To release the resources related to shader objects and programs, you can delete those objects by calling the respective deletion function. If either the shader program or object is currently active when it's deleted, the object is only marked for deletion. It is deleted when the program is no longer in use, or the shader object is detached from all shader program objects.

void **glDeleteShader**(GLuint *shader*);

Deletes *shader*. If *shader* is currently linked to one or more active shader programs, the object is tagged for deletion and deleted once the shader program is no longer being used by any shader program.

void **glDeleteProgram**(GLuint *program*);

Deletes *program* immediately if not currently in use in any context, or tagged for deletion when the program is no longer in use by any contexts.

To determine if an identifier is currently in use as either a shader program or object, calling **glIsProgram()** or **glIsShader()** will return a boolean value indicating if the identifier is in use.

GLboolean **glIsProgram**(GLuint *program*);

Returns GL_TRUE if *program* is the name of a shader program. If *program* is zero, or non-zero and not the name of a shader object, GL_FALSE is returned.

GLboolean **glIsShader**(GLuint *shader*);

Returns GL_TRUE if *shader* is the name of a shader object. If *shader* is zero, or non-zero and not the name of a shader object, GL_FALSE is returned.

To aid in shader development, the OpenGL call **glValidateProgram()** can help verify that a shader will execute given the current OpenGL state. Depending upon the underlying OpenGL implementation, program validation may also return hints about performance characteristics or other useful information specific to that shader's execution for that OpenGL implementation. You would validate a program in the same manner you

would compile it by merely calling **glValidateProgram()** once all of the necessary shader objects were attached to the shader program. Similarly, you can query the results of the validation step by calling **glGetProgramiv()** with an argument of GL_VALIDATE_STATUS.

void **glValidateProgram**(GLuint *program*);

Validates *program* against the current OpenGL state settings. After validation, the value of GL_VALIDATE_STATUS will be set to either GL_TRUE, indicating that the program will execute in the current OpenGL environment, or GL_FALSE otherwise. The value of GL_VALIDATE_STATUS status can be queried by calling **glGetProgramiv()**.

The OpenGL Shading Language

This section provides an overview of the shading language used within OpenGL, commonly called GLSL. GLSL shares many traits with C++ and Java, and is used for authoring both vertex and fragment shaders, although certain features are only available for one type of shader. We will first describe GLSL's requirements, types, and other language constructs that are shared between vertex and fragment shaders, and then discuss the features unique to each type of shader.

Creating Shaders with GLSL

The Starting Point

A shader program, just like a "C" program, starts execution in **main()**. Every GLSL shader program begins life as:

```
void
main()
{
    // Your code goes here
}
```

The "//" construct is a comment and terminates at the end of the current line. "C"-type, multi-line comments—the /* and */ type—are also supported.

However, unlike ANSI "C," **main()** does not return an integer value; it is declared void.

While this is a perfectly legal GLSL vertex- or fragment-shader program that compiles and even runs, its functionality leaves something to be desired. We will continue by describing variables and their operation.

Also, as with C and its derivative languages, statements are terminated with a semicolon.

Declaring Variables

GLSL is a strongly typed language; that is, every variable must be declared and have an associated type. Variable names conform to the same rules as those for C: You can use letters, numbers, and the underscore character (_) to compose variable names. A digit cannot be the first character in a variable name.

Table 15-1 shows the basic types available in GLSL.

Type	Description
float	IEEE-like floating-point value
int	signed integer value
uint	unsigned integer value
bool	Boolean value

Table 15-1 Basic Data Types in GLSL

An additional set of types is named *samplers*, which are used as opaque handles for accessing texture maps. The various types of samplers and their uses are discussed in "Accessing Texture Maps in Shaders" on page 707.

Note: An OpenGL implementation is not required to implement these types as stringently as one might like. As long as their operation is semantically and operationally correct, the underlying implementation may vary. For example, integers may be stored in floating-point registers. It is not a good idea to assume particular numeric outcomes, such as the maximum-sized integer value being 2^{15}, based upon these types.

Variable Scoping

While all variables must be declared, they may be declared any time before their use (unlike "C," where they must be the first statements in a block of code). The scoping rules of GLSL closely parallel those of C++:

- Variables declared outside of any function definition have global scope and are visible to all functions within the shader program.
- Variables declared within a set of curly braces (e.g., function definition, block following a loop or "if" statement, and so on) exist within the scope of those braces only.
- Loop iteration variables, such as *i* in the loop

  ```
  for (int i = 0; i < 10; ++i) {
      // loop body
  }
  ```

  are only scoped for the body of the loop.

Variable Initialization

Variables may also be initialized when declared. For example:

```
int    i, numParticles = 1500;
float  force, g = -9.8;
bool   falling = true;
```

Integer constants may be expressed as octal, decimal, or hexadecimal values. An optional minus sign before a numeric value negates the constant, and a trailing 'u' or 'U' denotes an unsigned integer value.

Floating-point values must include a decimal point, unless described in scientific format (e.g., 3E-7), and may optionally include an 'f' or 'F' suffix as in "C."

Boolean values are either *true* or *false*, and can be initialized to either of those values or as the result of a operation that resolves to a boolean expression.

Constructors

As mentioned, GLSL is a strongly typed language, even more so than C++. In general, there is no implicit conversion between values, except in a few cases. For example,

```
int f = 10.0;
```

will result in a compilation error due to assigning a constant floating-point value to an integer variable. Integer values and vectors will be implicitly

converted into the equivalent floating-point value or vector. Any other conversion of values requires using a conversion function (a C++-like constructor). For example,

```
float f = 10.0;
int   ten = int(f);
```

uses the **int()** function to do the conversion. Likewise, the other types also have conversion functions: **float()**, **uint()**, **bool()**. These functions also illustrate another feature of GLSL: operator overloading, whereby each function takes various input types, but all use the same base function name. We will discuss more on functions in a bit.

Aggregate Types

Three of GLSL's primitive types can be combined to better match core OpenGL's data values and to ease computational operations.

First, GLSL supports vectors of two, three, or four dimensions for each of the primitive types. Also, matrices of floats are available. Table 15-2 lists the valid vector and matrix types.

Base Type	2-D vec	3-D vec	4-D vec	Matrix Types
float	*vec2*	*vec3*	*vec4*	*mat2, mat3, mat4* *mat2x2, mat2x3, mat2x4,* *mat3x2, mat3x3, mat3x4* *mat4x2, mat4x3, mat4x4*
int	*ivec2*	*ivec3*	*ivec4*	—
uint	*uvec2*	*uvec3*	*uvec4*	—
bool	*bvec2*	*bvec3*	*bvec4*	—

Table 15-2 GLSL Vector and Matrix Types

Matrix types that list both dimensions, such as *mat4x3*, use the first value to specify the number of columns, the second the number of rows.

Variables declared with these types can be initialized similar to their scalar counterparts:

```
vec3  velocity = vec3(0.0, 2.0, 3.0);
```

and converting between types is equally accessible:

```
ivec3 steps = ivec3(velocity);
```

Vector constructors can also be used to truncate or lengthen a vector. If a longer vector is passed into the constructor of a smaller vector, the vector is truncated to the appropriate length.

```
vec4 color;
vec3 RGB = vec3(color); // now RGB only has three elements
```

Likewise, vectors are lengthened in somewhat the same manner. Scalar values can be promoted to vectors, as in

```
vec3 white = vec3(1.0);  // white = ( 1.0, 1.0, 1.0 )
vec4 translucent = vec4(white, 0.5);
```

Matrices are constructed in the same manner and can be initialized to either a diagonal matrix or a fully populated matrix.

In the case of diagonal matrices, a single value is passed into the constructor, and the diagonal elements of the matrix are set to that value, with all others being set to zero, as in

$$m = mat3(4.0) = \begin{bmatrix} 4.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 4.0 \end{bmatrix}$$

Matrices can also be created by specifying the value of every element in the matrix in the constructor. Values can be specified by combinations of scalars and vectors, as long as enough values are provided, and each column is specified in the same manner. Additionally, matrices are specified in column-major order, meaning the values are used to populate columns before rows (which is the opposite of how "C" initializes two-dimensional arrays).

For example, we could initialize a 3 × 3 matrix in any of the following ways:

```
mat3 M = mat3(1.0, 2.0, 3.0,
              4.0, 5.0, 6.0,
              7.0, 8.0, 9.0 );

vec3 column1 = vec3(1.0, 2.0, 3.0);
vec3 column2 = vec3(4.0, 5.0, 6.0);
vec3 column3 = vec3(7.0, 8.0, 9.0);

mat3 M = mat3(column1, column2, column3);
```

or even

```
vec2 column1 = vec2(1.0, 2.0);
vec2 column2 = vec2(4.0, 5.0);
vec2 column3 = vec2(7.0, 8.0);

mat3 M = mat3(column1, 3.0,
              column2, 6.0,
              column3, 9.0);
```

all yielding the same matrix

$$M = \begin{bmatrix} 1.0 & 4.0 & 7.0 \\ 2.0 & 5.0 & 8.0 \\ 3.0 & 6.0 & 9.0 \end{bmatrix}$$

Accessing Elements in Vectors and Matrices

The individual elements of vectors and matrices can be accessed and assigned. Vectors support two types of element access: a named-component method and an array-like method. Matrices use a two-dimensional array-like method.

Components of a vector can be accessed by name, as in

```
float red = color.r;
float v_y = velocity.y;
```

or using a zero-based index scheme. The following yield identical results to the above:

```
float red = color[0];
float v_y = velocity[1];
```

In fact, as shown in Table 15-3, there are three sets of component names available, all of which do the same thing. The multiple sets are useful for clarifying the operations that you're doing.

Component Accessors	Description
(x, y, z, w)	components associated with positions
(r, g, b, a)	components associated with colors
(s, t, p, q)	components associated with texture coordinates

Table 15-3 Vector Component Accessors

A common use for component-wise access to vectors is for *swizzling* components, as you might do with colors, perhaps for color space conversion. For example, you could do the following to specify a luminance value based on the red component of an input color:

```
vec3 luminance = color.rrr;
```

Likewise, if you needed to move components around in a vector, you might do:

```
color = color.abgr; // reverse the components of a color
```

The only restriction is that only one set of components can be used with a variable in one statement. That is, you can't do:

```
vec4 color = otherColor.rgz;// Error: 'z' is from a
// different group
```

Also, a compile-time error will be raised if you attempt to access an element that's outside of what the type supports. For example,

```
vec2 pos;
float zPos = pos.z; // Error: no 'z' component in 2D vectors
```

Matrix elements can be accessed using the array notation. Either a single scalar value or an array of elements can be accessed from a matrix:

```
mat4 m = mat4(2.0);
vec4 zVec = m[2];
float yScale = m[1][1]; // or m[1].y works as well
```

Structures

You can also logically group together collections of different types in a structure. Structures are convenient for passing groups of associated data into functions. When a structure is defined, it automatically creates a new type, and implicitly defines a constructor function that takes the types of the elements of the structure as parameters.

```
struct Particle {
    float lifetime;
    vec3 position;
    vec3 velocity;
};

Particle p = Particle(10.0, pos, vel); // pos, vel are vec3's
```

Likewise, to reference elements of a structure, use the familiar "dot" notation as you would in "C."

Arrays

GLSL also supports one-dimensional arrays of any type, including structures. As with "C," arrays are indexed using brackets ([ ]). The range of elements in an array of size *n* is 0 ... *n–1*. Unlike "C," however, negative array indices are not permitted, nor are two-dimensional arrays.

Arrays can be declared sized or unsized. You might use an unsized array as a forward declaration of an array variable and later redeclare it to the appropriate size. Array declarations use the bracket notation, as in:

```
float coeff[3]; // an array of 3 floats
float[3] coeff; // same thing
int   indices[]; // unsized. Redeclare later with a size
```

Arrays are first-class types in GLSL, meaning they have constructors and can be used as function parameters and return types. To statically initialize an array of values, you would use a constructor in the following manner:

```
float coeff[3] = float[3]( 2.38, 3.14, 42.0 );
```

The dimension value on the constructor is optional.

Additionally, similar to Java, GLSL arrays have an implicit method for reporting their number of elements: the **length**() method. If you would like to operate on all the values in an array, here is an example using the **length**() method:

```
for ( int i = 0; i < coeff.length(); ++i ) {
    coeff[i] *= 2.0;
}
```

Storage Qualifiers

Types can also have modifiers that affect their behavior. There are four modifiers defined in GLSL, as shown in Table 15-4.

Type Modifier	Description
const	Labels a variable as a read-only, compile-time constant.
in	Specifies that the variable is an input to the shader stage.
out	Specifies that the variable is an output from a shader stage.
uniform	Specifies that the value is passed to the shader from the application and is constant across a given primitive.

Table 15-4 GLSL Type Modifiers

Note: In GLSL versions prior to 1.30, vertex shader input variables were qualified with the keyword "attribute." Likewise, fragment shader input variables (which correspond to vertex shader outputs) were qualified with the "varying" keyword. In anticipation of potentially adding more shading stages, both of those keywords were replaced by the more generic "in" and "out" variations. In Version 1.40, both "attribute" and "varying" were removed (though they remain accessible using the GL_ARB_compatibility extension).

All storage qualifiers are valid for globally scoped variables. Additionally, *const* is applicable to local variables and function parameters.

Const Storage Qualifier

Just as with "C," the *const* type modifier indicates that the variable is read-only. For example, the statement

```
const float Pi = 3.141529
```

sets the variable *Pi* to an approximation of π. With the addition of the *const* modifier, it becomes an error to write to a variable after its declaration, so they must be initialized when declared.

In Storage Qualifier

The *in* modifier is used to qualify inputs into a shader stage. Those inputs may be vertex attributes (for vertex shaders) or interpolated variables (for fragment shaders).

Fragment shaders can further qualify their input values using some additional keywords that are valid only in combination with the *in* keyword. Those keywords are described in Table 15-5.

in Keyword Qualifier	Description
centroid	Forces the sampling of a fragment input variable to be within the area covered by the primitive for the pixel when multisampling is enabled.
smooth	Interpolates the fragment input variable in a perspective-correct manner.

Table 15-5 Additional *in* Keyword Qualifiers (for Fragment Shader Inputs)

in Keyword Qualifier	Description
flat	Doesn't interpolate the fragment input (i.e., the input will be the same for all fragments, as in flat shading).
noperspective	Linearly interpolates the fragment variable.

Table 15.5 (continued) Additional *in* Keyword Qualifiers (for Fragment Shader Inputs)

For example, if you would like a flat-shaded, centroid-sampled fragment input, you would specify

```
flat centroid in fragment;
```

in your fragment shader.

Out Storage Qualifier

The *out* modifier is used to qualify outputs from a shader stage—for example, the transformed homogenous coordinates from a vertex shader, or the final fragment color from a fragment shader.

Vertex shaders can further qualify their output values using the *centroid* keyword, which has the same meaning here as for fragment inputs. In addition, any vertex shader output qualified by *centroid* must have a matching fragment shader input variable that is also *centroid* qualified (i.e., the fragment shader has to have a variable with the identical declaration as in the vertex shader).

Uniform Storage Qualifier

The *uniform* modifier specifies that a variable's value will be specified by the application before the shader's execution and does not change across the primitive being processed. Uniform variables are shared between vertex and fragment shaders and must be declared as global variables. Any type of variable, including structures and arrays, can be specified as uniform.

Consider a shader that uses an additional color in shading a primitive. You might declare a uniform variable to pass that information into your shaders. In the shaders, you would make the declaration:

```
uniform vec4 BaseColor;
```

Within your shaders, you can reference *BaseColor* by name, but to set its value in your application, you need to do a little extra work. The GLSL

compiler creates a table of all uniform variables when it links your shader program. To set *BaseColor*'s value from your application, you need to obtain the index of *BaseColor* in the table, which is done using the **glGetUniformLocation()** routine.

GLint **glGetUniformLocation**(GLuint *program*, const char **name*)

Returns the index of the uniform variable *name* associated with the shader *program*. *name* is a null-terminated character string with no spaces. A value of minus one (–1) is returned if *name* does not correspond to a uniform variable in the active shader *program*, or if a reserved shader variable name (those starting with `gl_` prefix) is specified.

name can be a single variable name, an element of an array (by including the appropriate index in brackets with the name), or a field of a structure (by specifying *name*, then "." followed by the field name, as you would in the shader program). For arrays of uniform variables, the index of the first element of the array may be queried either by specifying only the array name (for example, "arrayName"), or by specifying the index to the first element of the array (as in "arrayName[0]").

The returned value will not change unless the shader program is relinked (see **glLinkProgram()**).

Once you have the associated index for the uniform variable, you can set the value of the uniform variable using the **glUniform*()** or **glUniformMatrix*()** routines.

void **glUniform**{1234}{if ui}(GLint *location*, *TYPE value*);
void **glUniform**{1234}{if ui}**v**(GLint *location*, GLsizei *count*,
const *TYPE* **values*);
void **glUniformMatrix**{234}**fv**(GLint *location*, GLsizei *count*,
GLboolean *transpose*, const GLfloat **values*);
void **glUniformMatrix**{2x3,2x4,3x2,3x4,4x2,4x3}**fv**(GLint *location,*
GLsizei *count*, GLboolean *transpose*,
const GLfloat **values*);

Sets the value for the uniform variable associated with the index *location* The vector form loads *count* sets of values (from one to four values, depending upon which **glUniform*()** call is used) into the uniform

variables starting *location*. If location is the start of an array, *count* sequential elements of the array are loaded.

The floating-point forms can be used to load a single float, a floating-point vector, an array of floats, or an array of vectors of floats.

The integer forms can be used to update a single integer, an integer vector, an array of integers, or an array of integer vectors. Additionally, individual and arrays of texture samplers can also be loaded.

For **glUniformMatrix**{234}**fv()**, *count* sets of 2 × 2, 3 × 3, or 4 × 4 matrices are loaded from *values*.

For **glUniformMatrix**{2x3,2x4,3x2,3x4,4x2,4x3}**fv()**, *count* sets of like-dimensioned matrices are loaded from *values*. If *transpose* is GL_TRUE, *values* are specified in row-major order (like arrays in "C"); or if GL_FALSE is specified, *values* are taken to be in column-major order (ordered in the same manner as **glLoadMatrix()**).

Example 15-4 demonstrates obtaining a uniform variable's index and assigning values.

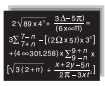
Example 15-4 Obtaining a Uniform Variable's Index and Assigning Values

```
GLint timeLoc; /* Uniform index for variable "time" in shader */
GLfloat timeValue; /* Application time */

timeLoc = glGetUniformLocation(program, "time");
glUniform1f(timeLoc, timeValue);
```

Uniform variables can also be declared within *named uniform blocks* that enable sharing and other features for shaders. Those uniform variables aren't accessible using the routines we just discussed, and they require usage of other routines described in the next section.

Uniform Blocks


Advanced

Advanced

As your shader programs become more complex, it's likely that the number of uniform variables they use will increase. Often the same uniform value is

used within several shader programs. As uniform locations are generated when a shader is linked (i.e., when **glLinkProgram()** is called), the indices may change, even though (to you) the values of the uniform variables are identical. *Uniform buffer objects* provide a method to optimize both accessing uniform variables and enabling sharing of uniform values across shader programs.

As you might imagine, that given uniform variables can exist both in your application and in a shader, you'll need to both modify your shaders and use OpenGL routines to set up uniform buffer objects.

Note: Uniform blocks were added into OpenGL Version 3.1.

Specifying Uniform Variables Blocks in Shaders

To access a collection of uniform variables using routines such as **glMapBuffer()** (see "Buffer Objects" in Chapter 2 for more details), you need to slightly modify their declaration in your shader. Instead of declaring each uniform variable individually, you group them, just as you would do in a structure, in a *uniform block*. A uniform block is specified using the *uniform* keyword. You then enclose all the variables you want in that block within a pair of braces, as shown in Example 15-5.

Example 15-5 Declaring a Uniform Variable Block

```
uniform Matrices {
    mat4 ModelView;
    mat4 Projection;
    mat4 Color;
};
```

All types, with the exception of samplers, are permitted to be within a uniform block. Additionally, uniform blocks must be declared at global scope.

Uniform Block Layout Control

A variety of qualifiers are available to specify how to lay out the variables within a uniform block. These qualifiers can be used either for each individual uniform block or to specify how all subsequent uniform blocks are arranged (after specifying a layout declaration). The possible qualifiers are detailed in Table 15-6.

Layout Qualifier	Description
shared	Specify that the uniform block is shared among multiple programs. (This is the default sharing setting).
packed	Lay out the uniform block to minimize its memory use; however, this generally disables sharing across programs.
std140	Use the default layout as described in the OpenGL specification for uniform blocks.
row_major	Cause matrices in the uniform block to be stored in a row-major element ordering.
column_major	Specify matrices should be stored in a column-major element ordering. (This is the default ordering.)

Table 15-6 Layout Qualifiers for Uniform Blocks

For example, to specify that a single uniform block is shared and has row-major matrix storage, you would declare it in the following manner:

```
layout (shared, row_major) uniform { ... };
```

The multiple qualifying options must be separated by commas within the parentheses. To affect the layout of all subsequent uniform blocks, use the following construct:

```
layout (packed, column_major) uniform;
```

With this specification, all uniform blocks declared after that line will use that layout until the global layout is changed, or unless they include a layout override specific to their declaration.

Accessing Uniform Variables Declared in a Uniform Block

While uniform blocks are named, the uniform variables declared within them are not qualified by that name. That is, a uniform block doesn't scope a uniform variable's name, so declaring two variables of the same name within two uniform blocks of different names will cause an error. Using the block name is not necessary when accessing a uniform variable, however.

Accessing Uniform Blocks from Your Application

Because uniform variables form a bridge to share data between shaders and your application, you need to find the offsets of the various uniform variables inside the named uniform blocks in your shaders. Once you know the location of those variables, you can initialize them with data, just as you would any type of buffer object (using calls such as **glBufferData()**, for example).

To start, let's assume that you already know the names of the uniform blocks used inside the shaders in your application. The first step in initializing the uniform variables in your uniform block is to obtain the index of the block for a given program. Calling **glGetUniformBlockIndex()** returns an essential piece of information required to complete the mapping of uniform variables into your application's address space.

GLuint **glGetUniformBlockIndex**(GLuint *program*,
const char ***uniformBlockName*)

Returns the index of the named uniform block specified by *uniformBlockName* associated with *program*. If *uniformBlockName* is not a valid uniform block of *program*, GL_INVALID_INDEX is returned.

To initialize a buffer object to be associated with your uniform block, you'll need to bind a buffer object to a GL_UNIFORM_BUFFER target using the **glBindBuffer()** routine (see "Creating Buffer Objects" in Chapter 2 for details).

Once we have a buffer object initialized, we need to determine how large to make it to accommodate the variables in the named uniform block from our shader. To do so, we use the routine **glGetActiveUniformBlockiv()**, requesting the GL_UNIFORM_BLOCK_DATA_SIZE, which returns the size of the block as generated by the compiler (the compiler may decide to eliminate uniform variables that aren't used in the shader, depending on which uniform block layout you've selected). **glGetActiveUniformBlockiv()** can be used to obtain other parameters associated with a named uniform block. See "The Query Commands" in Appendix B for the complete list of options.

After obtaining the index of the uniform block, we need to associate a buffer object with that block. The most common method for doing so is to call

either **glBindBufferRange()** or, if all the buffer storage is used for the uniform block, **glBindBufferBase()**.

void **glBindBufferRange**(GLenum *target*, GLuint *index*, GLuint *buffer*, GLintptr *offset*, GLsizeiptr *size*);
void **glBindBufferBase**(GLenum *target*, GLuint *index*, GLuint *buffer*);

Associates the buffer object *buffer* with the named uniform block associated with *index*. *target* can either be GL_UNIFORM_BUFFER (for uniform blocks) or GL_TRANSFORM_FEEDBACK_BUFFER (for use with transform feedback; see "Transform Feedback" on page 722). *index* is the index associated with a uniform block. *offset* and *size* specify the starting index and range of the *buffer* that is to be mapped to the uniform buffer.

Calling **glBindBufferBase()** is identical to calling **glBindBufferRange()** with *offset* equal to zero and *size* equal to the size of the buffer object.

These calls can generate various OpenGL errors: A GL_INVALID_VALUE is generated if *size* is less than zero; if *offset* + *size* is greater than the size of the buffer; if either *offset* or *size* is not a multiple of 4; or if *index* is less than zero, or greater than or equal to the value returned when querying GL_MAX_UNIFORM_BUFFER_BINDINGS.

Once the association between a named uniform block and a buffer object is made, you can initialize or change values in that block by using any of the commands that affect a buffer's values, as described in "Updating Data Values in Buffer Objects" in Chapter 2.

You may also want to specify the binding for a particular named uniform block to a buffer object, as compared to the process of allowing the linker to assign a block binding and then querying the value of that assignment after the fact. You might follow this approach if you have numerous shader programs will share a uniform block. It avoids having the block be assigned a different index for each program. To explicitly control a uniform block's binding, call **glUniformBlockBinding()** before calling **glLinkProgram()**.

GLint **gUniformBlockBinding**(GLuint *program*, GLuint *uniformBlockIndex*, GLuint *uniformBlockBinding*)

Explicitly assigns *uniformBlockIndex* to *uniformBlockBinding* for *program*.

The layout of uniform variables in a named uniform block is controlled by the layout qualifier specified when the block was compiled and linked. If you used the default layout specification, you will need to determine the offset and date-store size of each variable in the uniform block. To do so, you will use the pair of calls: **glGetUniformIndices()**, to retrieve the index of a particular named uniform variable, and **glGetActiveUniformsiv()**, to get the offset and size for that particular index, as shown in Example 15-6.

void **glGetUniformIndices**(GLuint *program*, GLsizei *uniformCount*, const char ***uniformNames*, GLuint ****uniformIndices*);

Returns the indices associated with the *uniformCount* uniform variables specified by name in the array *uniformNames* in the array *uniformIndices* for *program*. Each name in *uniformNames* is assumed to be NULL terminated, and both *uniformNames* and *uniformIndices* have *uniformCount* elements in each array.

If a name listed in *uniformNames* is not the name of an active uniform variables, the value GL_INVALID_INDEX is returned in the corresponding element in *uniformIndices*.

Example 15-6 Initializing Uniform Variables in a Named Uniform Block: ubo.c

```
/* Vertex and fragment shaders that share a block of uniforms
** named "Uniforms" */

const char* vShader = {
    "#version 140\n"
    "uniform Uniforms {"
    "    vec3  translation;"
    "    float scale;"
    "    vec4  rotation;"
    "    bool  enabled;"
    "};"
    "in vec2  vPos;"
    "in vec3  vColor;"
    "out vec4  fColor;"
    "void main()"
    "{"
    "    vec3   pos = vec3( vPos, 0.0 );"
    "    float  angle = radians( rotation[0] );"
    "    vec3   axis = normalize( rotation.yzw );"
    "    mat3   I = mat3( 1.0 );"

    "    mat3   S = mat3(       0, -axis.z,  axis.y, "
```

```
    "                       axis.z,       0, -axis.x, "
    "                      -axis.y,  axis.x,       0 );"
    "    mat3   uuT = outerProduct( axis, axis );"
    "    mat3   rot = uuT + cos(angle)*(I - uuT) + sin(angle)*S;"
    "    pos *= scale;"
    "    pos *= rot;"
    "    pos += translation;"
    "    fColor = vec4( scale, scale, scale, 1 );"
    "    gl_Position = vec4( pos, 1 );"
    "}"
};

const char* fShader = {
    "#version 140\n"
    "uniform Uniforms {"
    "    vec3  translation;"
    "    float scale;"
    "    vec4  rotation;"
    "    bool  enabled;"
    "};"
    "in vec4  fColor;"
    "out vec4 color;"
    "void main()"
    "{"
    "    color = fColor;"
    "}"
};

/* Helper function to convert GLSL types to storage sizes */
size_t
TypeSize( GLenum type )
{
    size_t  size;

#define CASE( Enum, Count, Type ) \
    case Enum: size = Count * sizeof(Type); break

    switch( type ) {
        CASE( GL_FLOAT,             1,  GLfloat );
        CASE( GL_FLOAT_VEC2,        2,  GLfloat );
        CASE( GL_FLOAT_VEC3,        3,  GLfloat );
        CASE( GL_FLOAT_VEC4,        4,  GLfloat );
        CASE( GL_INT,               1,  GLint );
        CASE( GL_INT_VEC2,          2,  GLint );
        CASE( GL_INT_VEC3,          3,  GLint );
        CASE( GL_INT_VEC4,          4,  GLint );
        CASE( GL_UNSIGNED_INT,      1,  GLuint );
```

```
        CASE( GL_UNSIGNED_INT_VEC2, 2,  GLuint );
        CASE( GL_UNSIGNED_INT_VEC3, 3,  GLuint );
        CASE( GL_UNSIGNED_INT_VEC4, 4,  GLuint );
        CASE( GL_BOOL,              1,  GLboolean );
        CASE( GL_BOOL_VEC2,         2,  GLboolean );
        CASE( GL_BOOL_VEC3,         3,  GLboolean );
        CASE( GL_BOOL_VEC4,         4,  GLboolean );
        CASE( GL_FLOAT_MAT2,        4,  GLfloat );
        CASE( GL_FLOAT_MAT2x3,      6,  GLfloat );
        CASE( GL_FLOAT_MAT2x4,      8,  GLfloat );
        CASE( GL_FLOAT_MAT3,        9,  GLfloat );
        CASE( GL_FLOAT_MAT3x2,      6,  GLfloat );
        CASE( GL_FLOAT_MAT3x4,      12, GLfloat );
        CASE( GL_FLOAT_MAT4,        16, GLfloat );
        CASE( GL_FLOAT_MAT4x2,      8,  GLfloat );
        CASE( GL_FLOAT_MAT4x3,      12, GLfloat );
        default:
            fprintf( stderr, "Unknown type: 0x%x\n", type );
            exit( EXIT_FAILURE );
            break;
    }
#undef CASE

    return size;
}


void
init()
{
    GLuint  program;

    glClearColor( 1, 0, 0, 1 );

    /* Compile and load vertex and fragment shaders (see
    ** LoadProgram.c */
    program = LoadProgram( vShader, fShader );
    glUseProgram( program );

    /* Initialize uniform values in uniform block "Uniforms" */
    {
        GLuint   uboIndex;
        GLint    uboSize;
        GLuint   ubo;
        GLvoid  *buffer;

        /* Find the uniform buffer index for "Uniforms", and
        ** determine the block's sizse*/
```

```
uboIndex = glGetUniformBlockIndex( program, "Uniforms" );

glGetActiveUniformBlockiv( program, uboIndex,
   GL_UNIFORM_BLOCK_DATA_SIZE, &uboSize );

buffer = malloc( uboSize );

if ( buffer == NULL ) {
   fprintf( stderr, "Unable to allocate buffer\n" );
   exit( EXIT_FAILURE );
}
else {
   enum { Translation, Scale, Rotation,
      Enabled, NumUniforms };

   /* Values to be stored in the buffer object */
   GLfloat    scale = 0.5;
   GLfloat    translation[] = { 0.1, 0.1, 0.0 };
   GLfloat    rotation[] = { 90, 0.0, 0.0, 1.0 };
   GLboolean  enabled = GL_TRUE;

   /* Since we know the names of the uniforms
   ** in our block, make an array of those values */
   const char* names[NumUniforms] = {
     "translation",
     "scale",
     "rotation",
     "enabled"
   };


   /* Query the necessary attributes to determine
   ** where in the buffer we should write
   ** the values */
   GLuint    indices[NumUniforms];
   GLint     size[NumUniforms];
   GLint     offset[NumUniforms];
   GLint     type[NumUniforms];

   glGetUniformIndices( program, NumUniforms,
      names, indices );
   glGetActiveUniformsiv( program, NumUniforms, indices,
      GL_UNIFORM_OFFSET, offset );
   glGetActiveUniformsiv( program, NumUniforms, indices,
      GL_UNIFORM_SIZE, size );
   glGetActiveUniformsiv( program, NumUniforms, indices,
      GL_UNIFORM_TYPE, type );
```

```
            /* Copy the uniform values into the buffer */
            memcpy( buffer + offset[Scale], &scale,
               size[Scale] * TypeSize(type[Scale]) );
            memcpy( buffer + offset[Translation], &translation,
               size[Translation] * TypeSize(type[Translation]) );
            memcpy( buffer + offset[Rotation], &rotation,
               size[Rotation] * TypeSize(type[Rotation]) );
            memcpy( buffer + offset[Enabled], &enabled,
               size[Enabled] * TypeSize(type[Enabled]) );

            /* Create the uniform buffer object, initialize
            ** its storage, and associated it with the shader
            ** program */
            glGenBuffers( 1, &ubo );
            glBindBuffer( GL_UNIFORM_BUFFER, ubo );
            glBufferData( GL_UNIFORM_BUFFER, uboSize,
               buffer, GL_STATIC_RAW );

            glBindBufferBase( GL_UNIFORM_BUFFER, uboIndex, ubo );
         }
      }
      ...
}
```

Computational Invariance

GLSL does not guarantee that two identical computations in different shaders will result in exactly the same value. The situation is no different than for computational applications executing on the CPU, where the order of compiled instructions may result in tiny differences due to the accumulation order of instructions. These tiny errors may be an issue for multipass algorithms that expect positions to be computed exactly the same for each shader pass. GLSL has a method of enforcing this type of invariance between shaders by using the *invariant* keyword.

invariant Qualifier

The *invariant* qualifier may be applied to any output varying variables of a vertex shader. The variable may be a built-in variable or a user-defined one. For example:

```
invariant gl_Position;

invariant centroid varying vec3 Color;
```

As you may recall, varying variables are used to pass data from a vertex shader into a fragment shader. Invariant variables must be declared *invariant* in both the vertex and fragment shader. The *invariant* keyword may be applied at any time before use of the variable in the shader and may be used to modify previously declared variables.

For debugging, it may be useful to impose invariance on all varying variables in shader. This can be accomplished by using the vertex shader preprocessor pragma

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

Global invariance in this manner is useful for debugging; however, it may likely have an impact on the shader's performance. Guaranteeing invariance usually disables optimizations that may have been performed by the GLSL compiler.

Statements

The real work in a shader is done by computing values and making decisions. In the same manner as C++, GLSL has a rich set of operators for constructing arithmetic operations for computing values and a standard set of logical constructs for controlling shader execution.

Arithmetic Operations

No text describing a language is complete without the mandatory table of operator precedence (see Table 15-7). The operators are ordered in decreasing precedence. In general, the types being operated on must be the same, and for vector and matrices, the operands must be of the same dimension.

Precedence	Operators	Accepted Types	Description
1	( )	—	Grouping of operations
2	[ ]	arrays	Array subscripting
	f( )	functions	Function calls and constructors
	. (period)	structures	Structure field or method access
	++ --	int, float, vec*, mat*	Post-increment and -decrement
3	++ --	int, float, vec*, mat*	Pre-increment and -decrement
	+ - !	int, float, vec*, mat*	Unary operations: explicit positive or negative value, negation

Table 15-7 GLSL Operators and Their Precedence

Precedence	Operators	Accepted Types	Description
4	* /	int, float, vec*, mat*	Multiplicative operations
5	+ -	int, float, vec*, mat*	Additive operations
6	< > <= >=	int, float, vec*, mat*	Relational operations
7	== !=	int, float, vec*, mat*	Equality operations
8	&&	bool	Logical and operation
9	^^	bool	Logical exclusive-or operation
10	||	bool	Logical or operation
11	a ? b : c	bool int, float, vec*, mat*	Selection operation (inline "if" operation; if (a) then (b) else (c))
12	= += -= *= /=	int, float, vec*, mat*	Assignment Arithmetic assignment
13	, (comma)	—	Sequence of operations

Table 15-7 (continued) GLSL Operators and Their Precedence

Note: This table lists all currently implemented operators of GLSL. Various operations that exist in C++ (%, the modulus operator, for example) are currently reserved but not implemented in GLSL.

Overload Operators

Most operators in GLSL are *overloaded*, meaning that they operate on a varied set of types. Specifically, arithmetic operations (including pre- and post-increment and -decrement) for vectors and matrices are well-defined in GLSL. For example, to multiply a vector and a matrix (recalling that the order of terms is important—matrix multiplication is non-commutative, for all you math-heads), use the following operation:

```
vec3 v;
mat3 m;
vec3 result = v * m;
```

The normal restrictions apply, that the dimensionality of the matrix and the vector must match. Additionally, scalar multiplication with a vector and matrix will produce the expected result. One notable exception is that the multiplication of two vectors will result in component-wise multiplication

of components; however, multiplying two matrices will result in normal matrix multiplication.

```
vec2 a, b, c;
mat2 m, u, v;
c = a * b; //c = ( a.x*b.x, a.y*b.y )
m = u * v; //m = (u00*v00+u01*v10u00*v01+u01*v11
              u01*v00+u11*v10u10*v01+u11*v11 )
```

Additional common vector operations (e.g., dot and cross products) are supported by function calls, as well as various per-component operations on vectors and matrices.

Logical Operations

GLSL's logical control structures are the popular if-then-else and switch statements. As with the "C" language the else clause is optional, and multiple statements require a block.

```
if ( truth ) {
    // true clause
} else {
    // false clause
}
```

Similar to the situation in C, switch statements are available (starting with GLSL 1.30) in their familiar form:

```
switch( int_value ) {
  case n:
    // statements
    break;

  case m:
    // statements
    break;

  default:
    // statements
    break;
}
```

GLSL switch statements also support "fall-through" cases—a case statement that does not end with a break statement. They do require a break for the final case in the block (before the closing brace).

Looping Constructs

GLSL supports the familiar "C" form of **for**, **while**, and **do ... while** loops.

The **for** loop permits the declaration of the loop iteration variable in the initialization clause of the **for** loop. The scope of iteration variables declared in this manner is only for the lifetime of the loop.

```
for (int i = 0; i < 10; ++i ) {
    ...
}

while (n < 10) {
    ...
}

do {
    ...
} while (n < 10);
```

Flow Control Statements

Additional control statements beyond conditionals and loops are available in GLSL. Table 15-8 describes available flow-control statements.

Statement	Description
break	Terminates execution of the block of a loop, and continues execution after the scope of that block.
continue	Terminates the current iteration of the enclosing block of a loop, resuming execution with the next iteration of the loop.
return [result]	Returns from the current subroutine, optionally providing a value to be returned from the function (assuming return value matches the return type of the enclosing function).
discard	Discards the current fragment and ceases shader execution. Discard statements are only valid in fragment shader programs.

Table 15-8 GLSL Flow-Control Statements

The `discard` statement is available only in fragment programs. The execution of the fragment shader may be terminated at the execution of the `discard` statement, but this is implementation dependent.

Functions

Functions permit you to replace occurrences of common code with a function call. This, of course, allows for smaller code, and less chances for errors. GLSL defines a number of built-in functions, which are listed in Appendix I, "Built-In OpenGL Shading Language Variables and Functions,"[2] as well as support for user-defined functions. User-defined functions can be defined in a single shader object, and reused in multiple shader programs.

Declarations

Function declaration syntax is very similar to "C," with the exception of the access modifiers on variables:

```
returnType functionName([accessModifier] type1 variable1,
                        [accessModifier] type2 varaible2,
                        ... )
{
    // function body
    return returnValue; // unless returnType is void
}
```

Function names can be any combination of letters, numbers, and the underscore character, with the exception that it can neither begin with a digit nor with `gl_`.

Return types can be any built-in GLSL type and user-defined structure; arrays are not available as return values. If a function doesn't return a value, its return type is `void`.

Parameters to functions can be of any type, including arrays (which must specify their size).

Functions must be either declared, or prototyped, before their use. Just as in C++, the compiler must have seen the function's definition before its use or an error will be raised. If a function is used in a shader object other than the one where it's defined, a prototype must be declared. A prototype is merely the function's signature without its accompanying body. Here's a simple example:

```
float HornerEvalPolynomial(float coeff[10], float x);
```

[2] This appendix is available online at `http://www.opengl-redbook.com/appendices/`.

Parameters Access Modifiers

While functions in GLSL are able to modify and return values after their execution, there's no concept of a pointer or reference, as in "C" or C++. Rather, parameters of functions have associated access modifiers indicating if the value should be copied into, or out of, a function after execution. Table 15-9 describes the available parameter access modifiers in GLSL.

Access Modifier	Description
in	value copied into a function (default if not specified)
const in	read-only value copied into a function
out	value copied out of a function (undefined upon entrance into the function)
inout	value copied into and out of a function

Table 15-9 GLSL Function Parameter Access Modifiers

The `in` keyword is optional. If a variable does not include an access modifier, then an "in" modifier is implicitly added to the parameter's declaration. However, if the variable's value needs to be copied out of a function, it must either be tagged with an "out" (for write-only variables) or an "inout" (for read-write variables). Writing to an variable not tagged with one of these modifiers will generate a compile-time error.

Additionally, to verify at compile time that a function doesn't modify an input-only variable, adding a "const in" modifier will cause the compiler to check that the variable is not written to in the function.

Using OpenGL State Values in GLSL Programs

Almost all values that you set in using the OpenGL API are accessible from within vertex and fragment shader programs. A comprehensive list of GLSL built-in variables is provided in Appendix I, "Built-In OpenGL Shading Language Variables and Functions."[3]

Accessing Texture Maps in Shaders

GLSL also supports accessing texture maps in both vertex and fragment shaders. To access a texture map, GLSL makes an association between an

[3] This appendix is available online at `http://www.opengl-redbook.com/appendices/`.

active texture unit (see “Steps in Multitexturing” in Chapter 9) configured in the OpenGL application, and a variable declared in a shader. Such variables use one of the *sampler* data types shown in Table 15-10 to allow the shader program access to the texture map’s data. The dimensions of the associated texture map must match the type of the sampler.

Sampler Name	Description
sampler1D isampler1D usample1D	Accesses a 1D texture map
sampler2D isampler2D usampler2D	Accesses a 2D texture map
sampler3D isampler3D usampler3D	Accesses a 3D texture map
samplerCube isamplerCube usamplerCube	Accesses a cube map (for reflection mapping)
sampler1DArray isampler1DArray usampler1DArray	Accesses an array of 1D texture maps
sampler2DArray isampler2DArray usampler2DArray	Accesses an array of 2D texture maps
sampler2DRect isampler2DRect usampler2DRect	Accesses a 2D texture rectangle
sampler1DShadow	Accesses a 1D shadow map
sampler2DShadow	Accesses a 2D shadow map
samplerCubeShadow	Accesses a cube map of shadowmaps
sampler1DArrayShadow	Accesses an array of 1D shadow maps
sampler2DArrayShadow	Accesses an array of 2D shadow maps

Table 15-10 Fragment Shader Texture Sampler Types

Sampler Name	Description
sampler2DRectShadow	Accesses a 2D shadow texture rectangle
samplerBuffer isamplerBuffer usamplerBuffer	Accesses a texture buffer

Table 15-10 (continued) Fragment Shader Texture Sampler Types

Samplers must be declared as uniform variables in the shader and must have their value assigned from within the OpenGL application. Samplers may also be used as parameters in functions, but must be used with samplers of matching type.

Samplers must have a texture unit assigned to them before their use in a shader, and can only be initialized by **glUniform1i()**, or **glUniform1iv()**, with the index of the texture unit that the sampler should use (see Example 15-7).

Example 15-7 Associating Texture Units with Sampler Variables

```
GLint texSampler; /* sampler index for shader variable "tex" */

texSampler = glGetUniformLocation(program, "tex");
glUniform1i(texSampler, 2); /* Set "tex" to use GL_TEXTURE2 */
```

Accessing Textures in GLSL

Sampling a texture map from within a GLSL shader uses the sampler variable that you've declared and associated with a texture unit. There are a number of texture access routines (described in "Texture Lookup Functions" in Appendix I, "Built-in OpenGL Shading Language Variables and Functions,"[4]) for accessing all OpenGL supported texture map types.

In Example 15-8, we sample the two-dimensional texture map associated with the sampler2D variable *tex*, and combine the results with the fragment's color, providing the same results as using GL_MODULATE mode for the texture environment mode.

Example 15-8 Sampling a Texture Within a GLSL Shader

```
uniform sampler2D tex;

void main()
{
    gl_FragColor = gl_Color * texture2D(tex, gl_TexCoord[0].st);
}
```

[4] This appendix is available online at `http://www.opengl-redbook.com/appendices/`.

Even though the example seems simple, that's truly all there is to do. However, much more interesting applications are enabled when the values in a texture map do not necessarily represent colors, but other data used for subsequent computation after retrieving the results from the texture map. Specifically, one application is using the values in one texture map as indices into another, described in "Dependent Texture Reads" below. The same results can be accomplished using the combiner texture environment (see "Texture Combiner Functions" in Chapter 9), but are much simpler to do using shaders.

The results you compute after sampling a texture are controlled by the code you write in your shader, but how the texture map is sampled is still controlled by the state settings in your application. For example, you control whether a texture map contains mipmaps, and how those mipmaps are sampled, as well as the filters used for resolving the returned texel values (basically, the parameters you set with **glTexParameter*()**). From inside the shader, you can control the biasing of mipmap selection, and using projective-texture techniques (see Appendix I, "Built-In OpenGL Shading Language Variables and Functions,"[5] for details and suitable functions).

Dependent Texture Reads

During the execution of a shader that employs texture mapping, you'll use texture coordinates to specify locations in texture maps and retrieve the resulting texel values. While texture coordinates are supplied for each active texture unit, you can use any values you might like as texture coordinates (assuming matching dimensionality) for use with a sampler, including values you may have just sampled from another texture map. Passing the results of one texture access as texture coordinates into another texture access operation is generally termed a *dependent texture read*, indicating that the results of the second operation are dependent on the first operation.

This is easy to implement in GLSL and is illustrated in Example 15-9.

Example 15-9 Dependent Texture Reads in GLSL

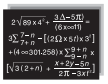
```
uniform sampler1D coords;
uniform sampler3D volume;

void main()
{
    vec3 texCoords = texture1D(coords, gl_TexCoord[0].s);
    vec3 volumeColor = texture3D(volume, texCoords);
    ...
}
```

[5] This appendix is available online at `http://www.opengl-redbook.com/appendices/`.

Texture Buffers

Advanced


Advanced

While GLSL makes arrays available, both as statically initialized values inside shaders, and as collections of values presented as an array in a uniform variable, you might occasionally need an array that exceeds the size limits available in those two options. Previous to OpenGL Version 3.1, you would have likely stored such a table of values in a texture map, and then manipulate the texture coordinates to obtain access to the values you wanted in the texture, usually in a less than straightforward manner. A more intuitive and direct solution to that data storage problem is the *texture buffer*. A texture buffer is a special type of buffer object, similar to a one-dimensional texture, that you can index using an integer value (like a normal array index) in your shader, but that offers the more expansive resources of texture memory, thereby allowing larger data sets.

You create a texture buffer just as you would any other buffer object. First you call **glBindBuffer()** with a target of GL_TEXTURE_BUFFER to create the object, and then you call **glBufferData()** (for example) to initialize its data. To bind that buffer to a texture buffer, you call **glTexBuffer()**.

void **glTexBuffer**(GLenum *target*, GLenum *internalFormat*, GLuint *buffer*);

Associates the buffer object *buffer* with *target*, causing the format of the data in *buffer* to be interpreted as having the format of *internalFormat*. *target* must be GL_TEXTURE_BUFFER, and *internalFormat* may be any of the sized texture formats: G, GL_R8, GL_R16, GL_R16F, GL_R32F, GL_R8I, GL_R16I, GL_R32I, GL_R8UI, GL_R16UI, GL_R32UI, GL_RG8, GL_RG16, GL_RG16F, GL_RG32F, GL_RG8I, GL_RG16I, GL_RG32I, GL_RG8UI, GL_RG16UI, GL_RG32UI, GL_RGBA8, GL_RGBA16, GL_RGBA16F, GL_RGBA32F, GL_RGBA8I, GL_RGBA16I, GL_RGBA32I, GL_RGBA8UI, GL_RGBA16UI, GL_RGBA32UI.

Similar to other texture maps, you specify which texture unit to associate the texture buffer with by calling **glActiveTexture()**.

Shader Preprocessor

The first step in compilation of a GLSL shader is parsing by the preprocessor. Similar to the "C" preprocessor, there are a number of directives for creating conditional compilation blocks and defining values. However, unlike the "C" preprocessor, there is no file inclusion (`#include`).

Preprocessor Directives

Table 15-11 lists the preprocessor directives accepted by the GLSL preprocessor and their functions.

Preprocessor Directive	Description
`#define` `#undef`	Control the definition of constants and macros similar to the C preprocessor

`#if`
`#ifdef`
`#ifndef`
`#else`
`#elif`
`#endif`
	Conditional code management similar to the C preprocessor, including the **defined** operator.
	Conditional expressions evaluate integer expressions and defined values (as specified by `#define`) only.

`#error` *`text`*	Cause the compiler to insert *text* (up to the first newline character) into the shader information log
`#pragma` *`options`*	Control compiler specific options
`#extension` *`options`*	Specify compiler operation with respect to specified GLSL extensions
`#version` *`number`*	Mandate a specific version of GLSL version support
`#line` *`options`*	Control diagnostic line numbering

Table 15-11 GLSL Preprocessor Directives

Macro Definition

The GLSL preprocessor allows macro definition in much the same manner as the "C" preprocessor, with the exception of the string substitution and concatenation facilities. Macros might define a single value, as in

```
#define NUM_ELEMENTS 10
```

or with parameters like

```
#define LPos(n) gl_LightSource[(n)].position
```

Additionally, there are several predefined macros for aiding in diagnostic messages (that you might issue with the `#error` directive, for example), as shown in Table 15-12.

Macro	Definition
__LINE__	Line number defined by one more than the number of newline characters processed and modified by the `#line` directive
__FILE__	Source string number currently being processed
__VERSION__	Integer representation of the OpenGL Shading Language version

Table 15-12 GLSL Preprocessor Predefined Macros

Likewise, macros (excluding those defined by GLSL) may be undefined by using the `#undef` directive. For example

```
#undef LPos
```

Preprocessor Conditionals

Identical to the processing by the "C" preprocessor, the GLSL preprocessor provides conditional code inclusion based on macro definition and integer constant evaluation.

Macro definition may be determined in two ways: Either using the `#ifdef` directive

```
#ifdef NUM_ELEMENTS
...
#endif
```

or using the **defined** operator with the `#if` or `#elif` directives

```
#if defined(NUM_ELEMENTS) && NUM_ELEMENTS > 3
...
#elif NUM_ELEMENTS < 7
...
#endif
```

Compiler Control

The `#pragma` directive provides the compiler additional information regarding how you would like your shaders compiled.

Optimization Compiler Option

The `optimize` option instructs the compiler to enable or disable optimization of the shader from the point where the directive resides forward in the shader source. You can enable or disable optimization by issuing either

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

and

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

respectively. These options may only be issued outside of a function definition. By default, optimization is enabled for all shaders.

Debug Compiler Option

The `debug` option enables or disables additional diagnostic output of the shader. You can enable or disable debugging by issuing either

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

and

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

respectively. Similar to the `optimize` option, these options may only be issued outside of a function definition, and by default, debugging is disabled for all shaders.

Global Shader Compilation Option

One final `#pragma` directive is available, `STDGL`. This option is currently used to enable invariance in the output of varying values. See “invariant Qualifier” on page 701 for details.

Extension Processing in Shaders

GLSL, like OpenGL itself, may be enhanced by extensions. As vendors may include extensions specific to their OpenGL implementation, it’s useful to have some control over shader compilation in light of possible extensions that a shader may use.

The GLSL preprocessor uses the `#extension` directive to provide instructions to the shader compiler regarding how extension availability

should be handled during compilation. For any, or all, extensions, you can specify how you would like the compiler to proceed with compilation.

```
#extension extension_name : <directive>
```

where *extensions_name* uses the same extension name returned by calling **glGetString**(GL_EXTENSIONS) or

```
#extension all : <directive>
```

to affect the behavior of all extensions.

The options available are shown in Table 15-13.

Directive	Description
require	Flag an error if the extension is not supported, or if the `all` extension specification is used.
enable	Give a warning if the particular extensions specified are not supported, or flag an error if the `all` extension specification is used.
warn	Give a warning if the particular extensions specified are not supported, or give a warning if any extension use is detected during compilation.
disable	Disable support for the particular extensions listed (that is, have the compiler act as if the extension is not supported even if it is) or all extensions if `all` is present, issuing warnings and errors as if the extension were not present.

Table 15-13 GLSL Extension Directive Modifiers

Vertex Shader Specifics

You can send data from your application into a vertex program using several mechanisms:

- By using the standard OpenGL vertex data interface (those calls that are legal between a **glBegin**() and a **glEnd**()), such as **glVertex***(), **glNormal***(), and so on. These values can vary on a per-vertex basis and are considered to be built-in attribute variables.
- By declaring uniform variables. These values remain constant across a geometric primitive.

- By declaring attribute variables, which can be updated on a per-vertex basis, in addition to the standard vertex state. (This is effectively the only method for specifying vertex attributes in OpenGL Version 3.1, unless the GL_ARB_compatibility extension is available to you.)

Likewise, a vertex program must output some data (and optionally update other variables) for continued processing by the remaining vertex processing by the OpenGL pipeline, and possibly an accompanying fragment program.

The outputs that need to be written include:

- `gl_Position`, which must be updated by the vertex program and contain the homogeneous coordinate of the vertex after modelview and projection transformation.
- Various other built-in variables that are declared varying for passing data into the fragment pipeline. These include colors, texture coordinates, and other per-fragment data. They are described in "Varying Output Variables" on page 721.
- User-defined varying variables.

Figure 15-5 illustrates the inputs and outputs of a vertex program.

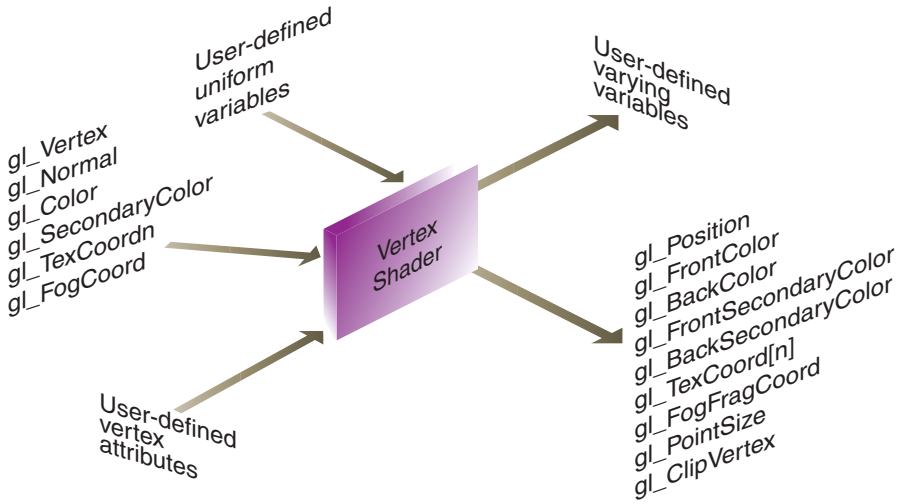


Figure 15-5 GLSL Vertex Shader Input and Output Variables

Built-In Attribute Input Variables

Table 15-14 shows variables representing those that are globally available in a vertex shader. The variables reflect the current OpenGL state, as set by the corresponding routine.

Variable	Type	Specifying function	Description
gl_Vertex	vec4	**glVertex**	Vertex's world-space coordinate
gl_Color	vec4	**glColor**	Primary color value
gl_SecondaryColor	vec4	**glSecondaryColor**	Secondary color value
gl_Normal	vec4	**glNormal**	Lighting normal
gl_MultiTexCoord*n*	vec4	**glMultiTexCoord(** *n, ...* **);**	Texture unit *n*'s texture coordinates, with *n* = 0 ... 7.
gl_FogCoord	float	**glFogCoord**	Fog coordinate
gl_VertexID	int	—	Index of current vertex since the start of the last rendering call
gl_InstanceID	int	**glDrawArraysInstanced** **glDrawElementsInstanced**	The instance ID for the associated primitives

Table 15-14 Vertex Shader Attribute Global Variables

User-Defined Attribute Variables

User-defined attribute variables are global variables that associate values passed from the OpenGL application to a vertex shader executing within the OpenGL implementation.

Attribute variables can be defined as float, floating-point vectors (vec*), or matrices (mat*).

Hint: In general, attribute variables are implemented as vec4's internal to OpenGL. If you have a number of single floating-point variables that you wish to use as vertex attributes, consider combining those values into one or more vec* structures. While declaring a single float is supported, it uses an entire vec4 to represent the single value.

To use user-defined attribute variables, OpenGL needs to know how to match the name of the variable you specified in your shader program with values that you pass into a shader. Similar to how OpenGL handles uniform variables, when a shader program is linked, the linker generates a table of variable names for attribute variables. To determine the maximum number of user-defined vertex attributes, call **glGetIntegerv()** with a parameter of GL_MAX_VERTEX_ATTRIBS.

To determine which index you need to update for the respective variable in the vertex program, you'll call **glGetAttribLocation()**, with the name of your variable, and the index corresponding to that name will be returned.

GLint **glGetAttribLocation**(GLuint *program*, const char **name*);

Returns the index associated with *name* for the shader *program*. *name* must be a null-terminated character string matching the declaration in *program*. If *name* is not an active variable, or it's the name of a built-in attribute variable, or an error occurs, a value of –1 is returned.

For example, in a vertex shader, you might declare

```
varying vec3 displacement;
```

Assuming that the shader compiled and linked appropriately, you would determine the index of "displacement" by calling:

```
int index = glGetAttribLocation(program, "displacement");
```

You can also explicitly set the binding of an attribute variable to an index using the **glBindAttribLocation()** call; however, this must be done before the shader program is linked.

void **glBindAttribLocation**(GLint *program*, GLuint *index*,
const char **name*);

Explicitly specifies which *index* location *name* should be assigned the next time *program* is linked. *index* must be an integer between zero and GL_MAX_VERTEX_ATTRIBS – 1, and *name* must be a null-terminated string. Built-in attribute variables (those beginning with `gl_`) will generate a GL_INVALID_OPERATION error.

To set the value associated with the returned index, you'll use a version of the **glVertexAttrib*()** function.

void **glVertexAttrib**{1234}{sfd}(GLuint *index*, *TYPE values*);
void **glVertexAttrib**{123}{sfd}**v**(GLuint *index*, const *TYPE *values*);
void **glVertexAttrib4**{bsifd ub us ui}**v**(GLuint *index*, *TYPE values*);
void **glVertexAttrib4Nub**(GLuint *index*, *TYPE values*);
void **glVertexAttrib4N**{bsi ub us ui}**v**(GLuint *index*, const *TYPE *values*);
void **glVertexAttribI**{1234}{i ui}(GLuint *index*, *TYPE values*);
void **glVertexAttribI4**{bsi ub us ui}**v**(GLint *index*, const *TYPE *values*);

Specifies the values for vertex attribute variables associated with *index* to *values*. For calls that do not explicitly set all four values, default values of 0.0 (0 for signed- and unsigned-integer values) will be set for the *y*- and *z*-coordinates, and 1.0 (1 for signed- and unsigned-integer values) for the *w*-coordinate.

Specifying values for *index* zero is identical to calling **glVertex*()**, with the same values.

The normalized version will convert integer input values into the range zero to one, using the mappings specified in Table 4-1 on page 198.

Matrices are updated by specifying consecutive values for *index*. *values* specified will be used to update the respective columns of the matrix.

While attribute variables are floating point, there's no restriction on the type of input data used to initialize their values. Specifically, integer-type input values can be normalized into the range of zero to one, using **glVertexAttrib4N*()**, before being assigned into the attribute variable.

Scalar (single-value), vector, and matrix attribute values can be set using **glVertexAttrib*()**. For the matrix case, multiple calls are required. In particular, for a matrix of dimension *n*, you would call **glVertexAttrib*()**, with indices: *index, index+1, ..., index+n–1*.

Vertex shaders also augment the vertex array facility (see "Vertex Arrays" on page 70). As with other types of vertex data, values for vertex attribute variables can be stored in vertex arrays and updated by calling **glDrawArrays()**, **glArrayElement()**, and so on. To specify the array to be used to update a variable's value, call **glVertexAttribPointer()**.

void **glVertexAttribPointer**(GLuint *index*, GLint *size*, GLenum *type*, GLboolean *normalized*, GLsizei *stride*, const GLvoid* *pointer*);

Specifies where the data values for *index* can be accessed. *pointer* is the memory address of the first set of values in the array. *size* represents the number of components to be updated per vertex. *type* specifies the data type (GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE) of each element in the array. *normalized* indicates that the vertex data should be normalized before being stored (in the same manner as **glVertexAttrib4N*()**). *stride* is the byte offset between consecutive elements in the array. If *stride* is 0, the data is assumed to be tightly packed.

As with other types of vertex arrays, specifying the array is only one part of the process. Each client-side vertex array needs to be enabled. Compared to using **glEnableClientState()**, arrays of vertex attributes are enabled by calling **glEnableVertexAttribArray()**.

void **glEnableVertexAttribArray**(GLuint *index*);
void **glDisableVertexAttribArray**(GLuint *index*);

Specifies that the vertex array associated with variable *index* be enabled or disabled. *index* must be a value between 0 and GL_MAX_VERTEX_ATTRIBS – 1.

Special Output Variables

The values shown in Table 15-15 are available for writing (and reading after being written) in a vertex shader. `gl_Position` specifies the vertex's final position upon exit of the vertex shader and is required to be written in the shader.

Variable Name	Type	Description
gl_Position	vec4	Transformed vertex position (in eye coordinates).
gl_PointSize	float	Point size of vertex.
gl_ClipVertex	vec4	Vertex position to be used with user-defined clipping planes. This value must be in the same coordinate system as the clipping planes: eye-coordinates or object-coordinates.

Table 15-15 Vertex Shader Special Global Variables

While you're able to set the output vertex position to any homogenous coordinate you might like, the final value of `gl_Position` is usually computed within a vertex program as:

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

Or, if you're employing a multipass algorithm (one rendering multiple images from the same geometry) that needs to have the results of the vertex shader and fixed function pipeline be consistent, set `gl_Position` using the following code:

```
gl_Position = ftransform();
```

`gl_PointSize` controls the output size of a point, similar to **glPointSize()**, but on a per-vertex basis. To control the size of points from within vertex programs, call **glEnable()** with a value of GL_VERTEX_PROGRAM_POINT_SIZE, which overrides any current point size that may have been specified.

User-defined clipping planes, as specified by **glClipPlane()**, can be used by writing a homogeneous coordinate into the `gl_ClipVertex` variable. For clipping to proceed correctly, the clipping plane specified and the coordinate written into `gl_ClipVertex` must be in the same coordinate space. The common space for clipping is eye-coordinates. You can transform the current vertex into eye-coordinates for clipping by executing:

```
gl_ClipVertex = gl_ModelViewMatrix * gl_Vertex;
```

Varying Output Variables

Table 15-16 represents those variables that can be written to in a vertex shader and have their values readable in a fragment shader. The values in the fragment shader are iterated across the fragments (or samples, if multisampling) of the primitives.

Variable Name	Type	Description
gl_FrontColor	vec4	Primary color to be used for front-facing primitives
gl_BackColor	vec4	Primary color to be used for back-facing primitives
gl_FrontSecondaryColor	vec4	Secondary color to be used for front-facing primitives

Table 15-16 Vertex Shader Varying Global Variables

Variable Name	Type	Description
gl_BackSecondaryColor	vec4	Secondary color to be used for back-facing primitives
gl_TexCoord[*n*]	vec4	n^{th} Texture coordinate values
gl_FogFragCoord	vec4	Fragment fog coordinate value

Table 15-16 (continued) Vertex Shader Varying Global Variables

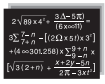
A vertex shader has the capability of setting both the front- and back-face color values for a vertex. Be default, regardless of the values set from within a vertex shader, the front-facing color will be chosen. This behavior can be modified by calling **glEnable()**, with a value of GL_VERTEX_PROGRAM_TWO_SIDE, which causes OpenGL to select colors based on the orientation of the underlying primitive.

Texture Mapping in Vertex Shaders

Texture mapping is available in vertex shaders. Using textures in vertex shaders is identical to the process described in "Accessing Textures in GLSL" on page 709, with one minor exception. Automatic mipmap selection is not done in vertex shaders. However, you can manually select which mipmap level using the **texture*Lod** routines in GLSL.

To determine if your implementation is capable of using textures in vertex shaders, query GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS using **glIntegerv()**. If a nonzero value is returned, texturing is supported.

Transform Feedback


Advanced

Advanced

Transform feedback allows an application to record the transformed primitives generated by rendering (and before clipping) using a vertex shader, into a buffer object, somewhat similar to the situation described in "Feedback" in Chapter 13, but with more flexible control of the data recorded.

Using transform feedback is a two-step process:

1. Specifying the mapping of outputs of a vertex shader into one or more buffer objects.
2. Rendering while in transform feedback mode.

To accomplish step 1, we'll call **glTransformFeedbackVaryings()** before linking our vertex shader. This function will set the output ordering of the varyings that we want to capture and specify how the data will be written out.

void **glTransformFeedbackVaryings**(GLuint *program*, GLsizei *count*, const char ***varyings*, GLenum *bufferMode*);

Assigns the ordering for *count* varying variables specified by (null-terminated) names in the array *varyings* for *program*. *bufferMode* must be either GL_SEPARATE_ATTRIBS, which specifies that the varyings should be written to *count* separate buffer objects, or GL_INTERLEAVED_ATTRIBS, which writes the *count* varying values contiguously into a single buffer object.

A program may fail to link if *count* exceeds the number of available varyings for output. A GL_INVALID_VALUE error will be generated if *program* is not a valid program object, or if *bufferMode* is set to GL_SEPARATE_ATTRIBS and *count* is greater than the value returned for a query of GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS.

Once the program has been successfully linked, and the ordering of varyings set, we can attach buffer objects to receive the transformed outputs. To bind a buffer, we use **glBindBufferRange()** (or **glBindBufferBase()**), associating the appropriate number of buffers if the *bufferMode* was set to GL_SEPARATE_ATTRIBS, or a single buffer if the *bufferMode* was set to GL_INTERLEAVED_ATTRIBS. The size of the buffer (as specified either by **glBindBufferRange()** or when the buffer was sized by calling **glBufferData()**, for example) dictates the number of attribute values recorded for the specified primitives. When the available buffer space is exhausted, no more primitives will be recorded.

To capture the transformed output, you enter transform feedback mode by calling **glBeginTransformFeedback()**, and specifying the type of primitives you would like recorded, as described in Table 15-17. The complete set of specified varyings is written for each vertex (up to space limitations in the buffer).

Transform Feedback Primitive Type	Permitted OpenGL Primitive Type
GL_POINTS	GL_POINTS
GL_LINES	GL_LINES GL_LINE_LOOP GL_LINE_STRIP
GL_TRIANGLES	GL_TRIANGLES GL_TRIANGLE_FAN GL_TRAINGLE_STRIP GL_QUADS GL_QUAD_STRIP GL_POLYGONS

Table 15-17 Transform Feedback Primitives and Their Permitted OpenGL Rendering Types

You then issue rendering commands. Vertices are transformed by your shader, and the varying variables you specified in step 1 are recorded.

void **glBeginTransformFeedback**(GLenum *primitiveMode*);
void **glEndTransformFeedback**(void);

Enters and exits transform feedback mode. *primitiveMode* must be one of GL_POINTS, GL_LINES, or GL_TRIANGLES, which represents the type of output written to the associated buffer objects.

A GL_INVALID_OPERATION error is generated if a command renders an OpenGL primitive type that is incompatible with the current transform feedback mode.

Example 15-10 illustrates the process of capturing vertex values, surface normals, and texture coordinates using transform feedback. Note that the input consists of only the vertex positions; all of the other values are generated by the vertex shader.

Example 15-10 Using Transform Feedback to Capture Geometric Primitives: xfb.c

```
GLuint
LoadTransformFeedbackShader(const char* vShader, GLsizei count,
   const GLchar** varyings)
{
    GLuint  shader, program;
    GLint   completed;
```

```
program = glCreateProgram();

/*
**  --- Load and compile the vertex shader ---
*/

if (vShader != NULL) {
    shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(shader, 1, &vShader, NULL);
    glCompileShader(shader);
    glGetShaderiv(shader, GL_COMPILE_STATUS, &completed);

    if (!completed) {
       GLint  len;
       char*  msg;

       glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
       msg = (char*) malloc(len);
       glGetShaderInfoLog(shader, len, &len, msg);
       fprintf(stderr, "Vertex shader compilation "
       "failure:\n%s\n", msg);
       free(msg);

       glDeleteProgram(program);

       exit(EXIT_FAILURE);
    }

   glAttachShader(program, shader);
}

glTransformFeedbackVaryings(program, count, varyings,
   GL_INTERLEAVED_ATTRIBS);

/*
**  --- Link program ---
*/
glLinkProgram(program);
glGetProgramiv(program, GL_LINK_STATUS, &completed);

if (!completed) {
   GLint  len;
   char*  msg;

   glGetProgramiv(program, GL_INFO_LOG_LENGTH, &len);
   msg = (char*) malloc(len);
   glGetProgramInfoLog(program, len, &len, msg);
```

```
        fprintf(stderr, "Program link failure:\n%s\n", msg);
        free(msg);

        glDeleteProgram(program);

        exit(EXIT_FAILURE);
    }

    return program;
}

void
init()
{
    /* Vertex shader generating our output values */
    const char vShader[] = {
    "#version 140\n" "in vec2  coords;"
    "out vec2 texCoords;"
    "out vec3 normal;"
    "void main() {"
    "   float angle = radians(coords[0]);"
    "   normal = vec3(cos(angle), sin(angle), 0.0);"
    "   texCoords = normal.xy;"
    "   gl_Position = vec4(normal.xy, coords[1], 1.0);"
    "}"
    };

    /* List (and ordering) of varying values written to the
    ** transform buffer */
    const char *varyings[] = {
        "texCoords",
        "normal",
        "gl_Position"
    };

    GLuint program;
    GLuint query;
    GLint  count;

    /* Load shader program and set up varyings */
    program = LoadTransformFeedbackShader(vShader, 3,
        varyings);

    glUseProgram(program);

    glGenQueries(1, &query);

    /* Bind to transform-feedback buffer */
```

```
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, xfb);

    glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN,

       query);
    glBeginTransformFeedback(GL_POINTS);
    glDrawArrays(GL_POINTS, 0, 2*(NumSlices+1));
    glEndTransformFeedback();
    glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);

    glGetQueryObjectiv(query, GL_QUERY_RESULT, &count);

    fprintf(stderr, "%d primitives written\n", count);
    ...
}
```

Fragment Shader Specifics

Like vertex programs, your OpenGL application can send data directly to a fragment program, as well as having OpenGL provide values for use in your program. All of that data, along with the inputs into a fragment shader, result in the final color and depth value of a fragment. After execution of a fragment shader, the computed values continue through the OpenGL fragment pipeline to the fragment test and blending stages.

Figure 15-6 describes the inputs and outputs of a fragment programs.

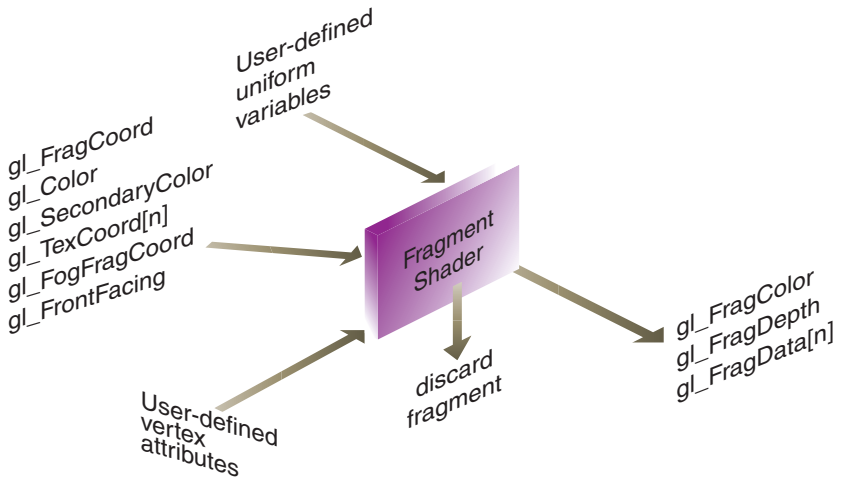


Figure 15-6 Fragment Shader Built-In Variables

Input Values

Fragment shaders receive the iterated values of the final vertex pipeline outputs. These include the fragment's position, resolved primary and secondary colors, a set of texture coordinates, and a fog coordinate distance for the fragment. All of these values are described in Table 15-18.

Variable	Type	Description
gl_FragCoord	vec4	(Read only) Position of the fragment, including the z-component, which represents the fixed-function computed depth value
gl_FrontFacing	bool	(Read only) Specifies if the fragment belongs to a front-facing primitive
gl_Color	vec4	Primary color of the fragment
gl_SecondayColor	vec4	Secondary color of the fragment
gl_TexCoord[*n*]	vec4	n^{th} texture coordinates for the fragment
gl_FogFragCoord	float	Fragment's fog coordinate, either specified as the z-coordinate of the primitive in eye space or the interpolated fog coordinate
gl_PointCoord	vec2	Fragment's location for a point sprite in the range [0.0, 1.0]. The value is undefined if the current primitive is not a point sprite or if point sprites are disabled

Table 15-18 Fragment Shader Varying Global Variables

Special Output Values

Input values are combined in a fragment program to produce the final values for fragment, as shown in Table 15-19.

Variable	Type	Description
gl_FragColor	vec4	Final color of the fragment
gl_FragDepth	float	Final depth of the fragment
gl_FragData[*n*]	vec4	Data value written to the n^{th} data buffer

Table 15-19 Fragment Shader Output Global Variables

`gl_FragColor` is the final color of the fragment. While writing to `gl_FragColor` isn't required output from a fragment program, the results of the final fragment color for writing into the color buffer are undefined.

`gl_FragDepth` is the final depth of the fragment and is utilized in the depth test. While you cannot modify the *(x, y)* coordinate of a fragment, the depth value can be modified.

Finally, additional data may be written from a fragment program. The `gl_FragData` array allows for the writing of various data into extra buffers, as described below.

Rendering to Multiple Output Buffers

A fragment shader may output values to multiple buffers simultaneously using the `gl_FragData` array. Writing a value into array element `gl_FragData[n]` will cause the color to be written into the appropriate fragment in the buffer in element *n* of the array passed to **glDrawBuffers()**.

A fragment shader may write to either `gl_FragColor` or `gl_FragData` in a shader, but not both.

User-Defined Fragment Shader Outputs

You might find that using relatively nondescript names such as `gl_FragColor` is not suitable for specifying the output of your shader. While GLSL doesn't care, the next programmer to look at your code (which you probably forgot to comment) might not immediately understand that your shader does not output a color, but rather the results of evaluating a Bessel function of the second kind (or some other mathematical archana).

With GLSL version 1.30, you can give more suitable names to the outputs from your fragment shaders, and you can control the mapping of those outputs to the various bound drawing buffers. You can either specify the mapping of fragment output name to a buffer *before* linking or query the program *after* linking to determine the layout of variables.

To use the first approach, call **glBindFragDataLocation()** before calling **glLinkProgram()**.

void **glBindFragDataLocation**(GLuint *program*, GLuint *colorNumber*, const char **name*);

Specifies the output of fragment variable *name* should be written to color buffer *colorNumber* for GLSL *program*. This affects only programs that have not yet been linked.

A GL_INVALID_VALUE error is generated if *colorNumber* is less than zero or greater than the value returned when querying GL_MAX_DRAW_BUFFERS. A GL_INVALID_OPERATION error is generated if *name* begins with `gl_`.

After linking, you can retrieve the mapping by calling **glGetFragDataLocation():**

GLint **glGetFragDataLocation**(GLuint *program*, const char **name*);

Returns the color buffer index associated with the output of *name* from GLSL *program.*

A GL_INVALID_OPERATION error is generated if *program* did not successfully link, and **glGetFragDataLocation()** is called. A value of –1 is returned if *name* is not a fragment program output associated with *program* or if another error occurred.

Appendix A

Basics of GLUT: The OpenGL Utility Toolkit

This appendix describes a subset of the OpenGL Utility Toolkit (GLUT) originally developed by Mark Kilgard. We use an open-source version of GLUT named freeglut (`http://freeglut.sourceforge.net/`) developed by Pawel W. Olszta, with contributions from Andreas Umbach and Steve Baker. GLUT has become a popular library for OpenGL programmers because it standardizes and simplifies window and event management. GLUT has been ported atop a variety of OpenGL implementations, including both the X Window System and Microsoft Windows.

This appendix has the following major sections:

- "Initializing and Creating a Window"
- "Handling Window and Input Events"
- "Loading the Color Map"
- "Initializing and Drawing Three-Dimensional Objects"
- "Managing a Background Process"
- "Running the Program"

(See "How to Obtain the Sample Code" on page xli for information about how to obtain the source code for GLUT.)

With GLUT, your application structures its event handling to use callback functions. (This method is similar to using the Xt Toolkit, also known as the X Intrinsics, with a widget set.) For example, first you open a window and register callback routines for specific events. Then you create a main loop without an exit. In that loop, if an event occurs, its registered callback functions are executed. On completion of the callback functions, flow of control is returned to the main loop.

Initializing and Creating a Window

Before you can open a window, you must specify its characteristics. Should it be single-buffered or double-buffered? Should it store colors as RGBA values or as color indices? Where should it appear on your display? To specify the answers to these questions, call **glutInit()**, **glutInitDisplayMode()**, **glutInitWindowSize()**, and **glutInitWindowPosition()** before you call **glutCreateWindow()** to open the window.

void **glutInit**(int *argc*, char ***argv*);

glutInit() should be called before any other GLUT routine, because it initializes the GLUT library. **glutInit()** will also process command line options, but the specific options are window system dependent. For the X Window System, -iconic, -geometry, and -display are examples of command line options, processed by **glutInit()**. (The parameters to **glutInit()** should be the same as those to **main()**.)

void **glutInitDisplayMode**(unsigned int *mode*);

Specifies a display mode (such as RGBA or color-index, or single- or double-buffered) for windows created when **glutCreateWindow()** is called. You can also specify that the window have an associated depth, stencil, and/or accumulation buffer. The *mask* argument is a bitwise ORed combination of GLUT_RGBA or GLUT_INDEX, GLUT_SINGLE or GLUT_DOUBLE, and any of the buffer-enabling flags: GLUT_DEPTH, GLUT_STENCIL, or GLUT_ACCUM. For example, for a double-buffered, RGBA-mode window with a depth and stencil buffer, use GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL. The default value is GLUT_RGBA | GLUT_SINGLE (an RGBA, single-buffered window).

void **glutInitContextVersion**(int *majorVersion*, int *minorVersion*);

Specifies the major and minor versions of the OpenGL implementation that you want a context created for. To use OpenGL Version 3.0 or greater, you need to call this routine before calling **glutCreateWindow()**, due to the different context creation semantics introduced by OpenGL Version 3.0.

void **glutInitWindowSize**(int *width*, int *height*);
void **glutInitWindowPosition**(int *x*, int *y*);

Requests windows created by **glutCreateWindow()** to have an initial size and position. The arguments (*x, y*) indicate the location of a corner of the window, relative to the entire display. *width* and *height* indicate the window's size (in pixels). The initial window size and position are hints and may be overridden by other requests.

int **glutCreateWindow**(char **name*);

Opens a window with previously set characteristics (display mode, width, height, and so on). The string *name* may appear in the title bar if your window system does that sort of thing. The window is not initially displayed until **glutMainLoop()** is entered, so do not render into the window until then.

The value returned is a unique integer identifier for the window. This identifier can be used for controlling and rendering to multiple windows (each with an OpenGL rendering context) from the same application.

Handling Window and Input Events

After the window is created, but before you enter the main loop, you should register callback functions using the following routines.

void **glutDisplayFunc**(void (**func*)(void));

Specifies the function that's called whenever the contents of the window need to be redrawn. The contents of the window may need to be redrawn when the window is initially opened, when the window is popped and window damage is exposed, and when **glutPostRedisplay()** is explicitly called.

void **glutReshapeFunc**(void (**func*)(int *width*, int *height*));

Specifies the function that's called whenever the window is resized or moved. The argument *func* is a pointer to a function that expects two arguments, the new width and height of the window. Typically, *func* calls **glViewport()**, so that the display is clipped to the new size, and it redefines the projection matrix so that the aspect ratio of the projected image matches the viewport, avoiding aspect ratio distortion. If **glutReshapeFunc()** isn't called or is deregistered by passing NULL, a default reshape function is called, which calls **glViewport**(*0*, *0*, *width*, *height*).

void **glutKeyboardFunc**(void (**func*)(unsigned char *key*, int *x*, int *y*));

Specifies the function, *func*, that's called when a key that generates an ASCII character is pressed. The *key* callback parameter is the generated ASCII value. The *x* and *y* callback parameters indicate the location of the mouse (in window-relative coordinates) when the key was pressed.

void **glutMouseFunc**(void (**func*)(int *button*, int *state*, int *x*, int *y*));

Specifies the function, *func*, that's called when a mouse button is pressed or released. The *button* callback parameter is GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. The *state* callback parameter is either GLUT_UP or GLUT_DOWN, depending on whether the mouse has been released or pressed. The *x* and *y* callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

void **glutMotionFunc**(void (**func*)(int *x*, int *y*));

Specifies the function, *func*, that's called when the mouse pointer moves within the window while one or more mouse buttons are pressed. The *x* and *y* callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

void **glutPostRedisplay**(void);

Marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by **glutDisplayFunc()** will be called.

Loading the Color Map

If you're using color-index mode, you might be surprised to discover there's no OpenGL routine to load a color into a color-lookup table. This is because the process of loading a color map depends entirely on the window system. GLUT provides a generalized routine to load a single color index with an RGB value, **glutSetColor()**.

void **glutSetColor**(GLint *index*, GLfloat *red*, GLfloat *green*, GLfloat *blue*);

Loads the index in the color map, *index*, with the given *red*, *green*, and *blue* values. These values are normalized to lie in the range [0.0, 1.0].

Initializing and Drawing Three-Dimensional Objects

Many sample programs in this guide use three-dimensional models to illustrate various rendering properties. The following drawing routines are included in GLUT to avoid having to reproduce the code to draw these models in each program. The routines render all their graphics in immediate mode. Each three-dimensional model comes in two flavors: wireframe without surface normals, and solid with shading and surface normals. Use the solid version when you're applying lighting. Only the teapot generates texture coordinates.

void **glutWireSphere**(GLdouble *radius,* GLint *slices*, GLint *stacks*);
void **glutSolidSphere**(GLdouble *radius,* GLint *slices*, GLint *stacks*);

void **glutWireCube**(GLdouble *size*);
void **glutSolidCube**(GLdouble *size*);

void **glutWireTorus**(GLdouble *innerRadius*, GLdouble *outerRadius*,
GLint *nsides,* GLint *rings*);
void **glutSolidTorus**(GLdouble *innerRadius*, GLdouble *outerRadius*,
GLint *nsides,* GLint *rings*);

void **glutWireIcosahedron**(void);
void **glutSolidIcosahedron**(void);

void **glutWireOctahedron**(void);
void **glutSolidOctahedron**(void);

void **glutWireTetrahedron**(void);
void **glutSolidTetrahedron**(void);

void **glutWireDodecahedron**(GLdouble *radius*);
void **glutSolidDodecahedron**(GLdouble *radius*);

void **glutWireCone**(GLdouble *radius*, GLdouble *height,* GLint *slices*, GLint *stacks*);
void **glutSolidCone**(GLdouble *radius*, GLdouble *height,* GLint *slices*, GLint *stacks*);

void **glutWireTeapot**(GLdouble *size*);
void **glutSolidTeapot**(GLdouble *size*);

Managing a Background Process

You can specify a function that's to be executed if no other events are pending—for example, when the event loop would otherwise be idle—with **glutIdleFunc()**. This is particularly useful for continuous animation or other background processing.

void **glutIdleFunc**(void (**func)*(void));

Specifies the function, *func*, to be executed if no other events are pending. If NULL (zero) is passed in, execution of *func* is disabled.

Running the Program

After all the setup is completed, GLUT programs enter an event processing loop, **glutMainLoop()**.

void **glutMainLoop**(void);

Enters the GLUT processing loop, never to return. Registered callback functions will be called when the corresponding events instigate them.

This page intentionally left blank

Appendix B

State Variables

This appendix lists the queryable OpenGL state variables, their default values, and the commands for obtaining the values of these variables, and contains the following major sections:

- "The Query Commands"
- "OpenGL State Variables"

The Query Commands

In addition to the basic commands to obtain the values of simple state variables (commands such as **glGetIntegerv()** and **glIsEnabled()**, which are described in "Basic State Management" in Chapter 2), there are other specialized commands to return more complex state variables. The prototypes for these specialized commands are listed here. Some of these routines, such as **glGetError()** and **glGetString()**, have been discussed in more detail elsewhere in the book.

To find out when you need to use these commands and their corresponding symbolic constants, use the tables in the next section, "OpenGL State Variables."

void **glGetActiveAttrib**(GLuint *program*, GLuint *index*, GLsizei *bufSize*, GLsizei **length*, GLint **size*, GLenum **type*, char **name*);

void **glGetActiveUniformBlockiv**(GLuint *program*, GLuint *uniformBlockIndex*, GLenum *pname*, GLint **params*);

GLint **glGetActiveUniformName**(GLuint *program*, GLuint *uniformIndex*, GLsizei *bufSize*, GLsizei **length*, char **uniformName*)

void **glGetActiveUniformsiv**(GLuint *program*, GLsizei *uniformCount*, const GLuint **uniformIndices*, GLenum *pname*, GLint **params*)

void **glGetAttachedShaders**(GLuint *program*, GLsizei *maxCount*, GLsizei **count*, GLuint **shaders*);

void **glGetBufferSubData(**GLenum *target*, GLintptr *offset*, GLsizeiptr *size*, GLvoid* *data*);

void **glGetBufferParameteriv**(GLenum *target*, GLenum *pname*, GLint **params*);

void **glGetBufferPointerv**(GLenum *target*, GLenum *pname,* GLvoid ***pointer*);

void **glGetClipPlane**(GLenum *plane*, GLdouble **equation*);

void **glGetColorTable**(GLenum *target*, GLenum *pname*, GLenum *type*, GLvoid **table*);

void **glGetColorTableParameter**{if}**v**(GLenum *target*, GLenum *pname*,
TYPE ***params*);

void **glGetCompressedTexImage(**GLenum *target*, GLint *lod*,
GLvoid ****pixels*);

void **glGetConvolutionFilter**(GLenum *target*, GLenum *format*,
GLenum *type*, GLvoid ****image*);

void **glGetConvolutionParameter**{if}**v**(GLenum *target*, GLenum *pname*,
TYPE ***params*);

GLenum **glGetError**(void);

void **glGetHistogram**(GLenum *target*, GLboolean reset, GLenum *format*,
GLenum *type*, GLvoid ****values*);

void **glGetHistogramParameter**{if}**v**(GLenum *target*, GLenum *pname*,
TYPE ***params*);

void **glGetLight**{if}**v**(GLenum *light*, GLenum *pname*, *TYPE* ***params*);

void **glGetMap**{ifd}**v**(GLenum *target*, GLenum *query*, *TYPE* ****v*);

void **glGetMaterial**{if}**v**(GLenum *face*, GLenum *pname*, *TYPE* ***params*);

void **glGetMinmax**(GLenum *target*, GLboolean reset, GLenum *format*,
GLenum *type*, GLvoid ****values*);

void **glGetMinmaxParameter**{if}**v** (GLenum *target*, GLenum *pname*,
TYPE ***params*);

void **glGetPixelMap**{f ui us}**v**(GLenum *map*, *TYPE* ****values*);

void **glGetPolygonStipple**(GLubyte ****mask*);

void **glGetProgramInfoLog**(GLuint *program*, GLsizei *bufSize*,
GLsizei ****length*, GLchar ****infoLog*);

void **glGetProgramiv**(GLuint *program*, GLenum *pname*, GLint ****params*);

void **glGetQueryiv**(GLenum *target*, GLenum *pname*, GLint ****params*);

void **glGetQueryObjectiv**(GLuint *id*, GLenum *pname*, GLint ****params*);

void **glGetQueryObjectuiv**(GLuint *id*, GLenum *pname*, GLuint ****params*);

void **glGetSeparableFilter**(GLenum *target*, GLenum *format*,
GLenum *type*, GLvoid ****row*, GLvoid ****column*,
GLvoid ****span*);

void **glGetShaderInfoLog**(GLuint *shader*, GLsizei *bufSize*, GLsizei **length*, GLchar **infoLog*);

void **glGetShaderiv**(GLuint *shader*, GLenum *pname*, GLint **params*);

void **glGetShaderSource**(GLuint *shader*, GLsizei *bufSize*, GLsizei **length*, GLchar **source*);

const GLubyte * **glGetString**(GLenum *name*);

const GLubyte * **glGetStringi**(GLenum *name*, GLuint *index*);

void **glGetTexEnv**{if}**v**(GLenum *target*, GLenum *pname*, *TYPE* **params*);

void **glGetTexGen**{ifd}**v**(GLenum *coord*, GLenum *pname*, *TYPE* **params*);

void **glGetTexImage**(GLenum *target*, GLint *level*, GLenum *format*, GLenum *type*, GLvoid **pixels*);

void **glGetTexLevelParameter**{if}**v**(GLenum *target*, GLint *level*, GLenum *pname*, *TYPE* **params*);

void **glGetTexParameter**{if}**v**(GLenum *target*, GLenum *pname*, *TYPE* **params*);

void **glGetUniform**{if}**v**(GLuint *program*, GLint *location*, TYPE **params*);

void **glGetVertexAttrib**{ifd}**v**(GLuint *index*, GLenum *pname*, *TYPE* **params*);

void **glGetVertexAttribPointerv**(GLuint *index*, GLenum *pname*, GLvoid ***pointer*);

GLboolean **glIsBuffer**(GLuint *buffer*);

GLboolean **glIsList**(GLuint *list*);

GLboolean **glIsProgram**(GLuint *program*);

GLboolean **glIsQuery**(GLuint *id*);

GLboolean **glIsShader**(GLuint *shader*);

GLboolean **glIsTexture**(GLuint *texObject*);

void **gluGetNurbsProperty**(GLUnurbsObj **nobj*, GLenum *property*, GLfloat **value*);

const GLubyte * **gluGetString**(GLenum *name*);

void **gluGetTessProperty**(GLUtesselator **tess*, GLenum *which*, GLdouble **data*);

OpenGL State Variables

The following pages contain tables that list the names of queryable state variables. For each variable, the tables list a description of it, its attribute group, its initial or minimum value, and the suggested **glGet*()** command to use for obtaining it. State variables that can be obtained using **glGetBooleanv()**, **glGetIntegerv()**, **glGetFloatv()**, or **glGetDoublev()** are listed with just one of these commands—the one that's most appropriate given the type of data to be returned. (Some vertex-array variables can be queried only with **glGetPointerv()**.) These state variables can't be obtained using **glIsEnabled()**. However, state variables for which **glIsEnabled()** is listed as the query command can also be obtained using **glGetBooleanv()**, **glGetIntegerv()**, **glGetFloatv()**, and **glGetDoublev()**. State variables for which any other command is listed as the query command can be obtained only by using that command.

Note: When querying texture state, such as GL_TEXTURE_MATRIX, in an implementation where the GL_ARB_multitexture extension is defined, the values returned reference the currently active texture unit only. See "Multitexturing" on page 467 for details.

If one or more attribute groups are listed, the state variable belongs to the listed group or groups. If no attribute group is listed, the variable doesn't belong to any group. **glPushAttrib()**, **glPushClientAttrib()**, **glPopAttrib()**, and **glPopClientAttrib()** may be used to save and restore all state values that belong to an attribute group (see "Attribute Groups" in Chapter 2 for more information).

All queryable state variables have initial values; however, those that are implementation-dependent may not have an initial value listed. If no initial value is listed, you need to consult the section where that variable is discussed.

More detail on all of the query functions and values is available online at `http://www.opengl.org/sdk/docs/man`.

Current Values and Associated Data

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_CURRENT_COLOR	Current color	current	(1, 1, 1, 1)	glGetIntegerv(), glGetFloatv()
GL_CURRENT_SECONDARY_COLOR	Current secondary color	current	(0, 0, 0, 1)	glGetIntegerv(), glGetFloatv()
GL_CURRENT_INDEX	Current color index	current	1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_TEXTURE_COORDS	Current texture coordinates	current	(0, 0, 0, 1)	glGetFloatv()
GL_CURRENT_NORMAL	Current normal	current	(0, 0, 1)	glGetFloatv()
GL_CURRENT_FOG_COORD	Current fog coordinate	current	0	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_POSITION	Current raster position	current	(0, 0, 0, 1)	glGetFloatv()
GL_CURRENT_RASTER_DISTANCE	Current raster distance	current	0	glGetFloatv()
GL_CURRENT_RASTER_COLOR	Color associated with raster position	current	(1, 1, 1, 1)	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_SECONDARY_COLOR	Secondary color associated with raster position	current	(0, 0, 0, 1)	glGetIntegerv(), glGetFloatv()

Table B-1 State Variables for Current Values and Associated Data

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_CURRENT_RASTER_INDEX	Color index associated with raster position	current	1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_TEXTURE_COORDS	Texture coordinates associated with raster position	current	(0, 0, 0, 1)	glGetFloatv()
GL_CURRENT_RASTER_POSITION_VALID	Raster position valid bit	current	GL_TRUE	glGetBooleanv()
GL_EDGE_FLAG	Edge flag	current	GL_TRUE	glGetBooleanv()

Table B-1 (continued) State Variables for Current Values and Associated Data

Vertex Array Data State (Not Included in Vertex Array Object State)

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_CLIENT_ACTIVE_TEXTURE	Active texture unit, for texture-coordinate array	vertex-array	GL_TEXTURE0	glGetIntegerv()
GL_ARRAY_BUFFER_BINDING	Current buffer binding	vertex-array	0	glGetIntegerv()
GL_PRIMITIVE_RESTART	Primitive restart enable	—	GL_FALSE	glIsEnabled()
GL_PRIMITIVE_RESTART_INDEX	Primitive restart index value	—	0	glGetIntegerv()

Table B-2 Vertex Array Data State Variables

Vertex Array Object State

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_VERTEX_ARRAY	Vertex array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_VERTEX_ARRAY_BINDING	Vertex array object binding	vertex-array	0	glGetIntegerv()
GL_VERTEX_ARRAY_SIZE	Coordinates per vertex	vertex-array	4	glGetIntegerv()
GL_VERTEX_ARRAY_TYPE	Type of vertex coordinates	vertex-array	GL_FLOAT	glGetIntegerv()

Table B-3 Vertex Array Object State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_VERTEX_ARRAY_STRIDE	Stride between vertices	vertex-array	0	glGetIntegerv()
GL_VERTEX_ARRAY_POINTER	Pointer to the vertex array	vertex-array	NULL	glGetPointerv()
GL_NORMAL_ARRAY	Normal array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_NORMAL_ARRAY_TYPE	Type of normal coordinates	vertex-array	GL_FLOAT	glGetIntegerv()
GL_NORMAL_ARRAY_STRIDE	Stride between normals	vertex-array	0	glGetIntegerv()
GL_NORMAL_ARRAY_POINTER	Pointer to the normal array	vertex-array	NULL	glGetPointerv()
GL_FOG_COORD_ARRAY	Fog coordinate array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_FOG_COORD_ARRAY_TYPE	Type of fog coordinate components	vertex-array	GL_FLOAT	glGetIntegerv()
GL_FOG_COORD_ARRAY_STRIDE	Stride between fog coordinates	vertex-array	0	glGetIntegerv()
GL_FOG_COORD_ARRAY_POINTER	Pointer to the fog coordinate array	vertex-array	NULL	glGetPointerv()
GL_COLOR_ARRAY	RGBA color array enable	vertex-array	GL_FALSE	glIsEnabled()

Table B-3 (continued) Vertex Array Object State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_COLOR_ARRAY_SIZE	Color components per vertex	vertex-array	4	glGetIntegerv()
GL_COLOR_ARRAY_TYPE	Type of color components	vertex-array	GL_FLOAT	glGetIntegerv()
GL_COLOR_ARRAY_STRIDE	Stride between colors	vertex-array	0	glGetIntegerv()
GL_COLOR_ARRAY_POINTER	Pointer to the color array	vertex-array	NULL	glGetPointerv()
GL_SECONDARY_COLOR_ARRAY	Secondary color array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_SECONDARY_COLOR_ARRAY_SIZE	Secondary color components per vertex	vertex-array	3	glGetIntegerv()
GL_SECONDARY_COLOR_ARRAY_TYPE	Type of secondary color components	vertex-array	GL_FLOAT	glGetIntegerv()
GL_SECONDARY_COLOR_ARRAY_ STRIDE	Stride between secondary colors	vertex-array	0	glGetIntegerv()
GL_SECONDARY_COLOR_ARRAY_ POINTER	Pointer to the secondary color array	vertex-array	NULL	glGetPointerv()
GL_INDEX_ARRAY	Color-index array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_INDEX_ARRAY_TYPE	Type of color indices	vertex-array	GL_FLOAT	glGetIntegerv()

Table B-3 (continued) Vertex Array Object State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_INDEX_ARRAY_STRIDE	Stride between color indices	vertex-array	0	glGetIntegerv()
GL_INDEX_ARRAY_POINTER	Pointer to the index array	vertex-array	NULL	glGetPointerv()
GL_TEXTURE_COORD_ARRAY	Texture-coordinate array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_TEXTURE_COORD_ARRAY_SIZE	Texture coordinates per element	vertex-array	4	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_TYPE	Type of texture coordinates	vertex-array	GL_FLOAT	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_STRIDE	Stride between texture coordinates	vertex-array	0	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_POINTER	Pointer to the texture-coordinate array	vertex-array	NULL	glGetPointerv()
GL_VERTEX_ATTRIB_ARRAY_ENABLED	Vertex attrib array enable	vertex-array	GL_FALSE	glGetVertexAttribiv()
GL_VERTEX_ATTRIB_ARRAY_SIZE	Vertex attrib array size	vertex-array	4	glGetVertexAttribiv()
GL_VERTEX_ATTRIB_ARRAY_STRIDE	Vertex attrib array stride	vertex-array	0	glGetVertexAttribiv()
GL_VERTEX_ATTRIB_ARRAY_TYPE	Vertex attrib array type	vertex-array	GL_FLOAT	glGetVertexAttribiv()

Table B-3 (continued) Vertex Array Object State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_VERTEX_ATTRIB_ARRAY_ NORMALIZED	Vertex attrib array normalized	vertex-array	GL_FALSE	glGetVertexAttribiv()
GL_VERTEX_ATTRIB_ARRAY_INTEGER	Vertex attrib array has unconverted integer values	vertex-array	GL_FALSE	glGetVertexAttribiv()
GL_VERTEX_ATTRIB_ARRAY_POINTER	Vertex attrib array pointer	vertex-array	NULL	glGetVertexAttrib Pointer()
GL_EDGE_FLAG_ARRAY	Edge flag array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_EDGE_FLAG_ARRAY_STRIDE	Stride between edge flags	vertex-array	0	glGetIntegerv()
GL_EDGE_FLAG_ARRAY_POINTER	Pointer to the edge-flag array	vertex-array	NULL	glGetPointerv()
GL_ARRAY_BUFFER_BINDING	Current buffer binding	vertex-array	0	glGetIntegerv()
GL_VERTEX_ARRAY_BUFFER_BINDING	Vertex array buffer binding	vertex-array	0	glGetIntegerv()
GL_NORMAL_ARRAY_BUFFER_BINDING	Normal array buffer binding	vertex-array	0	glGetIntegerv()
GL_COLOR_ARRAY_BUFFER_BINDING	Color array buffer binding	vertex-array	0	glGetIntegerv()
GL_INDEX_ARRAY_BUFFER_BINDING	Index array buffer binding	vertex-array	0	glGetIntegerv()

Table B-3 (continued) Vertex Array Object State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_TEXTURE_COORD_ARRAY_BUFFER_BINDING	Texture-coordinate array buffer binding	vertex-array	0	glGetIntegerv()
GL_EDGE_FLAG_ARRAY_BUFFER_BINDING	Edge flag array buffer binding	vertex-array	0	glGetIntegerv()
GL_SECONDARY_COLOR_ARRAY_BUFFER_BINDING	Secondary color array buffer binding	vertex-array	0	glGetIntegerv()
GL_FOG_COORD_ARRAY_BUFFER_BINDING	Fog-coordinate array buffer binding	vertex-array	0	glGetIntegerv()
GL_ELEMENT_ARRAY_BUFFER_BINDING	Element array buffer binding	vertex-array	0	glGetIntegerv()
GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	Vertex attribute array buffer binding	vertex-array	0	glGetVertexAttribiv()
GL_VERTEX_ARRAY_BINDING	Current vertex array object binding	vertex-array	0	glGetIntegerv()

Table B-3 (continued) Vertex Array Object State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_BUFFER_SIZE	Buffer data size	—	0	glGetBufferParameteriv()
GL_BUFFER_USAGE	Buffer usage pattern	—	GL_STATIC_DRAW	glGetBufferParameteriv()
GL_BUFFER_ACCESS	Buffer access flag	—	GL_READ_WRITE	glGetBufferParameteriv()
GL_BUFFER_ACCESS_FLAGS	Extended buffer access flags	—	0	glGetBufferParameteriv()
GL_BUFFER_MAPPED	Buffer map flag	—	GL_FALSE	glGetBufferParameteriv()
GL_BUFFER_MAP_POINTER	Mapped buffer pointer	—	NULL	glGetBufferPointerv()
GL_BUFFER_MAP_OFFSET	Start of mapped buffer range	—	0	glGetBufferParameteriv()
GL_BUFFER_MAP_LENGTH	Size of mapped buffer range	—	0	glGetBufferParameteriv()

Table B-4 Vertex Buffer Object State Variables

Transformation

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_COLOR_MATRIX	Color matrix stack	—	Identity	glGetFloatv()
GL_TRANSPOSE_COLOR_MATRIX	Stack of transposed color matrices	—	Identity	glGetFloatv()
GL_MODELVIEW_MATRIX	Modelview matrix stack	—	Identity	glGetFloatv()
GL_TRANSPOSE_MODELVIEW_MATRIX	Stack of transposed modelview matrices	—	Identity	glGetFloatv()
GL_PROJECTION_MATRIX	Projection matrix stack	—	Identity	glGetFloatv()
GL_TRANSPOSE_PROJECTION_MATRIX	Stack of transposed projection matrices	—	Identity	glGetFloatv()
GL_TEXTURE_MATRIX	Texture matrix stack	—	Identity	glGetFloatv()
GL_TRANSPOSE_TEXTURE_MATRIX	Stack of transposed texture matrices	—	Identity	glGetFloatv()
GL_VIEWPORT	Viewport origin and extent	viewport	—	glGetIntegerv()
GL_DEPTH_RANGE	Depth range near and far	viewport	0, 1	glGetFloatv()
GL_COLOR_MATRIX_STACK_DEPTH	Color matrix stack pointer	—	1	glGetIntegerv()

Table B-5 Transformation State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_MODELVIEW_STACK_DEPTH	Modelview matrix stack pointer	—	1	glGetIntegerv()
GL_PROJECTION_STACK_DEPTH	Projection matrix stack pointer	—	1	glGetIntegerv()
GL_TEXTURE_STACK_DEPTH	Texture matrix stack pointer	—	1	glGetIntegerv()
GL_MATRIX_MODE	Current matrix mode	transform	GL_MODELVIEW	glGetIntegerv()
GL_NORMALIZE	Current normal normalization on/off	transform/ enable	GL_FALSE	glIsEnabled()
GL_RESCALE_NORMAL	Current normal rescaling on/off	transform/ enable	GL_FALSE	glIsEnabled()
GL_CLIP_DISTANCE*i* (replacing GL_ CLIP_PLANE*i* in Version 3.0 and greater)	User clipping-plane coefficients	transform	(0, 0, 0, 0)	glGetClipPlane()
GL_CLIP_DISTANCE*i* (replacing GL_ CLIP_PLANE*i* in Version 3.0 and greater)	*i*th user clipping plane enabled	transform/ enable	GL_FALSE	glIsEnabled()

Table B-5 (continued) Transformation State Variables

Coloring

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_FOG_COLOR	Fog color	fog	(0, 0, 0, 0)	glGetFloatv()
GL_FOG_INDEX	Fog index	fog	0	glGetFloatv()
GL_FOG_DENSITY	Exponential fog density	fog	1.0	glGetFloatv()
GL_FOG_START	Linear fog start	fog	0.0	glGetFloatv()
GL_FOG_END	Linear fog end	fog	1.0	glGetFloatv()
GL_FOG_MODE	Fog mode	fog	GL_EXP	glGetIntegerv()
GL_FOG	True if fog enabled	fog/enable	GL_FALSE	glIsEnabled()
GL_FOG_COORD_SRC	Source of coordinates for fog calculation	fog	GL_FRAGMENT_DEPTH	glGetIntegerv()
GL_COLOR_SUM	True if color sum is enabled	fog/enable	GL_FALSE	glIsEnabled()
GL_SHADE_MODEL	glShadeModel() setting	lighting	GL_SMOOTH	glGetIntegerv()
GL_CLAMP_VERTEX_COLOR	Vertex clamping color	lighting/enable	GL_TRUE	glGetIntegerv()
GL_CLAMP_FRAGMENT_COLOR	Fragment clamping color	color-buffer/enable	GL_FIXED_ONLY	glGetIntegerv()
GL_CLAMP_READ_COLOR	Read color clamping	color-buffer/enable	GL_FIXED_ONLY	glGetIntegerv()

Table B-6 Coloring State Variables

Lighting

See also Tables 5-1 and 5-3 for initial values.

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_LIGHTING	True if lighting is enabled	lighting /enable	GL_FALSE	glIsEnabled()
GL_COLOR_MATERIAL	True if color tracking is enabled	lighting	GL_FALSE	glIsEnabled()
GL_COLOR_MATERIAL_PARAMETER	Material properties tracking current color	lighting	GL_AMBIENT_ AND_DIFFUSE	glGetIntegerv()
GL_COLOR_MATERIAL_FACE	Face(s) affected by color tracking	lighting	GL_FRONT_ AND_BACK	glGetIntegerv()
GL_AMBIENT	Ambient material color	lighting	(0.2, 0.2, 0.2, 1.0)	glGetMaterialfv()
GL_DIFFUSE	Diffuse material color	lighting	(0.8, 0.8, 0.8, 1.0)	glGetMaterialfv()
GL_SPECULAR	Specular material color	lighting	(0.0, 0.0, 0.0, 1.0)	glGetMaterialfv()
GL_EMISSION	Emissive material color	lighting	(0.0, 0.0, 0.0, 1.0)	glGetMaterialfv()
GL_SHININESS	Specular exponent of material	lighting	0.0	glGetMaterialfv()
GL_LIGHT_MODEL_AMBIENT	Ambient scene color	lighting	(0.2, 0.2, 0.2, 1.0)	glGetFloatv()
GL_LIGHT_MODEL_LOCAL_VIEWER	Viewer is local	lighting	GL_FALSE	glGetBooleanv()

Table B-7 Lighting State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_LIGHT_MODEL_TWO_SIDE	Use two-sided lighting	lighting	GL_FALSE	glGetBooleanv()
GL_LIGHT_MODEL_COLOR_ CONTROL	Color control	lighting	GL_SINGLE_ COLOR	glGetIntegerv()
GL_AMBIENT	Ambient intensity of light i	lighting	(0.0, 0.0, 0.0, 1.0)	glGetLightfv()
GL_DIFFUSE	Diffuse intensity of light i	lighting	—	glGetLightfv()
GL_SPECULAR	Specular intensity of light i	lighting	—	glGetLightfv()
GL_POSITION	Position of light i	lighting	(0.0, 0.0, 1.0, 0.0)	glGetLightfv()
GL_CONSTANT_ATTENUATION	Constant attenuation factor	lighting	1.0	glGetLightfv()
GL_LINEAR_ATTENUATION	Linear attenuation factor	lighting	0.0	glGetLightfv()
GL_QUADRATIC_ATTENUATION	Quadratic attenuation factor	lighting	0.0	glGetLightfv()
GL_SPOT_DIRECTION	Spotlight direction of light i	lighting	(0.0, 0.0, −1.0)	glGetLightfv()
GL_SPOT_EXPONENT	Spotlight exponent of light i	lighting	0.0	glGetLightfv()
GL_SPOT_CUTOFF	Spotlight angle of light i	lighting	180.0	glGetLightfv()
GL_LIGHTi	True if light i enabled	lighting /enable	GL_FALSE	glIsEnabled()
GL_COLOR_INDEXES	c_a, c_d, and c_s for color-index lighting	lighting	0, 1, 1	glGetMaterialfv()

Table B-7 (continued) Lighting State Variables