

CHAPTER 20

Programmable Pipelines

Programmers mutiny! We're going to throw off our shackles and take over the engine room!

The OpenGL Shading Language (abbreviated as GLSL, also called GLSLang) was developed originally by the OpenGL ARB (Architecture Review Board) to give programmers direct control over parts of the graphics processing pipeline. Included in the core of OpenGL 2.0, released in 2004, it represented the first major upgrade of OpenGL since its creation in 1992. The way the GLSL operates is to allow the programmer to write programs, called *shaders*, to supplant parts of the graphics pipeline formerly of fixed-functionality.

Historically, the GLSL evolved as a response to the increasing capabilities of graphics cards (also called graphics processing units, or GPUs) and the need to expose these capabilities to the application programmer. Before the standardization of the GLSL, a programmer had to write code in hardware-specific assembly language to access individual GPU features – a difficult and inefficient task at best. Just as high-level programming languages like C evolved from assembly in order to hide low-level and hardware-specific calls from the developer and give her a structured and stable environment, so did the GLSL.

As a language, the GLSL itself is based on C, so coding will not be a problem for us. In addition to most of C's functionality, the GLSL necessarily has dedicated functions and variables for the shaders to interact with each other, as well as with the main OpenGL program to which they are attached (and, thereby, with the rest of the pipeline).

The goal of this chapter is not an extensive coverage of the GLSL, but a thorough introduction. We begin in Section 20.1 with an overview of the programmable pipeline, learn how to attach shaders to an OpenGL program and run over the data types of the GLSL. Section 20.2 describes

how a program communicates with its shaders, shipping them data from the fixed-functionality parts of the pipeline, and how the shaders communicate between themselves. The communication interface between these entities is implemented by means of specially qualified variables. It is precisely because shaders are programmable and at the same time have access to almost all hitherto opaque data in the pipeline that the GLSL is so powerful.

Per-pixel lighting – versus the per-vertex lighting of the first-generation fixed pipeline – is a popular practical application of the GLSL that we'll see in Section 20.3. In Section 20.4 we learn how to import and manipulate textures in the programmable pipeline. We conclude in Section 20.5.

20.1 GLSL Basics

Shaders are the programs written by the user to replace fixed-functionality. One each of two kinds of shaders can be attached to an OpenGL program – *vertex shaders* and *fragment shaders*. Both kinds are written in the same C-like GLSL but target different parts of the graphics pipeline. The vertex shader operates on incoming vertex data (including coordinates, normal values, colors, etc.) before handing over its output for clipping and then rasterization. The fragment shader, on the other hand, operates on fragments in the raster before passing them on to the bottom stage of the classical pipeline, which remains intact, to perform its own per-fragment operations such as the scissor, alpha, stencil and depth tests, as well as blending. Figure 20.1 is a simplified diagram. Observe that texturing is within the purview of the fragment shader, which is justified technically as it is a process of combining texels with fragments in the raster.

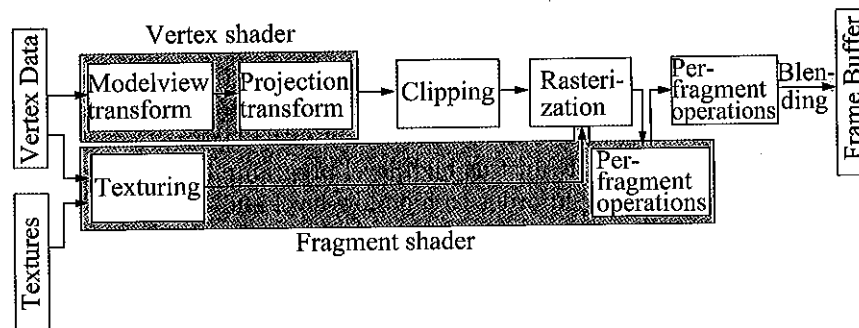


Figure 20.1: GLSL pipeline simplified: shaded regions indicate parts of the classical pipeline now programmable.

A vertex shader operates on each vertex coming down the pipeline, while a fragment shader on each fragment in the raster. Once attached, shaders do have minimum responsibilities that they have to discharge and that cannot be left to fixed parts of the pipeline. The vertex shader, for example, must

output at le
to operate o
to fixed-fun
functionality
Likewise, th
decide to di

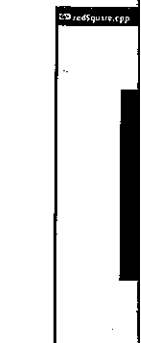
20.1.1

Time now t
code. Our g
square. cpp

Experiment Square.

Note: For
Appendix B
the GLSL su
sure, when
two shader

Now, re
amount of
passThrough
a red square



Figure

Before w
program itse
new lines of

output at least the vertex's coordinates in world space. It cannot, say, choose to operate only on a vertex's normal values, leaving coordinate computation to fixed-functionality (though, of course, it is free to just mimic the fixed-functionality computation of vertex coordinates and not do anything more). Likewise, the fragment shader must at least assign each fragment a color or decide to discard it from the pipeline altogether.

20.1.1 Attaching Shaders

Time now to go see what we have been talking about. In other words, let's code. Our guinea pig will be our trusty workhorse from way back when – `square.cpp`.

Experiment 20.1. Fire up `redSquare.cpp` in the folder `Code/GLSL/Red-Square`.

Note: For how to set up the environment to run GLSL programs see Appendix B. Each of our GLSL programs is in a similarly named folder in the GLSL subdirectory of `Code`, with two accompanying shader files. Make sure, when running a GLSL program, to keep it in the same directory as its two shader files.

Now, `redSquare.cpp` is *exactly* `square.cpp` with the *barest* minimum amount of code added to be able to attach a vertex shader, called `passThrough.vs`, and a fragment shader, called `red.fs`. The output is a red square in the OpenGL window, as in Figure 20.2(a). **End**

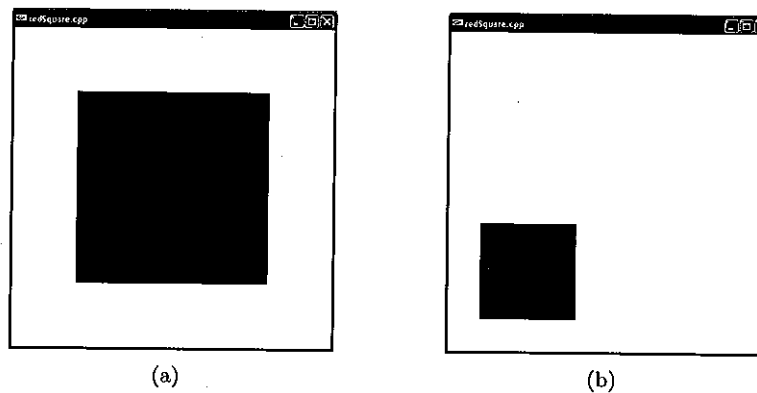


Figure 20.2: Screenshots: (a) `redSquare.cpp` (b) Experiment 20.2.

Before we get to the shaders, let's see what, in fact, has been added to the program itself to "GLSLify" it. Comments starting with "NEW" indicate new lines of code.

Linking in the OpenGL Extension Wrangler Library (GLEW), as we have done with directives at the top of the program, is advisable for the run-time support of OpenGL extensions that GLEW provides.

The integer globals `programHandle`, `vertexShaderHandle` and `fragmentShaderHandle` are to store references to program, vertex shader and fragment shader objects, respectively. The routine `readShader()` is a generic routine to read external text files, and not GLSL-specific.

The `setShaders()` routine is the critical one that initializes the two shaders. Let's go through it line by line. The commands

```
char* vertexShader = readShader(vertexShaderFile);  
char* fragmentShader = readShader(fragmentShaderFile);
```

read the two shader source files, which are simply text files, and store them internally as character strings. Next,

```
programHandle = glCreateProgram();  
vertexShaderHandle = glCreateShader(GL_VERTEX_SHADER);  
fragmentShaderHandle = glCreateShader(GL_FRAGMENT_SHADER);
```

create an empty program object and two empty shader objects, returning references to them. The pair of statements

```
glShaderSource(vertexShaderHandle, 1,  
               (const char**) &vertexShader, NULL);  
glShaderSource(fragmentShaderHandle, 1,  
               (const char**) &fragmentShader, NULL);
```

attaches the vertex and fragment shader sources to the respective shader objects. The two statement pairs

```
glCompileShader(vertexShaderHandle);  
glCompileShader(fragmentShaderHandle);  
  
glAttachShader(programHandle, vertexShaderHandle);  
glAttachShader(programHandle, fragmentShaderHandle);
```

compile the vertex and fragment shader source code strings and attach the resulting shader objects to the program object. Finally,

```
glLinkProgram(programHandle);  
glUseProgram(programHandle);
```

links the program and creates an executable which is installed into the current rendering state.

The `main()` routine actually sets the programmable pipeline into motion by initializing GLEW and invoking the shaders. That's it. That is all the additional overhead to attaching shaders – essentially, the `setShaders()` routine, which is the same across all our GLSL programs.

Minimal Shaders

Now let's see what happens. Apparently not much.

Note: Our vertex shader uses the extension `GL_EXT_shader_text_files`, as text files by the way.

The only operation

```
gl_Position =
```

in `passThrough.v` pipeline, its coordinate matrices and the `modelView` the fixed pipeline called a *pass-through*.

Note: The name of the program state set is `glVertex*()`.

As indicated by `gl_ModelViewPro`

```
gl_ProjectionM
```

and can replace it to multiply a vertex by the `modelView` is to use the special

```
gl_Position =
```

which guarantees

The fragment shader fragment colors

```
gl_FragColor
```

20.1.2 Data

It seems, even if the `modelView` is attached to `redS` (e.g., `vec4`) to `gl_ProjectionM`.

To begin with

```
float
```

to declare a single. In addition, the `modelView` vectors, respectively.

Minimal Shaders

Now let's see what the shaders associated with `redSquare.cpp` do. Apparently not much. The vertex shader is `passThrough.vs`.

Note: Our vertex shaders all have the extension `.vs`, while fragment shaders the extension `.fs`, but this is not a standard, and they are treated simply as text files by the programming environment.

The only operational statement

```
gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex
```

in `passThrough.vs` is fairly intuitive. As each vertex comes down the pipeline, its coordinate vector is multiplied by the modelview and projection matrices and the result returned in a position vector. This is precisely what the fixed pipeline does and, as this particular shader does no more, it is called a *pass-through* vertex shader.

Note: The naming convention is `gl_state` for a variable containing the program state set by `glstate*`, e.g., `gl_Vertex` for the variable set by `glVertex*()`.

As indicated by comments in `passThrough.vs`, the *derived* matrix `gl_ModelViewProjectionMatrix` is equal to the product

```
gl_ProjectionMatrix * gl_ModelViewMatrix
```

and can replace it. An even more optimized option, if it is desired only to multiply a vertex's coordinates by the projection and modelview matrices, is to use the special command

```
gl_Position = ftransform()
```

which guarantees the same result as fixed-functionality.

The fragment shader `red.fs` is equally simple-minded, setting all fragment colors to red with the single statement

```
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0)
```

20.1.2 Data Types

It seems, even from the very concise shaders `passThrough.vs` and `red.fs` attached to `redSquare.cpp`, that the GLSL has vector and matrix data types (e.g., `vec4`) to manipulate state variables such as `gl_Vertex`, `gl_Position`, `gl_ProjectionMatrix` and such. This is correct.

To begin with, the GLSL has the classical C types

```
float      int      bool
```

to declare a single floating point, integer and Boolean quantity, respectively. In addition, the GLSL can declare 2-, 3- and 4-component floating point vectors, respectively, with

vec2 vec3 vec4

Likewise, integer vectors are declared with

ivec2 ivec3 ivec4

and Boolean vectors with

bvec2 bvec3 bvec4

Matrices, which are always floating point and square, of sizes 2×2 , 3×3 or 4×4 are declared, respectively, with

mat2 mat3 mat4

The final set of special non-C data types, called *samplers*, are handles to access textures:

sampler1D sampler2D sampler3D
samplerCube sampler1DShadow sampler2DShadow

The first three declare handles to 1D, 2D and 3D textures, respectively; the fourth a handle to a cube-mapped texture; the fifth and sixth handles to 1D and 2D depth textures, respectively, with comparison.

Finally, the complex data types structures (*struct*) and arrays (*[]*), functioning just as in C, are available as well.

20.1.3 Swizzling

Components of a vector can be selected and even rearranged and duplicated using the *swizzle* operator *."*. Vector components are accessed using the following three sets of names:

x, y, z, w r, g, b, a s, t, p, q

For example, *x*, *r* and *s* each denotes the first component; *y*, *g* and *t* each the second component; and so on. The sets cannot be mixed though. The following snippet illustrates how the swizzle operator behaves on the right-hand side – read the comments:

```
vec4 pos1 = vec4(1.0, 2.0, 3.0, 4.0);
vec4 pos2 = pos1.yxzw; // Rearrangement: pos2 = (2.0, 1.0, 3.0, 4.0)
vec4 pos3 = pos1.rrba; // Duplication: pos3 = (1.0, 1.0, 3.0, 4.0)
vec4 pos4 = vec4(pos1.xyz, 5.0); // pos4 = (1.0, 2.0, 3.0, 5.0).
vec2 pos5 = pos1.xy; // pos5 = (1.0, 2.0).
vec4 pos6 = pos1.xgga; // Illegal: mixing names from different sets.
```

On the left-hand side, though, the swizzle operator does not accept repeated components:

```
vec4 pos1 = vec4(1.0, 2.0, 3.0, 4.0);
pos1.xy = vec2(5.0, 6.0); // pos1.xy = (5.0, 6.0)
pos1.yx = vec2(5.0, 6.0); // pos1.yx = (5.0, 6.0)
pos1.xx = vec2(5.0, 6.0); // pos1.xx = (5.0, 6.0)
```

Here's a simple application of *swizzle*

Experiment 20.2. Replace the *swizzle* with

```
void main()
{
    vec4 scaledPos = vec4(0.5 * pos1.x, 0.5 * pos1.y, 0.5 * pos1.z, 0.5 * pos1.w);
    gl_Position = gl_ModelViewProjMatrix * scaledPos;
}
```

As expected, the *xy*-values of Figure 20.2(b) for a screenshot.

Exercise 20.1. (Program 20.1) Replace the first line of the new shader is

```
vec4 scaledPos = 0.5 * pos1;
instead of
vec4 scaledPos = vec4(pos1.x, pos1.y, pos1.z, pos1.w);
what happens? Why?
```

20.2 Communicating with the GPU

Even the simple shaders require some amount of rudimentary vertex shader programming. The current modelview matrix, of course, to the incoming *gl_Vertex*. It outputs the *gl_Position*. The fragment shader defines the *gl_FragColor*.

20.2.1 Overview

A very simple overview of the program has – that the shaders only (like *gl_Vertex*) incoming data and

```
vec4 pos1 = vec4(1.0, 2.0, 3.0, 4.0);
pos1.xy = vec2(5.0, 6.0); // pos1 = (5.0, 6.0, 3.0, 4.0).
pos1.yx = vec2(5.0, 6.0); // pos1 = (6.0, 5.0, 3.0, 4.0).
pos1.xx = vec2(5.0, 6.0); // Illegal - x is repeated.
```

Here's a simple application of swizzling in a vertex shader.

Experiment 20.2. Replace the vertex shader code for `redSquare.cpp` with

```
void main()
{
    vec4 scaledPos = vec4(0.5 * gl_Vertex.xy, 0.0, 1.0);
    gl_Position = gl_ModelViewProjectionMatrix * scaledPos;
}
```

As expected, the *xy*-values of the square's vertices are both halved. See Figure 20.2(b) for a screenshot. **End**

Exercise 20.1. (Programming) In the preceding experiment, if the first line of the new shader is simply

```
vec4 scaledPos = 0.5 * gl_Vertex;
```

instead of

```
vec4 scaledPos = vec4(0.5 * gl_Vertex.xy, 0.0, 1.0);
```

what happens? Why?

20.2 Communication

Even the simple shaders attached to `redSquare.cpp` are evidently doing some amount of rudimentary communication with the program. The vertex shader `passThrough.vs` accesses the program's state – particularly, the current modelview and projection matrices through the variables `gl_ModelViewMatrix` and `gl_ProjectionMatrix`, respectively – in addition, of course, to the incoming vertex's coordinates contained in the variable `gl_Vertex`. It outputs the vertex's transformed coordinates into the variable `gl_Position`. The fragment shader `red.fs`, too, outputs to a variable `gl_FragColor` defining the current fragment's color.

20.2.1 Overview

A very simple overview of the GLSL's communication scheme is as follows.

The program has so-called *built-in* variables – all prefixed with “gl_” – that the shaders can access. Built-in variables are of two kinds: read-only (like `gl_Vertex`, `gl_ModelViewMatrix`, etc.) for the shaders to access incoming data and the program's state, and writable (like `gl_Position`,

`gl_FragColor`, etc.) for the shaders to output the result of their computation for use by the program (in other words, back to the pipeline).

Shaders, too, have built-in variables for communication between one another. In addition to the built-in variables, the programmer can define variables herself, called *user-defined* obviously, for the program to communicate with the shaders, as well as for inter-shader communication.

Before we write programs with our own user-defined variables, here's another with only built-in variables, but with somewhat more going on than `redSquare.cpp`.

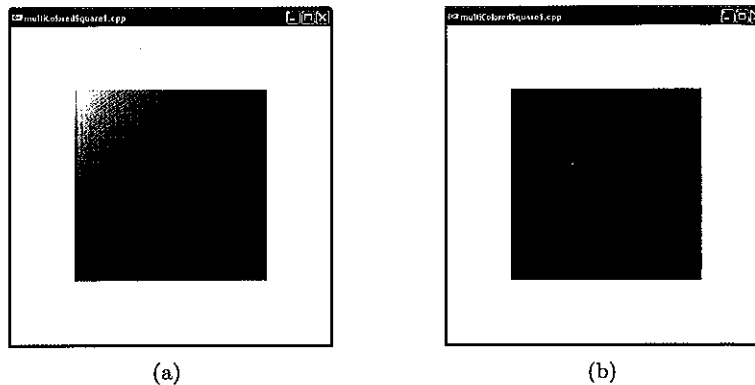


Figure 20.3: Screenshots of `multiColoredSquare1.cpp`: (a) Front (b) Back.

Experiment 20.3. Run `multiColoredSquare1.cpp`. The program itself is a copy of `redSquare.cpp`, except for a different color at each square vertex *and* enabling of two-sided coloring with a call to `glEnable(GL_VERTEX_PROGRAM_TWO_SIDE)` in the setup routine. The output initially is a multi-colored square (Figure 20.3(a)).

The vertex shader `simpleColorizer.vs` writes out both a front and a back color to the built-in variables `gl_FrontColor` and `gl_BackColor`, respectively:

```
gl_FrontColor = gl_Color;
gl_BackColor = vec4(1.0, 0.0, 0.0, 1.0);
```

It reads the front color from the user-defined colors, which it accesses through the built-in state variable `gl_Color`, while the back color is a fixed red.

The fragment shader `passThrough.fs`, on the other hand, simply sets

```
gl_FragColor = gl_Color;
```

Now, the way the GLSL works, the fragment shader *does not* receive its `gl_Color` values from the program; rather they are computed by interpolation from either the `gl_FrontColor` or `gl_BackColor` values specified in the

vertex shader, depending on whether the fragment is facing the vertex shader using `gl_FrontColor` or the back of the vertex shader to access its state. To keep the context in mind, the vertex shader does no more than pass through the color it is called a *pass-through*.

As the square is facing the fragment shader, the fragment color is taken from `gl_FrontColor` in the program. Conversely, if the square is facing the back of the vertex shader, the fragment color is taken from `gl_BackColor`.

A fun way to reveal the back of the square is to replace `gl_FragColor = gl_Color;` with `gl_FragColor = gl_BackColor;`. Accordingly, replace

```
void main()
{
    gl_FrontColor = gl_Color;
    gl_BackColor = vec4(1.0, 0.0, 0.0, 1.0);

    vec4 transposePc = gl_Color;
    // coordinate vector

    gl_Position = gl_ModelViewProjectionMatrix * gl_Position;
```

to see now a back-facing square.

20.2.2 Specifying Attributes

We see next how to specify attributes for two shaders, as well as how to use them. Particularly, these are the same as in Section 20.1.2, but with a few changes.

attribute

attribute *qualified* (e.g., `gl_FragColor`), which is simply, attributes) to the vertex shader. The vertex shader receives the incoming vertex; the fragment shader receives the fragment as well. Built-in variables are also available among others.

Vertex shaders cannot be accessed directly by the program.

For efficiency of the pipeline, point scalars, vector

vertex shader, depending on the visible face. The use of the same name `gl_Color` to represent actually different variables in the two shaders – the vertex shader using `gl_Color` to access the program, while the fragment shader to access its sibling – can be a source of confusion. One needs to keep the context in mind when using this variable. As the current fragment shader does no more than assign colors interpolated from the vertex shader, it is called a *pass-through* fragment shader.

As the square itself is oriented counter-clockwise and, therefore, front-facing, the fragment shader computes its `gl_Color` values by interpolation from `gl_FrontColor`, which in turn tracks the vertex color values as specified in the program. Consequently, a multi-colored square is drawn.

A fun way to reverse the square's orientation next is with a bit of swizzling. Accordingly, replace the vertex shader code with

```
void main()
{
    gl_FrontColor = gl_Color;
    gl_BackColor = vec4(1.0, 0.0, 0.0, 1.0);

    vec4 transposePos = gl_Vertex.yxzw; // Interchanges x and y
    // coordinate values, reversing the order of the vertices.

    gl_Position = gl_ModelViewProjectionMatrix * transposePos;
}
```

to see now a back-facing red square (Figure 20.3(b)).

End

20.2.2 Specifying Interface Variables with Qualifiers

We see next how variables at the interfaces between the program and the two shaders, as well as at that between the shaders themselves, are specified. Particularly, these variables will be of the GLSL types described earlier in Section 20.1.2, but each with an additional *qualifier* from the following:

attribute	uniform	varying
-----------	---------	---------

attribute *qualifier*: Attribute-qualified variables (or attribute variables or, simply, attributes) are used by the program to communicate per-vertex data to the vertex shader. An attribute is read by the vertex shader with each incoming vertex; therefore, it can be updated by the program for each vertex as well. Built-in attributes include `gl_Vertex`, `gl_Color` and `gl_Normal`, among others.

Vertex shaders can only read attributes, not write them. Attributes cannot be accessed by a fragment shader at all.

For efficiency of implementation, attribute types are restricted to floating point scalars, vectors and matrices.

uniform qualifier: Uniform-qualified variables (or uniform variables or, simply, uniforms) are used by the program to communicate per-primitive data to either shader or to both together. A uniform is not read and cannot be updated within a `gl_Begin(primitive)-gl_End()` pair in the program, only outside. Therefore, its value remains constant through a primitive and it cannot be used to send per-vertex values. Built-in uniforms include `gl_ModelViewMatrix`, `gl_ProjectionMatrix` and `gl_ModelViewProjectionMatrix`, among others.

Shaders can only read uniforms, not write them.

varying qualifier: Varying-qualified variables (or varying variables or, simply, varyings) are used by the vertex shader to communicate data to the fragment shader. There is, however, an important particularity of varyings: the vertex shader writes them per-vertex; subsequently, however, this data is *interpolated in a perspective-correct manner* across each primitive's fragments before being read by the fragment shader. For this reason varyings are often called *interpolators*, which actually is a more appropriate term. Built-in varyings include `gl_Color`, `gl_FrontColor` and `gl_BackColor`, among others.

Fragment shaders can only read varyings, not write them.

Remark 20.1. As we learned from Experiment 20.3, the vertex shader writes to the built-in varyings `gl_FrontColor` and `gl_BackColor`, while the fragment shader reads the built-in varying `gl_Color`, which is interpolated from one of the first two, depending on the primitive's visible face. Moreover, `gl_Color` is the name of an attribute as well, which is accessed by the vertex shader to read the program's color state.

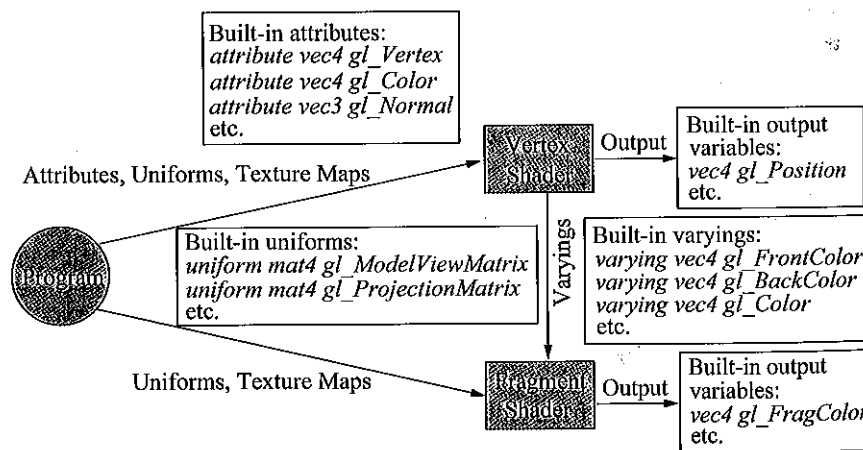


Figure 20.4: GLSL's communication scheme and a few popular built-in variables.

Figure 20.4
tion scheme
Texture map
we'll see the

Remark 20.1.
type qualifier
in, out and

Let's get
apply to a us

Experiment
as redSquare
because of the

What we h
color the squ
the position
RGB's within
color (0.8, 0.2,
the color (0.2,

Since the fi
a varying is u
positionToCo

varying vec
in both shaders

vertexColor

in the vertex sh
xyz-values, whi

gl_FragColor

in the fragment
tacking a 1 on f

Let's code a

Experiment 2
cylinder.cpp,
control the num
Press the up/do
keys to change t
shows the cylind

The waviness
cylinder.cpp vi

variables
cate per-
m is not
nd() pair
constant
ex values.
atrix and

variables or,
te data to
ularity of
y, however,
cross each
c. For this
is a more
Color and

tex shader
c, while the
interpolated
. Moreover,
y the vertex

-in output
bles:
gl_Position

gs:
gl_FrontColor
gl_BackColor
gl_Color

-in output
bles:
gl_FragColor

-in variables.

Figure 20.4 illustrates the qualifier definitions in the GLSL's communication scheme and shows a few of the most commonly used built-in variables. Texture maps are communicated by the program to the shaders as well and we'll see their use soon.

Remark 20.2. In addition to the three above, the GLSL defines four other type qualifiers: `const` (just like its C namesake for compile-time constants), `in`, `out` and `inout` (the latter three qualify formal function parameters).

Let's get our feet wet with the easiest of the three interface qualifiers to apply to a user-defined variable: `varying`.

Experiment 20.4. Run `multiColoredSquare2.cpp`. The code is exactly as `redSquare.cpp`, except this time the output is a multi-colored square because of the new shaders. Figure 20.5 is a screenshot. **End**

What we have done in `multiColoredSquare2.cpp` is use the shaders to color the square's vertices in a somewhat offbeat manner: by converting the position of each into a color by dividing its *xyz*-values by 100 to get RGB's within 0 to 1. For example, the vertex at (80.0, 20.0, 0.0) gets the color (0.8, 0.2, 0.0), a reddish hue; likewise, the vertex at (20.0, 80.0, 0.0) gets the color (0.2, 0.8, 0.0), which is more green.

Since the fragment shader `positionToColor.fs` cannot read `gl_Vertex`, a `varying` is used to pass the needed data to it from the vertex shader `positionToColor.vs`. In particular, the global declaration

```
varying vec3 vertexColor;
```

in both shaders links them through the `varying vertexColor`. Next

```
vertexColor = 0.01 * gl_Vertex.xyz;
```

in the vertex shader writes `vertexColor` with one-hundredth of the vertex *xyz*-values, while

```
gl_FragColor = vec4(vertexColor, 1.0);
```

in the fragment shader turns `vertexColor` into a legitimate color vector by tacking a 1 on for the alpha value.

Let's code a couple of uniforms next.

Experiment 20.5. Run `wavyCylinder1.cpp`. This program, based on `cylinder.cpp`, draws a cylinder with a wavy surface, allowing the user to control the number of waves, as well as change its color from red to green. Press the up/down arrow keys to change the waviness, the left/right arrow keys to change the color and 'x'-'Z' keys to turn the cylinder. Figure 20.6 shows the cylinder initially. **End**

The waviness and color capabilities of `wavyCylinder1.cpp` are added into `cylinder.cpp` via the vertex and fragment shaders, respectively. The vertex

Section 20.2

COMMUNICATION

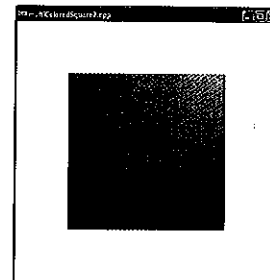


Figure 20.5: Screenshot of `multiColoredSquare2.cpp`.

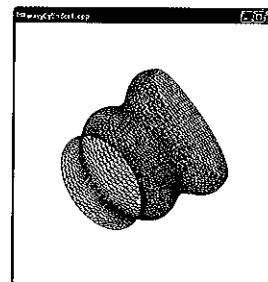


Figure 20.6: Screenshot of `wavyCylinder1.cpp`.

shader receives a parameter value from the program through the uniform `waveParamShader` to control the number of waves, while the fragment shader receives a parameter value from the program through the uniform `colorParamShader` to determine the cylinder's color. Let's understand the shaders themselves before deciphering how the uniforms are linked to the program.

The vertex shader `waves1.vs` first reads the cylinder's vertex coordinates `gl_Vertex` into the local vector variable `v`. Subsequently, the statements

```
v.x *= 1 + 0.1 * sin(waveParamShader * PI * v.z);
v.y *= 1 + 0.1 * sin(waveParamShader * PI * v.z);
```

scale the cross-section of the cylinder, which is parallel to the xy -plane, by the factor

$$1 + 0.1 \sin(\text{waveParamShader} * \pi * v.z)$$

This scaling factor follows a sine function along the z -axis, which is the axis of the cylinder. The greater the parameter `waveParamShader`, the more rapidly does the scaling factor vary with z -value `v.z`. Finally,

```
gl_Position = gl_ModelViewProjectionMatrix * v;
```

outputs the position of the cylinder vertex.

The fragment shader `colorInterpolate.fs` is simple. The statement

```
gl_FragColor = vec4(1.0 - colorParamShader, colorParamShader,
                    0.0, 1.0);
```

uses the parameter `colorParamShader` to interpolate the fragment color between red and green.

Let's see now how the uniforms are actually linked to the program. The global declaration

```
uniform float waveParamShader = 2.0;
```

in the vertex shader declares the uniform `waveParamShader`. It will be linked to a counterpart variable `waveParamProgram`, which has been declared globally in the program. The two statements

```
parameterLocation = glGetUniformLocation(programHandle,
                                         "waveParamShader");
glUniform1f(parameterLocation, waveParamProgram);
```

in the `specialKeyInput()` routine – where `waveParamProgram` is manipulated – complete the linking: the first statement obtains the location of the uniform `waveParamShader`, while the second binds it to the program variable `waveParamProgram`.

Likewise, the fragment shader uniform `colorParamShader` is linked to its program counterpart `colorParamProgram`.

Coding an attribute is next.

Experin
are exact
in the me
we discus

The s
same as i
wavePara
to its vert
in the vert
inside the

```
for(j
{
glB
for
{
}
glEn
}
```

in the dra
Program p
which yield
shader wav

The me
counterpart
from that o

```
glBindA
```

just before
attribute s
value 1). Su

```
glVertex
```

in the triang
associated
Program.

Experiment 20.6. Run `wavyCylinder2.cpp`. The output and controls are exactly as for `wavyCylinder1.cpp`. The difference between the two is in the mechanism by which the cross-section of the cylinder is scaled, which we discuss next. **End**

The scaling factor applied to the cylinder in `wavyCylinder2.cpp`, the same as in `wavyCylinder1.cpp` and parametrized by the program variable `waveParamProgram` as well, is computed by the program itself and shipped to its vertex shader `waves2.vs` via an attribute, rather than being computed in the vertex shader as for `wavyCylinder1.cpp`. The triangle strip definition inside the for loop

```
for(j = 0; j < q; j++)
{
    glBegin(GL_TRIANGLE_STRIP);
    for(i = 0; i <= p; i++)
    {
        scaleFactorProgram = 1 + 0.1 * sin(waveParamProgram * PI *
                                           h(i,j+1));
        glVertexAttribf(attributeIndex, scaleFactorProgram);
        glArrayElement( (j+1)*(p+1) + i );

        scaleFactorProgram = 1 + 0.1 * sin(waveParamProgram * PI *
                                           h(i,j));
        glVertexAttribf(attributeIndex, scaleFactorProgram);
        glArrayElement( j*(p+1) + i );
    }
    glEnd();
}
```

in the `drawScene()` routine computes the scaling factor `scaleFactorProgram` per vertex, using a formula parametrized by `waveParamProgram`, which yields the same value as the corresponding formula in the vertex shader `waves1.vs` of `wavyCylinder1.cpp`.

The method of linking the program variable `scaleFactorProgram` to its counterpart vertex shader attribute `scaleFactorShader` is a little different from that of linking to a uniform. The command

```
glBindAttribLocation(programHandle, attributeIndex,
                     "scaleFactorShader");
```

just before the for loop in `drawScene()`, associates the vertex shader attribute `scaleFactorShader` with the index `attributeIndex` (current value 1). Subsequently, each command

```
glVertexAttribf(attributeIndex, scaleFactorProgram);
```

in the triangle strip definition sets the value of `scaleFactorShader`, already associated with `attributeIndex`, to the current value of `scaleFactorProgram`.

The fragment shader `colorInterpolate.fs` of `wavyCylinder2.cpp` is copied over from `wavyCylinder1.cpp`.

Exercise 20.2. (Programming) Rewrite `throwBall.cpp` with the help of shaders – in particular, replace the call to translate the ball in the program with statements to change its coordinates in the vertex shader.

20.3 Per-Pixel Lighting

We come now to an application which definitively takes the GLSL beyond first-generation OpenGL: *per-pixel* lighting. The context is bump mapping, which we first encountered as a special effect in Section 13.7. The idea of bump mapping is to give an illusion of detail on a surface by perturbing its normals so that light reflects off it as though it were actually detailed. We applied this idea in Experiment 13.12 of Section 13.7 to make a plane appear corrugated in the program `bumpMapping.cpp`. We remarked then that bump mapping is particularly effective with the per-pixel lighting of Phong's shading model, where normal values are interpolated across primitives, rather than being fixed at vertices. We'll demonstrate now that indeed this is the case.

First, though, we'll warm up by replicating, via a vertex shader, *per-vertex* lighting – which consists of Phong lighting at each vertex, followed by Gouraud shading to interpolate colors through the primitives. Per-vertex lighting is implemented in the traditional fixed OpenGL pipeline and is what we have seen in all our lit programs to date.

Experiment 20.7. Run `bumpMappingPerVertexLighting.cpp`, which is code-wise almost exactly `bumpMapping.cpp`, but with a couple of shaders attached. Interaction is the same as well: press space to toggle between bump mapping on and off. Figures 20.7(a) and (b) are screenshots of `bumpMapping.cpp` and `bumpMappingPerVertexLighting.cpp`, respectively, doing bump mapping. Yes, they are exactly the same and we'll see momentarily why! End

The big difference between `bumpMapping.cpp` and `bumpMappingPerVertexLighting.cpp` is that is all the lighting calculations for the latter happen in its vertex shader `perVertexLightingSimple.vs`. These calculations, however, replicate exactly those of the first-generation pipeline by implementing Phong's lighting model to determine the color intensities at each vertex, and, moreover, with environment parameters set as for `bumpMapping.cpp`. Subsequently, the fragment shader `passThrough.fs` applies Gouraud shading by interpolating the vertex intensities through each triangle, again as in the first-generation pipeline. This is the reason, then, for the identical output from the two programs. We'll soon examine the vertex shader `perVertexLightingSimple.vs` in detail.

Figure 2
Lighting

Before
where a
in unifor
for mate

A Few

Below is
lighting
will find

```
// Ma  
unif  
  
// Ma  
stru  
{  
  v  
  v  
  v  
  v  
  f  
};  
  
unif  
unif  
  
// L  
stru  
{  
  v  
  v  
  v  
  v
```

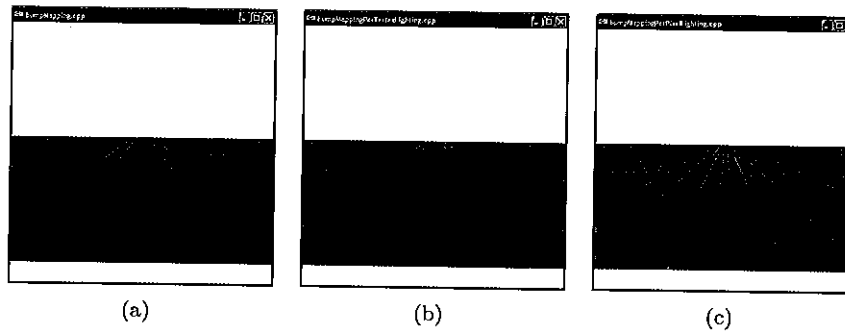


Figure 20.7: Screenshots of (a) bumpMapping.cpp (b) bumpMappingPerVertexLighting.cpp (c) bumpMappingPerPixelLighting.cpp.

Before examining the vertex shader `perVertexLightingSimple.vs`, where all the action is, let's first see a general definition of those built-in uniforms through which it accesses the current OpenGL state, particularly for material and lighting properties.

A Few of GLSL's Built-in Uniforms

Below is a code-like listing of those built-in uniforms that come in handy in lighting applications. It is extracted from the orange book [115], where you will find a listing of all GLSL built-ins.

```
// Matrix State
uniform mat3 gl_NormalMatrix;    // Derived

// Material State
struct gl_MaterialParameters
{
    vec4 emission;    // Ecm
    vec4 ambient;    // ACM
    vec4 diffuse;    // Dcm
    vec4 specular;    // Scm
    float shininess;    // Srm
};

uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;

// Light State
struct gl_LightSourceParameters
{
    vec4 ambient;    // Acli
    vec4 diffuse;    // Dcli
    vec4 specular;    // Scli
    vec4 position;    // Ppli
```

```

vec4 halfVector;           // Derived: Hi
vec3 spotDirection;        // Sdli
float spotExponent;        // Srli
float spotCutoff;          // Crli
                           // (range: [0.0, 90.0], 180.0)
float spotCosCutoff;       // Derived: cos(Crli)
                           // (range: [1.0, 0.0], -1.0)

float constantAttenuation; // K0
float linearAttenuation;   // K1
float quadraticAttenuation; // K2
};

uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];

struct gl_LightModelParameters
{
    vec4 ambient; // Acs
};

uniform gl_LightModelParameters gl_LightModel;

```

Now let's get to the vertex shader `perVertexLightingSimple.vs`. The first statement

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

multiplies the current normal, stored in the attribute `gl_Normal`, by the uniform `gl_NormalMatrix`, which is derived from the modelview matrix as the correct transformation for normals (see Section 11.11.5 for the derivation of the normal matrix from the modelview). Then it normalizes the result with the help of the built-in function `normalize()`.

Next

```
lightDirection = normalize(gl_LightSource[0].position.xyz)
```

extracts the normalized light direction as a 3-vector from the light's position vector. The position's *w* value, which is 0 because the given light source is directional, is ignored. Keep in mind, as well, that the light direction is the same for all vertices, again because the light source is directional.

The statement

```
halfVector = normalize(gl_LightSource[0].halfVector.xyz);
```

returns the normalized halfway vector between light and eye directions. The `halfVector` field of `gl_LightSource[0]` is a derived one storing the bisector between `gl_LightSource[0].position` and the vector `[0, 0, 1, 0]`. Therefore, it is correct to use the value of the preceding equation as the halfway vector *only if* the viewpoint is infinite (an infinite viewpoint is where the eye is assumed in the positive *z*-direction – see Section 11.4 for a discussion of

local versus infinite viewpoints). The program, in fact, asserts an infinite viewpoint.

The next few statements

```
emission = gl_FrontMaterial.emission;
globalAmbient = gl_LightModel.ambient * gl_FrontMaterial.ambient;
ambient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
diffuse = max(dot(normal, lightDirection), 0.0)
           * (gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse);
specular = pow( max(dot(normal, halfVector), 0.0),
                gl_FrontMaterial.shininess )
           * (gl_LightSource[0].specular * gl_FrontMaterial.specular);
gl_FrontColor = emission + globalAmbient + ambient + diffuse +
                specular;
```

need little narration. If you refer to the first lighting equation (11.10) in Section 11.2.4, then the statements above are a word for word implementation (assuming only one light source, of course). Note that, as there is neither a spotlight nor distance attenuation to take into account, implementing the first equation (11.10) effectively implements the full OpenGL lighting equation (11.12).

The pass-through fragment shader completes the job of Gouraud shading the interiors of primitives by applying interpolated color values. Note that there is no user-defined communication – it's all through built-in variables.

Experiment 20.8. If you are skeptical that we have actually replicated fixed-functionality lighting calculations in the vertex shader `perVertexLightingSimple.vs` and wondering if we are still somehow sneaking the output from fixed-functionality, then replace the `gl_FrontColor` specification in that shader with

```
gl_FrontColor = vec4(1.0, 0.0, 0.0, 1.0);
```

Figure 20.8 is a screenshot. There is no doubt, is there, that it's the vertex shader that's in charge of color calculation?!

End

With an understanding of how to code per-vertex lighting in shaders, the upgrade next to per-pixel is not hard.

Experiment 20.9. Run `bumpMappingPerPixelLighting.cpp`. The program itself is identical to `bumpMappingPerVertexLighting.cpp` – the difference is in the shaders, which now implement Phong shading, or per-pixel lighting as it is called. Again, press space to toggle between bump mapping on and off. Figure 20.7(c) is a screenshot.

End

Let's first look at the vertex shader `PerPixelLightingSimple.vs` of `bumpMappingPerPixelLighting.cpp`. The statements

```
normal = normalize(gl_NormalMatrix * gl_Normal);
lightDirection = normalize(gl_LightSource[0].position.xyz);
halfVector = normalize(gl_LightSource[0].halfVector.xyz);
```



Figure 20.8: Screenshot of Experiment 20.8.

are copied over from the per-vertex shader `perPixelLightingSimple.vs`, but now the left-side variables are all three global varyings, rather than local variables, in order to transmit their values to the fragment shader for interpolation across triangles.

The variables `emission`, `globalAmbient` and `ambient`, as well, are transmitted as varyings to the fragment shader after their values have been set by the statements

```
emission = gl_FrontMaterial.emission;
globalAmbient = gl_LightModel.ambient * gl_FrontMaterial.ambient;
ambient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
```

These three light components will then simply be interpolated across interior pixels. Nothing further need be done, as these components are the same as in the case of per-vertex lighting, even in the interior of triangles.

The two statements next pass the light's diffuse and specular color intensities, respectively scaled by the material's corresponding color values, as varyings to the fragment shader for interpolation, *and further attenuation per pixel*, as we shall see:

```
diffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse;
specular = gl_LightSource[0].specular * gl_FrontMaterial.specular;
```

Next let's see what the fragment shader `perPixelLightingSimple.fs` does. First,

```
normalPerPixel = normalize(normal);
```

normalizes the received interpolated normal values. There is no need to do likewise for the received interpolated light direction and halfway vector values because the two are constant over all vertices, given that the program has a single directional light source and an infinite viewpoint; therefore, their interpolated values are constant, as well, and of unit length already.

The next two statements, respectively, attenuate the interpolated diffuse and specular values received from the vertex shader with a factor computed exactly as in the case of per-vertex lighting, *except* now the interpolated normal value is used at each pixel:

```
diffusePerPixel = max(dot(normalPerPixel, lightDirection), 0.0) *
    diffuse;
specularPerPixel = pow( max(dot(normalPerPixel, halfVector), 0.0),
    gl_FrontMaterial.shininess ) * specular;
```

Finally,

```
gl_FragColor = emission + globalAmbient + ambient +
    diffusePerPixel + specularPerPixel;
```

accumulates all the color components and applies them per fragment.

The superiority of per-pixel lighting is apparent upon comparing Figures 20.7(a) and (b) with Figure 20.7(c): the corrugations are far more sharply defined in the latter.

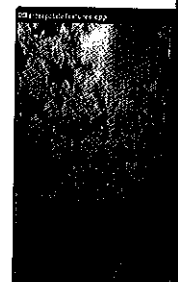
Exercise 20
GLSL to mov
front and back
per-pixel lit ve

Exercise 20
lightAndMate
attenuation an

20.4 Te

Textures can n
difficulty, but n
next in a progr

Experiment
user to interpo
a square. Figu
configurations.



(a)

Figure 20.9: Scr

The progra
texture coordin
links to the sh
Specifically, the

```
glActiveTex  
glBindTextu  
parameterLo  
glUniformli
```

links the sample
textureInterp

Exercise 20.3. (Programming) Rewrite `litCylinder.cpp`, using the GLSL to move lighting calculation over to the shaders. Mind that both front and back faces of the cylinder are visible. Write both per-vertex and per-pixel lit versions.

Exercise 20.4. (Programming) Write a per-pixel lit version of `lightAndMaterial1.cpp`. Make sure to take into account distance attenuation and that both lights are positional.

Section 20.4
TEXTURES

20.4 Textures

Textures can not only be imported into the programmable pipeline without difficulty, but manipulated in there to great effect. We'll see a simple example next in a program which interpolates between two textures.

Experiment 20.10. Run `interpolateTextures.cpp`, which allows the user to interpolate between (or, blend, if you like) two textures painted on a square. Figure 20.9 shows screenshots of the start, a part way and end configurations. **End**

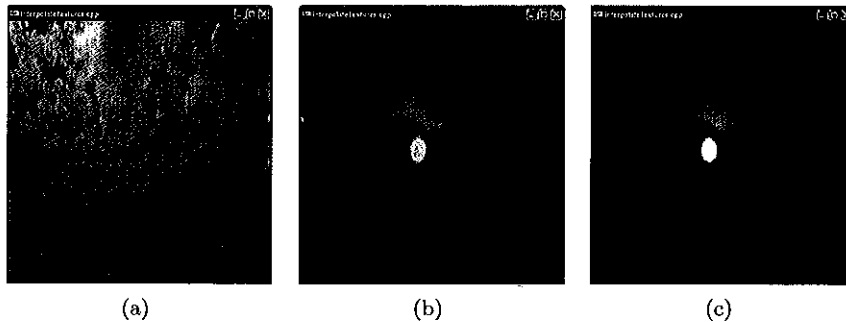


Figure 20.9: Screenshots of `interpolateTextures.cpp`: (a) Start (b) Part way (c) End.

The program itself simply loads and activates two textures, assigns texture coordinates to the vertices of the square and, most importantly, links to the shaders, passing to them the texture maps and coordinates. Specifically, the statements

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture[0]);
parameterLocation = glGetUniformLocation(programHandle, "skyTexture");
glUniform1i(parameterLocation, 0);
```

links the sampler `skyTexture`, declared as a uniform in the fragment shader `textureInterpolator.fs`, with the sky image. A similar set of statements

links the sampler `nightskyTexture`, also declared in the fragment shader, with the night sky image.

The assignments

```
gl_TexCoord[0] = gl_MultiTexCoord0;  
gl_TexCoord[1] = gl_MultiTexCoord0;
```

in the vertex shader `textureSimple.vs` simply access the texture coordinates to use with the two textures – which happen to be the same in this case – and place them in the built-in varyings on the left for interpolation and transmission to the fragment shader. Finally, the equation

```
gl_FragColor = (1.0 - paramShader) * texture2D(skyTexture,  
gl_TexCoord[0].st) +  
paramShader * texture2D(nightSkyTexture,  
gl_TexCoord[1].st);
```

in the fragment shader, parametrized by the uniform `paramShader`, interpolates between the two textures.

Exercise 20.5. (Programming) Rewrite `litTexturedCylinder.cpp` with the help of shaders. In particular, you will have to implement the `GL_MODULATE` option to combine light with texture.

20.5 Summary, Notes and More Reading

This chapter gave an introduction to the programmable pipeline and, particularly, OpenGL's Shading Language, the GLSL, which is used to do the programming. The canonical source for all things OpenGL, including the GLSL is, of course, the OpenGL site [99]. The standard text reference for the GLSL is by Rost & Licea-Kane [115], known as the orange book.

Particularly exciting is that, with OpenGL ES 2.0, shaders have gone mobile with a vengeance: anything that can be done in a shader has been removed from fixed-functionality and *has to be done* in a shader! OpenGL ES 2.0 is a “lean, mean, shadin’ machine” as the OpenGL site calls it. The mobile shading language, GLSL ES, itself is very similar to the desktop version of this chapter, so the reader should be able to begin coding shaders for small devices without trouble.

Another popular high-level shading language is Cg (or, C for Graphics) developed by Nvidia and Microsoft, which, in fact, is nearly identical to Microsoft's own proprietary HLSL (High Level Shading Language). The particular advantage of Cg for Windows developers is that it allows simultaneous development of shaders for DirectX and OpenGL. However, the GLSL, being part of OpenGL 2.0, enjoys wider vendor and platform support.