CHAPTER 15

GLSL Shaders

We will conclude our journey through 3D graphics with an exciting topic: GLSL shaders. Shader programming has a fame of being the tool of choice for creating striking real-time computer graphics. While this is true, it is also true that coding shaders is not easy because they require an advanced understanding of how computers convert numbers into images on the screen, which often involves a great deal of mathematics. In this chapter, we will try to go through the basic concepts and applications of GLSL shaders step by step, so readers should end up with a good foundation to continue learning more about this powerful tool.

What Is a Shader?

This is a good question to start our chapter with! A quick answer could be that a shader is a piece of code that generates an image on the screen from input data representing a 2D or 3D scene. This code runs on the Graphics Processing Unit (GPU) of the computer (either desktop, laptop, phone, or watch). In the previous chapters, we already saw how to enter some of that data with the Processing API: vertices, colors, textures, lights, etc., but we did not need to write any shader code. But everything that Processing draws on the screen with the P2D and P3D renderers is the output of a specific default shader running behind the scenes. Processing handles these default shaders transparently so that we don't need to worry about them, and we can simply use the Processing's drawing functions to create all kinds of shapes, animations, and interactions.

Processing offers a set of advanced functions and variables that allows us to replace the default shaders with our own, written in a language called GLSL, from "OpenGL Shading Language" (OpenGL refers to a programming interface for 2D and 3D graphics; Processing uses OpenGL internally to talk to the GPU). This opens many exciting possibilities: rendering 3D scenes using more realistic lighting and texturing algorithms, applying image postprocessing effects in real time, creating complex procedural objects that would be very hard or impossible to generate with the regular drawing API, and sharing shader effects between desktop, mobile, and web platforms with minimal code changes. All of this surely sounds great, but to be able to write our own shaders, we first need to understand how they work and how they can be used to modify and extend the drawing capabilities of Processing. The next section will provide a brief overview of the inner workings of shaders before we jump into writing GLSL code.

The Graphics Pipeline: Vertex and Pixel Shaders

All modern GPUs in our computing devices implement a well-defined sequence of stages from input data to final output on the screen, called the "graphics pipeline." We will look at the main stages in the graphics pipeline from the perspective of a simple Processing sketch so we can understand how the data we enter in Processing goes through this pipeline. This sketch, shown in Listing 15-1, draws a quad with lights and some geometric transformations using the functions we learned in Chapters 13 and 14.

A. Colubri, *Processing for Android*, https://doi.org/10.1007/978-1-4842-9585-4_15

Listing 15-1. Processing sketch that draws a lit rotating rectangle in 3D

```
float angle;

void setup() {
  size(400, 400, P3D);
  noStroke();
}

void draw() {
  background(255);
  perspective();
  camera();
  pointLight(255, 255, 255, 200, 200, -300);
  translate(200, 200);
  rotateY(angle += 0.01);
  beginShape(QUADS);
  normal(0, 0, 1);
  fill(50, 50, 200);
  vertex(-100, +100);
  vertex(+100, +100);
  fill(200, 50, 50);
  vertex(+100, -100);
  vertex(-100, -100);
  endShape();
}
```

The data we send to the graphics pipeline from the sketch consists of the global parameters of the scene (e.g., projection, camera, and lighting setup, model or geometry transformations applied on the shapes such as rotations and translations) and the attributes of each individual vertex that form our shapes (including position, color, normal vectors, and texture coordinates). Another important piece of information is the type of shape (in this case, QUAD), which tells the GPU how the vertices should be connected with each other (we saw all the different shape types supported by Processing in Chapter 4). The first stage in the pipeline is the "vertex shader." It calculates the position of each vertex in 3D space after applying setup and geometric transformations, as well as its color as the result of the lights in the scene and the material attributes of the vertex. The type of shape is also considered in this stage so that by connecting the vertices with each other according to the type, the output of the vertex shader will be the list of pixels that should be painted with a specific color to draw the desired shape on the screen. This output is called "fragment data," which comprises not only the coordinate of each pixel on the screen that should be painted but also its color (and potentially other attributes that are defined per pixel). This output from the vertex shader in turn represents the input of the "fragment shader" (called this way since it operates on fragments). The fragment shader outputs the final color for each pixel on the screen. This sequence of operations is depicted in Figure 15-1.

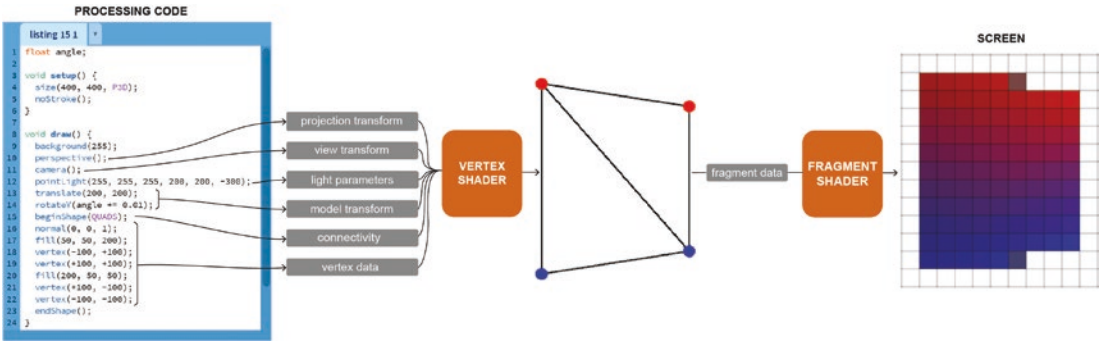

Figure 15-1. *Diagram of the vertex and fragment stages in the graphics pipeline*

■ **Note** This is a very simplified representation of the graphics pipeline; there are several other stages that are not shown here so we can focus on its most important elements. References to learn shader programming in more depth are provided at the end of this section.

A key feature that makes shaders so important in graphics programming is their speed. In principle, we could potentially draw any 2D or 3D scene without using shaders. For example, Processing allows us to set the color of each pixel on the screen directly from our sketch, simply by looping over all the pixels one by one, as we can see in Listing 15-2.

Listing 15-2. Processing sketch that sets the color of each pixel on the screen

```
fullScreen();
for (int x = 0; x < width; x++) {
  for (int y = 0; y < height; y++) {
    set(x, y, color(map(x, 0, width, 0, 255), map(y, 0, height, 0, 255), 255));
  }
}
```

While this code is perfectly valid and will generate a gradient of color covering the entire screen, it works sequentially, going one pixel at a time and setting its color. This will not run very fast, especially for devices with high-resolution screens. For example, a resolution of 2340×1080 pixels or more is very common for current Android devices, meaning that there is a total of at least 2,527,200 pixels to paint in every frame. If we now consider that a smooth animation often requires drawing 60 frames per second, this means that we should be able to calculate and set a color (at least) 15,1632,000 times per second, which, if we do sequentially one pixel at a time, may significantly slow down our sketch. On the other hand, vertex and fragment shaders are exceptionally fast, allowing real-time rendering even for very high vertex and pixel counts. This is because instead of processing vertices or fragments one by one, they work in parallel. Therefore, they can process many vertices or fragments simultaneously, even if the calculations on each vertex or fragment are quite complicated. We can think of shaders as little snippets of code that run on each vertex (or fragment), all at the same time. We don't need to worry about looping over all the vertices (or fragments); the GPU will do that for us and with the added benefit of doing it in parallel to ensure smooth real-time animations!

The remainder of this chapter will provide an overview of GLSL shaders within Processing. However, there are many other programming tools and environments that support shader programming, and thanks to the use of GLSL as the common language, shader code and techniques can be ported back and forth between these tools and Processing. For example, we can use shaders in web apps through WebGL, the JavaScript API for rendering interactive 2D and 3D graphics in any compatible web browser. *The Book of Shaders* (https://thebookofshaders.com), by Patricio Gonzalez Vivo and Jen Lowe, is an excellent learning resource on shaders, which focuses on the use of fragment shaders with WebGL. The p5.js library also supports GLSL shaders, and P5.js Shaders (https://itp-xstory.github.io/p5js-shaders), by Casey Conchinha and Louise Lessél, is another guide that can also be useful for Processing users. Online tools specifically designed for creating and sharing shaders are excellent places to learn and experiment, such as the following:

- Shadertoy: www.shadertoy.com/
- GLSL Sandbox: https://glslsandbox.com/
- Vertex Shader Art: www.vertexshaderart.com/

The Processing shader examples by Gene Kogan (https://github.com/genekogan/Processing-Shader-Examples) is a good collection of GLSL shaders specifically to be run inside Processing and shows how to adapt shader effects from online tools such as the GLSL Sandbox to be compatible with Processing's shader API. The shader examples for Processing by Adam Ferriss (https://github.com/aferriss/shaderExamples) is another useful resource for shader programming in Processing, with a version for p5.js (https://github.com/aferriss/p5jsShaderExamples).

The PShader Class

In the previous section, we saw that the two stages in the graphics pipeline are the vertex and fragment shaders. Both are needed to specify a complete, working pipeline. In Processing, we write the GLSL code for the fragment and vertex shaders in separate files, which then are combined to form a single "shader program" than can be run by the GPU. The word "program" is often skipped, with the assumption that when we say shader, we are referring to a complete shader program involving both fragment and vertex shaders.

A shader (program) is encapsulated in Processing by the PShader class. A PShader object is created with the loadShader() function that takes the file names of the vertex and fragment shaders as the arguments. If we only provide one file name, then Processing will expect that the file name corresponds to the fragment shader and will use a default vertex shader to complete the program. Listing 15-3 shows a sketch loading and using a basic shader that renders a shape using its current fill color, and includes the code of the fragment and vertex shaders after the sketch's code. The files frag.glsl and vert.glsl should be saved in the data folder of the sketch. The output of this code is shown in Figure 15-2.

Listing 15-3. Sketch that uses a shader to draw a shape using its fill color

```
PShader simple;
float angle;

void setup() {
  fullScreen(P3D);
  simple = loadShader("frag.glsl", "vert.glsl");
  shader(simple);
  noStroke();
}
```

```
void draw() {
  background(255);
  translate(width/2, height/2);
  rotateY(angle);
  beginShape(QUADS);
  normal(0, 0, 1);
  fill(50, 50, 200);
  vertex(-200, +200);
  vertex(+200, +200);
  fill(200, 50, 50);
  vertex(+200, -200);
  vertex(-200, -200);
  endShape();
  angle += 0.01;
}
```

frag.glsl

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

varying vec4 vertColor;

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

vert.glsl

```
uniform mat4 transform;

attribute vec4 position;
attribute vec4 color;

varying vec4 vertColor;

void main() {
  gl_Position = transform * position;
  vertColor = color;
}
```

Notice how the shader is set using the shader() function, which takes the shader object we want to use as its argument. This function works like other Processing functions that set the style of the scene, in the sense that Processing will try to use this shader to render all subsequent shapes. We can set a new shader at any time by calling shader() again with the appropriate argument, and if we want Processing to go back to its built-in shaders, we just call resetShader(). In this example, the output of the custom shader is identical to what Processing would render without it. This is so because our shader is replicating the default shader that Processing uses to draw scenes without lights or textures. We will go through the details of this first custom shader in the next section and will learn how to make some changes to modify its output.

Figure 15-2. *Output of our first custom shader from Listing 15-3*

Anatomy of a Simple Shader

Listing 15-3 included the source code of the vertex and fragment shaders we used to arrive to the image in Figure 15-2. Let's look at them separately to start understanding the inputs, calculation, and outputs of each stage and how they work together to render the shapes in our Processing sketch. First, the vertex shader:

```
uniform mat4 transform;

attribute vec4 position;
attribute vec4 color;

varying vec4 vertColor;

void main() {
  gl_Position = transform * position;
  vertColor = color;
}
```

The vertex shader's code starts with the declaration of four variables: transform, position, color, and vertColor, followed by the instructions inside the main() function, which the GPU runs automatically on each vertex. The declared variables have types, in this case, mat4 and vec4, which mean that they hold either a matrix of 4×4 elements (in the case of transform) or vectors of four elements (in the case of position, color, and vertColor). Each one of these variables also has a "storage qualifier," which is uniform for the transform matrix, attribute for the position and color vectors, and varying for vertColor. We will explain what these qualifiers mean in a second. But first, we should note that by inspecting this code, we have encountered a defining feature of shaders: they operate on vectors and matrices, which are very convenient mathematical objects to represent and work with 3D data. The details of mathematics for computer graphics (mostly consisting of vector and matrix algebra) are beyond the scope of this chapter; here, we will only say that by multiplying the position vector by the transform matrix (which combines the effect of the geometry and camera transformations), we obtain the coordinates of the vector in screen space and then assign them to the gl_Position vector, which is a built-in GLSL variable that stores the result of this calculation and sends it down the pipeline:

$$gl\_Position = transform \cdot position = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

In this formula, we can see that the transform matrix has 16 elements (since it's 4 rows by 4 columns) and that the position has x, y, z, and w coordinates (with x, y, and z coming from our sketch code and w being a normalization factor needed for perspective calculations and determined automatically by Processing). Unless we want to manipulate these numbers to do some advanced calculation ourselves, we don't need to worry about them at all; GLSL will carry out the matrix-vector multiplication for us. The only other line of code in the shader is the vertColor = color assignment; this is simply passing along the color vector (which contains the red, green, blue, and alpha values we set with the fill() function in our sketch) to the next stage in the pipeline, using the vertColor variable for this purpose.

As we briefly mentioned before, the shader variables have a storage qualifier, either uniform, attribute, or varying. Now that we have seen what each of these variables does, it will make more sense to describe what the qualifiers mean. The transform matrix, for example, has the uniform qualifier, which indicates that it is the constant for all the vertices going through the vertex shader. This is reasonable, since all vertices have the same transformation applied to them. Then, we have the position and color qualified as attributes; this means that these variables store values defined for each individual vertex that passes through the shader. Finally, variables qualified as varying are variables that communicate information between the vertex and the fragment shaders. Here, we only have one varying variable, vertColor, which will be used, immediately after the vertex stage, to calculate the color of the fragments going into the fragment shader. The gl_Position variable is also a varying, but since it is a built-in variable from GLSL, we don't need to declare it like the other variables at the top of the vertex shader's code.

Having dissected the elements of the vertex shader in our example, we can move on to the fragment shader:

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

varying vec4 vertColor;

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

We can see that the vertColor varying variable that we had as the output in the vertex shader now appears in the declaration of the variables of the fragment shader, also as a varying variable. In general, any varying variable we declare in the vertex shader should also appear in the declaration section of the corresponding fragment shader. The implementation of the fragment shader, inside the main() function, is very simple; we just assign the vertColor variable, containing the color computed in the vertex stage, to the gl_FragColor, another built-in GLSL variable that sends the output of the fragment calculation to the corresponding pixel on the screen. The ifdef section at the top is required to make the shader compatible with mobile devices. It sets the precision of the float and integer numbers to medium, which should be fine for most devices. We need to remember to include this section in all our fragment shaders.

■ **Note** Knowledge of vector and matrix linear algebra is very important if we want to write our own shaders. There are many books on the topic; some good free online resources are the following: Linear Algebra section from Khan Academy (www.khanacademy.org/math/linear-algebra), Scratch Pixel (www.scratchapixel.com/), Immersive Linear Algebra (https://immersivemath.com), and the OpenGL tutorials from Song Ho Ahn (www.songho.ca/opengl/).

We already noted that our shader does not generate anything different from the default Processing output, but now that we have a basic understanding of the GLSL code, we should be able to make some small changes. For instance, we could try setting a constant color in the fragment shader, as shown in Listing 15-4 (the rest of the code, including sketch and vertex shader, is identical to that in Listing 15-3).

Listing 15-4. Fragment shader that sets green as the fragment color

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

varying vec4 vertColor;

void main() {
  gl_FragColor = vec4(0, 1, 0, 1);
}
```

The output of this shader is shown in Figure 15-3. Even though we are sending the same fill colors from our sketch as before, the fragment shader is ignoring the value from vertColor and using a constant value for all fragments generated by the rectangle. We should also keep in mind that in GLSL, the components of a color are between 0 and 1, so that's why we use the (0, 1, 0, 1) value to output green (0 = red, 1 = green, 0 = blue, 1 = alpha).

Figure 15-3. *Output of the shader in Listing 15-4, with constant fragment color*

One handy feature of shaders is the possibility of easy reuse. Since they implement graphics calculations at the level of each individual vertex or fragment, they are independent of the details of the geometry in our sketch, so we could apply the same shaders across different projects. For example, in Listing 15-5, we could load the same shaders from Listing 15-4, but this time the shape is a static rectangle that covers the entire screen.

Listing 15-5. Reusing our basic custom shader with a different shape

```
PShader simple;

void setup() {
  fullScreen(P3D);
  simple = loadShader("frag.glsl", "vert.glsl");
  shader(simple);
  noStroke();
}

void draw() {
  rect(0, 0, width, height);
}
```

The output of this sketch should be the entire screen of the device painted green, since the rectangle goes from (0, 0) to (width, height) and so it generates fragments for all the pixels on the screen.

Defining Custom Uniforms

As we have seen so far, shaders are a very powerful tool for graphics programming, but our learning needs to move forward by taking small steps; the downside is that shaders require very specialized knowledge in a new language, GLSL, as well as good understanding of vector and matrix algebra to fully take advantage of their capabilities. However, even at this initial stage, we can already try some interesting possibilities by defining custom uniform variables to pass parameters and user input to our shaders.

In Listing 15-6, we define two custom uniforms: `resolution` and `pointer`. In the first one, we will store the (width, height) values from Processing, while in the second, we will store the (mouseX, mouseY) coordinates. We can take advantage of another built-in varying variable available in the fragment shader, gl_FragCoord, which contains the (x, y) screen coordinates of the fragment. Using all these variables, we can construct a dynamic gradient that depends on the position of the mouse pointer on the screen.

Listing 15-6. Dynamic gradient using mouse pointer and screen resolution

```
PShader gradient;

void setup() {
  fullScreen(P3D);
  gradient = loadShader("frag.glsl", "vert.glsl");
  gradient.set("resolution", float(width), float(height))  ;
  noStroke();
}

void draw() {
  shader(gradient);
  fill(255);
```

```
  gradient.set("pointer", float(mouseX), float(mouseY));
  rect(0, 0, width, height);
  resetShader();
  fill(255);
  ellipse(mouseX, mouseY, 100, 100);
}
```

frag.glsl

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

uniform vec2 resolution;
uniform vec2 pointer;

varying vec4 vertColor;

void main() {
  float maxr = pointer.x / resolution.x;
  float maxg = pointer.y / resolution.y;
  float gradx = gl_FragCoord.x / resolution.x;
  float grady = gl_FragCoord.y / resolution.y;
  gl_FragColor = vec4(maxr * gradx * vertColor.r, maxg * grady * vertColor.g,
  vertColor.b, 1);
}
```

vert.glsl

```
uniform mat4 transform;

attribute vec4 position;
attribute vec4 color;

varying vec4 vertColor;

void main() {
  gl_Position = transform * position;
  vertColor = color;
}
```

The output of this code is shown in Figure 15-4. By moving the pointer around, the shader calculates the maximum red and green component of the gradient by dividing the pointer coordinates by the resolution (let's remember that colors in GLSL need to be normalized between 0 and 1), while the position along the gradient in x and y is determined by dividing the fragment coordinates by the resolution, again resulting in a 0–1 number. The Processing function to set the value of a uniform is PShader.set(), and it is important to note that the (width, height) and (mouseX, mouseY) values are interpreted as float numbers so they are received by the shader as floating-point vectors. In fact, the gradient can get modulated by the fill color of the rectangle, since the varying vertColor that passes the color from the vertices is used to multiply each component in the gl_FragColor. Finally, we use the resetShader() function in the sketch to disable our custom gradient shader and draw an ellipse using the default Processing shader.

Even though this example is very simple, it demonstrates how we can perform per-pixel calculations using custom parameters we send from our sketch to the shaders. This same approach can be applied to generate much more complex shader effects.

Figure 15-4. *Output of the dynamic gradient shader*

Types of Shaders in Processing

In all the shader examples we have considered so far, we only run sketches without any lighting or texturing, so shapes always appear rendered with a flat color. Can we use those same shaders as soon as we incorporate lights in our scene and apply textures to the shapes? The answer is no, because a shader for rendering lit shapes requires additional attributes and uniform variables from Processing. These additional variables are not needed by shaders to render flat-colored shapes without lights or textures, so the source code of the shaders must be different. Similarly, a shader for rendering textured shapes needs its own uniforms and attributes that would not be required otherwise. In summary, we have four possible scenarios, each requiring its own type of shader:

- There are no lights and no textures: Use a color shader.
- There are lights but no textures: Use a light shader.
- There are textures but no lights: Use a texture shader.
- There are both textures and lights: Use a texlight shader.

So depending on the current configuration of our sketch (we could be drawing flat-colored shapes at some point and textured or lit shapes later), we need to provide a shader of the correct type at each moment. For example, if we are drawing textured shapes but we set a color shader, Processing will ignore it and use its default texture shader. Processing is also capable of autodetecting the type of shader we are trying to use depending on the type of uniforms and attributes defined in the shader code. We will learn more about each type in the next sections.

Color Shaders

We used color shaders in Listings 15-3 through 15-6, since in those sketches we had neither lights nor textures. A color shader only requires the transform uniform to convert the raw vertex positions into pixels and the position and color attributes per vertex. Since shaders are reusable, we could just drop the color shader from Listing 15-3 in the next listing, Listing 15-7, where we use a blue sphere shape to demonstrate the flat shading.

Listing 15-7. Applying a color shader to a sphere

```
PShape globe;
float angle;
PShader colorShader;

void setup() {
  fullScreen(P3D);
  colorShader = loadShader("colorfrag.glsl", "colorvert.glsl");
  globe = createShape(SPHERE, 300);
  globe.setFill(color(#4C92F2));
  globe.setStroke(false);
}

void draw() {
  background(0);
  shader(colorShader);
  translate(width/2, height/2);
  rotateY(angle);
  shape(globe);
  angle += 0.01;
}
```

We didn't need to list the fragment and vertex shaders, since they are identical to those in Listing 15-3. The output of this code is shown in Figure 15-5.

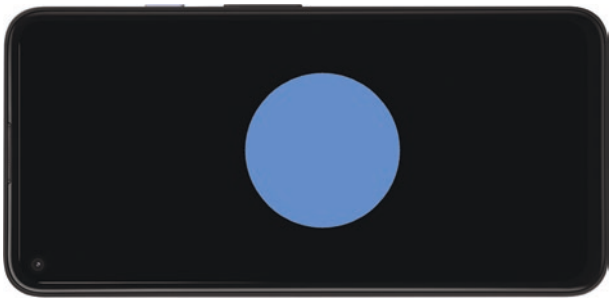

Figure 15-5. *Output of the color shader applied to a sphere*

Texture Shaders

Rendering textured shapes requires additional uniforms and attributes in the shader. Let's look at the sketch in Listing 15-8, together with the accompanying fragment and vertex shaders.

Listing 15-8. Sketch for textured rendering (no lights)

```
PImage earth;
PShape globe;
float angle;
PShader texShader;
```

```
void setup() {
  fullScreen(P3D);
  texShader = loadShader("texfrag.glsl", "texvert.glsl");
  earth = loadImage("earthmap1k.jpg");
  globe = createShape(SPHERE, 300);
  globe.setTexture(earth);
  globe.setStroke(false);
}

void draw() {
  background(0);
  shader(texShader);
  translate(width/2, height/2);
  rotateY(angle);
  shape(globe);
  angle += 0.01;
}
```

texfrag.glsl

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

uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
  gl_FragColor = texture2D(texture, vertTexCoord.st) * vertColor;
}
```

texvert.glsl

```
uniform mat4 transform;
uniform mat4 texMatrix;

attribute vec4 position;
attribute vec4 color;
attribute vec2 texCoord;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
  gl_Position = transform * position;

  vertColor = color;
  vertTexCoord = texMatrix * vec4(texCoord, 1.0, 1.0);
}
```

In this listing, we have a new uniform in the vertex shader called texMatrix, which properly scales the texture coordinates of each vertex, passed in the additional attribute texCoord. In the fragment shader, we have another new uniform variable, of type sampler2D, called texture, and that gives the shader access to the texture data. The GLSL function texture2D() allows us to retrieve the content of the texture at the position specified by the texture coordinate vertTexCoord. The result of this sketch is shown in Figure 15-6.

Figure 15-6. *Output of the texture shader applied to a sphere*

The texture2D() function becomes very handy to implement many different kinds of effects using textures. For example, we can pixelate the texture very easily by modifying the texture coordinate values, vertTexCoord.st, so that they are binned within a given number of cells. We can make this number into a uniform parameter controlled by user input, as shown in Listing 15-9, with a typical rendering in Figure 15-7 (vertex shader is omitted as it is identical to the one in Listing 15-8).

Listing 15-9. Sketch for pixelated texture rendering

```
PImage earth;
PShape globe;
float angle;
PShader pixShader;

void setup() {
  fullScreen(P3D);
  pixShader = loadShader("pixelated.glsl", "texvert.glsl");
  earth = loadImage("earthmap1k.jpg");
  globe = createShape(SPHERE, 300);
  globe.setTexture(earth);
  globe.setStroke(false);
}

void draw() {
  background(0);
  shader(pixShader);
  pixShader.set("numBins", int(map(mouseX, 0, width, 0, 100)));
  translate(width/2, height/2);
  rotateY(angle);
  shape(globe);
  angle += 0.01;
}
```

pixelated.glsl

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

uniform int numBins;
uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
  int si = int(vertTexCoord.s * float(numBins));
  int sj = int(vertTexCoord.t * float(numBins));
  gl_FragColor = texture2D(texture, vec2(float(si) / float(numBins), float(sj) /
  float(numBins))) * vertColor;
}
```

Figure 15-7. *Output of the pixelated texture shader applied to a sphere*

Light Shaders

As we learned in Chapter 14, lighting a 3D scene involves placing one or more light sources in the virtual space, defining their parameters (http://www.learnopengles.com/android-lesson-two-ambient-and-diffuse-lighting/), such as type (point, spotlight) and color (diffuse, ambient, specular). It also requires using a mathematical model that takes in those parameters to generate convincing lit surfaces. This is a huge topic in computer graphics; without going into any details, we would only say that all lighting models we can implement with the help of GLSL shaders are approximations to how light works in the real world. The model we will use in this section is probably one of the simplest; it evaluates the light intensity at each vertex of the object as the dot product between the vertex normal and the direction vector between the vertex and light positions. This model represents a point light source that emits light equally in all directions, as illustrated in Figure 15-8.

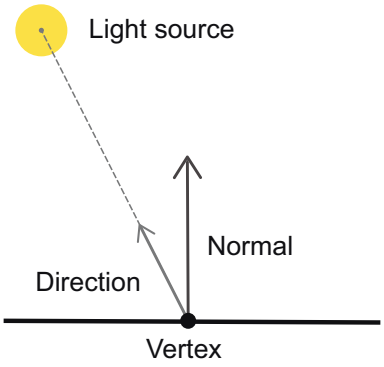

Figure 15-8. *Diagram illustrating the basic lighting model in the light shader*

Using the same geometry from the previous examples, we can now write a simple shader to render the scene with a single point light. To do so, we need some extra uniform variables in the vertex shader: lightPosition, which holds the position of the light source, and normalMatrix, which is a 3×3 matrix to convert the normal vector to the appropriate coordinates to perform the lighting calculations. The full sketch and shader code is provided in Listing 15-10.

Listing 15-10. Sketch with a simple lighting shader

```
PShape globe;
float angle;
PShader lightShader;

void setup() {
  fullScreen(P3D);
  lightShader = loadShader("lightfrag.glsl", "lightvert.glsl");
  globe = createShape(SPHERE, 300);
  globe.setStroke(false);
}

void draw() {
  background(0);
  shader(lightShader);
  pointLight(255, 255, 255, width, height/2, 500);
  translate(width/2, height/2);
  rotateY(angle);
  shape(globe);
  angle += 0.01;
}
```

lightfrag.glsl

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

```
varying vec4 vertColor;

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

lightvert.glsl

```
uniform mat4 modelview;
uniform mat4 transform;
uniform mat3 normalMatrix;

uniform vec4 lightPosition;

attribute vec4 position;
attribute vec4 color;
attribute vec3 normal;

varying vec4 vertColor;

void main() {
  gl_Position = transform * position;
  vec3 ecPosition = vec3(modelview * position);
  vec3 ecNormal = normalize(normalMatrix * normal);

  vec3 direction = normalize(lightPosition.xyz - ecPosition);
  float intensity = max(0.0, dot(direction, ecNormal));
  vertColor = vec4(intensity, intensity, intensity, 1) * color;
}
```

In the vertex shader, the ecPosition variable is the position of the input vertex expressed in eye coordinates, not screen coordinates, since it is obtained by multiplying the vertex position by the modelview matrix, which encodes the camera (view) and geometry (model) transformations, but not the projection transformations. Similarly, by multiplying the input normal vector with the normalMatrix, we obtain its coordinates in the eye coordinates system. Once all the vectors are expressed in the same system, they can be used to calculate the intensity of the incident light at each vertex. From inspecting the formula used in the shader, we could see that the intensity is directly proportional to the angle between the normal and the vector between the vertex and the light source.

In this example, there is a single point light, but Processing can send to the shader up to eight different lights and their associated parameters. The full list of light uniforms that can be used to get this information in the shader is as follows:

- uniform int lightCount: Number of active lights
- uniform vec4 lightPosition[8]: Position of each light
- uniform vec3 lightNormal[8]: Direction of each light (only relevant for directional and spot lights)
- uniform vec3 lightAmbient[8]: Ambient component of light color
- uniform vec3 lightDiffuse[8]: Diffuse component of light color
- uniform vec3 lightSpecular[8]: Specular component of light color

- uniform vec3 lightFalloff[8]: Light falloff coefficients
- uniform vec2 lightSpot[8]: Light spot parameters (cosine of light spot angle and concentration)

The values in these uniforms completely specify any lighting configuration set in the sketch using the ambientLight(), pointLight(), directionalLight(), and spotLight() functions in Processing. However, a valid light shader doesn't need to declare all these uniforms; for instance, in the previous listing, we only needed the lightPosition uniform. The output of this listing is shown in Figure 15-9.

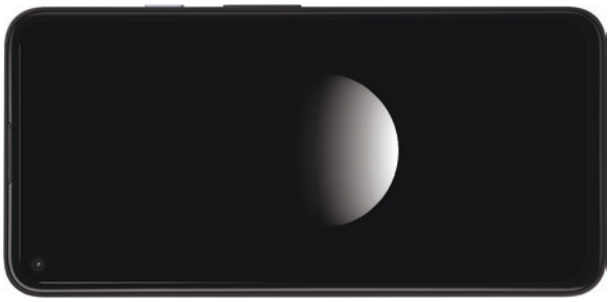

Figure 15-9. *Output of the light shader applied to a sphere*

Texlight Shaders

Finally, a textlight shader incorporates the uniforms from both light and texture shaders, and that's why it's called that way. We can integrate the code from the previous sections in the sketch shown in Listing 15-11. Its output is provided in Figure 15-10.

Listing 15-11. Sketch combining texturing and per-vertex lighting

```
PImage earth;
PShape globe;
float angle;
PShader texlightShader;

void setup() {
  fullScreen(P3D);
  texlightShader = loadShader("texlightfrag.glsl", "texlightvert.glsl");
  earth = loadImage("earthmap1k.jpg");
  globe = createShape(SPHERE, 300);
  globe.setTexture(earth);
  globe.setStroke(false);
}

void draw() {
  background(0);
  shader(texlightShader);
  pointLight(255, 255, 255, width, height/2, 500);
  translate(width/2, height/2);
  rotateY(angle);
```

```
  shape(globe);
  angle += 0.01;
}
```

texlightfrag.glsl

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

uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
  gl_FragColor = texture2D(texture, vertTexCoord.st) * vertColor;
}
```

texlightvert.glsl

```
uniform mat4 modelview;
uniform mat4 transform;
uniform mat3 normalMatrix;
uniform mat4 texMatrix;

uniform vec4 lightPosition;

attribute vec4 position;
attribute vec4 color;
attribute vec3 normal;
attribute vec2 texCoord;

varying vec4 vertColor;
varying vec4 vertTexCoord;

void main() {
  gl_Position = transform * position;
  vec3 ecPosition = vec3(modelview * position);
  vec3 ecNormal = normalize(normalMatrix * normal);

  vec3 direction = normalize(lightPosition.xyz - ecPosition);
  float intensity = max(0.0, dot(direction, ecNormal));
  vertColor = vec4(intensity, intensity, intensity, 1) * color;

  vertTexCoord = texMatrix * vec4(texCoord, 1.0, 1.0);
}
```

Figure 15-10. *Output of the texlight shader applied to a sphere*

■ **Note** We cannot use a texlight shader to render a sketch only with textures or only with lights; in those cases, we will need a separate texture or light shader.

Image Postprocessing Filters

As we saw with the pixelated texture example from Listing 15-9, the fragment shader can be used to run image postprocessing effects very efficiently by taking advantage of the parallel nature of the GPUs. For example, let's imagine that we want to render a texture using only black and white colors: black if the luminance of the original color at a given pixel in the image is below a threshold and white if it is above. This can be implemented with the texture shader in Listing 15-12.

Listing 15-12. Sketch with a black and white shader

```
PImage earth;
PShape globe;
float angle;
PShader bwShader;

void setup() {
  fullScreen(P3D);
  bwShader = loadShader("bwfrag.glsl");
  earth = loadImage("earthmap1k.jpg");
  globe = createShape(SPHERE, 300);
  globe.setTexture(earth);
  globe.setStroke(false);
}

void draw() {
  background(0);
  shader(bwShader);
  translate(width/2, height/2);
  rotateY(angle);
  shape(globe);
  angle += 0.01;
}
```

bwfrag.glsl

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

uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

const vec4 lumcoeff = vec4(0.299, 0.587, 0.114, 0);

void main() {
  vec4 col = texture2D(texture, vertTexCoord.st);
  float lum = dot(col, lumcoeff);
  if (0.5 < lum) {
    gl_FragColor = vertColor;
  } else {
    gl_FragColor = vec4(0, 0, 0, 1);
  }
}
```

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

uniform sampler2D texture;

varying vec4 vertColor;
varying vec4 vertTexCoord;

const vec4 lumcoeff = vec4(0.299, 0.587, 0.114, 0);

void main() {
  vec4 col = texture2D(texture, vertTexCoord.st);
  float lum = dot(col, lumcoeff);
  if (0.5 < lum) {
    gl_FragColor = vertColor;
  } else {
    gl_FragColor = vec4(0, 0, 0, 1);
  }
}
```

The fragment shader reads the texture at position vertTexCoord.st and uses the color value to compute the luminance and then the two alternative outputs based on the threshold, which in this case is 0.5, resulting in the black and white rendering in Figure 15-11. We can notice that this time the loadShader() function only receives the file name of the fragment shader. How does Processing complete the entire shader

program? The answer is that it uses the default vertex stage for texture shaders. Because of this, and since the varying variables are first declared in the vertex stage, the fragment shader needs to follow the varying names adopted in the default shader. In this case, the varying variables for the fragment color and texture coordinate must be named vertColor and vertTexCoord, respectively.

Figure 15-11. *Output of the black and white shader applied to a sphere*

Convolution filters (https://lodev.org/cgtutor/filtering.html) can also be implemented easily in the fragment shader. Given the texture coordinates of a fragment, vertTexCoord, the neighboring pixels in the texture (also called "texels") can be sampled using the texOffset uniform. This uniform is set automatically by Processing and contains the vector (1/width, 1/height), with width and height being the resolution of the texture. These values are precisely the offsets along the horizontal and vertical directions needed to sample the color from the texels around vertTexCoord.st. For example, vertTexCoord.st + vec2(texOffset.s, 0) is the texel exactly one position to the right. Listing 15-13 shows the implementation of a standard emboss filter, and Figure 15-12 shows its output.

Listing 15-13. Sketch with an emboss shader

```
PImage earth;
PShape globe;
float angle;
PShader embossShader;

void setup() {
  fullScreen(P3D);
  embossShader = loadShader("embossfrag.glsl");
  earth = loadImage("earthmap1k.jpg");
  globe = createShape(SPHERE, 300);
  globe.setTexture(earth);
  globe.setStroke(false);
}

void draw() {
  background(0);
  shader(embossShader);
  translate(width/2, height/2);
  rotateY(angle);
  shape(globe);
  angle += 0.01;
}
```

embossfrag.glsl

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

uniform sampler2D texture;
uniform vec2 texOffset;

varying vec4 vertColor;
varying vec4 vertTexCoord;

const vec4 lumcoeff = vec4(0.299, 0.587, 0.114, 0);

void main() {
  vec2 tc0 = vertTexCoord.st + vec2(-texOffset.s, -texOffset.t);
  vec2 tc1 = vertTexCoord.st + vec2(               0.0, -texOffset.t);
  vec2 tc2 = vertTexCoord.st + vec2(-texOffset.s,                0.0);
  vec2 tc3 = vertTexCoord.st + vec2(+texOffset.s,               0.0);
  vec2 tc4 = vertTexCoord.st + vec2(                0.0, +texOffset.t);
  vec2 tc5 = vertTexCoord.st + vec2(+texOffset.s, +texOffset.t);

  vec4 col0 = texture2D(texture, tc0);
  vec4 col1 = texture2D(texture, tc1);
  vec4 col2 = texture2D(texture, tc2);
  vec4 col3 = texture2D(texture, tc3);
  vec4 col4 = texture2D(texture, tc4);
  vec4 col5 = texture2D(texture, tc5);

  vec4 sum = vec4(0.5) + (col0 + col1 + col2) - (col3 + col4 + col5);
  float lum = dot(sum, lumcoeff);
  gl_FragColor = vec4(lum, lum, lum, 1.0) * vertColor;
}
```

Figure 15-12. *Output of the emboss shader applied to a sphere*

All these postprocessing effects were implemented as texture shaders since they only require the texture image as an input. Another way of using them is through the filter() function in Processing (https://processing.org/reference/filter_.html). This function applies a filter effect on an image that we can select from a list of predefined effects (threshold, invert, posterize, etc.), or using a shader object. We can reuse any of our texture shaders to use in filter(); for example, Listing 15-14 shows the use of the black and white shader we had in Listing 15-12, but this time applied on a flat image drawn on the screen. The result is displayed in Figure 15-13.

Listing 15-14. Using the black and white shader as a filter

```
PImage earth;
PShader bwShader;

void setup() {
  fullScreen(P2D);
  bwShader = loadShader("bwfrag.glsl");
  earth = loadImage("earthmap1k.jpg");
}

void draw() {
  image(earth, 0, 0, width, height);
  filter(bwShader);
}
```

Figure 15-13. *Output of the black and white shader applied as a filter image*

Day-to-Night Earth Shader and Live Wallpaper

To conclude this chapter on shader programming and cap the section on 3D graphics, let's take on a final project where we put our newly gained GLSL skills to good use! We can continue with the Earth theme we had throughout the examples in the chapter while introducing a few advanced techniques. Let's start with an observation we could derive from the texture shader examples: the fragment shader always gets the color values from a single texture. However, we can read texels from several textures at the same time, and this feature enables shader programmers to implement very sophisticated effects (such as realistic lighting with shadows and other visual nuances). As we pointed out before, most of these effects would also require some advanced mathematics, so we will try to keep our code as simple as possible.

An idea for a shader that requires reading from two textures simultaneously would be to render the Earth so that the night lights caused by human activity become visible as the planet rotates around its axis. The site Solar System Scope contains equirectangular maps for many objects in the Solar System that can be freely shared under the Creative Commons Attribution 4.0 International license (`www.solarsystemscope.com/textures/`), including the Earth's Day and night maps (Figure 15-14) we will use for this project.

We know how to implement a shader in Processing to render a textured object (e.g., Listing 15-8) with a single texture, and we will now learn how to read more than one texture from our shader. It is not difficult; all we need to do is to declare two sampler2D uniform variables in the fragment shader and manually set those uniforms from the Processing code with the corresponding image object, as it is shown in Listing 15-15.

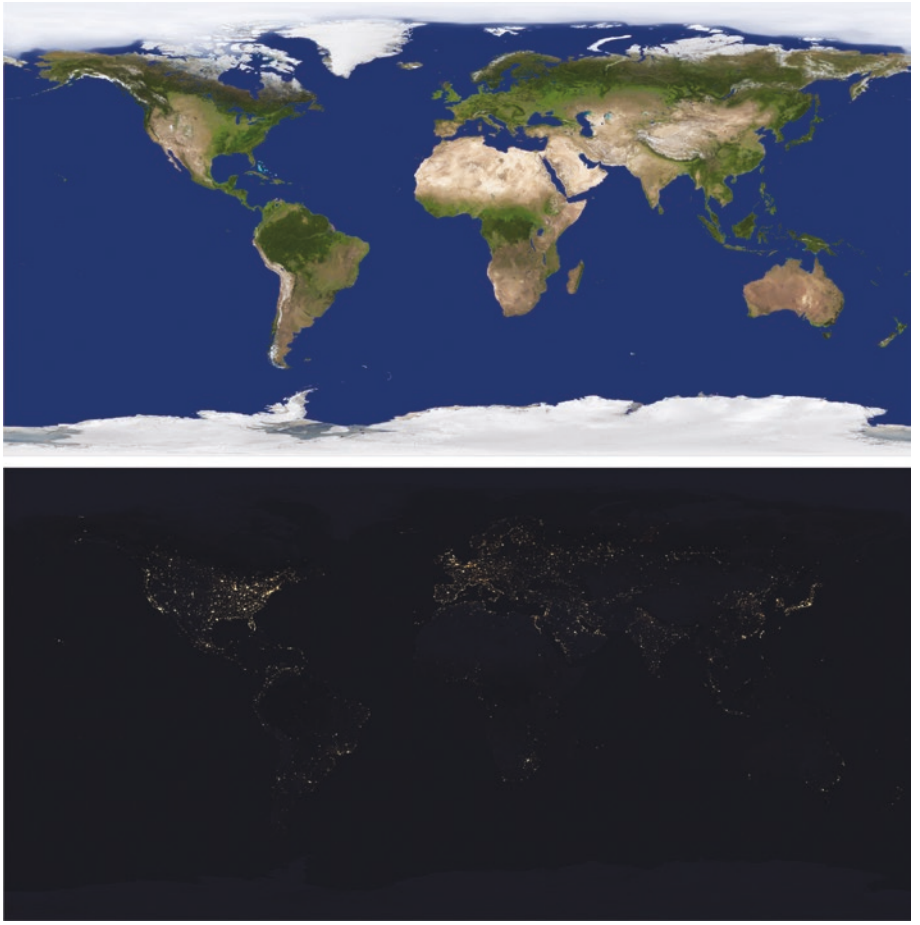

Figure 15-14. *Day and night maps of the Earth from the Solar System Scope repository*

Listing 15-15. Sketch with two images used for texturing a sphere

```
PShape earth;
PShader earthShader;

float viewRotation;

void setup() {
  fullScreen(P3D);

  PImage earthDay = loadImage("Solarsystemscope_texture_2k_earth_daymap.jpg");
  PImage earthNight = loadImage("Solarsystemscope_texture_2k_earth_nightmap.jpg");

  earthShader = loadShader("EarthFrag.glsl");
  earthShader.set("dayTexture", earthDay);
  earthShader.set("nightTexture", earthNight);

  earth = createShape(SPHERE, 400);
  earth.setStroke(false);
}

void draw() {
  background(0);

  earthShader.set("mixFactor",  map(mouseX, 0, width, 0, 1));
  viewRotation += 0.001;

  translate(width/2, height/2);

  shader(earthShader);
  pushMatrix();
  rotateY(viewRotation);
  shape(earth);
  popMatrix();
}
```

EarthFrag.glsl

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

uniform sampler2D dayTexture;
uniform sampler2D nightTexture;

uniform float mixFactor;

varying vec4 vertColor;
varying vec4 vertTexCoord;
```

```
void main() {
  vec2 st = vertTexCoord.st;
  vec4 dayColor = texture2D(dayTexture, st);
  vec4 nightColor = texture2D(nightTexture, st);
  gl_FragColor = mix(dayColor, nightColor, mixFactor) * vertColor;
}
```

In the fragment shader, we also define another uniform variable, mixFactor, where we store a value between 0 and 1 to mix both textures in different proportions depending on the horizontal position of the mouse. GLSL provides a built-in function precisely called mix that allows us to create a linear interpolation between two values (in this case, the texels sampled from the day and night textures). Figure 15-15 shows the output of this code.

Figure 15-15. *Sphere textured with a mixture of day and night maps of the Earth*

However, for our day-to-night effect, we need to mix the two textures according to the angle around the Earth's sphere. The values corresponding to the sunrise and sunset are where the textures should transition into one another. These angles can be calculated in the vertex shader using the sphere's (x, y, z) coordinates we receive from Processing in the position attribute. By applying the formulas to convert between Cartesian and spherical coordinates (https://en.wikipedia.org/wiki/Spherical_coordinate_system#Cartesian_coordinates), we should be able to get the azimuthal angle θ, which corresponds to the longitude in the geographical coordinate system. We do this in Listing 15-16.

Listing 15-16. Calculation of spherical coordinates in the vertex shader

```
PShape earth;
PImage earthDay;PShape earth;
PShader earthShader;

float viewRotation;

void setup() {
  fullScreen(P3D);

  PImage earthDay = loadImage("Solarsystemscope_texture_2k_earth_daymap.jpg");
  PImage earthNight = loadImage("Solarsystemscope_texture_2k_earth_nightmap.jpg");
```

```
  PGraphicsOpenGL pgl = (PGraphicsOpenGL)g;
  pgl.textureWrap(REPEAT);

  earthShader = loadShader("EarthFrag.glsl", "EarthVert.glsl");
  earthShader.set("dayTexture", earthDay);
  earthShader.set("nightTexture", earthNight);
  earthShader.set("width", width);
  earthShader.set("height", height);

  earth = createShape(SPHERE, 400);
  earth.setStroke(false);
}

void draw() {
  background(0);

  float targetAngle = map(mouseX, 0, width, 0, TWO_PI);
  viewRotation += 0.05 * (targetAngle - viewRotation);

  translate(width/2, height/2);

  shader(earthShader);
  pushMatrix();
  rotateY(viewRotation);
  shape(earth);
  popMatrix();
}
PImage earthNight;
PShader earthShader;

float viewRotation;

void setup() {
  fullScreen(P3D);

  earthDay = loadImage("Solarsystemscope_texture_2k_earth_daymap.jpg");
  earthNight = loadImage("Solarsystemscope_texture_2k_earth_nightmap.jpg");

  earthShader = loadShader("EarthFrag.glsl");
  earthShader.set("dayTexture", earthDay);
  earthShader.set("nightTexture", earthNight);

  earth = createShape(SPHERE, 400);
  earth.setStroke(false);
}

void draw() {
  background(0);

  earthShader.set("mixFactor",  map(mouseX, 0, width, 0, 1));
  viewRotation += 0.001;
```

```
  translate(width/2, height/2);

  shader(earthShader);
  pushMatrix();
  rotateY(viewRotation);
  shape(earth);
  popMatrix();
}
```

EarthFrag.glsl

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

#define TWO_PI 6.2831853076

uniform sampler2D dayTexture;
uniform sampler2D nightTexture;
varying vec4 vertColor;
varying vec4 vertTexCoord;
varying float azimuth;

void main() {
  vec2 st = vertTexCoord.st;
  vec4 dayColor = texture2D(dayTexture, st);
  vec4 nightColor = texture2D(nightTexture, st);
  gl_FragColor = mix(dayColor, nightColor, azimuth / TWO_PI) * vertColor;
}
```

EarthVert.glsl

```
uniform mat4 transform;
uniform mat4 modelview;
uniform mat4 texMatrix;

attribute vec4 position;
attribute vec4 color;
attribute vec2 texCoord;

varying vec4 vertColor;
varying vec4 vertTexCoord;
varying float azimuth;

uniform int width;
uniform int height;

#define PI 3.1415926538
```

```
void main() {
  gl_Position = transform * position;
  vec3 v = position.xyz - vec3(float(width)/2.0, float(height)/2.0, 0.0);
  azimuth = PI - sign(v.z) * acos(v.x / length(v.xz));
  vertColor = color;
  vertTexCoord = texMatrix * vec4(texCoord, 1.0, 1.0);
}
```

The calculation of the azimuth angle uses the vector v, which is the position minus (width/2, height/2, 0). The reason for this is because the formula assumes that the (x, y, z) coordinates are centered at zero, but in Processing, the origin of coordinates is located at the upper left corner, and a translation to (width/2, height/2, 0) is applied to have the Earth sphere rendered at the center of the screen. Because of this, the position coordinates are centered at (width/2, height/2, 0); the subtraction in vertex shader gets us the zero-centered coordinates that we can use to calculate the azimuth angle from the x and z values (since the equatorial plane of the Earth is the XZ plane in Processing). The calculation sign(v.z) * acos(v.x / length(v.xz)) returns an angle between -π and +π, and subtracting that from the PI constant (defined at the top of the vertex shader since GLSL does not include a built-in constant for π) gives us a value between 0 and 2π. This value is passed down to the fragment shader in the varying float azimuth, which is divided by the constant TWO_PI defined separately to have the mixing factor between 0 and 1 in mix(dayColor, nightColor, azimuth / TWO_PI). This is similar to what we had in the previous listing, but now the interpolation uses the azimuth angle, so the day and night textures are mixed following the longitude around the Earth. This takes us closer to our intended effect, as seen in Figure 15-16.

We need to add the rotation of the Earth, so the transition areas between day and night move around the sphere as time passes on. Instead of rotating the sphere, we could alternatively translate the texture coordinates along the horizontal direction, which would rotate the texture around the static object; rotating the vertices makes the calculations more involved because the azimuth angle would vary alongside the rotation. Listing 15-17 shows only the changes we need to make to our previous sketch to have this implemented; the vertex shader is not included since it remains the same from Listing 15-16.

Figure 15-16. *Interpolation between day and night Earth maps using the azimuth angle*

Listing 15-17. Calculation of spherical coordinates in the vertex shader

```
PShape earth;
PShader earthShader;

float viewRotation;
float earthRotation;

void setup() {
  ...
  PGraphicsOpenGL pgl = (PGraphicsOpenGL)g;
  pgl.textureWrap(REPEAT);
}

void draw() {
  background(0);

  earthShader.set("earthRotation", earthRotation % TWO_PI);
  earthRotation += 0.001;
  ...
}
```

EarthFrag.glsl

```
...
uniform float earthRotation;

void main() {
  float s = vertTexCoord.s;
  float t = vertTexCoord.t;
  vec2 st = vec2(s - earthRotation / TWO_PI, t);
  ...
}
```

We added a new variable in the sketch code, earthRotation, to store the rotation angle of the Earth as it revolves around its axis. We increase this angle by a small amount in each frame but make it vary only between 0 and TWO_PI when setting the value of the corresponding uniform in the shader by taking the modulus with earthRotation % TWO_PI. In this way, if, for example, earthRotation is 3π, the module would return π (which represents the same angle). Meanwhile, in the fragment shader, we add the earthRotation value, normalized to the 0–1 range by dividing it by TWO_PI, to the s coordinate of the vertex texture coordinate (the minus sign is to ensure that the rotation happens in the correct direction from east to west). We also use a new function in the shader code, textureWrap(REPEAT), so the texture coordinates wrap around (0, 1) as the values change along the s direction when adding earthRotation in the shader. Even though the angle is normalized, we could still have an s value outside (0, 1), for example, 0.5 + 07 = 1.2, but this would get wrapped around to 0.2, thanks to the textureWrap() setting. This is an advanced function that's not part of the regular Processing API, so to access it, we need the underlying renderer object g as a PGraphicsOpenGL variable, which is the renderer type when using P2D or P3D, and only then we can call the advanced functions available in the OpenGL renderers.

With all of this, we finally have our day-to-night effect fully working! We would need a few last touches to have the rotation of the Earth linked to the actual "real" time and the view of the Earth showing the geographical location of the user. For the first feature, linking the rotation to the real time, we can use the java.time package that provides utilities to retrieve the current Coordinated Universal Time (UTC). For the

second feature, setting the view according to the users' location, we can use the Ketai library as we did in Chapter 9. None of this affects the shaders from the last listing; the changes in the sketch code are shown in Listing 15-18.

Listing 15-18. Calculation of spherical coordinates in the vertex shader

```
import java.time.Instant;
import java.time.ZoneOffset;
import ketai.sensors.*;
...

KetaiLocation location;
float longitude;

void setup() {
  ...
  location = new KetaiLocation(this);
}
void draw() {
background(0);

  Instant instant = Instant.now();
  int hour = instant.atZone(ZoneOffset.UTC).getHour();
  int minute = instant.atZone(ZoneOffset.UTC).getMinute();
  float utcTime = 60 * hour + minute;
  earthRotation = map(utcTime, 0, 1439, HALF_PI - PI, HALF_PI + PI);
  earthShader.set("earthRotation", earthRotation);
  earthShader.set("earthRotation", earthRotation % TWO_PI);

  viewRotation = longuitude - earthRotation;
  ...
}

void onLocationEvent(double lat, double lon) {
  longuitude = radians((float)lon + 210);
}
```

We see how we can retrieve the hour and minute of the current UTC using the instant.atZone(ZoneOffset.UTC).getHour() and getMinute() calls once we have the instant object from Instant.now(). Then, we convert the hour and minute values into the total amount of minutes since midnight with float utcTime = 60 * hour + minute and then map that value onto the Earth rotation angle. As the Earth revolves around itself, we want to view it from a vantage point flying over the user's geographical location. With Ketai, it is easy to get the (latitude, longitude) values from the GPS (remember to set the appropriate permissions with the sketch permissions tool) in the onLocationEventHandler() function. The latitude is not used since we only need the longitude to calculate the view rotation together with the Earth's rotation. By doing all of this, the output of our sketch is kind of a "world clock" since it visualizes the zone of daylight at the location of the user running the sketch and keeps updating it in real time. The only inaccuracy in this last version of the code is that it does not incorporate the declination angle that varies seasonally as the Earth tilts on its axis of rotation and the rotation of the Earth around the Sun (https://solarsena.com/solar-declination-angle-calculator/). Therefore, our rendering is only correct during the equinoxes, which is when the declination angle of the Earth is exactly zero. Adding this should not be too hard, and it is left as an exercise for the readers. ☺

We could even run our sketch as a live wallpaper, so it updates continuously in the background of our screen, showing the current time of the day or night. In this case, we could add a frameRate(1) call to setup, to make it refresh the screen just once every second to reduce battery usage. Running it faster than that would not be necessary in any case since the Earth's rotation is virtually unnoticeable in real time from second to second. Figure 15-17 shows some typical outputs of the wallpaper during different times of the day.

As the final step, we should create a set of icons for our day-to-night live wallpaper so it can be packaged and distributed on the Google Play Store!

Figure 15-17. *Final version of the day-to-night sketch, installed as a live wallpaper*

Summary

We only scratched the surface of a fascinating topic, shader programming, but we were able to cover the main types of shaders we can use in Processing. These shaders allow us to draw colored, textured, and lit scenes, and we apply them to create a live wallpaper that renders the Earth's day-to-night changes in real time. Writing GLSL shaders is not easy, it requires math skills especially in vector and matrix algebra, as well as a good understanding of how the graphics pipeline in modern GPUs works, but all of that can be gained with enough practice and dedication. Processing provides several functions to incorporate shaders into 2D and 3D sketches, which should be useful for those users who want to delve deeper into the world of shaders.