

22-3 The OpenGL Shading Language

GLSL is a C-like language designed to directly support the development of shaders. It has a wide variety of data types for representing typical shading data items such as vectors, colors, and matrices, along with a collection of built-in operators that simplify manipulation of those data items.

The designers of GLSL attempted to create a shading language that met a number of fairly ambitious goals. They wanted a high-level, easy-to-use programming language that would work well with OpenGL. It needed to be as hardware-independent as possible, to allow the same shaders to be used with graphics hardware from different manufacturers. In addition, because graphics hardware continues to evolve, the language should not be tied to a particular type or generation of hardware; rather, it should be powerful enough to take advantage of the capabilities of the underlying hardware and flexible enough in its design that it can accommodate the rapid evolution of that hardware.

Although the language looks very much like C or C++, it's important to remember that GLSL really isn't either one of them. There are several differences in the way that function parameters are handled, and the language is much stricter with respect to type checking issues. In addition, many familiar C and C++ data types and language constructs (such as pointer variables and widespread implicit conversion between data types) were intentionally not included in GLSL.

Like OpenGL, GLSL has evolved since it was first introduced. New features have been added, and existing features have been *deprecated* (that is, marked for future removal from the language) in favor of new features. Depending on which version of GLSL your implementation supports, some features may not be available. The following code can be used to determine your versions of OpenGL and GLSL:

```
printf ("OpenGL version: %s\n",
        (char *) glGetString (GL_VERSION));

printf ("GLSL version: %s\n",
        (char *) glGetString (GL_SHADING_LANGUAGE_VERSION));
```

The first statement prints out a string containing version information for your OpenGL implementation, and the second prints out the corresponding GLSL version information.

We also note that our discussion of GLSL will, necessarily, be incomplete. The language has far too many features to allow us to explore them completely in a single chapter. Instead, we will discuss enough details of shader creation and use that you may begin to experiment with your own shaders. For more in-depth study of GLSL, any of the GLSL references described at the end of this chapter should suffice.

Shader Structure

Most GLSL programs will contain both a vertex shader and a fragment shader. Each shader contains a main routine which, in fact, is a function named `main`. Shaders may also contain supporting functions, as well as global variables to facilitate communication between the vertex and fragment shaders.

A shader's main routine will vary depending on what functionality is required, but certain operations must be performed. As mentioned earlier, a vertex shader will be executed on every vertex that passes through the pipeline;

even if it does nothing else, it must transform the vertex into clip space. This is accomplished by multiplying the vertex by the modelview matrix, and then multiplying that result by the projection matrix. Each of these values is available to the vertex shader as a built-in global variable. These built-in global variables all have names beginning with the character sequence `gl_`; to transform the vertex into clip space, the vertex shader must use the vertex position (`gl_Vertex`) and the contents of the modelview and projection matrices (`gl_ModelViewMatrix`, `gl_ProjectionMatrix`). The transformed vertex must be placed in the global variable `gl_Position` so that the next stage in the OpenGL pipeline can use it. Here is an example of a minimal vertex shader:

```
void main ()
{
    gl_Position = gl_ProjectionMatrix *
                  (gl_ModelViewMatrix * gl_Vertex);
}
```

There are other ways to perform this transformation. An additional global variable named `gl_ModelViewProjectionMatrix` contains the product of the projection and modelview matrices and can be used to reduce the transformation to one multiplication. Because this operation is standard, a built-in function is also available to perform it:

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

Another common operation performed in vertex shaders is the assignment of a color to the vertex. This is accomplished by assigning a color value to the global variable `gl_FrontColor`, as follows:

```
void main ()
{
    gl_Position = gl_ProjectionMatrix *
                  (gl_ModelViewMatrix * gl_Vertex);
    gl_FrontColor = gl_Color;
}
```

The `gl_Color` variable contains whatever color the OpenGL application associated with the vertex by calling `glColor`. As you might expect, there is also a `gl_BackColor` global variable, which can be used when two-sided lighting is being used in the OpenGL program (see Section 17-11 for a discussion of two-sided lighting).

Fragment shaders are responsible for computing the color associated with a fragment. At minimum a fragment shader must assign that color to the global variable `gl_FragColor`. The shader can compute the color, or can retrieve whatever color was assigned by the vertex shader, as in this example:

```
void main ()
{
    gl_FragColor = gl_Color;
}
```

It is important to note that although it appears that this fragment shader is accessing the same global variable used by the vertex shader, the contents of `gl_Color` are modified by the OpenGL pipeline between the execution of the two shaders.

The fragment
depending
belongs to.

Using Shaders

Unlike Ope
Instead, the
The process
both a verte

1. Create
2. Attach
3. Comp
4. Create
5. Attach
6. Link t

The sha
of character
shader sou
program as
a file into a
determines
the file into

```
//inclu
//inclu
/*
** Gre
** who
** the
** a r
** //
```

```
GLchar
FTI
GLD
in
```

```
/*
if
//
fp
if
/*
fa
co
re
```

The fragment shader will see either the `gl_FrontColor` or the `gl_BackColor`, depending upon which side of the primitive the fragment being processed belongs to.

Using Shaders in OpenGL

Unlike OpenGL programs themselves, shader programs are not precompiled. Instead, they are compiled during the execution of the OpenGL program itself. The process involves a series of steps; for example, assuming that we are using both a vertex and a fragment shader, we would do the following:

1. Create two shader objects.
2. Attach the source for each shader to its shader object.
3. Compile the shaders.
4. Create a program object.
5. Attach the shader objects to the program object.
6. Link the program.

The shader source code must be a null-terminated C-style string (a sequence of characters, followed by a trailing byte containing the value 0). Commonly, the shader source code is put into a text file, from which it is read into the OpenGL program as a single string. Here is an example function that reads the contents of a file into a dynamically allocated string buffer. The function opens the file and determines how many characters are in it. It then allocates a string buffer, reads the file into that as a single string, and returns the pointer to the string buffer.

```
#include <stdio.h>
#include <stdlib.h>

/*
** Create a null-terminated string from the contents of a file
** whose name is supplied as a parameter. Return a pointer to
** the string, unless something goes wrong, in which case return
** a null pointer.
*/

GLchar *readTextFile( const char *name ) {
    FILE *fp;
    GLchar *content = NULL;
    int count=0;

    /* verify that we were actually given a name */
    if (name == NULL) return NULL;

    /* attempt to open the file */
    fp = fopen( name, "rt" ); /* open the file */
    if (fp == NULL) return NULL;

    /* determine the length of the file */
    fseek (fp, 0, SEEK_END);
    count = ftell (fp);
    rewind( fp );
```

```

/* allocate a buffer and read the file into it */
if( count > 0 ) {
    content = (GLchar *) malloc (sizeof(char) * (count+1));
    if( content != NULL ) {
        count = fread (content, sizeof(char), count, fp);
        content[count] = '\0';
    }
}

fclose (fp);

return content;
}

```

To create our shader program, we must first create two shader objects:

```

GLuint vertShader, fragShader;

vertShader = glCreateShader (GL_VERTEX_SHADER);
fragShader = glCreateShader (GL_FRAGMENT_SHADER);

```

Each call to `glCreateShader` returns a *handle* that is associated with a shader object. We use this handle whenever we need to refer to the shader object, such as when we want to attach source code to it.

Next, we read in the source for each shader. There is no restriction on the names of shader source files; assuming that our vertex shader is in a file named `simpleShader.vert` and our fragment shader is in a file named `simpleShader.frag`, we can read them into our program as follows:

```

GLchar *vertSource, *fragSource;

vertSource = readTextFile ("simpleShader.vert");
if (vertSource == NULL) {
    fputs ("Failed to read vertex shader\n", stderr);
    exit (EXIT_FAILURE);
}

fragSource = readTextFile ("simpleShader.frag");
if (fragSource == NULL) {
    fputs ("Failed to read fragment shader\n", stderr);
    exit (EXIT_FAILURE);
}

```

Now that we have the source strings, we must attach them to the shaders:

```

glShaderSource (vertShader, 1,
                (const GLchar **) &vertSource, NULL);

glShaderSource (fragShader, 1,
                (const GLchar **) &fragSource, NULL);

free (vertSource);
free (fragSource);

```

This fu
object.
is the r
pointer
the str
of the s
objects

Th

gl

gl

It is a g
pilation
the sta

GL

gl

if

}

gl

if

}

Or

shader

GL

pr

gl

gl

gl

The gl

dle to

gl

if

}

On

error o

This function allows us to attach several shader source strings to the same shader object. The first parameter is the shader object to be used. The second parameter is the number of source strings to be attached; the third parameter is an array of pointers to the strings. Finally, the fourth parameter tells `glShaderSource` that the strings are terminated by null characters. `glShaderSource` makes a copy of the string contents, so once we have attached the source strings to the shader objects, we can deallocate the strings to reduce our memory use.

The next step is to compile the shaders:

```
glCompileShader (vertShader);
glCompileShader (fragShader);
```

It is a good idea to verify that the compilation succeeded. We can retrieve the compilation status with the `glGetShaderiv` function. If the compilation succeeded, the status will be `GL_TRUE`:

```
GLint status;

glGetShaderiv (vertShader, GL_COMPILE_STATUS, &status);
if (status != GL_TRUE ) {
    fputs ("Error in vertex shader compilation\n", stderr);
    exit (EXIT_FAILURE);
}

glGetShaderiv (fragShader, GL_COMPILE_STATUS, &status);
if (status != GL_TRUE ) {
    fputs ("Error in fragment shader compilation\n", stderr);
    exit (EXIT_FAILURE);
}
```

Once we have compiled them, we create our program object, attach the shaders to it, and link the program:

```
GLuint program;

program = glCreateProgram ();

glAttachShader (program, vertShader);
glAttachShader (program, fragShader);

glLinkProgram (program);
```

The `glCreateProgram` function allocates a program object and returns its handle to us. Again, it is a good idea to verify that the link operation succeeded:

```
glGetProgramiv (vertShader, GL_LINK_STATUS, &status);
if (status != GL_TRUE ) {
    fputs( "Error when linking shader program\n", stderr );
    exit (EXIT_FAILURE);
}
```

Our error checking here is very rudimentary—all it reveals is *whether* an error occurred, not *what* the error was. We can get more information about what

happened by retrieving the shader or program information log. To do this, we first ask for the length of the log, and then retrieve the log into a string buffer, which we can then print out. Here is an example, using dynamically allocated buffers:

```
GLint length;
GLsizei num;
char *log;

glGetShaderiv (vertShader, GL_INFO_LOG_LENGTH, &length);
if (length > 0) {
    log = (char *) malloc (sizeof(char) * length);
    glGetShaderInfoLog (vertShader, length, &num, log);
    fprintf (stderr, "%s\n", log);
}

glGetProgramiv (program, GL_INFO_LOG_LENGTH, &length);
if (length > 0) {
    log = (char *) malloc (sizeof(char) * length);
    glGetProgramInfoLog (program, length, &num, log);
    fprintf (stderr, "%s\n", log);
}
```

The information log functions have the same parameter list. The first parameter is the object whose log we want to retrieve. The fourth parameter is the buffer into which the log will be placed, expressed as a null-terminated string; the second parameter is the size of that buffer (so that the function will not overrun the buffer). The function will place the number of bytes written into the buffer (not including the trailing null) into the third parameter.

We can have any number of shader program objects in our OpenGL program, which allows us to apply different shaders to each object in our scene. To use a shader program, we make it the active shader before drawing the object to which it will be applied:

```
glUseProgram (program);
```

Once we activate a shader, it will be applied to every object that we draw until we activate a different shader. If we have activated a shader for one or more objects and then want to “deactivate” it, we call `glUseProgram` again but give it a value of 0 as the program object:

```
glUseProgram (0);
```

Finally, during execution, we may want to delete shader objects or program objects when we are done with them. The functions `glDeleteShader` and `glDeleteProgram` are used to do this. Each takes an object handle for the appropriate type of object (shader or program) as its only parameter. The memory associated with the object will be deallocated, and the object handle is marked as unused. Deleting a program object detaches the shaders associated with it but does not delete them; they are still usable, and they can be attached to another program object. We can explicitly detach a shader object from a program object with the function `glDetachShader`, which takes the program object given as its first parameter and detaches the shader object given as the second parameter from it.

If we delete a shader object before the program object it is attached to is deleted, the actual deletion is deferred until the program object is deleted.

Similar
the dele

Basic D

The set
C-famil
either d
categori
grouped
In g
and may
can be i
dependi

Scal
(bool),
values,
range of
are avail

Vectors

Vectors
four com
values; a
ivec2, u
for exam
value, or
is done u
four elem

vec4

GLSL
ables can
second h
vector el
named p
tion.y,
point in
names r,
a texture
that the v
positio

It is a
called sw
instead o
examples

vec4

v.xyz

v.xyz

v.rgb

v.y

v.sp

Similarly, if we delete a program object while it is still the active shader program, the deletion will be deferred until the shader program is no longer active.

Basic Data Types

The set of data types provided by GLSL is significantly larger than what is found in C-family languages. At the same time, some familiar types from those languages either don't exist in GLSL, or exist in modified form. GLSL data types can be categorized as scalar types, vectors, matrices, and samplers. Any of these can be grouped using structure and array capabilities.

In general, variable declarations have the same form as those in C and C++, and may occur anywhere wherever needed within shader source code. Variables can be initialized at declaration time; however, the syntax for initialization varies depending on the type of variable being initialized.

Scalar types are limited to integer (`int`), unsigned integer (`uint`), Boolean (`bool`), and floating-point (`float`). Boolean variables have only two possible values, `true` and `false`. Integer and floating-point variables have the usual range of values possible in most programming languages, and most C operators are available, with the exception of bitwise operators.

Vectors

Vectors of each of the four scalar types are available, and can have two, three, or four components. Declarations of `vec2`, `vec3`, and `vec4` contain floating-point values; a one-character prefix is added for the other scalar types (for example, `ivec2`, `uvec2`, and `bvec2`). The vector types can be used for any kind of data—for example, a `vec4` could contain a red, green, blue, and alpha (RGBA) color value, or the *x*, *y*, *z*, and *w* components of a point, and so on. Vector initialization is done using the constructor syntax of C++; for example, we could initialize the four elements of a `vec4` to the values 1.0, 2.0, 3.0, and 4.0 as follows:

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

GLSL provides several mechanisms for manipulating vectors. Vector variables can be subscripted like arrays, with the first element having subscript 0, the second having subscript 1, and so on. In addition, structure-like referencing of vector elements is possible. For example, the four elements of a `vec4` variable named `position` can be accessed with the expressions `position.x`, `position.y`, `position.z`, and `position.w`, respectively, treating the variable as a point in space. However, the same four elements can also be accessed with the names `r`, `g`, `b`, and `a`, treating it as an RGBA color, or as `s`, `t`, `p`, and `q`, treating it as a texture coordinate. The only compile-time type checking done here is verifying that the vector is large enough to contain the requested element; `position.y`, `position.b`, and `position.p` all access the third element of the vector.

It is also possible to access collections of vector elements using a technique called *swizzling*. Swizzling is a generalization of the structure-access mechanism; instead of a single element name, multiple names can be used. Here are several examples:

```
vec4 v;

v.xyzw // a vec4 identical to v
v.xyz  // a vec3 containing the first three elements of v
v.rgb  // a vec3 containing the first three elements
v.y    // a float containing the second element
v.sp   // a vec2 containing the first and third elements
```

Element names can also be listed in order or out of order, or they can be duplicated—the only restriction is that they must be from the same name set (*xyzw*, *rgba*, or *stpq*):

```
vec4 a = vec4(1.0, 2.0, 3.0, 4.0);
vec3 b = v.yzx;      // (2.0, 3.0, 1.0)
vec4 c = v.rrb;      // (1.0, 1.0, 3.0, 3.0)
```

Arithmetic operators are overloaded to allow multiplication between vectors and matrices.

Matrices

Matrices of floating-point values can be declared. Square matrices (that is, $n \times n$ elements) can be declared as *mat2*, *mat3*, and *mat4* variables. Non-square matrices can be declared as *matmxn*, where *m* is the number of columns and *n* the number of rows. Elements of matrices can be accessed using array notation. It is possible to access an entire column at once by using a single subscript, or a single element by using two subscripts. As in OpenGL, matrices are stored in *column-major* order, so the first subscript is the column number and the second is the row number. For example, assuming the declaration *mat4 m*, *m[2]* is a *vec4* containing the third column, and *m[1][3]* is a *float* containing the second element of the fourth row. Initialization is done using constructor syntax, listing the elements in column-major order:

```
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
```

creates the matrix

$$m = \begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix} \quad (22-2)$$

Arithmetic operators are overloaded to allow matrix manipulation.

Structures and Arrays

Structures and arrays are similar to their C counterparts. Arrays can be created from any type, including vectors, matrices, structures, and scalars. Structure members can be of any type known to the shader compiler at the time of declaration, including other structures and arrays. A structure declaration is considered automatically to be a type declaration; variables of the structure type are declared simply by using the structure name tag. For example:

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

light desklamp;
light spotlights[4];
```

As mentioned earlier, GLSL is a much stricter language with respect to data types than either C or C++. Because there is a Boolean type, conditional expressions must always be Boolean, unlike C and C++ (which allow the use of any

expressio
expressio
or unsig
versions
requeste

Control

GLSL pro
for, whi
break an
version c
else co
statemen
declared

As n
conversio
nectives
results, a

A sp
purpose
buffer. W
to be dis
operation

GLSL Fu

Function
differenc
return ty
Return ty
be recurs
function
function

Nam
tuple decl
laration l

Para
match ex
must hav
not gene
paramete

Fund
times kn
paramete
formal p
paramete
paramete
call is ig
tial conte
inout p
execution

expression whose value can be implicitly converted to integer as conditional expressions). Implicit type conversions are limited to conversions from integer or unsigned integer to float, either as scalars or as vectors. All other type conversions must be explicit; rather than using C-style *type casting*, conversion is requested using C++ constructor syntax.

Control Structures

GLSL provides most of the usual C control structures. Looping constructs include `for`, `while`, and `do-while` loops. Variable can be declared within loops, and the `break` and `continue` statements perform the expected operations. In the original version of GLSL, selection statements were limited to `if-then` and `if-then-else` constructs. GLSL version 1.30 introduced `switch` statements, but `goto` statements and labels are not available. Unlike C and C++, variables cannot be declared inside `if` statements.

As mentioned earlier, conditional expressions must be Booleans; no implicit conversion from numeric types to Boolean types is provided. The Boolean connectives (`&&` and `||`) are *short-circuited*, as in C and C++, and produce Boolean results, as do the relational operators.

A special statement, `discard`, is available for use in fragment shaders. Its purpose is to prevent the fragment shader from making any change to the frame buffer. When a `discard` is executed, the fragment being processed is marked to be discarded. The shader may or may not continue to execute, but whatever operations it performs will have no effect on the frame buffer.

GLSL Functions

Function declarations and calls are much like C++ function calls, with a few differences. Every function must be declared with an explicit return type; the return type `void` is allowed, indicating that the function does not return a value. Return types can be any type, including arrays and structures. Functions cannot be recursive in any form, including indirect recursion (that is, it is illegal for a function to call itself, or for it to call another function that then calls the first function again).

Names of functions can be overloaded based on parameter type; that is, multiple declarations of a function are allowed within a shader, so long as each declaration has the same return type and the parameter lists are all clearly distinct.

Parameter type checking is always performed. All actual parameters must match exactly the type of the corresponding formal parameter. Array parameters must have explicit sizes. A function declaration with an empty parameter list is not generic, as in C, but rather indicates that the function must be called without parameters.

Function parameters in GLSL are passed using *call by value-return* (sometimes known as *call by value-result*). Parameters are *qualified* as `in`, `out`, or `inout` parameters; `in` parameters can be qualified further as `const`, indicating that the formal parameter cannot be modified within the function. For `in` and `inout` parameters, the actual parameter supplied in the call is copied into the formal parameter; in the case of `out` parameters, the actual parameter supplied in the call is ignored (although the formal parameter is readable in the shader, its initial contents are undefined). If no qualifier is used, `in` is assumed. For `out` and `inout` parameters, the last value assigned to the formal parameter during the execution of the function is copied back to the original actual parameter when the

function returns. (These types of actual parameters must not be literals, but must, for obvious reasons, be actual variables.)

Arrays and structures can be passed as parameters to the functions. However, arrays are not passed by reference—instead, the contents of the array is copied into the formal parameter, as with all other parameter types.

As might be expected, a large number of built-in functions are available in GLSL. These range from angle conversions (degrees and radians), trigonometric operations, exponentiation and logarithm functions, and vector and matrix geometric operations.

Communicating with OpenGL

Because the main routine of a shader takes no parameters, communication with the rest of the OpenGL program is achieved by way of global variables. As with function parameters, global variables are typically qualified based on how they are used to convey information to the shader from the OpenGL program or between the vertex and fragment shaders. In the latter case, the same global variable will be declared in both shader sources but may have different qualifiers in the two shaders. Global variable qualifiers are similar to those for function parameters, with a few differences.

The OpenGL program uses uniform global variables to communicate data into all types of shaders. Generally, they contain data that does not change frequently. Shaders can read uniform variables, but cannot write to them.

The `in` qualifier is used in all types of shaders to indicate data that is being given to the shader from previous stages in the pipeline. In a vertex shader, the source is typically the OpenGL program, and the type of the variable is limited to a numeric scalar or vector (Booleans are not allowed) or a matrix.

In a fragment shader, the source of data read from an `in` global variable can be the OpenGL program or the vertex shader; in the latter case, the variable must exactly match an `out`-qualified variable in the vertex shader. Commonly, this data is interpolated—for example, there may be several fragments generated by the pipeline from a set of vertices, and each fragment will be sent through the fragment shader separately, so the contents of the `in` variable may vary between executions of the shader.

Values being produced by any type of shader for use in later stages in the pipeline are defined with the `out` qualifier. Other than the built-in global variables discussed earlier, `out` variables are the only way that results can be sent from the vertex shader to the fragment shader. The same variable must be declared with the same size and type as an `in` variable in the fragment shader.

In GLSL versions prior to 1.30, the `in` and `out` qualifiers did not exist. Global variables holding per-vertex data coming from OpenGL into a vertex shader was marked with the qualifier `attribute`, and output global variables had the qualifier `varying`. In fragment shaders, global variables of any type (coming in from the vertex shader, or going out to later stages in the pipeline) were tagged as `varying`. While `attribute` and `varying` qualifiers are still recognized in version 4.10.6 of GLSL (the current release as this book was written), their use should be limited, as they may be removed from future versions.

Communicating information from an OpenGL program to vertex and fragment shaders through global variables is not quite as simple as we might like. Because these variables are defined in the shader source code, they aren't known when the OpenGL program is compiled, and thus the program cannot access them directly. Instead, the OpenGL program must first request the location of the variable in the current shader program object, and only then can it write data into the global for use by the shader. We request the location of a uniform variable in

this m
GL
1c
where
contai
we ha
follow

GL
GL
gl
gl
If
modif
type a

GL
gl
gl
gl
gl

There
GL

gl
gl
gl
gl

Simila
ger va
the fu
tribl
Th
means
object

22-4

Now t
shader
tively

this manner:

```
GLint location;

location = glGetUniformLocation (program, "variable");
```

where *program* is a program object handle, and *variable* is a null-terminated string containing the name of the uniform global variable that we want to access. Once we have the location, we can retrieve the contents of the variable with one of the following functions:

```
GLint i;
GLfloat f;

glGetUniformiv (program, location, &i );
glGetUniformfv (program, location, &f );
```

If we have the location of a uniform variable in a program object, we can modify its contents. To do this, we must know not only its location, but also its type and the number of elements that it contains:

```
GLfloat v1, v2, v3, v4;

glUniform1f (location, v1);
glUniform2f (location, v1, v2);
glUniform3f (location, v1, v2, v3);
glUniform4f (location, v1, v2, v3, v4);
```

There are also array versions of these routines:

```
GLfloat va[4];

glUniform1fv (location, 1, va);
glUniform2fv (location, 2, va);
glUniform3fv (location, 3, va);
glUniform4fv (location, 4, va);
```

Similarly, we use `glUniform*i` and `glUniform*iv` to write to uniform integer variables. Modification of attribute variables is handled similarly, with the functions `glGetAttribLocation`, `glVertexAttrib1f`, `glVertexAttrib1fv`, and so on.

The `glUniform` functions do not take a program object parameter. This means that they can only write to shader variables found in the active program object (that is, the one selected by the most recent call to `glUseProgram`).

22-4 Shader Effects

Now that we have some understanding of the structure and capabilities of GLSL shaders, it is time to see some examples. Again, note that these examples are relatively simple; showing the full power of shaders is beyond the scope of this text.

A Phong Shader

Recall the Phong illumination model described in Equation 22-1. This can be implemented quite easily in GLSL. For simplicity, we will assume that `GL_LIGHT0` has been enabled as a directional light source in the scene, and that each object has been defined with appropriate material properties.

To implement Phong shading, we need to know where our light source is. Information about active OpenGL lights is available in a built-in global variable named `gl_LightSource`, which is a uniform array with one element per OpenGL light source. Each element of the array is a structure containing a number of fields that describe the light. For our purposes, the most important of these are *ambient*, *diffuse*, *specular*, and *position*. These are all `vec4` fields; the first three contain the ambient, diffuse, and specular characteristics of the light source, and the fourth contains the light's position. The expression

```
gl_LightSource[0].diffuse
```

gives us the diffuse light emitted by `GL_LIGHT0`.

We also need to know what the material properties are for the object being shaded. These are available through a global variable named `gl_FrontMaterial`. This variable is also a structure, with *ambient*, *diffuse*, and *specular* fields containing these characteristics for the object. To compute the interaction of the diffuse light and the surface, we multiply these two fields:

```
gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse
```

There is also a `gl_BackMaterial` global variable which we can use for two-sided lighting.

Our Phong implementation will be a relatively simple pair of shaders. All calculations will be done in the vertex shader, and the fragment shader will only copy the computed color into the `gl_FragColor` variable. We start by computing the ambient light contribution as the product of the object's ambient reflective characteristics and the ambient illumination from the light source:

```
vec4 color;
```

```
color = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
```

To compute the diffuse contribution, we will need to know the surface normal, the direction to the light, and the view direction. To use the dot-product method, all three of these vectors must be normalized. The surface normal is available in the global variable `gl_Normal`; however, like `gl_Vertex`, it is in object coordinates, so we must transform it before we use it. We do this by multiplying it by the global variable `gl_NormalMatrix`, and we normalize the result with the built-in `normalize` function:

```
vec3 normal;
```

```
normal = normalize( gl_NormalMatrix * gl_Normal );
```

We next need to normalize the direction to the light. Our light is directional, which means that its position in OpenGL is actually the direction the light is

shining.
malize t

vec

lig

If the li
between
Con
are nor
clamp th

flo

Ndo

We
add it to

col

If th
This rec
vector. I
eye pos
so we c
vertex t
the refle
pointing
have the
specular

if(

(

)

shining. We can take the position, convert it to a three-element vector, and normalize the result:

```
vec3 lightdir;

lightdir = normalize( vec3( gl_LightSource[0].position ) );
```

If the light was positional, we could compute its direction as the difference between the vertex position and the light position.

Computing the cosine of the angle between these vectors is easy because they are normalized vectors. To ensure that we don't get a negative cosine, we will clamp the result to 0.0:

```
float NdotL;

NdotL = max( dot(normal, lightdir), 0.0 );
```

We now have enough information to compute the diffuse contribution and add it to the final color:

```
color += NdotL *
    (gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse);
```

If the cosine value is positive, we also want to include the specular highlight. This requires that we compute the view vector and the reflection of the light vector. In eye coordinates, the view vector can be computed by subtracting the eye position from the vertex position. However, the eye position is the origin, so we can just use the vertex position and negate it to get the vector from the vertex to the eye position. We can use the built-in `reflect` function to compute the reflection of the light vector around the surface normal; our light vector is pointing from the light to the vertex, though, so we must negate it. Once we have those, we can compute their dot-product (clamping it to 0.0), calculate the specular contribution, and add it to the computed color:

```
if( NdotL > 0.0 )
{
    vec3 view, reflection;
    float RdotV;

    view = vec3( -normalize(gl_ModelViewMatrix * gl_Vertex) );
    reflection = normalize( reflect(-lightdir, normal) );
    RdotV = max( dot(reflection, view), 0.0 );

    color += gl_FrontMaterial.specular *
        gl_LightSource[0].specular *
        pow( RdotV, gl_FrontMaterial.shininess );
}
```

Finally, we must assign the computed color to the global `gl_FragColor` variable and transform the vertex. Here is the complete vertex shader:

```
// Phong vertex shader

void main() {
    vec3 normal, lightdir;
    vec4 color;
    float NdotL;

    color = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;

    normal = normalize(gl_NormalMatrix * gl_Normal);
    lightdir = normalize( vec3(gl_LightSource[0].position) );
    NdotL = max( dot(normal, lightdir), 0.0 );

    color += NdotL *
        (gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse);

    if( NdotL > 0.0 )
    {
        vec3 view, reflection;
        float RdotV;

        view = vec3( -normalize(gl_ModelViewMatrix * gl_Vertex) );
        reflection = normalize( reflect(-lightdir, normal) );
        RdotV = max( dot( reflection, view ), 0.0 );

        color += gl_FrontMaterial.specular *
            gl_LightSource[0].specular *
            pow( RdotV, gl_FrontMaterial.shininess );
    }

    gl_FrontColor = color;
    gl_Position = ftransform();
}
```

Because we performed all the color calculations in the vertex shader, the fragment shader is very simple:

```
// Phong fragment shader

void main()
{
    gl_FragColor = gl_Color;
}
```

Color Plate 32 shows a scene containing three `gluSpheres`, illuminated by a single directional light, drawn using this shader pair.

All the computation in our example was done in the vertex shader, and the fragment shader simply used that result. We could have done the color calculations in the fragment shader, but we would still need to do the normal, light direction, and view vector calculations in the vertex shader because of the need to access the per-vertex variables. The resulting vectors would be communicated to the fragment shader through global variables.

Texture Mapping

Texture mapping is another operation that is relatively easy to implement using shaders. It can be implemented by directly mapping each location on the surface of an object to a point within the texture, or by modifying the Phong shader shown earlier to take color information from the texture image rather than from the object's material properties.

We first set up the texture within our OpenGL program as discussed in Chapter 18, by creating a texture object with `glGenTextures`, binding it with `glBindTexture`, setting our desired texture parameters with calls to `glTexParameter`, and then defining the texture itself with `glTexImage`. To use the texture within a shader, however, two additional steps are required. We must tell the shader where to find the texture, and the shader itself must gain access to the texture data.

In Chapter 18, our discussion of surface texture mapping in OpenGL assumed that we could apply only a single texture at a time to the surface of an object. This was, in fact, a simplification of OpenGL's texture-mapping capabilities. OpenGL actually supports *multitexturing*—that is, the ability to apply more than one texture to the surface of an object. It does this through the use of *texture units*. The number of texture units is implementation-dependent; the following example code queries the OpenGL state to determine the number of texture units in this implementation:

```
GLint units;

glGetIntegerv (GL_MAX_TEXTURE_UNITS, &units);
```

When we define a texture, it is defined within the *active texture unit*. All texture parameter settings and image data are assigned to that unit. The active unit is selected with a call to `glActiveTexture`, as follows:

```
glActiveTexture (GL_TEXTURE0);
```

This selects texture unit 0 as the active unit. (This is actually the default texture unit, so this function call is unnecessary unless we have selected a different unit and want to switch back to unit 0.)

After we bind our texture object to the texture unit, we must tell our shader which texture unit we are using. We do this by writing into a global *sampler* variable in the shader. Samplers are special types of data items in GLSL that have access to all the texture information in a texture unit. The shader code uses a sampler to identify the texture unit to be accessed, but the sampler itself is *opaque* to the shader. It cannot be directly read or written by GLSL code—it can only be passed as a parameter to a texture access function within our shader.

Samplers come in many forms. We create samplers for one-, two-, and three-dimensional floating-point textures with the types `sampler1D`, `sampler2D`, and `sampler3D`. Samplers can also be created for integer or unsigned integer textures, cube-map textures, shadow map textures, and other variations. To create a

sampler for a basic two-dimensional texture, for example, we would use a declaration like this one in our fragment shader:

```
uniform sampler2D textureID;
```

In the OpenGL program, we assign the texture unit sequence number to the sampler. If our active shader program is `texShader` and we want to use texture unit 0, we find the location of the sampler variable and assign the sequence number to it:

```
GLint texloc;

texloc = glGetUniformLocation (texShader, "textureID");
glUniform1i (texloc, 0);
```

Note that we assign the texture unit sequence number (0), not the OpenGL symbolic constant (`GL_TEXTURE0`), to the sampler variable.

A shader program that maps a two-dimensional texture directly to the surface of an object is very straightforward. Generally, the vertex shader takes care of setting up all necessary texture coordinates, and the fragment shader accesses the texture and uses it to determine the color of the fragment. The texture coordinates for texture unit 0 that correspond to the current vertex are available to the vertex shader in a global variable named `gl_MultiTexCoord0`. These coordinates must be communicated to the rest of the pipeline for interpolation; this is achieved by assigning them to the first slot in a global array of vectors named `gl_TexCoord`.

Here is a simple vertex shader that copies the existing texture coordinates for interpolation:

```
void main()
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = ftransform();
}
```

The fragment shader must use the interpolated coordinates to access the texture image and determine the fragment color accordingly. Because the sampler variable is opaque, the shader must use built-in functions to access the texture data. The built-in function `texture2D` takes a sampler variable and a coordinate position as its parameters, and returns the texture data as a `vec4` value. Here is a simple fragment shader that uses the texture data directly as the fragment color:

```
uniform sampler2D textureID;

void main ()
{
    vec4 color = texture2D(textureID, gl_TexCoord[0].st);
    gl_FragColor = color;
}
```

The
face o
it to t
Plate
sphere
The
the te
result
color
object.

Bump

Another
an obje
a funct
then ap
mappin
but is s
pipelin

To
mal at
each fr
and ho
applic
face, an
the ima

On
differen
surface
we mus
this iss
the par
vector a
third ve
perform

The
the tang
compor

where (
(N_x, N_y, N_z)
matrix t
tem arou
change f
Our
relief ma

The result of using this shader pair to map a texture image of the surface of the Earth to a square polygon is shown in Color Plate 33. We can use it to texture-map any object for which texture coordinates are defined. Color Plate 34 shows an application of the same texture image to a GLU quadric sphere.

This example texture-mapping shader is very simplistic because it uses the texture color information directly as the fragment color. A more realistic result could be obtained, for instance, by modifying a Phong shader to use color information from a texture image instead of the material properties of the object.

Bump Mapping

Another application of texture mapping is the simulation of surface roughness on an object. The technique known as **bump mapping** (discussed in Chapter 18) uses a function to perturb the normal vector at a point on the surface of an object, and then applies a standard illumination model to calculate color at that point. Bump mapping is relatively easy to implement in an interactive program using shaders, but is significantly more difficult to implement using the original fixed-function pipeline.

To bump-map an image, we must decide how far to perturb the surface normal at each point on the object. We can do this computationally as we process each fragment, or we can precompute the changes to be applied at each point and hold them in a special type of texture called a **normal map**. If we are also applying a texture to the surface of the object as well as bump-mapping the surface, an obvious source of bump-map information is the color variation within the image.

One complication in bump mapping is the fact that we must work with several different coordinate spaces. The incoming information that we use to compute surface colors is typically in either object coordinates or eye coordinates; however, we must do our displacement calculations in texture space. Typically, we resolve this issue by converting everything into texture space. To do this, we compute the partial derivative vector P'_u discussed in Section 18-3. We then normalize this vector and the surface normal and take their cross-product, which produces a third vector that is orthogonal to the first two. These three vectors are used to perform the transformation.

The normalized P'_u vector is called the **tangent vector**. The cross-product of the tangent vector and the surface normal is called the **binormal vector**. From the components of these vectors, we build the transformation matrix

$$\mathbf{M} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix} \quad (22-3)$$

where (T_x, T_y, T_z) is the tangent vector, (B_x, B_y, B_z) is the binormal vector, and (N_x, N_y, N_z) is the normal vector. Multiplying an object-space vector by this matrix transforms it into **tangent space**. Tangent space is a local coordinate system around the point being shaded, and the tangent and binormal vectors may change from point to point across the surface.

Our bump-mapping shader will be a simplified form of a technique known as **relief mapping**. We will use the texture image to determine the surface color at the

shading point, as in our earlier texture-mapping example. However, we will also use it to calculate the displacements to be applied to our surface normals—that is, the apparent roughness of the surface will be determined by the color variations in our texture image.

To implement bump mapping as a shader pair, we must divide the work between the vertex and fragment shaders. In addition to the transformations we have seen in earlier examples, the vertex shader will compute the light and view vectors and will transform them into tangent space for use by the fragment shader. The fragment shader, in turn, will use the color information from the texture image to calculate the height variation for this fragment, and it also will perform a simple diffuse shading calculation using the texture color information.

To compute the transformation matrix in our vertex shader, we need the tangent and binormal vectors in addition to the surface normal. We are given the surface normal; we can either compute a tangent vector from it or use one supplied by the OpenGL program. We can calculate the tangent vector fairly easily by computing the cross-products between the surface normal and the *y* and *z* axes, and then selecting the longer of the two cross-products and normalizing it.

If we choose to have the OpenGL program supply the tangent vector, it must be written into a global attribute variable used by the vertex shader. We obtain its location using `glGetAttribLocation` and write three values into it as follows:

```
GLfloat tangVector[3];
GLint tangentLoc;

tangentLoc = glGetAttribLocation (bumpshader, "tangent");
glVertexAttrib3fv (tangentLoc, tangVector);
```

In the vertex shader, `tangent` is declared globally as an attribute variable. We must also declare the view and light vector variables as varying variables:

```
attribute vec3 tangent;
varying vec3 light, view;
```

In the vertex shader, we transform the surface normal, and compute the binormal vector. Once we have these three vectors, we can compute the view and light vectors, and transform them into tangent space. A fast way to perform the transformation is to take advantage of the fact that we can compute the three result values from multiplying the transformation matrix by a vector using the built-in dot product function. For example, given the transformed light vector and the three tangent-space vectors, we can transform the light vector into tangent space as follows:

```
vec3 tmp;
tmp.x = dot( light, tangent );
tmp.y = dot( light, binorm );
tmp.z = dot( light, normal );
light = tmp;
```

locat
com

Th
fragme
along
To
of the
point
of a 50
functio

fl
(

)

Cre
by first
position
coordina
centered
consisti
vertex.
the cross
vectors.
The
normal

We complete the vertex shader by performing the usual copying of the texture location and transformation of the vertex position into clip space. Here is the completed vertex shader:

```

varying vec3 light, view;
attribute vec3 tangent;

void main()
{
    vec3 normal = vec3( normalize( gl_NormalMatrix * gl_Normal ) );
    vec3 binorm = normalize( cross( normal, tangent ) );

    view = -normalize( vec3( gl_ModelViewMatrix * gl_Vertex ) );
    light = normalize( vec3( gl_LightSource[0].position ) );

    vec3 tmp;
    tmp.x = dot( light, tangent );
    tmp.y = dot( light, binorm );
    tmp.z = dot( light, normal );
    light = tmp;

    tmp.x = dot( view, tangent );
    tmp.y = dot( view, binorm );
    tmp.z = dot( view, normal );
    view = tmp;

    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = ftransform();
}

```

The bump-mapping fragment shader is more complicated than our previous fragment shaders. Globally, we need to define the light and view vector variables, along with our texture sampler variable.

To compute the height offset from a color value, we will compute the average of the red and green components and then smooth out variations from point to point by taking 1.5 percent of the color average and adding that to 98.5 percent of a 50 percent gray value. We compute the blended value using the built-in `mix` function:

```

float height( vec3 color )
{
    float avg = (color.r + color.g) / 2.0;
    return mix( avg, 0.5, 0.985 );
}

```

Creating the perturbed surface normal at a point on the texture is achieved by first creating a small triangle around the point on the surface. We calculate the positions of the triangle's vertices by adding three different offsets to the texture coordinate to locate three points at 0°, 120°, and 240° around an imaginary circle centered on the coordinate point. For each of these vertices, we create a vector consisting of the *s* and *t* offsets and a height offset calculated from the color at the vertex. We then create the normal vector from these three vectors by computing the cross-product of vectors formed by taking the difference between pairs of the vectors.

The rest of our fragment shader is straightforward. We compute the modified normal vector for the texture coordinate, and then compute ambient and diffuse

color contributions based on the texture color and the modified normal. Here is the completed fragment shader:

```

varying vec3 light, view;
uniform sampler2D textureID;

// Calculate height offset
float height( vec3 color ) {
    float avg = (color.r + color.g)/2.0;
    return mix( avg, .5, .985 );
}

// Create modified surface normal
vec3 modNormal( vec2 point ) {

    // Create the small triangle - first, the s and t
    // distances from the center point
    vec2 d0 = vec2( 0, 0.001 );
    vec2 d1 = vec2( -0.000866, -0.0005 );
    vec2 d2 = vec2( 0.000866, -0.0005 );

    // Calculate the triangle vertex positions
    vec2 p0 = point + d0;
    vec2 p1 = point + d1;
    vec2 p2 = point + d2;

    // Compute the height offset for each vertex
    float h0 = height( vec3( texture2D( textureID, p0 ) ) );
    float h1 = height( vec3( texture2D( textureID, p1 ) ) );
    float h2 = height( vec3( texture2D( textureID, p2 ) ) );

    // Create the three vectors
    vec3 v0 = vec3( d0, h0 );
    vec3 v1 = vec3( d1, h1 );
    vec3 v2 = vec3( d2, h2 );

    // Compute the modified normal vector
    return normalize( vec3( cross( v1-v0, v2-v0 ) ) );
}

void main() {
    vec4 base = texture2D( textureID, gl_TexCoord[0].st );
    vec3 bump = modNormal( gl_TexCoord[0].st );
    vec4 color = gl_LightSource[0].ambient * base;

    float NdotL = max( dot(bump, light), 0.0 );
    color += NdotL * ( gl_LightSource[0].diffuse * base );

    gl_FragColor = color;
}

```

Color Plate 35 shows the results of using this shader pair to apply the Earth surface texture image used in previous examples to a square polygon. Compare

this to
Plate 3

22-5

Compu
graphic
work d
graphic
of grap
original

As
became
capabil
oped, w
of differ
ing lang
operatio

The
ing prog
program
tion of v
through
perform
using a
used to

T A B

Summary

F

glCreat
glShade
glCompil
glGetSha
glGetSha
glCreate
glAttach
glGetPro
glGetPro
glUsePro
glGetUni
glGetUni
glUnifor
glGetAtt
glGetAtt
glVertex

this to the direct application of the texture image to the polygon shown in Color Plate 33 to see the effect of the bump mapping calculation.

22-5 Summary

Computer graphics libraries have evolved over time to match the capabilities of graphics hardware. In the beginning, graphics programmers were required to work directly with the hardware available to them. Libraries of commonly used graphics routines were developed in an attempt to standardize the development of graphics programs, culminating in APIs such as the OpenGL library and its original fixed-function internal pipeline.

As graphics hardware continued to evolve, the fixed-function pipeline became more limiting because it could not take advantage of improved hardware capabilities. To solve this problem, a programmable pipeline model was developed, which allowed graphics programmers more control over the functionality of different stages in the pipeline through the use of programmable shaders. Shading languages were created to simplify the task of performing common shading operations.

The OpenGL Shading Language (GLSL) was developed as a way of integrating programmable shading operations into the OpenGL pipeline. GLSL provides programmable "hooks" into the pipeline at critical stages, allowing the manipulation of vertices, object geometries, surface tessellation, and fragment manipulation through the use of shader programs. Given the flexibility of GLSL, it is possible to perform easily shading tasks that would be difficult or impossible to accomplish using a fixed-function graphics pipeline. Table 22-1 lists the OpenGL functions used to create and communicate with GLSL shader programs.

TABLE 22-1

Summary of OpenGL GLSL-Related Functions

Function	Description
<code>glCreateShader</code>	Creates a shader object.
<code>glShaderSource</code>	Attaches shader source code to a shader object.
<code>glCompileShader</code>	Compiles shader source code.
<code>glGetShaderiv</code>	Queries shader object state.
<code>glGetShaderInfoLog</code>	Retrieves shader object messages.
<code>glCreateProgram</code>	Creates a shader program object.
<code>glAttachShader</code>	Attaches a shader object to a shader program object.
<code>glGetProgramiv</code>	Queries shader program object state.
<code>glGetProgramInfoLog</code>	Retrieves shader program messages.
<code>glUseProgram</code>	Activates a shader program.
<code>glGetUniformLocation</code>	Obtains the location of a global shader uniform variable.
<code>glGetUniform*</code>	Reads the contents of a global shader uniform variable.
<code>glUniform*</code>	Writes the contents of a global shader uniform variable.
<code>glGetAttribLocation</code>	Obtains the location of a vertex shader attribute variable.
<code>glGetAttrib*</code>	Reads the contents of a vertex shader attribute variable.
<code>glVertexAttrib*</code>	Writes the contents of a vertex shader attribute variable.

REFERENCES

Shade trees are discussed in Cook (1984). Ken Perlin's PSE is described in Perlin (1985), and his original noise implementation can be found on his website, at <http://cs.nyu.edu/~perlin/>. RenderMan is presented in Upstill (1989) and Apodaca and Gritz (2000). The official RISpec can be found online at <https://renderman.pixar.com/products/rispec/index.htm>, and a number of RenderMan implementations (both commercial and open-source) can be found on the Internet. Relief texture mapping is discussed in Oliveira, Bishop and McAllister (2000) and in Policarpo, Oliveira, and Comba (2005).

Official specifications for GLSL can be found at <http://www.opengl.org/>, along with sample programs and tutorial guides. GLSL is also discussed in Shreiner (2010). Finally, more complete treatments of GLSL can be found in Rost and Licea-Kane (2010) and in Bailey and Cunningham (2009).

EXERCISES

- 22-1 Determine whether or not your OpenGL installation supports GLSL. If it does, determine the version of GLSL it supports.
- 22-2 Write a function which takes two null-terminated strings as its parameters and returns the `GLuint` identifier for a shader program object. The parameters contain the names of vertex and fragment shader source files.
- 22-3 Write a program using the functions in the previous exercise to draw a square in the center of the display window and use the vertex shader program to color the square red.
- 22-4 The example Phong shader performs its color calculations at each vertex. Convert it into a shader that does the color calculations in the fragment shader. Remember that some critical values must be computed in the vertex shader.
- 22-5 Write a program to display an origin-centered tetrahedron on a black background using the shader you wrote in the previous exercise to shade the object. Add the ability to rotate the object around the *y*-axis using keyboard input.
- 22-6 Modify the example Phong shader to use a light source specified through a global shader variable rather than just using light source 0.
- 22-7 Modify the example Phong shader to work with multiple light sources. Use a global shader variable to tell the shader how many light sources are active.
- 22-8 Modify the program in Exercise 22-5 to add two more light sources to the scene. Use the shader you developed in the previous exercise to shade

the object in the scene. The positions and orientations of the light sources should be taken as input parameters to the program.

- 22-9 Modify the simple texture mapping shader so that it performs Phong calculations using color information from the texture image instead of the material properties of the object being shaded.
- 22-10 Write program to display an origin-centered cube on a black background using the shader you wrote in the previous exercise to shade the object with a textured image on each of the faces of the cube. Provide the ability to rotate the cube about the *y* and *z* axes using keyboard input.
- 22-11 Modify the program and shader used in the previous exercise to add two more light sources to the scene and have the shader texture the object using lighting information from all three lights. The positions and orientations of the light sources should be taken as input parameters to the program.

IN MORE DEPTH

- 22-1 By modifying the examples in the chapter, write a shader program to apply the texture patterns you developed in Chapter 18 to the objects in your scene. If you wrote a bump map for any of those objects, modify the example in the chapter to produce a shader program for that as well. Replace the existing texture and bump maps with the shader programs and note the visual differences between the two approaches to texture mapping, if any.
- 22-2 Consider the pattern created by four ceramic tiles arranged in a 2×2 pattern on a floor or wall. Each tile has a width and height, and the grout line between adjacent tiles also has a width. The total width of this pattern is $2 \times \text{tileWidth} + 2 \times \text{groutWidth}$, and similarly the total height is $2 \times \text{tileHeight} + 2 \times \text{groutWidth}$. We can apply this pattern to a surface as a procedural texture map without using an actual texture image fairly easily. The *s* texture coordinate specifies a position between the left edge ($s = 0$) and the right edge ($s = 1$); similarly, the *t* coordinate specifies a position between the bottom edge ($t = 0$) and the top edge ($t = 1$). Because we know the width and height of each tile and the width of the grout line, we can use the *s* and *t* coordinates to determine whether this point on the "texture" is covered by tile or grout. Create a shader that applies this type of procedural texture to an object. Communicate the width, height, and color of a tile and the width and color of the grout line to your shader through global uniform variables.