allowed granularity of scope. If nothing is specified, the granularity is a shader (i.e., 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.

The number sign (#) on a line by itself is ignored. Any directive not described in this section will cause the compiler to generate an error message. The shader will subsequently be treated as ill-formed.

1.9 Preprocessor Expressions

Preprocessor expressions can contain the following operators as shown in Table 1.3.

Table 1.3 Preprocessor operators

Operator	Description
+ - ~ ! defined	unary
*** / %**	multiplicative
+ -	additive
<< >>	bit-wise shift
< > <= >=	relational
== !=	equality
& ^ |	bit-wise
&& ||	logical

They have precedence, associativity, and behavior matching the standard C preprocessor.

Preprocessor expressions may be executed on the processor running the compiler and not on the graphics processor that will execute the shader. Precision is allowed to be this host processor's precision and hence will likely be different from the precision available when executing expressions in the core language.

As with the core language, string types and operations are not provided. None of the hash-based operators (#, ##, etc.) is provided, nor is a preprocessor **sizeof** operator.

1.10 Error Handling

Compilers accept some ill-formed programs because it is impossible to detect all ill-formed programs. For example, completely accurate detection of usage of an uninitialized variable is not possible. Such ill-formed shaders may execute differently on different platforms. Therefore, the OpenGL Shading Language specification states that portability is ensured only for well-formed programs.

OpenGL Shading Language compilers are encouraged to detect ill-formed programs and issue diagnostic messages, but are not required to do so for all cases. Compilers are required to return messages regarding shaders that are lexically, grammatically, or semantically incorrect. Shaders that generate such error messages cannot be executed. The OpenGL entry points for obtaining any diagnostic messages are discussed in the OpenGL 2.0 specification.

1.11 Summary

The OpenGL Shading Language is a high-level procedural language designed specifically for the OpenGL environment. This language allows applications to specify the behavior of programmable, highly parallel graphics hardware. It contains constructs that allow succinct expression of graphics shading algorithms in a way that is natural for programmers experienced in C and C++.

The OpenGL Shading Language includes support for scalar, vector, and matrix types; structures and arrays; sampler types that are used to access textures; data type qualifiers that are used to define shader input and output; constructors for initialization and type conversion; and operators and flow control statements like those in C/C++.