Chapter 6

The OpenGL ES Shading Language (GLSL ES)

This chapter takes a break from examining WebGL sample programs and explains the essential features of the OpenGL ES Shading Language (GLSL ES) in detail.

As you have seen, shaders are the core mechanism within WebGL for constructing 3DCG programs, and GLSL ES is the dedicated programming language for writing those shader programs. This chapter covers:

- Data, variables, and variable types
- Vector, matrix, structure, array, and sampler types
- Operators, control flow, and functions
- Attributes, uniform, and varying variables
- Precision qualifiers
- Preprocessor and directives

By the end of this chapter, you will have a good understanding of GLSL ES and how to use it to write a variety of shaders. This knowledge will help you tackle the more complex 3D manipulations introduced in Chapters 7 through 9. Note that language specifications can be quite dry, and for some of you, this may be more detail than you need. If so, it's safe to skip this chapter and use it as a reference when you look at the examples in the rest of the book.

Recap of Basic Shader Programs

As you can see from Listings 6.1 and 6.2, you can construct shader programs in a similar manner to constructing programs using the C programming language.

Listing 6.1 Example of a Simple Vertex Shader

```
// Vertex shader
attribute vec4 a_Position;
attribute vec4 a_Color;
uniform mat4 u_MvpMatrix;
varying vec4 v_Color;
void main() {
  gl_Position = u_MvpMatrix * a_Position;
  v_Color = a_Color;
}
```

Variables are declared at the beginning of the code, and then the `main()` routine defines the entry point for the program.

Listing 6.2 Example of a Simple Fragment Shader

```
// Fragment shader
#ifdef GLSL_ES
precision mediump float;
#endif
varying vec4 v_Color;
void main() {
  gl_FragColor = v_Color;
}
```

The version of GLSL ES dealt with in this chapter is 1.00. However, you should note that WebGL does not support all features defined in GLSL ES 1.00[1]; rather, it supports a subset of 1.00 with core features needed for WebGL.

Overview of GLSL ES

The **GLSL ES** programming language was developed from the OpenGL Shading Language (GLSL) by reducing or simplifying functionality, assuming that the target platforms were consumer electronics or embedded devices such as smart phones and game consoles. A prime goal was to allow hardware manufacturers to simplify the hardware needed to execute GLSL ES programs. This had two key benefits: reducing power consumption by devices and, perhaps more importantly, reducing manufacturing costs.

GLSL ES supports a limited (and partially extended) version of the C language syntax. Therefore, if you are familiar with the C language, you'll find it easy to understand GLSL ES. Additionally, the shading language is beginning to be used for general-purpose processing such as image processing and numerical computation (so called GPGPU), meaning that GLSL ES has an increasingly wide application domain, thus increasing the benefits of studying the language.

1 http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

Hello Shader!

By tradition, most programming books begin with a "Hello World!" example, or in our case the corresponding shader program. However, because you have already seen several shader programs in previous chapters, let's skip that and take a look at the basics of GLSL ES, using Listing 6.1 and Listing 6.2 shown earlier.

Basics

Like many programming languages, you need to pay attention to the following two items when you write shader programs using GLSL ES:

- The programs are case-sensitive (`marina` and `Marina` are different).
- A semicolon (;) must be specified at the end of each command.

Order of Execution

Once a JavaScript program is loaded, program lines are executed in the order in which they were written—sequentially starting from the first program line. However, like C, shader programs are executed from the function `main()` and therefore must have one (and only one) `main()` function that cannot have any parameters. Looking back, you can see that each shader program shown in Listing 6.1 and Listing 6.2 defines a single `main()`.

You must prepend the keyword `void` to `main()`, which indicates that the function has no return value. (See the section "Functions" later in this chapter.) This is different from JavaScript, where you can define a function using the keyword `function`, and you don't have to worry whether the function returns a value. In GLSL ES, if the function returns a value, you must specify its data type in front of the function name, or if it doesn't return a value, specify `void` so that the system doesn't expect a return value.

Comments

As with JavaScript, you can write comments in your shader program, and in fact use the same syntax as JavaScript. So, the following two types of comment are supported:

- // characters followed by any sequence of characters up to the end of line:

```
int kp = 496; // kp is a Kaprekar number
```

- /* characters, followed by any sequence of characters (including new lines), followed by the */ characters:

```
/* I have a day off today.
   I want to take a day off tomorrow.
 */
```

Data (Numerical and Boolean Values)

GLSL ES supports only two data types:

- **Numerical value:** GLSL ES supports integer numbers (for example, 0, 1, 2) and floating point numbers (for example, 3.14, 29.98, 0.23571). Numbers without a decimal point (.) are treated as integer numbers, and those with a decimal point are treated as floating point numbers.
- **Boolean value:** GLSL ES supports `true` and `false` as boolean constants.

GLSL ES does not support character strings, which may initially seem strange but makes sense for a 3D graphics language.

Variables

As you have seen in the previous chapters, you can use any variable names you want as long as the name follows the basic naming rules:

- The character set for variables names contains only the letters a–z, A–Z, the underscore (_), and the numbers 0–9.
- Numbers are not allowed to be used as the first character of variable names.
- The keywords shown in Table 6.1 and the reserved keywords shown in Table 6.2 are not allowed to be used as variable names. However, you can use them as part of the variable name, so the variable name `if` will result in error, but `iffy` will not.
- Variable names starting with `gl_`, `webgl_`, or `_webgl_` are reserved for use by OpenGL ES. No user-defined variable names may begin with them.

Table 6.1 Keywords Used in GLSL ES

`attribute`	`bool`	`break`	`bvec2`	`bvec3`	`bvec4`
`const`	`continue`	`discard`	`do`	`else`	`false`
`float`	`for`	`highp`	`if`	`in`	`inout`
`Int`	`invariant`	`ivec2`	`ivec3`	`ivec4`	`lowp`
`mat2`	`mat3`	`mat4`	`medium`	`out`	`precision`
`return`	`sampler2D`	`samplerCube`	`struct`	`true`	`uniform`
`varying`	`vec2`	`vec3`	`vec4`	`void`	`while`

Table 6.2 Reserved Keywords for Future Version of GLSL ES

`asm`	`cast`	`class`	`default`
`double`	`dvec2`	`dvec3`	`dvec4`
`enum`	`extern`	`external`	`fixed`
`flat`	`fvec2`	`fvec3`	`fvec4`
`goto`	`half`	`hvec2`	`hvec3`
`hvec4`	`inline`	`input`	`interface`
`long`	`namespace`	`noinline`	`output`
`packed`	`public`	`sampler1D`	`sampler1DShadow`
`sampler2DRect`	`sampler2DRectShadow`	`sampler2DShadow`	`sampler3D`
`sampler3DRect`	`short`	`sizeof`	`static`
`superp`	`switch`	`template`	`this`
`typedef`	`union`	`unsigned`	`using`
`volatile`			

GLSL ES Is a Type Sensitive Language

GLSL ES does not require the use of `var` to declare variables, but it does require you to specify the type of data a variable will contain. As you have seen in the sample programs, you declare variables using the form

```
<data type> <variable name>
```

such as `vec4 a_Position`.

As discussed, when you define a function like `main()`, you must also specify the data type of the return value of the function. Equally, the type of data on the left side of the assignment operation (=) and that of data on the right side must have the same type; otherwise, it will result in an error.

For these reasons, GLSL ES is called a **type sensitive language**, meaning that it belongs to a class of languages that require you to specify and pay attention to types.

Basic Types

GLSL ES supports the basic data types shown in Table 6.3.

Table 6.3 GLSL Basic Types

Type	Description
`float`	The data type for a single floating point number. It indicates the variable will contain a single floating point number.
`int`	The data type for a single integer number. It indicates the variable will contain a single integer number.
`bool`	The data type for a boolean value. It indicates the variable will contain a boolean value.

Specifying the data type for variables allows the WebGL system to check errors in advance and process the program efficiently. The following are examples of variable declarations using basic types.

```
float klimt;  // The variable will contain a single floating number
int utrillo;  // The variable will contain a single integer number
bool doga;    // The variable will contain a single boolean value
```

Assignment and Type Conversion

Assignments of values to variables are performed using the assignment operator (`=`). As mentioned, because GLSL ES is a type-sensitive language, if the data type of the left-side variable is not equal to that of the assigned data (or variable), it will result in an error:

```
int i = 8;       // OK
float f1 = 8;    // Error
float f2 = 8.0;  // OK
float f3 = 8.0f; // Error: Expressions like 8.0f used in C are not allowed.
```

Semantically, 8 and 8.0 are the same values. However, when you assign 8 to a floating point variable `f1`, it will result in an error. In this case, you would see the following error message:

```
failed to compile shader: ERROR: 0:11: '=' : cannot convert from 'const mediump int'
to 'float'.
```

If you want to assign an integer number to a floating point variable, you need to convert the integer number to a floating point number. This conversion is called **type conversion**. To convert an integer into a floating point number, you can use the built-in function `float()`, as follows:

```
int i = 8;
float f1 = float(i); // 8 is converted to 8.0 and assigned to f1
float f2 = float(8); // equivalent to the above operation
```

GLSL ES supports a number of other built-in functions for type conversion, which are shown in Table 6.4.

Table 6.4 The Built-In Functions for Type Conversion

Conversion	**Function**	**Description**
To an integer number	`int(float)`	The fractional part of the floating-point value is dropped (for example, 3.14 → 3).
	`int(bool)`	`true` is converted to 1, or `false` is converted to 0.
To a floating point number	`float(int)`	The integer number is converted to a floating point number (for example, 8 → 8.0).
	`float(bool)`	`true` is converted to 1.0, or `false` is converted to 0.0.
To a boolean value	`bool(int)`	0 is converted to `false`, or non-zero values are converted to `true`.
	`bool(float)`	0.0 is converted to `false`, or non-zero values are converted to `true`.

Operations

The operators applicable to the basic types are similar to those in JavaScript and are shown in Table 6.5.

Table 6.5 The Operators Available for the Basic Types

Operator	**Operation**	**Applicable Data Type**
`-`	Negation (for example, for specifying a negative number)	`int` or `float`.
`*`	Multiplication	`int` or `float`. The data type of the result of the operation is the same as operands.
`/`	Division	
`+`	Addition	
`-`	Subtraction	
`++`	Increment (postfix and prefix)	`int` or `float`. The data type of the result of the operation is the same as operands.
`--`	Decrement (postfix and prefix)	
`=`	Assignment	`int`, `float`, or `bool`
`+= -= *= /=`	Arithmetic assignment	`int` or `float`.

Operator	Operation	Applicable Data Type
`< > <= >=`	Comparison	`int` or `float`.
`== !=`	Comparison (equality)	`int`, `float`, or `bool`.
`!`	Not	`bool` or an expression that results in `bool` [1].
`&&`	Logical and	
`||`	Logical inclusive or	
`^^`	Logical exclusive or [2]	
`condition? expression1:expression2`	Ternary selection	*condition* is `bool` or an expression that results in `bool`. Data types other than array can be used in *expression1* and *expression2*.

[1] The second operand in a logical and (`&&`) operation is evaluated if and only if the first operand evaluates to `true`. The second operand in a logical or (`||`) operation is evaluated if and only if the first operand evaluates to `false`.

[2] If either the left-side condition or the right-side one is `true`, the result is `true`. If both sides are `true`, the result is `false`.

The followings are examples of basic operations:

```
int i1 = 954, i2 = 459;
int kp = i1 - i2; // 495 is assigned to kp.
float f = float(kp) + 5.5; // 500.5 is assigned to f.
```

Vector Types and Matrix Types

GLSL ES supports vector and matrix data types which, as you have seen, are useful when dealing with computer graphics. Both these types contain multiple data elements. A vector type, which arranges data in a list, is useful for representing vertex coordinates or color data. A matrix arranges data in an array and is useful for representing transformation matrices. Figure 6.1 shows an example of both types.

$$\begin{pmatrix} 3 & 7 & 1 \end{pmatrix} \qquad \begin{bmatrix} 3 & 7 & 1 \\ 1 & 5 & 3 \\ 4 & 0 & 7 \end{bmatrix}$$

Figure 6.1 A vector and a matrix

GLSL ES supports a variety of vector or matrix types, as shown in Table 6.6.

Table 6.6 Vector Types and Matrix Types

Category	Types in GLSL ES	Description
`Vector`	`vec2`, `vec3`, `vec4`	The data types for 2, 3, and 4 component vectors of floating point numbers
	`ivec2`, `ivec3`, `ivec4`	The data types for 2, 3, and 4 component vectors of integer numbers
	`bvec2`, `bvec3`, `bvec4`	The data types for 2, 3, and 4 component vectors of boolean values
`Matrix`	`mat2`, `mat3`, `mat4`	The data type for 2×2, 3×3, and 4×4 matrix of floating point numbers (with 4, 9, and 16 elements, respectively)

The following examples show the use of the vector and matrix types:

```
vec3 position;  // variable for 3-component vector of float
                // For example: (10.0, 20.0, 30.0)
ivec2 offset;   // variable for 2-component vector of integer
                // For example: (10, 20)
mat4 mvpMatrix; // the variable for 4×4 matrix of float
```

Assignments and Constructors

Assignment of data to variables of the type vector or matrix is performed using the `=` operator. Remember that the type of data on the left side of the assignment operation and that of the data/variable on the right side must be the same. In addition, the number of elements on the left side of the assignment operation must be equal to that of the data/variable on the right side. To illustrate that, the following example will result in an error:

```
vec4 position = 1.0; // vec4 variable requires four floating point numbers
```

In this case, because a `vec4` variable requires four floating point numbers, you need to pass four floating numbers in some way. A solution is to use the built-in functions with the same name of the data type so; for example, in the case of `vec4`, you can use the constructor `vec4()`. (See Chapter 2, "Your First Step with WebGL.") For example, to assign 1.0, 2.0, 3.0, and 4.0 to a variable of type `vec4`, you can use `vec4()` to bundle them into a single data element as follows:

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

Functions for making a value of the specified data type are called **constructor functions**, and the name of the constructor is always identical to that of the data type.

Vector Constructors

Vectors are critical in GLSL ES so, as you'd imagine, there are multiple ways to specify arguments to a vector constructor. For example:

```
vec3 v3 = vec3(1.0, 0.0, 0.5);  // sets v3 to(1.0, 0.0, 0.5)
vec2 v2 = vec2(v3);  // sets v2 to (1.0, 0.0) using the 1st and 2nd elements of v3
vec4 v4 = vec4(1.0); // sets v4 to (1.0, 1.0, 1.0, 1.0)
```

In the second example, the constructor ignores the third element of `v3`, and only the first and second elements of `v3` are used to create the new vector. Similarly, in the third example, if a single value is specified to a vector constructor, the value is used to initialize all components of the constructed vector. However, if more than one value is specified to a vector constructor but the number of the values is less than the number of elements required by the constructor, it will result in an error.

Finally, a vector can be constructed from multiple vectors:

```
vec4 v4b = vec4(v2, v4);   // sets (1.0, 0.0, 1.0, 1.0) to v4b
```

The rule here is that the vector is filled with values from the first vector (v2), and then any missing values are supplied by the second vector (v4).

Matrix Constructors

Constructors are also available for matrices and operate in a similar manner to vector constructors. However, you should make sure the order of elements stored in a matrix is in a column major order. (See Figure 3.27 for more details of "column-major order.") The following examples show different ways of using the matrix constructor:

- If multiple values are specified to a matrix constructor, a matrix is constructed using them in column major order:

```
mat4 m4 = mat4 (   1.0,  2.0,   3.0,   4.0,
                   5.0,  6.0,   7.0,   8.0,
                   9.0, 10.0,  11.0,  12.0,
                  13.0, 14.0,  15.0,  16.0 );
```

$$\begin{bmatrix} 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \\ 4.0 & 8.0 & 12.0 & 16.0 \end{bmatrix}$$

- If multiple vectors are specified to a matrix constructor, a matrix is constructed using the elements of each vector in column major order:

```
// two vec2 are used to construct a mat2
vec2 v2_1 = vec2(1.0, 3.0);
vec2 v2_2 = vec2(2.0, 4.0);
mat2 m2_1 = mat2(v2_1, v2_2); // 1.0 2.0
                              // 3.0 4.0
// vec4 is used to construct mat2
vec4 v4 = vec4(1.0, 3.0, 2.0, 4.0);
```

```
mat2 m2_2 = mat2(v4);  // 1.0 2.0
                       // 3.4 4.0
```

- If multiple values and multiple vectors are specified to a matrix constructor, a matrix is constructed using them in column major order:

```
// Two floating point numbers and vec2 are used to construct a mat2
mat2 m2 = mat2(1.0, 3.0, v2_2); // 1.0 2.0
                                // 3.0 4.0
```

- If a single value is specified to a matrix constructor, a matrix is constructed using the value as its diagonal elements:

```
mat4 m4 = mat4(1.0);  // 1.0 0.0 0.0 0.0
                      // 0.0 1.0 0.0 0.0
                      // 0.0 0.0 1.0 0.0
                      // 0.0 0.0 0.0 1.0
```

Similar to a vector constructor, if an insufficient number of values is specified to the constructor (but more than one), it will result in an error.

```
mat4 m4 = mat4(1.0, 2.0, 3.0); // Error. mat4 requires 16 elements.
```

Access to Components

To access the components in a vector or matrix, you can use the operators `.` and `[]`, as shown in the following subsections.

The . Operator

An individual component in a vector can be accessed by the variable name followed by period (.) and then the component name, as shown in Table 6.7.

Table 6.7 Component Names

Category	Description
`x, y, z, w`	Useful for accessing vertex coordinates.
`r, g, b, a`	Useful for accessing colors.
`s, t, p, q`	Useful for accessing texture coordinates. (Note that this book uses only `s` and `t`. `p` is used instead of `r` because `r` is used for colors.)

Because vectors are used for storing various types of data such as vertex coordinates, colors, and texture coordinates, three types of component names are supported to increase the readability of programs. However, any of the component names `x`, `r`, or `s` accesses the first component; any of `y`, `g`, or `t` accesses the second one; and so on, so you can use them interchangeably if you prefer. For example:

```
vec3 v3 = vec3(1.0, 2.0, 3.0);  // sets v3 to(1.0, 2.0, 3.0)
float f;

f = v3.x; // sets f to 1.0
f = v3.y; // sets f to 2.0
f = v3.z; // sets f to 3.0

f = v3.r; // sets f to 1.0
f = v3.s; // sets f to 1.0
```

As you can see from the comments of these examples, `x`, `r`, and `s` have different names but always access the first component. Attempting to access a component beyond the number of components in the vector will result in an error:

```
f = v3.w; // w requires access to the fourth element, which doesn't exist.
```

Multiple components can be selected by appending their names (from the same name set) after the period (.). This is known as **swizzling**. In the following example, `x`, `y`, `z`, and `w` will be used, but other sets of component names have the same effect:

```
vec2 v2;
v2 = v3.xy; // sets v2 to (1.0, 2.0)
v2 = v3.yz; // sets v2 to (2.0, 3.0). Any component can be omitted
v2 = v3.xz; // sets v2 to (1.0, 3.0). You can skip any component.
v2 = v3.yx; // sets v2 to (2.0, 1.0). You can reverse the order.
v2 = v3.xx; // sets v2 to (1.0, 1.0). You can repeat any component.

vec3 v3a;
v3a = v3.zyx; // sets v3a to (3.0, 2.0, 1.0). You can use all names.
```

The component name can also be used in the left-side expression of an assignment operator (`=`):

```
vec4 position = vec4(1.0, 2.0, 3.0, 4.0);
position.xw = vec2(5.0, 6.0); // position = (5.0, 2.0, 3.0, 6.0)
```

Remember, the component names must come from the same set so, for example, `v3.was` is not allowed.

The [ ] Operator

In addition to the . operator, the components of a vector or a matrix can be accessed using the array indexing operator `[]`. Note that the elements in a matrix are also read out in column major order. Just like JavaScript, the index starts from 0, so applying `[0]` to a matrix selects the first column in the matrix, `[1]` selects the second one, `[2]` selects the third one, and so on. The following shows an example:

```
mat4 m4 = mat4 ( 1.0,  2.0,  3.0,  4.0,
                 5.0,  6.0,  7.0,  8.0,
                 9.0, 10.0, 11.0, 12.0,
                13.0, 14.0, 15.0, 16.0);
vec4 v4 = m4[0]; // Retrieve the 1st column from m4: (1.0, 2.0, 3.0, 4.0)
```

In addition, two `[]` operators can be used to select a column and then a row of a matrix:

```
float m23 = m4[1][2]; // sets m23 to the third component of the second
                      // column of m4 (7.0).
```

A component name can be used to select a component in conjunction with the `[]` operator, as follows:

```
float m32 = m4[2].y; // sets m32 to the second component of the third
                     // column of m4 (10.0).
```

One restriction is that only a **constant index** can be specified as the index number in the `[]` operator. The constant index is defined as

- A integral literal value (for example, 0 or 1)
- A global or local variable qualified as `const`, excluding function parameters (see the section "const Variables")
- Loop indices (see the section "Conditional Control Flow and Iteration")
- Expressions composed from any of the preceding

The following examples use the type `int` constant index:

```
const int index = 0;  // "const" keyword specifies the variable is a
                      // read-only variable.
vec4 v4a = m4[index]; // is the same as m4[0]
```

The following example uses an expression composed of constant indices as an index.

```
vec4 v4b = m4[index + 1]; // is the same as m4[1]
```

Remember, you cannot use an `int` variable without the `const` qualifier as an index because it is not a constant index (unless it is a loop index):

```
int index2 = 0;
vec4 v4c = m4[index2]; // Error: because index2 is not a constant index.
```

Operations

You can apply the operators shown in Table 6.8 to a vector or a matrix. These operators are similar to the operators for basic types. Note that the only comparative operators available for a vector and matrix are `==` and `!=`. The `<`, `>`, `<=`, and `>=` operators cannot be used for comparisons of vectors or matrices. In such cases, you can use built-in functions such as `lessThan()`. (See Appendix B, "Built-In Functions of GLSL ES 1.0.")

Table 6.8 The Operators Available for a Vector and a Matrix

Operators	Operation	Applicable Data Types
`*`	Multiplication	`vec[234]` and `mat[234]`. The operations on `vec[234]` and `mat[234]` will be explained below. The data type of the result of operation is the same as the operands.
`/`	Division	
`+`	Addition	
`-`	Subtraction	
`++`	Increment (postfix and prefix)	`vec[234]` and `mat[234]`. The data type of the result of this operation is the same as the operands.
`--`	Decrement (postfix and prefix)	
`=`	Assignment	`vec[234]` and `mat[234]`.
`+=, -=, *=, /=`	Arithmetic assignment	`vec[234]` and `mat[234]`.
`==, !=`	Comparison	`vec[234]` and `mat[234]`. With `==`, if all components of the operands are equal, the result is `true`. For `!=`, if any of components of the operands are not equal, then the result is `true` [1].

[1] If you want component-wise equality, you can use the built-in function `equal()` or `notEqual()`. (See Appendix B.)

Note that when an arithmetic operator operates on a vector or a matrix, it is operating independently on each component of the vector or matrix in component-wise order.

Examples

The following examples show frequently used cases. In the examples, we assume that the types of variables are defined as follows:

```
vec3 v3a, v3b, v3c;
mat3 m3a, m3b, m3c;
float f;
```

Operations on a Vector and Floating Point Number

An example showing the use of the + operator:

```
// The following example uses the + operator, but the -, *, and /
// operators also have the same effect.
v3b = v3a + f;   // v3b.x = v3a.x + f;
                 // v3b.y = v3a.y + f;
                 // v3b.z = v3a.z + f;
```

For example, `v3a = vec3(1.0, 2.0, 3.0)` and `f = 1.0` will result in `v3b=(2.0, 3.0, 4.0)`.

Operations on Vectors

These operators operate on each component of a vector:

```
// The following example uses the + operator, but the -, *, and /
// operators also have the same effect.
v3c = v3a + v3b;  // v3a.x + v3b.x;
                  // v3a.y + v3b.y;
                  // v3a.z + v3b.z;
```

For example, `v3a = vec3(1.0, 2.0, 3.0)` and `v3b = vec3(4.0, 5.0, 6.0)` will result in `v3c=(5.0, 7.0, 9.0)`.

Operations on a Matrix and a Floating Point Number

These operators operate on each component of the matrix:

```
// The following example uses the + operator, but the -, *, and /
// operators also have the same effect.
m3b = m3a * f;   // m3b[0].x = m3a[0].x * f; m3b[0].y = m3a[0].y * f;
                 // m3b[0].z = m3a[0].z * f;
                 // m3b[1].x = m3a[1].x * f; m3b[1].y = m3a[1].y * f;
                 // m3b[1].z = m3a[1].z * f;
                 // m3b[2].x = m3a[2].x * f; m3b[2].y = m3a[2].y * f;
                 // m3b[2].z = m3a[2].z * f;
```

Multiplication of a Matrix and a Vector

For multiplication, the result is the sum of products of each element in a matrix and vector. This result is the same as Equation 3.5 that you saw back in Chapter 3, “Drawing and Transforming Triangles”:

```
v3b = m3a * v3a;  // v3b.x = m3a[0].x * v3a.x + m3a[1].x * v3a.y
                  //                           + m3a[2].x * v3a.z;
                  // v3b.y = m3a[0].y * v3a.x + m3a[1].y * v3a.y
                  //                           + m3a[2].y * v3a.z;
                  // v3b.z = m3a[0].z * v3a.x + m3a[1].z * v3a.y
                  //                           + m3a[2].z * v3a.z;
```

Multiplication of a Vector and a Matrix

Multiplication of a vector and a matrix is possible, as you can see from the following expressions. Note that this result is different from that when multiplying a matrix by a vector:

```
v3b = v3a * m3a; // v3b.x = v3a.x * m3a[0].x + v3a.y * m3a[0].y
                 //                           + v3a.z * m3a[0].z;
                 // v3b.y = v3a.x * m3a[1].x + v3a.y * m3a[1].y
                 //                           + v3a.z * m3a[1].z;
                 // v3b.z = v3a.x * m3a[2].x + v3a.y * m3a[2].y
                 //                           + v3a.z * m3a[2].z;
```

Multiplication of Matrices

This is the same as Equation 4.4 in Chapter 4, "More Transformations and Basic Animation":

```
m3c = m3a * m3b; // m3c[0].x = m3a[0].x * m3b[0].x + m3a[1].x * m3b[0].y
                 //                                 + m3a[2].x * m3b[0].z;
                 // m3c[1].x = m3a[0].x * m3b[1].x + m3a[1].x * m3b[1].y
                 //                                 + m3a[2].x * m3b[1].z;
                 // m3c[2].x = m3a[0].x * m3b[2].x + m3a[1].x * m3b[2].y
                 //                                 + m3a[2].x * m3b[2].z;

                 // m3c[0].y = m3a[0].y * m3b[0].x + m3a[1].y * m3b[0].y
                 //                                 + m3a[2].y * m3b[0].z;
                 // m3c[1].y = m3a[0].y * m3b[1].x + m3a[1].y * m3b[1].y
                 //                                 + m3a[2].y * m3b[1].z;
                 // m3c[2].y = m3a[0].y * m3b[2].x + m3a[1].y * m3b[2].y
                 //                                 + m3a[2].y * m3b[2].z;

                 // m3c[0].z = m3a[0].z * m3b[0].x + m3a[1].z * m3b[0].y
                 //                                 + m3a[2].z * m3b[0].z;
                 // m3c[1].z = m3a[0].z * m3b[1].x + m3a[1].z * m3b[1].y
                 //                                 + m3a[2].z * m3b[1].z;
                 // m3c[2].z = m3a[0].z * m3b[2].x + m3a[1].z * m3b[2].y
                 //                                 + m3a[2].z * m3b[2].z;
```

Structures

GLSL ES also supports user-defined types, called **structure**s, which aggregate other already defined types using the keyword `struct`. For example:

```
struct light {   // defines the structure "light"
  vec4 color;
  vec3 position;
}
light l1, l2;    // declares variable "l1" and "l2" of the type "light"
```

This example defines the new structure type `light` that consists of two members: the variable `color` and `position`. Then two variables `l1` and `l2` of type `light` are declared after the definition. Unlike C, the `typedef` keyword is not necessary because, by default, the name of the structure becomes the name of the type.

In addition, as a convenience, variables of the new type can be declared with the definition of the structure, as follows:

```
struct light {   // declares structure and its variable all together
  vec4 color;    // color of a light
  vec3 position; // position of a light
} l1;            // variable "l1" of the structure
```

Assignments and Constructors

Structures support the standard constructor, which has the same name as the structure. The arguments to the constructor must be in the same order and of the same type as they were declared in the structure. Figure 6.2 shows an example.

$$\text{l1 = light(}\underbrace{\text{vec4(0.0, 1.0, 0.0, 1.0)}}_{\text{color}}\text{,}\underbrace{\text{vec3(8.0, 3.0, 0.0)}}_{\text{position}}\text{);}$$

Figure 6.2 A constructor of structure

Access to Members

Each member of a structure can be accessed by appending the variable name with a period (.) and then the member name. For example:

```
vec4 color = l1.color;
vec3 position = l1.position;
```

Operations

For each member in the structure, you can use any operators allowed for that member's type. However, the operators allowed for the structure itself are only the assignment (`=`) and comparative operators (`==` and `!=`); see Table 6.9.

Table 6.9 The Operators Available for a Structure

Operator	Operation	Description
`=`	Assignment	The assignment and comparison operators are not allowed for the structures that contain arrays or sampler types
`==, !=`	Comparison	

When using the `==` operator, the result is `true` if, and only if, all the members are component-wise equal. When using the !=, the result is `false` if one of the members is not component-wise equal.

Arrays

GLSL ES arrays have a similar form to the array in JavaScript, with only one-dimensional arrays being supported. Unlike arrays in JavaScript, the `new` operator is not necessary to create arrays, and methods such as `push()` and `pop()` are not supported. The arrays can be declared by a name followed by brackets (`[]`) enclosing their sizes. For example:

```
float floatArray[4]; // declares an array consisting of four floats
vec4 vec4Array[2];   // declares an array consisting of two vec4s
```

The array size must be specified as an **integral constant expression** greater than zero where the integral constant expression is defined as follows:

- A numerical value (for example, 0 or 1)
- A global or local variable qualified as `const`, excluding function parameters (see the section "const Variables")
- Expressions composed of both of the above

Therefore, the following will result in an error:

```
int size = 4;
vec4 vec4Array[size]; // Error. If you declare "const int size = 4;"
                      // it will not result in an error
```

Note that arrays cannot be qualified as `const`.

Array elements can be accessed using the array indexing operator ([]). Note that, like C, the index starts from 0. For example, the third element of the float Array defined earlier can be accessed as follows:

```
float f = floatArray[2];
```

Only an integral constant expression or uniform variable (see the section "Uniform Variables") can be used as an index of an array. In addition, unlike JavaScript or C, an array cannot be initialized at declaration time. So each element of the array must be initialized explicitly as follows:

```
vec4Array[0] = vec4(4.0, 3.0, 6.0, 1.0);
vec4Array[1] = vec4(3.0, 2.0, 0.0, 1.0);
```

Arrays support only `[]` operators. However, elements in an array do support the standard operators available for their type. For example, the following operator can be applied to the elements of `floatArray` or `vec4Array`:

```
// multiplies the second element of floatArray by 3.14
float f = floatArray[1] * 3.14;
// multiplies the first element of vec4Array by vec4(1.0, 2.0, 3.0, 4.0);
vec4 v4 = vec4Array[0] * vec4(1.0, 2.0, 3.0, 4.0);
```

Samplers

GLSL ES supports a dedicated type called sampler for accessing textures. (See Chapter 5, "Using Colors and Texture Images.") Two types of samplers are available: `sampler2D` and `samplerCube`. Variables of the `sampler` type can be used only as a uniform variable (see the section "Uniform Variables") or an argument of the functions that can access textures such as `texture2D()`. (See Appendix B.) For example:

```
uniform sampler2D u_Sampler;
```

In addition, the only value that can be assigned to the variable is a texture unit number, and you must use the WebGL method `gl.uniform1i()` to set the value. For example, `TexturedQuad.js` in Chapter 5 uses `gl.uniform1i(u_Sampler, 0)` to pass the texture unit 0 to the shader.

Variables of type `sampler` are not allowed to be operands in any expressions other than `=`, `==`, and `!=`.

Unlike other types explained in the previous sections, the number of `sampler` type variables is limited depending on the shader type (see Table 6.10). In the table, the keyword `mediump` is a precision qualifier. (This qualifier is explained in detail in the section "Precision Qualifiers," toward the end of this chapter.)

Table 6.10 Minimum Number of Variables of the Sampler Type

Shaders that Use the Variable	**Built-In Constants Representing the Maximum Number**	**Minimum Number**
Vertex shader	`const mediump int gl_MaxVertexTextureImageUnits`	0
Fragment shader	`const mediump int gl_MaxTextureImageUnits`	8

Precedence of Operators

Operator precedence is shown in Table 6.11. Note the table contains several operators that are not explained in this book but are included for reference.

Table 6.11 The Precedence of Operators

Precedence	**Operators**
1	parenthetical grouping (())
2	function calls, constructors (()), array indexing ([]), period (.)
3	increment/decrement (++, –), negate (-), **inverse(~)**, not(!)
4	multiplication (*), division (/), **remainder (%)**
5	addition (+), subtraction (-)
6	**bit-wise shift (<<, >>)**
7	comparative operators (<, <=, >=, >)
8	equality (==, !=)
9	**bit-wise and (&)**
10	**bit-wise exclusive or (^)**
11	**bit-wise or (|)**
12	and (&&)
13	exclusive or (^^)
14	or (||)
15	ternary selection (? :)
16	assignment (=), arithmetic assignments (+=, -=, *=, /=, **%=, <<=, >>=, &=, ^=, |=**)
17	sequence(,)

Bold font indicates operators reserved for future versions of GLSL.

Conditional Control Flow and Iteration

Conditional control flow and iteration in the shading language are almost the same as in JavaScript or C.

if Statement and if-else Statement

A conditional control flow can use either `if` or `if-else`. An `if-else` statement follows the pattern shown here:

```
if (conditional-expression1) {
  commands here are executed if conditional-expression1 is true.
} else if (conditional-expression2) {
  commands here are executed if conditional-expression1 is false but conditional-
  expression2 is true.
} else {
  commands here are executed if conditional-expression1 is false and conditional-
  expression2 is false.
}
```

The following shows a code example using the `if-else` statement:

```
if(distance < 0.5) {
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); // red
} else {
  gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0); // green
}
```

As shown in this example, the conditional expression in the `if` or `if-else` statement must be either a boolean value or an expression that becomes a boolean value. Boolean vector types, such as `bvec2`, are not allowed in the conditional expression.

`Switch` statement are not allowed, and you should note that usage of the `if` or `if-else` statement will slow down the shaders.

for Statement

The `for` statement can be used as follows:

```
for (for-init-statement; conditional-expression; loop-index-expression) {
  the commands which you want to execute repeatedly.
}
```

For example:

```
for (int i = 0; i < 3; i++) {
  sum += i;
}
```

Note that the loop index (`i` in the preceding example) of the `for` statement can be declared only in the *for-init-statement*. The *conditional-expression* can be omitted, and an empty condition becomes true. The `for` statement has the following restrictions:

- Only a single loop index is allowed. The loop index must have the type `int` or `float`.
- *loop-index-expression* must have one of the following forms (supposing that `i` is a loop index):

  `i++`, `i--`, `i+=`*constant-expression*, `i-=`*constant-expression*
- *conditional-expression* is a comparison between a loop index and an integral constant expression. (See the section "Array.")
- Within the body of the loop, the loop index cannot be assigned.

These limitations are in place so that the compiler can perform inline expansion of `for` statements.

continue, break, discard Statements

Just like JavaScript or C, `continue` and `break` statements are allowed only within a `for` statement and are generally used within `if` statements:

`continue` skips the remainder of the body of the innermost loop containing the continue, increases/decreases the loop index, and then moves to the next loop.

`break` exits the innermost loop containing the break. No further execution of the loop is done.

The following show examples of the `continue` statement:

```
for (int i = 0; i < 10; i++) {
  if (i == 8) {
     continue; // skips the remainder of the body of the innermost loop
  }
  // When i == 8, this line is not executed
}
```

The following shows an example of the `break` statement:

```
for (int i = 0; i < 10; i++) {
  if (i == 8) {
    break; // exits "for" loop
  }
  // When i >= 8, this line is not executed.
}
// When i == 8, this line is executed.
```

The `discard` statement is only allowed in fragment shaders and discards the current fragment, abandoning the operation on the current fragment and skipping to the next fragment. The use of `discard` will be explained in more detail in the section "Make a Rounded Point" in Chapter 10, "Advanced Techniques."

Functions

In contrast to the way functions are defined in JavaScript, the functions in GLSL ES are defined in the same manner as in C. For example:

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

Argument `type`s must use one of the data types explained in this chapter, and like `main()`, functions with no arguments are allowed. When the function returns no value, the `return` statement does not need to be included. In this case, `returnType` must be `void`. You can also specify a structure as the `returnValue`, but the structure returned cannot contain an array.

The following example shows a function to convert an RGBA value into a luminance value:

```
float luma (vec4 color) {
  float r = color.r;
  float g = color.g;
  float b = color.b;
  return 0.2126 * r + 0.7162 * g + 0.0722 * b;
  // The preceding four lines could be rewritten as follows:
  // return 0.2126 * color.r + 0.7162 * color.g + 0.0722 * color.b;
}
```

You can call the function declared above in the same manner as in JavaScript or C by using its name followed by a list of arguments in parentheses:

```
attribute vec4 a_Color; // (r, g, b, a) is passed
void main() {
  ...
  float brightness = luma(a_Color);
  ...
}
```

Note that an error will result if, when called, argument types do not match the declared parameter types. For example, the following will result in an error because the type of the parameter is float, but the caller passes an integer:

```
float square(float value) {
  return value * value;
}
void main() {
  ...
  float x2 = square(10); // Error: Because 10 is integer. 10.0 is OK.
  ...
}
```

As you can see from the previous examples, functions work just like those in JavaScript or C except that you cannot call the function itself from inside the body of the function (that is, a recursive call of the function isn't allowed). For the more technically minded, this is because the compilers can in-line function calls.

Prototype Declarations

When a function is called before it is defined, it must be declared with a prototype. The prototype declaration tells WebGL in advance about the types of parameters and the return value of the function. Note that this is different from JavaScript, which doesn't require a prototype. The following is an example of a prototype declaration for `luma()`, which you saw in the previous section:

```
float luma(vec4); // a prototype declaration
main() {
...
float brightness = luma(color); // luma() is called before it is defined.
...
}

float luma (vec4 color) {
  return 0.2126 * color.r + 0.7162 * color.g + 0.0722 * color.b;
}
```

Parameter Qualifiers

GLSL ES supports qualifiers for parameters that control the roles of parameters within a function. They can define that a parameter (1) is to be passed into a function, (2) is to be passed back out of a function, and (3) is to be passed both into and out of a function. (2) and (3) can be used just like a pointer in C. These are shown in Table 6.12.

Table 6.12 Parameter Qualifiers

Qualifiers	Roles	Description
`in`	Passes a value into the function	The parameter is passed by value. Its value can be referred to and modified in the function. The caller cannot refer to the modification.
`const in`	Passes a value into the function	The parameter is passed by constant value. Its value can be referred to but cannot be modified.
`out`	Passes a value out of the function	The parameter is passed by reference. If its value is modified, the caller can refer to the modification.
`inout`	Passes a value both into/out of the function	The parameter is passed by reference, and its value is copied in the function. Its value can be referred to and modified in the function. The caller can also refer to the modification.
`<none: default>`	Passes a value into the function	Same as `in`.

For example, `luma()` can return the result of its calculation using a parameter qualified by `out` instead of a return value, as follows:

```
void luma2 (in vec3 color, out float brightness) {
  brightness = 0.2126 * color.r + 0.7162 * color.g + 0.0722 * color.b;
}
```

Because the function itself no longer returns a value, the return type of this function is changed from `float` to `void`. Additionally, the qualifier `in`, in front of the first parameter, can be omitted because `in` is a default parameter qualifier.

This function can be used as follows:

```
luma2(color, brightness); // the result is stored into "brightness"
                          // same as brightness = luma(color)
```

Built-In Functions

In addition to user-defined functions, GLSL ES supports a number of built-in functions that perform operations frequently used in computer graphics. Table 6.13 gives an overview of the built-in functions in GLSL ES, and you can look at Appendix B for the detailed definition of each function.

Table 6.13 Built-In Functions in GLSL ES

Category	**Built-In Functions**
Angle functions	`radians` (converts degrees to radians), `degrees` (converts radians to degrees)
Trigonometry functions	`sin` (sine function), `cos` (cosine function), `tan` (tangent function), `asin` (arc sine function), `acos` (arc cosine function), and `atan` (arc tangent function)
Exponential functions	`pow` (x^y), `exp` (natural exponentiation), `log` (natural logarithm), `exp2` (2^x), `log2` (base 2 logarithm), `sqrt` (square root), and `inversesqrt` (inverse of `sqrt`)
Common functions	`abs` (absolute value), `min` (minimum value), `max` (maximum value), `mod` (remainder), `sign` (sign of a value), `floor` (floor function), `ceil` (ceil function), `clamp` (clamping of a value), `mix` (linear interpolation), `step` (step function), `smoothstep` (Hermite interpolation), and `fract` (fractional part of the argument)
Geometric functions	`length` (length of a vector), `distance` (distance between two points), `dot` (inner product), `cross` (outer product), `normalize` (vector with length of 1), `reflect` (reflection vector), and `faceforward` (converting normal when needed to "faceforward")
Matrix functions	`matrixCompMult` (component-wise multiplication)
Vector relational functions	`lessThan` (component-wise "<"), `lessThanEqual` (component-wise "<="), `greaterThan` (component-wise ">"), `greaterThanEqual` (component-wise ">="), `equal` (component-wise "=="), `notEqual` (component-wise "!="), `any` (true if any component is true), `all` (true if all components are true), and `not` (component-wise logical complement)
Texture lookup functions	`texture2D` (texture lookup in the 2D texture), `textureCube` (texture lookup in the cube map texture), `texture2DProj` (projective version of `texture2D()`), `texture2DLod` (level of detail version of `texture2D()`), `textureCubeLod` (lod version of `textureCube()`), and `texture2DProjLod` (projective version of `texture2DLod()`)

Global Variables and Local Variables

Just like JavaScript or C, GLSL ES supports both global variables and local variables. Global variables can be accessed from anywhere in the program, and local variables can be accessed only from within a limited portion of the program.

In GLSL ES, in a similar manner to JavaScript or C, variables declared "outside" a function become global variables, and variables declared "inside" a function become local variables. The local variables can be accessed only from within the function containing them. For this reason, the attribute, uniform, and varying variables described in the next section must be declared as global variables because they are accessed from outside the function.

Storage Qualifiers

As explained in the previous chapters, GLSL ES supports storage qualifiers for attribute, uniform, and varying variables (see Figure 6.3). In addition, a `const` qualifier is supported to specify a constant variable to be used in a shader program.

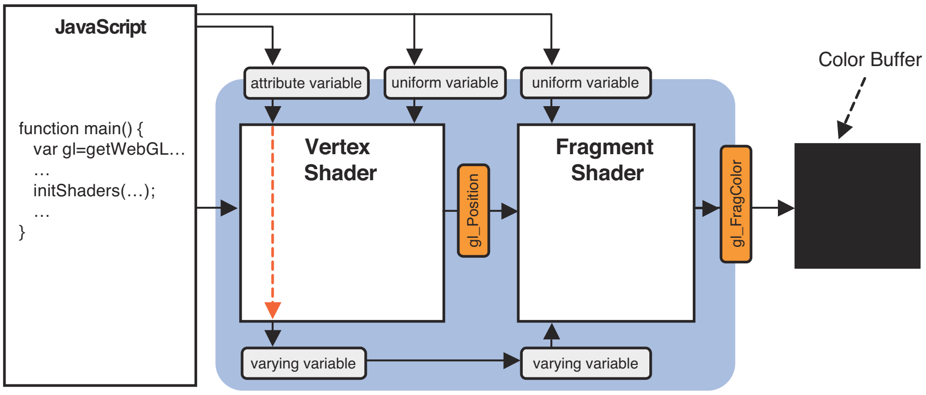

Figure 6.3 Attribute, uniform, and varying variables

const Variables

Unlike JavaScript, GLSL ES supports the `const` qualifier to specify a constant variable, or one whose value cannot be modified.

The `const` qualifier is specified in front of the variable type, just like an attribute variable. Variables qualified by `const` must be initialized at their declaration time; otherwise, they are unusable because no data can be assigned to them after their declaration. Some examples include:

```
const int lightspeed = 299792458;          // light speed (m/s)
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0); // red
const mat4 identity = mat4(1.0);           // identity matrix
```

Assigning data to the variable qualified by `const` will result in an error. For example:

```
const int lightspeed;
lightspeed = 299792458;
```

will result in the following error message:

```
failed to compile shader: ERROR: 0:11: 'lightspeed' : variables
with qualifier 'const' must be initialized
ERROR: 0:12: 'assign': l-value required (can't modify a const variable)
```

Attribute Variables

As you have seen in previous chapters, attribute variables are available only in vertex shaders. They must be declared as a global variable and are used to pass per-vertex data to the vertex shader. You should note that it is "per-vertex." For example, if there are two vertices, (4.0, 3.0, 6.0) and (8.0, 3.0, 0.0), data for each vertex can be passed to an attribute variable. However, data for other coordinates, such as (6.0, 3.0, 3.0), which is a halfway point between the two vertices and not a specified vertex, cannot be passed to the variable. If you want to do that, you need to add the coordinates as a new vertex. Attribute variables can only be used with the data types `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, and `mat4`. For example:

```
attribute vec4 a_Color;
attribute float a_PointSize;
```

There is an implementation-dependent limit on the number of attribute variables available, but the minimum number is 8. The limits on the number of each type of variable are shown in Table 6.14.

Table 6.14 The Limitation on the Number of Attribute, Uniform, and Varying Variables

Types of Variables		**The Built-In Constants for the Maximum Number**	**Minimum Number**
attribute variables		`const mediump int gl_MaxVertexAttribs`	8
uniform variables	Vertex shader	`const mediump int gl_MaxVertexUniformVectors`	128
	Fragment shader	`const mediump int gl_MaxFragmentUniformVectors`	16
varying variables		`const mediump int gl_MaxVaryingVectors`	8

Uniform Variables

Uniform variables are allowed to be used in both vertex and fragment shaders and must be declared as global variables. Uniform variables are read-only and can be declared as any data types other than array and structure. If a uniform variable of the same name and data type is declared in both a vertex shader and a fragment shader, it is shared between them. Uniform variables contain "uniform" (common) data, so your JavaScript program must only use them to pass such data. For example, because transformation matrices contain the uniform values for all vertices, they can be passed to uniform variables:

```
uniform mat4 u_ViewMatrix;
uniform vec3 u_LightPosition;
```

There is an implementation-dependent limit on the number of uniform variables that can be used (Table 6.14). Note that the limit in a vertex shader is different from that in a fragment shader.

Varying Variables

The last type of qualifier is `varying`. Varying variables also must be declared as global variables and are used to pass data from a vertex shader to a fragment shader by declaring a variable with the same type and name in both shaders. (See `v_Color` in Listing 6.1 and Listing 6.2.) The following are examples of varying variable declarations:

```
varying vec2 v_TexCoord;
varying vec4 v_Color;
```

Just like attribute variables, the varying variables can be declared only with the following data types: `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, and `mat4`. As explained in Chapter 5, the value of a varying variable written by a vertex shader is not passed to a fragment shader as is. Rather, the rasterization process between the vertex and fragment shaders interpolates the value according to the shape to be drawn, and then the interpolated value is passed per fragment. This interpolation process is the reason for the limitations on the data types that can be used with a varying variable.

The number of varying variables also has an implementation dependent limit. The minimum number is 8 (see Table 6.14).

Precision Qualifiers

Precision qualifiers were newly introduced in GLSL ES to make it possible to execute shader programs more efficiently and to reduce their memory size. As the name suggests, it is a simple mechanism to specify how much precision (the number of bits) each data type should have. Simply put, specifying higher precision data requires more memory and computation time, and specifying lower precision requires less. By using these qualifiers, you can exercise fine-grained control over aspects of performance and size. However, precision qualifiers are optional, and a reasonable default compromise can be specified using the following lines:

```
#ifdef GL_ES
precision mediump float;
#endif
```

Because WebGL is based on OpenGL ES 2.0, which was designed for consumer electronics and embedded systems, WebGL programs may end up executing on a range of hardware platforms. In some cases, the computation time and memory efficiency could be improved by using lower precision data types when performing calculations and operations. Perhaps more importantly, this also enables reduced power consumption and thus extended battery life on mobile devices.

You should note, however, that just specifying lower precision may lead to incorrect results within WebGL, so it's important to balance efficiency and correctness.

As shown in Table 6.15, WebGL supports three types of precision qualifiers: `highp` (high precision), `mediump` (medium precision), and `lowp` (lower precision).

Table 6.15 Precision Qualifiers

Precision Qualifiers	Descriptions	Default Range and Precision	
		`Float`	`int`
`highp`	High precision. The minimum precision required for a vertex shader.	$(-2^{62}, 2^{62})$ Precision: 2^{-16}	$(-2^{16}, 2^{16})$
`mediump`	Medium precision. The minimum precision required for a fragment shader. More than `lowp`, and less than `highp`.	$(-2^{14}, 2^{14})$ Precision: 2^{-10}	$(-2^{10}, 2^{10})$
`lowp`	Low precision. Less than `mediump`, but all colors can be represented.	$(-2, 2)$ Precision: 2^{-8}	$(-2^{8}, 2^{8})$

There are a couple of things to note. First, fragment shaders may not support `highp` in some WebGL implementations; a way to check this is shown later in this section. Second, the actual range and precision are implementation dependent, which you can check by using `gl.getShaderPrecisionFormat()`.

The following are examples of the declaration of variables using the precision qualifiers:

```
mediump float size;  // float of medium precision
highp vec4 position; // vec4 composed of floats of high precision
lowp vec4 color;     // vec4 composed of floats of lower precision
```

Because specifying a precision for all variables is time consuming, a default for each data type can be set using the keyword `precision`, which must be specified at the top of a vertex shader or fragment shader using the following syntax:

```
precision precision-qualifier name-of-type;
```

This sets the precision of the data type specified by *name-of-type* to the precision specified by *precision-qualifier*. In this case, variables declared without a precision qualifier have this default precision automatically set. For example:

```
precision mediump float; // All floats have medium precision
precision highp int;     // All ints have high precision
```

This specifies all data types related to float, such as `vec2` and `mat3`, to have medium precision, and all integers to have high precision. For example, because `vec4` consists of four float types, each float of the vector is set to medium precision.

You may have noticed that in the examples in previous chapters, you didn't specify precision qualifiers to the data types other than `float` in fragment shaders. This is because most data types have a default precision value; however, there is no default precision for `float` in a fragment shader. See Table 6.16 for details.

Table 6.16 Default Precision of Type

Type of Shader	Data Type	Default Precision
Vertex shader	`int`	`highp`
	`float`	`highp`
	`sampler2D`	`lowp`
	`samplerCube`	`lowp`
Fragment shader	`int`	`medium`
	`float`	**None**
	`sampler2D`	`lowp`
	`samplerCube`	`lowp`

The fact that there is no default precision for float requires programmers to carefully use floats in their fragment shaders. So, for example, using a float without specifying the precision will result in the following error:

```
failed to compile shader: ERROR: 0:1 : No precision specified for (float).
```

As mentioned, whether a WebGL implementation supports `highp` in a fragment shader is implementation dependent. If it is supported, the built-in macro `GL_FRAGMENT_PRECISION_HIGH` is defined (see the next section).

Preprocessor Directives

GLSL ES supports preprocessor directives, which are commands (directives) for the preprocessor stage before actual compilation. They are always preceded by a hash mark (#). The following example was used in `ColoredPoints.js`:

```
#ifdef GL_ES
precision mediump float;
#endif
```

These lines check to see if the macro `GL_ES` is defined, and if so the lines between `#ifdef` and `#endif` are executed. They are similar to an `if` statement in JavaScript or C.

The following three preprocessor directives are available in GLSL ES:

```
#if constant-expression
If the constant-expression is true, this part is executed.
#endif

#ifdef macro
If the macro is defined, this part is executed.
#endif

#ifndef macro
If the macro is not defined, this part is executed.
#endif
```

The `#define` is used to define macros. Unlike C, macros in GLSL ES cannot have macro parameters:

```
#define macro-name string
```

You can use `#undef` to undefine the macro:

```
#undef macro-name
```

You can use `#else` directives just like an `if` statement in JavaScript or C. For example:

```
#define NUM 100
#if NUM == 100
If NUM == 100 then this part is executed.
#else
If NUM != 100 then this part is executed.
#endif
```

Macros can use any name except for the predefined macros names shown in Table 6.17.

Table 6.17 Predefined Macros

Macro	Description
`GL_ES`	Defined and set to 1 in OpenGL ES 2.0
`GL_FRAGMENT_PRECISION_HIGH`	`highp` is supported in a fragment shader

So you can use the macro with preprocessor directives as follows:

```
#ifdef GL_ES
#ifdef GL_FRAGMENT_PRECISION_HIGH
precision highp float; // highp is supported. floats have high precision
#else
precision mediump float; // highp is not supported. floats have medium precision
#endif
#endif
```

You can specify which version of GLSL ES is used in the shader by using the `#version` directive:

```
#version number
```

Accepted versions include 100 (for GLSL ES 1.00) and 101 (for GLSL ES 1.01). By default, shaders that do not include a `#version` directive will be treated as written in GLSL ES version 1.00. The following example specifies version 1.01:

```
#version 101
```

The `#version` directive must be specified at the top of the shader program and can only be preceded by comments and white space.

Summary

This chapter explained the core features of the OpenGL ES Shading Language (GLSL ES) in some detail.

You have seen that the GLSL ES shading language has many similarities to C but has been specialized for computer graphics and has had unnecessary C features removed. The specialized computer graphics features include support for vector and matrix data types, special component names for accessing the components of a vector or matrix, and operators for a vector or matrix. In addition, GLSL ES supports many built-in functions for operations frequently used in computer graphics, all designed to allow you to create efficient shader programs.

Now that you have a better understanding of GLSL ES, the next chapter will return to WebGL and explore more sophisticated examples using this new knowledge.

This page intentionally left blank