Chapter 20

Programming 3D Graphics with OpenGL

In this chapter, we will talk extensively about working with the OpenGL ES graphics API on the Android Platform. OpenGL ES is a version of OpenGL that is optimized for embedded systems and other low-powered devices such as mobile phones.

The Android Platform supports OpenGL ES 1.0 and OpenGL ES 2.0. OpenGL ES 2.0 is available only from API level 8 corresponding to Android SDK release 2.2. As of this writing, there are some issues with using Java bindings to OpenGL ES 2.0. See the notes and recommendations on OpenGL ES 2.0 in a separate section towards the end of this chapter. The primary issue is the lack of support for OpenGL ES 2.0 in the emulator. Android 3.0 strengthened the opportunities for OpenGL by introducing Renderscript. Renderscript is meant for better performance by allowing to run native code programmed in a "c" like language. This code even may be executed on the GPU (Graphical Processing Unit). Renderscript is also designed to allow for cross-platform compatibility. When performance is not critical, programmers are still advised to use the Java bindings for much of the OpenGL work. Due to time limitations we didn't cover the Renderscript in this edition of the book. We have provided a reference URL to a Renderscript (from Google) programming guide at the end of this chapter.

The Android SDK distribution comes with a number of OpenGL ES samples. However, the documentation on how to get started with OpenGL ES is minimal to nonexistent in the SDK. The underlying assumption is that OpenGL ES is an open standard that programmers can learn from outside of Android. As a result, Android online OpenGL resources and the Android OpenGL code samples assume you're already familiar with OpenGL.

In this chapter, we will help you with these minor roadblocks. With few OpenGL prerequisites, by the end of this chapter, you'll be comfortable with programming in OpenGL. We will do this by introducing almost no mathematics (unlike many OpenGL books).

In the first section of the chapter, we'll provide an overview of OpenGL, OpenGL ES, and some competing standards.

In the second section, we will explain the theory behind OpenGL. This is a critical section to read if you are new to OpenGL. In this section, we will cover OpenGL coordinates, its idea of a camera, and the essential OpenGL ES drawing APIs.

In the third section, we will explain how you interact with the OpenGL ES API on Android. This section covers GLSurfaceView and the Renderer interface and how they work together to draw using OpenGL. In one of our simple examples, we will draw a simple triangle to show how drawing is impacted by changing the OpenGL scene setup APIs.

> **NOTE:** The OpenGL camera concept is similar but distinct from the Camera class in Android's graphics package, which you learned about in Chapter 6. Whereas Android's Camera object from the graphics package simulates 3D-like viewing capabilities by projecting a 2D view moving in 3D space, the OpenGL camera is a paradigm that represents a virtual viewing point. In other words, it models a real-world scene through the viewing perspective of an observer looking through a camera. You'll learn more in the subsection "Understanding the Camera and Coordinates" under "Using OpenGL ES." Both cameras are still separate from the handheld device's physical camera that you use to take pictures or shoot video.

In the fourth section, we will take you a bit deeper into OpenGL ES and introduce the idea of shapes. We will also cover textures and show you how to draw multiple figures during a single draw method. We will then cover the support for OpenGL ES 2.0 by briefly introducing OpenGL shaders and a quick sample. Please note up front that OpenGL ES 2.0 can only be tested on a real device.

We conclude the chapter with a list of resources we found as we researched material for this chapter.

So, let's look into the history and background of OpenGL!

Understanding the History and Background of OpenGL

OpenGL (originally called Open Graphics Library) is a 2D and 3D graphics API that was developed by Silicon Graphics, Inc. (SGI) for its UNIX workstations. Although SGI's version of OpenGL has been around for a long time, the first standardized spec of OpenGL emerged in 1992. Now widely adopted on all operating systems, the OpenGL standard forms the basis of much of the gaming, computer-aided design (CAD), and even virtual reality (VR) industries.

The OpenGL standard is currently being managed by an industry consortium called The Khronos Group (www.khronos.org), founded in 2000 by companies such as NVIDIA, Sun Microsystems, ATI Technologies, and SGI. You can learn more about the OpenGL spec at the consortium's web site at:

www.khronos.org/opengl/

The official documentation page for OpenGL is available here:

www.opengl.org/documentation/

As you can see from this documentation page, you have access to books and online resources dedicated to OpenGL. Of these, the gold standard is *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*, also known as the "red book" of OpenGL. You can find an online version of this book here:

www.glprogramming.com/red/

This book is quite good and quite readable. We did have some difficulty, however, unraveling the nature of *units* and *coordinates* that are used to draw. We'll try to clarify these important ideas regarding what you draw and what you see in OpenGL. These ideas center on setting up the OpenGL camera and a *viewing box*, also known as a *viewing volume* or *frustum*.

OpenGL ES

The Khronos Group is also responsible for two additional standards that are tied to OpenGL: OpenGL ES, and the EGL Native Platform Graphics Interface (known simply as EGL). As we mentioned, OpenGL ES is a smaller version of OpenGL intended for embedded systems.

NOTE: Java Community Process is also developing an object-oriented abstraction for OpenGL for mobile devices called Mobile 3D Graphics API (M3G). We will give you a brief introduction to M3G in the subsection "M3G: Another Java ME 3D Graphics Standard."

The EGL standard is essentially an enabling interface between the underlying operating system (OS) and the rendering APIs offered by OpenGL ES. Because OpenGL and OpenGL ES are general-purpose interfaces for drawing, each OS needs to provide a standard hosting environment for OpenGL and OpenGL ES to interact with. Android SDK, starting with the 1.5 release, hides these platform specifics quite well. You will learn about this in the second section titled "Interfacing OpenGL ES with Android."

The target devices for OpenGL ES include cell phones, appliances, and even vehicles. Because OpenGL ES has to be much smaller than OpenGL, many convenient functions have been removed. For example, drawing rectangles is not directly supported in OpenGL ES; you have to draw two triangles to make a rectangle.

As you start exploring Android's support for OpenGL, you'll focus primarily on OpenGL ES and its bindings to the Android OS through Java and EGL. You can find the documentation for OpenGL ES here:

www.khronos.org/opengles/documentation/opengles1_0/html/index.html

We kept returning to this reference as we developed this chapter because it identifies and explains each OpenGL ES API and describe the arguments for each. You'll find these APIs similar to Java APIs, and we'll introduce you to the key ones in this chapter.

OpenGL ES and Java ME

OpenGL ES, like OpenGL, is a C-based, flat API. Because the Android SDK is a Java-based programming API, you need a Java binding to OpenGL ES. Java ME has already defined this binding through JSR 239: Java Binding for the OpenGL ES API. JSR 239 itself is based on JSR 231, which is a Java binding for OpenGL 1.5. JSR 239 could have been strictly a subset of JSR 231, but it's not because it must accommodate some extensions to OpenGL ES that are not in OpenGL 1.5.

You can find the documentation for JSR 239 here:

http://java.sun.com/javame/reference/apis/jsr239/

This reference will give you a sense of the APIs available in OpenGL ES. It also provides valuable information about the following packages:

- javax.microedition.khronos.egl
- javax.microedition.khronos.opengles
- java.nio

The nio package is necessary because the OpenGL ES implementations take only byte streams as inputs for efficiency reasons. This nio package defines many utilities to prepare native buffers for use in OpenGL. You will see some of these APIs in action in the "glVertexPointer and Specifying Drawing Vertices" subsection under "Using OpenGL ES."

You can find documentation (although quite minimal) of the Android SDK's support for OpenGL at the following URL:

http://developer.android.com/guide/topics/graphics/opengl.html

On this page, the documentation indicates that the Android implementation mostly parallels JSR 239 but warns that it might diverge from it in a few places.

M3G: Another Java ME 3D Graphics Standard

JSR 239 is merely a Java binding on a native OpenGL ES standard. As mentioned briefly in the "OpenGL ES" subsection, Java provides another API to work with 3D graphics on mobile devices: M3G. This object-oriented standard is defined in JSR 184 and JSR 297, the latter being more recent. As per JSR 184, M3G serves as a lightweight, object-oriented, interactive 3D graphics API for mobile devices.

The object-oriented nature of M3G separates it from OpenGL ES. For details, visit the home page for JSR 184 at the following URL:

www.jcp.org/en/jsr/detail?id=184

The APIs for M3G are available in the Java package named

javax.microedition.m3g.*;

M3G is a higher-level API compared to OpenGL ES, so it should be easier to learn. However, the jury is still out on how well it will perform on handhelds. As of now, Android does not support M3G.

So far, we have laid out the options available in the OpenGL space for handheld devices. We have talked about OpenGL ES and also briefly about the M3G standard. We will now focus on understanding the fundamentals of OpenGL.

Fundamentals of OpenGL

This section will help you understand the concepts behind OpenGL and the OpenGL ES API. We'll explain all the key APIs. To supplement the information from this chapter, you might want to refer to the "Resources" section towards the end of this chapter. The resources there include the red book, JSR 239 documentation, and The Khronos Group API reference.

> **NOTE:** As you start using the OpenGL resources, you'll notice that some of the APIs are not available in OpenGL ES. This is where The Khronos Group's *OpenGL ES Reference Manual* comes in handy.

We will cover the following APIs in a fair amount of detail because they're central to understanding OpenGL and OpenGL ES:

- glVertexPointer
- glDrawElements
- glColor
- glClear
- gluLookAt
- glFrustum
- glViewport

As we cover these APIs, you'll learn how to

- Use the essential OpenGL ES drawing APIs.
- Clear the palette.
- Specify colors.
- Understand the OpenGL camera and coordinates.

Essential Drawing with OpenGL ES

In OpenGL, you draw in 3D space. You start out by specifying a series of points, also called vertices. Each of these points will have three values: one for the x coordinate, one for the y coordinate, and one for the z coordinate.

These points are then joined together to form a shape. You can join these points into a variety of shapes called *primitive shapes*, which include points, lines, and triangles in OpenGL ES. Note that in OpenGL, primitive shapes also include rectangles and polygons. As you work with OpenGL and OpenGL ES, you will continue to see differences whereby the latter has fewer features than the former. Here's another example: OpenGL allows you to specify each point separately, whereas OpenGL ES allows you to specify them only as a series of points in one fell swoop. However, you can often simulate OpenGL ES's missing features through other, more primitive features. For instance, you can draw a rectangle by combining two triangles.

OpenGL ES offers two primary methods to facilitate drawing:

- glVertexPointer
- glDrawElements

> **NOTE**: We'll use the terms "API" and "method" interchangeably when we talk about the OpenGL ES APIs.

You use glVertexPointer to specify a series of points or vertices, and you use glDrawElements to draw them using one of the primitive shapes mentioned earlier. We'll describe these methods in more detail, but first let's go over some nomenclature around the OpenGL API names.

The names of OpenGL APIs all begin with gl. Following gl is the method name. The method name is followed by an optional number such as 3, which points to either the number of dimensions—such as (x,y,z)—or the number of arguments. The method name is then followed by a data type such as f for float. (You can refer to any of the OpenGL online resources to learn the various data types and their corresponding letters.)

There's one more convention. If a method takes an argument either as a byte (b) or a float (f), then the method will have two names: one ending with b, and one ending with f.

Let's now look at each of the two drawing-related methods, starting with glVertexPointer.

glVertexPointer and Specifying Drawing Vertices

The glVertexPointer method is responsible for specifying an array of points to be drawn. Each point is specified in three dimensions, so each point will have three values: x, y, and z. Listing 20–1 shows how to specify three points in an array.

Listing 20–1. *Vertex Coordinates Example for an OpenGL Triangle*

```
float[] coords = {
     -0.5f, -0.5f, 0,   //p1: (x1,y1,z1)
      0.5f, -0.5f, 0,    //p2: (x1,y1,z1)
      0.0f,  0.5f, 0     //p3: (x1,y1,z1)
};
```

The structure in Listing 20–1 is a contiguous set of floats kept in a Java-based float array. Don't worry about typing or compiling this code anywhere yet—our goal at this point is just to give you an idea of how these OpenGL ES methods work. We will give you the working examples and code when we develop a test harness later to draw simple figures. We have also given you a link to a downloadable project in the reference section at the end of this chapter.

In Listing 20–1, you might be wondering what units are used for the coordinates in points p1, p2, and p3. The short answer is, as you model your 3D space, these coordinate units can be anything you'd like. But subsequently you will need to specify something called a *bounding volume* (or *bounding box*) that quantifies these coordinates.

For example, you can specify the bounding box as a cube with 5-inch sides or a cube with 2-inch sides. These coordinates are also known as *world coordinates* because you are conceptualizing your world independent of the physical device's limitations. We will explain these coordinates more in the subsection "Understanding the Camera and Coordinates." For now, assume that you are using a cube that is 2 units across all its sides and centered at (x=0,y=0,z=0). In other words, the center is at the center of the cube and the sides of the cube are 1 unit apart from the center.

> **NOTE**: The terms *bounding volume*, *bounding box*, *viewing volume*, *viewing box*, and *frustum* all refer to the same concept: the pyramid-shaped 3D volume that determines what is visible onscreen. You'll learn more in the "glFrustum and the Viewing Volume" subsection under "Understanding the Camera and Coordinates."

You can also assume that the origin is at the center of the visual display. The z axis will be negative going into the display (away from you) and positive coming out of the display (toward you); x will go positive as you move right and negative as you move left. However, these coordinates will also depend on the direction from which you are viewing the scene.

To draw the points in Listing 20–1, you need to pass them to OpenGL ES through the glVertexPointer method. For efficiency reasons, however, glVertexPointer takes a native buffer that is language-agnostic rather than an array of Java floats. For this, you need to convert the Java-based array of floats to an acceptable C-like native buffer. You'll need to use the java.nio classes to convert the float array into the native buffer. Listing 20–2 shows an example of using nio buffers.

Listing 20–2. *Creating NIO Float Buffers*

```
jva.nio.ByteBuffer vbb = java.nio.ByteBuffer.allocateDirect(3 * 3 * 4);
vbb.order(ByteOrder.nativeOrder());
java.nio.FloatBuffer mFVertexBuffer = vbb.asFloatBuffer();
```

In Listing 20–2, the byte buffer is a buffer of memory ordered into bytes. Each point has three floats because of the three axes, and each float is 4 bytes. So together you get 3 * 4 bytes for each point. Plus, a triangle has three points. So you need 3 * 3 * 4 bytes to hold all three float points of a triangle.

Once you have the points gathered into a native buffer, you can call glVertexPointer, as shown in Listing 20–3.

Listing 20–3. *glVertexPointer API Definition*

```
glVertexPointer(
    // Are we using (x,y) or (x,y,z) in each point
    3,
    // each value is a float value in the buffer
    GL10.GL_FLOAT,
    // Between two points there is no space
   0,
   // pointer to the start of the buffer
   mFVertexBuffer);
```

Let's talk about the arguments of glVertexPointer method. The first argument tells OpenGL ES how many dimensions there are in a point or a vertex. In this case, we specified 3 for x, y, and z. You could also specify 2 for just x and y. In that case, z would be zero. Note that this first argument is not the number of points in the buffer, but the number of dimensions used. So if you pass 20 points to draw a number of triangles, you will not pass 20 as the first argument; you would pass 2 or 3, depending on the number of dimensions used.

The second argument indicates that the coordinates need to be interpreted as floats. The third argument, called a stride, points to the number of bytes separating each point. In this case, it is zero because one point immediately follows the other. Sometimes you can add color attributes as part of the buffer after each point. If you want to do so, you'd use a stride to skip those as part of the vertex specification. The last argument is the pointer to the buffer containing the points.

Now that you know how to set up the array of points to be drawn, let's see how to draw this array of points using the glDrawElements method.

glDrawElements

Once you specify the series of points through glVertexPointer, you use the glDrawElements method to draw those points with one of the primitive shapes that OpenGL ES allows. Note that OpenGL is a state machine. It remembers the values set by one method when it invokes the next method in a cumulative manner. So you don't need to explicitly pass the points set by glVertexPointer to glDrawElements. glDrawElements will implicitly use those points. Listing 20–4 shows an example of this method with possible arguments.

Listing 20–4. *Example of glDrawElements*

```
glDrawElements(
      // type of shape
      GL10.GL_TRIANGLE_STRIP,
      // Number of indices
      3,
     // How big each index is
     GL10.GL_UNSIGNED_SHORT,
     // buffer containing the 3 indices
     mIndexBuffer);
```

The first argument indicates the type of geometrical shape to draw: GL_TRIANGLE_STRIP signifies a triangle strip. Other possible options for this argument are points only (GL_POINTS), line strips (GL_LINE_STRIP), lines only (GL_LINES), line loops (GL_LINE_LOOP), triangles only (GL_TRIANGLES), and triangle fans (GL_TRIANGLE_FAN).

The concept of a STRIP in GL_LINE_STRIP and GL_TRIANGLE_STRIP is to add new points while making use of the old ones. By doing so, you can avoid specifying all the points for each new object. For example, if you specify four points in an array, you can use strips to draw the first triangle out of (1,2,3) and the second one out of (2,3,4). Each new point will add a new triangle. (Refer to the OpenGL red book for more details.) You can also vary these parameters to see how the triangles are drawn as you add new points.

The idea of a FAN in GL_TRIANGLE_FAN applies to triangles where the first point is used as a starting point for all subsequent triangles. So you're essentially making a fan- or circle-like object with the first vertex in the middle. Suppose you have six points in your array: (1,2,3,4,5,6). With a FAN, the triangles will be drawn at (1,2,3), (1,3,4), (1,4,5), and (1,5,6). Every new point adds an extra triangle, similar to the process of extending a fan or unfolding a pack of cards.

The rest of the arguments of glDrawElements involve the method's ability to let you reuse point specification. For example, a square contains four points. Each square can be drawn as a combination of two triangles. If you want to draw two triangles to make up the square, do you have to specify six points? No. You can specify only four points and refer to them six times to draw two triangles. This process is called *indexing into the point buffer*.

Here is an example:

```
Points: (p1, p2, p3, p4)
Draw indices (p1, p2, p3,    p2,p3,p4)
```

Notice how the first triangle comprises p1, p2, p3 and the second one comprises p2, p3, p4. With this knowledge, the second argument of glDrawElements identifies how many indices there are in the index buffer.

The third argument to glDrawElements (see Listing 20–4) points to the type of values in the index array, whether they are unsigned shorts (GL_UNSIGNED_SHORT) or unsigned bytes (GL_UNSIGNED_BYTE).

The last argument of glDrawElements points to the index buffer. To fill up the index buffer, you need to do something similar to what you did with the vertex buffer. Start with a Java array and use the java.nio package to convert that array into a native buffer.

Listing 20–5 shows some sample code that converts a short array of {0,1,2} into a native buffer suitable to be passed to glDrawElements.

Listing 20–5. *Converting Java Array to NIO Buffers*

```
//Figure out how you want to arrange your points
short[] myIndecesArray = {0,1,2};

//get a short buffer
java.nio.ShortBuffer mIndexBuffer;

//Allocate 2 bytes each for each index value
ByteBuffer ibb = ByteBuffer.allocateDirect(3 * 2);
ibb.order(ByteOrder.nativeOrder());
mIndexBuffer = ibb.asShortBuffer();

//stuff that into the buffer
for (int i=0;i<3;i++)
{
  mIndexBuffer.put(myIndecesArray[i]);
}
```

Now that you've seen mIndexBuffer at work in Listing 20–5, you can revisit Listing 20–4 and better understand how the index buffer is created and manipulated.

NOTE: Rather than create any new points, the index buffer merely indexes into the array of points indicated through the glVertexPointer. This is possible because OpenGL remembers the assets set by the previous calls in a stateful fashion.

Now we'll look at two commonly used OpenGL ES methods: glClear and glColor.

glClear

You use the glClear method to erase the drawing surface. Using this method, you can reset the color, depth, and the type of stencils used. You specify which element to reset by the constant that you pass in: GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, or GL_STENCIL_BUFFER_BIT.

The color buffer is responsible for the pixels that are visible, so clearing it is equivalent to erasing the surface of any colors. The depth buffer is related to the pixels that are visible in a 3D scene, with depth referring to how far or close the object is.

The stencil buffer is a bit advanced to cover here, except to say this: you use it to create visual effects based on some dynamic criteria, and you use glClear to erase it.

NOTE: A stencil is a drawing template that you can use to replicate a drawing many times. For example, if you are using Microsoft Office Visio, all the drawing templates that you save as *.vss files are stencils. In the noncomputer drawing world, you create a stencil by cutting out a pattern in a sheet of paper or some other flat material. Then you can paint over that sheet and remove it, creating the impression that results in a replication of that drawing. What you see depends on what stencil or stencils are active. Clearing all of them will make everything drawn visible.

For your purposes, you can use this code to clear the color buffer:

```
//Clear the surface of any color
gl.glClear(gl.GL_COLOR_BUFFER_BIT);
```

Now let's talk about attaching a default color to what gets drawn.

glColor

You use glColor to set the default color for the subsequent drawing that takes place. In the following code segment, the method glColor4f sets the color to red:

```
//Set the current color
glColor4f(1.0f, 0, 0, 0.5f);
```

Recall the discussion about method nomenclature: 4f refers to the four arguments that the method takes, each of which is a float. The four arguments are components of red, green, blue, and alpha (color gradient). The starting values for each are (1,1,1,1). In this case, the color has been set to red with half a gradient (specified by the last alpha argument).

Although we have covered the basic drawing APIs, we still need to address a few things regarding the coordinates of the points that you specify in 3D space. The next subsection explains how OpenGL models a real-world scene through the viewing perspective of an observer looking through a camera.

Understanding OpenGL Camera and Coordinates

As you draw in 3D space, you ultimately must project the 3D view onto a 2D screen—much like capturing a 3D scene using a camera in the real world. This symbolism is formally recognized in OpenGL, so many concepts in OpenGL are explained in terms of a camera.

As you will see in this section, the part of your drawing that becomes visible depends on the location of the camera, the direction of the camera lens, the orientation of the camera (such as upside down or tilted), the zoom level, and the size of the capturing "film."

These aspects of projecting a 3D picture onto a 2D screen are controlled by three methods in OpenGL:

- gluLookAt controls the direction of the camera.
- glFrustum controls the viewing volume or zoom or the distance (from and to) you care about.
- glViewport controls the size of the screen or the size of the camera's film.

You won't be able to program anything in OpenGL unless you understand the implications of these three APIs. Let's elaborate on the camera symbolism further to explain how these three APIs affect what you see on an OpenGL screen. We will start with gluLookAt.

gluLookAt and the Camera Symbolism

Imagine you are taking photographs of a landscape involving flowers, trees, streams, and mountains. You arrive at a meadow; the scene that lies before you is equivalent to what you would like to draw in OpenGL. You can make these drawings big, like the mountains, or small, like the flowers—as long as they are all proportional to one another. The coordinates you'll use for these

drawings, as we hinted at earlier, are called *world coordinates*. Under these coordinates, you can establish a line to be 4 units long on the x axis by setting your points as (0,0,0) to (4,0,0).

As you prepare to take a photograph, you find a spot to place your tripod. Then you hook up the camera to the tripod. The location of your camera—not the tripod, but the camera itself—becomes the origin of your camera in the world. So you will need to take a piece of paper and write down this location, which is called the *eye point*.

If you don't specify an eye point, the camera is located at (0,0,0), which is the exact center of your screen. Usually you want to step away from the origin so that you can see the (x,y) plane that is sitting at the origin of z = 0. For argument's sake, suppose you position the camera at (0,0,5). This would move the camera off your screen toward you by 5 units.

You can refer to Figure 20–1 to visualize how the camera is placed.

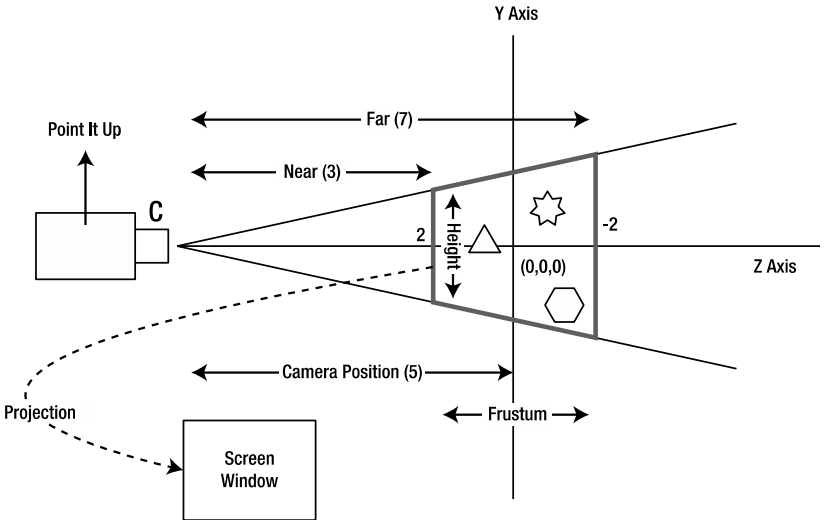

Figure 20–1. *OpenGL viewing concepts using the camera analogy*

Looking at Figure 20–1 you might wonder why the axes in the figure are y and z and not x and y. This is because we use the convention that the OpenGL camera looks down on the z axis if your normal plane of scene is the xy plane. This convention works fine because we usually associate the z axis as the axis of depth.

Once you place the camera, you start looking ahead or forward to see which portion of the scene you want to capture. You will position the camera in the direction you are looking. This far-off point that you are looking at is called a *view point* or a *look-at point*. This point specification is really a specification of the direction. If you specify your view point as (0,0,0), then the camera is looking along the z axis toward the origin from a distance of 5 units, assuming the camera is positioned at (0,0,5). You can see this in Figure 20–1 where the camera is looking down the z axis.

Imagine further that there is a rectangular building at the origin. You want to look at it not in a portrait fashion, but in a landscape fashion. What do you have to do? You obviously can leave the camera in the same location and still point it toward the origin, but now you need to turn the camera by 90 degrees (similar to tilting your head to see sideways). This is the *orientation* of the camera, as the camera is fixed at a given eye point and looking at a specific look-at point or direction. This orientation is called the *up vector*.

The up vector simply identifies the orientation of the camera (up, down, left, right, or at an angle). This orientation of the camera is also specified using a point. Imagine a line from the origin—not the camera origin, but the world-coordinate origin—to this point. Whatever angle this line subtends in three dimensions at the origin is the orientation of camera.

For example, an up vector for a camera might look like (0,1,0) or even (0,15,0), both of which would have the same effect. The point (0,1,0) is a point away from the origin along the y axis going up. This means you position the camera upright. If you use (0,-1,0), you would position the camera upside down. In both cases, the camera is still at the same point (0,0,5) and looking at the same origin (0,0,0). You can summarize these three coordinates like this:

- (0,0,5): Eye point (location of the camera)
- (0,0,0): Look-at point (direction the camera is pointing)
- (0,1,0): Up vector (whether the camera is up, down, or slanted)

You will use the gluLookAt method to specify these three points—the eye point, the look-at point, and the up vector, like so:

```
gluLookAt(gl, 0,0,5,    0,0,0,   0,1,0);
```

The arguments are as follows: the first set of coordinates belongs to the eye point, the second set of coordinates belongs to the look-at point, and the third set of coordinates belongs to the up vector with respect to the origin.

Let's now look at the viewing volume.

glFrustum and the Viewing Volume

You might have noticed that none of the points describing the camera position using gluLookAt deal with size. They deal only with positioning, direction, and orientation. How can you tell the camera where to focus? How far away is the subject you are trying to capture? How wide and how tall is the subject area? You use the OpenGL method glFrustum to specify the area of the scene that you are interested in.

If you were to imagine yourself sitting at a play, then the stage is your viewing volume. You really don't need to know what happens outside of that stage. However, you do care about the dimensions of this stage because you want to observe all that goes on upon/inside that stage.

Think of the scene area as bounded by a box, also called the *frustum* or *viewing volume* (this is the area marked by the bold border in the middle of Figure 20–1). Anything inside the box is captured, and anything outside the box is clipped and ignored. So how do you specify this viewing box? You first decide on the *near point*, or the distance between the camera and the beginning of the box. Then you can choose a *far point*, which is the distance between the camera and the end of the box. The distance between the near and far points along the z axis is the depth of the box. If you specify a near point of 50 and a far point of 200, then you will capture everything between those points and your box depth will be 150. You will also need to specify the left side of the box, the right side of the box, the top of the box, and the bottom of the box along the imaginary *ray* that joins the camera to the look-at point.

In OpenGL, you can imagine this box in one of two ways. One is called a *perspective projection*, which involves the frustum we've been talking about. This view, which simulates a natural camera-like function, involves a pyramidal structure in which the far plane serves as the base and the camera serves as the apex. The near plane cuts off the top of the pyramid, forming the frustum between the near plane and the far plane.

The other way to imagine the box involves thinking of it as a cube. This second scenario is called *orthographic projection* and is suited for geometrical drawings that need to preserve sizes despite the distance from the camera.

Listing 20–6 shows how to specify the frustum for our example.

Listing 20–6. *Specifying a Frustum through glFrustum*

```
//calculate aspect ratio first
float ratio = (float) w / h;

//indicate that we want a perspective projection
glMatrixMode(GL10.GL_PROJECTION);

//Specify the frustum: the viewing volume
gl.glFrustumf(
   -ratio,    // Left side of the viewing box
   ratio,     // right side of the viewing box
   1,         // top of the viewing box
   -1,        // bottom of the viewing box
   3,         // how far is the front of the box from the camera
   7);        // how far is the back of the box from the camera
```

Because we set the top to 1 and bottom to -1 in the preceding code (Listing 20–6), we have set the front height of the box to 2 units. We specify the sizes for the left and right sides of the frustum by using proportional numbers, taking into account the window's aspect ratio. This is why this code uses the window height and width to figure out the proportion. The code also assumes the area of action to be between 3 and 7 units along the z axis. Anything drawn outside these coordinates, relative to the camera, won't be visible.

Because we set the camera at (0,0,5) and pointing toward (0,0,0), 3 units from the camera toward the origin will be (0,0,2) and 7 units from the camera will be (0,0,-2). This leaves the origin plane right in the middle of your 3D box.

So now we've identified the size of our viewing volume. There's one more important API and it maps these sizes to the screen: glViewport.

glViewport and Screen Size

glViewport is responsible for specifying the rectangular area on the screen onto which the viewing volume will be projected. This method takes four arguments to specify the rectangular box: the x and y coordinates of the lower-left corner, followed by the width and height. Listing 20–7 is an example of specifying a view as the target for this projection.

Listing 20–7. *Defining a ViewPort through glViewPort*

```
glViewport(0,          // lower left "x" of the rectangle on the screen
        0,          // lower left "y" of the rectangle on the screen
        width,      // width of the rectangle on the screen
        height);    // height of the rectangle on the screen
```

If our window or view size is 100 pixels in height and the frustum height is 10 units, then every logical unit of 1 in the world coordinates translates to 10 pixels in screen coordinates.

So far we have covered some important introductory concepts in OpenGL. Understanding these OpenGL fundamentals is useful for learning how to write Android OpenGL code. With these prerequisites behind us, we'll now discuss what is needed to call the OpenGL ES APIs that we have learned in this section.

Interfacing OpenGL ES with Android

OpenGL ES, as indicated, is a standard that is supported by a number of platforms. At the core, it's a C-like API that addresses all of the OpenGL drawing chores. However, each platform and OS is different in the way it implements displays, screen buffers, and the like. These OS-specific aspects are left to each operating system to figure out and document. Android is no different.

- Starting with its 1.5 SDK, Android simplified the interaction and initialization process necessary to start drawing in OpenGL. This support is provided in the package android.opengl. The primary class that provides much of this functionality is GLSurfaceView, and it has an internal interface called GLSurfaceView.Renderer. Knowing these two entities is sufficient to make a substantial headway with OpenGL on Android.

Using GLSurfaceView and Related Classes

Starting with 1.5 of the SDK, the common usage pattern for using OpenGL is quite simplified. (Refer to the first edition of this book to see the Android 1.0–approach.) Here are the typical steps to draw using these classes:

1. Implement the Renderer interface.
2. Provide the Camera settings needed for your drawing in the implementation of the renderer.
3. Provide the drawing code in the onDrawFrame method of the implementation.
4. Construct a GLSurfaceView.
5. Set the renderer implemented in steps 1 to 3 in the GLSurfaceView.
6. Indicate whether you want animation or not to the GLSurfaceView.
7. Set the GLSurfaceView in an Activity as the content view. You can also use this view wherever you can use a regular view.

Let's start with how to implement the renderer interface.

Implementing the Renderer

The signature of the Renderer interface is shown in Listing 20–8.

Listing 20–8. *The Renderer Interface*

```
public static interface GLSurfaceView.Renderer
{
  void onDrawFrame(GL10 gl);
  void onSurfaceChanged(GL10 gl, int width, int height);
  void onSurfaceCreated(GL10 gl, EGLConfig config);
}
```

The main drawing happens in the onDrawFrame() method. Whenever a new surface is created for this view, the onSurfaceCreated() method is called. We can call a number of OpenGL APIs such as dithering, depth control, or any others that can be called outside of the immediate onDrawFrame() method.

Similarly, when a surface changes, such as the width and height of the window, the onSurfaceChanged() method is called. We can set up our camera and viewing volume here.

Even in the onDrawFrame() method there are lot of things that may be common for our specific drawing context. We can take advantage of this commonality and abstract these methods in another level of abstraction called an AbstractRenderer, which will have only one method that is left unimplemented called draw().

Listing 20–9 shows the code for the AbstractRenderer.

Listing 20–9. *The AbstractRenderer*

```
//filename: AbstractRenderer.java
import android.opengl.*;
//…Use Eclipse to resolve other imports
public abstract class AbstractRenderer
implements android.opengl.GLSurfaceView.Renderer
{
   public void onSurfaceCreated(GL10 gl, EGLConfig eglConfig) {
      gl.glDisable(GL10.GL_DITHER);
      gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,
            GL10.GL_FASTEST);
      gl.glClearColor(.5f, .5f, .5f, 1);
      gl.glShadeModel(GL10.GL_SMOOTH);
      gl.glEnable(GL10.GL_DEPTH_TEST);
   }

   public void onSurfaceChanged(GL10 gl, int w, int h) {
      gl.glViewport(0, 0, w, h);
      float ratio = (float) w / h;
      gl.glMatrixMode(GL10.GL_PROJECTION);
      gl.glLoadIdentity();
      gl.glFrustumf(-ratio, ratio, -1, 1, 3, 7);
   }

   public void onDrawFrame(GL10 gl)
   {
      gl.glDisable(GL10.GL_DITHER);
      gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
      gl.glMatrixMode(GL10.GL_MODELVIEW);
      gl.glLoadIdentity();
```

```
        GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        draw(gl);
    }
    protected abstract void draw(GL10 gl);
}
```

Having this abstract class is very useful, as it allows us to focus on just the drawing methods. We'll use this class to create a SimpleTriangleRenderer class; Listing 20–10 shows the source code.

Listing 20–10. *SimpleTriangleRenderer*

```
//filename: SimpleTriangleRenderer.java
public class SimpleTriangleRenderer extends AbstractRenderer
{
   //Number of points or vertices we want to use
    private final static int VERTS = 3;

    //A raw native buffer to hold the point coordinates
    private FloatBuffer mFVertexBuffer;

    //A raw native buffer to hold indices
    //allowing a reuse of points.
    private ShortBuffer mIndexBuffer;

    public SimpleTriangleRenderer(Context context)
    {
        ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
        vbb.order(ByteOrder.nativeOrder());
        mFVertexBuffer = vbb.asFloatBuffer();

        ByteBuffer ibb = ByteBuffer.allocateDirect(VERTS * 2);
        ibb.order(ByteOrder.nativeOrder());
        mIndexBuffer = ibb.asShortBuffer();

        float[] coords = {
                -0.5f, -0.5f, 0, // (x1,y1,z1)
                 0.5f, -0.5f, 0,
                 0.0f,  0.5f, 0
        };
        for (int i = 0; i < VERTS; i++) {
            for(int j = 0; j < 3; j++) {
                mFVertexBuffer.put(coords[i*3+j]);
            }
        }
        short[] myIndecesArray = {0,1,2};
```

```
        for (int i=0;i<3;i++)
        {
          mIndexBuffer.put(myIndecesArray[i]);
        }
        mFVertexBuffer.position(0);
        mIndexBuffer.position(0);
    }

   //overriden method
    protected void draw(GL10 gl)
    {
       gl.glColor4f(1.0f, 0, 0, 0.5f);
       gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glDrawElements(GL10.GL_TRIANGLES, VERTS,
              GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

Although there seems to be a lot of code here, most of it is used to define the vertices and then translate them to NIO buffers from Java buffers. Otherwise, the draw method is just three lines: set the color, set the vertices, and draw.

NOTE: Although we are allocating memory for NIO buffers, we never release them in our code. So who releases these buffers? How does this memory affect OpenGL?

According to our research, the java.nio package allocates memory space outside of the Java heap that can be directly used by such systems as OpenGL, File I/O, etc. The nio buffers are actually Java objects that eventually point to the native buffer. These nio objects are garbage collected. When they are garbage collected, they go ahead and delete the native memory. Java programs don't have to do anything special to free the memory.

However, the gc won't get fired unless memory is needed in the Java heap. This means you can run out of native memory and gc may not realize it. The Internet offers many examples on this subject where an out of memory exception will trigger a gc and then it's possible to inquire if memory is now available due to gc having been invoked.

Under ordinary circumstances—and this is important for OpenGL—you can allocate the native buffers and not worry about releasing allocated memory explicitly because that is done by the gc.

Now that we have a sample renderer, let's see how we can supply this renderer to a GLSurfaceView and have it show up in an Activity.

Using GLSurfaceView from an Activity

Listing 20–11 shows a typical activity that uses a GLSurfaceView along with a suitable renderer.

Listing 20–11. *A Simple OpenGLTestHarness Activity*

```
public class OpenGLTestHarnessActivity extends Activity {
  private GLSurfaceView mTestHarness;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      mTestHarness = new GLSurfaceView(this);
      mTestHarness.setEGLConfigChooser(false);
      mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
      mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
      //mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
      setContentView(mTestHarness);
   }
   @Override
   protected void onResume()    {
      super.onResume();
      mTestHarness.onResume();
   }
   @Override
   protected void onPause() {
      super.onPause();
      mTestHarness.onPause();
   }
}
```

Let's explain the key elements of this source code. Here is the code that instantiates the GLSurfaceView:

```
mTestHarness = new GLSurfaceView(this);
```

We then tell the view that we don't need a special EGL config chooser and the default will work by doing the following:

```
mTestHarness.setEGLConfigChooser(false);
```

Then we set our renderer as follows:

```
mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
```

Next, we use one of these two methods to allow for animation or not:

```
mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
//mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
```

If we choose the first line, the drawing is going to be called only once or, more accurately, whenever it needs to be drawn. If we choose the second option, our drawing code will be called repeatedly so that we can animate our drawings.

That's all there is to interfacing with OpenGL on Android.

Now we have all the pieces necessary to test this drawing. We have the activity in Listing 20–11, we have the abstract renderer in Listing 20–9, and the SimpleTriangleRenderer (Listing 20–10) itself. All we have to do is invoke the Activity class through any of our menu items using the following:

```
private void invokeSimpleTriangle()
{
    Intent intent = new Intent(this,OpenGLTestHarnessActivity.class);
    startActivity(intent);
}
```

Of course, we will have to register the activity in the Android manifest file, like so:

```
  <activity android:name=".OpenGLTestHarnessActivity"
            android:label="OpenGL Test Harness"/>
```

Although it's perfectly reasonable to design a standalone activity like the OpenGLTestHarnessActivity in Listing 20–11, we would like to propose an alternative that fits this chapter much better.

This need comes from the fact that we have a number of demos in this chapter. If we were to design a separate activity for each demo, we would end up with lot of code that looks very similar to what we have in Listing 20–11 and does not elucidate over and above. In addition, each of those activities needs to be registered in the manifest file.

With this in mind, let's create a unified activity that allows us to test all OpenGL ES 1.0 demos. The code is in Listing 20–12. It may look extensive compared to the activity listed in 20–11; however, if you look at the menu response for R.id.mid_opengl_simpletriangle, you'll see that we are doing essentially the same thing. As more menu options are implemented, we'll have more if statements, one each for the type of demo.

The other menu options will be explored as we go through the chapter. After Listing 20–12, we'll present the menu .xml file followed by an explanation of this multipurpose activity in a bit more detail.

Listing 20–12. *MultiviewTestHarness Activity*

```
//filename: MultiViewTestHarnessActivity.java
public class MultiViewTestHarnessActivity extends Activity {
  private GLSurfaceView mTestHarness;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      mTestHarness = new GLSurfaceView(this);
      mTestHarness.setEGLConfigChooser(false);

      Intent intent = getIntent();
      int mid = intent.getIntExtra("com.ai.menuid", R.id.mid_OpenGL_Current);
      if (mid == R.id.mid_OpenGL_SimpleTriangle)
      {
         mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
         mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
         setContentView(mTestHarness);
         return;
      }
      if (mid == R.id.mid_OpenGL_Current)
      {
          //Call someother OpenGL Renderer
          //and
          //return;
      }
      //otherwise do this
      mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
      mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
      setContentView(mTestHarness);
      return;
   }
   @Override
   protected void onResume()    {
      super.onResume();
      mTestHarness.onResume();
   }
   @Override
   protected void onPause() {
      super.onPause();
      mTestHarness.onPause();
   }
}
```

The menu file in Listing 20–13 supports the code in Listing 20–12. This file is called res/menu/main_menu.xml. We went ahead and created all the possible menu items for all the demos of this chapter.

Listing 20–13. *Main Menu File*

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <!-- This group uses the default category. -->
   <group android:id="@+id/menuGroup_Main">

      <item android:id="@+id/mid_OpenGL_SimpleTriangle"
         android:title="Simple Triangle" />

      <item android:id="@+id/mid_OpenGL_SimpleTriangle2"
         android:title="Two Triangles" />

      <item android:id="@+id/mid_OpenGL_AnimatedTriangle"
         android:title="Animated Triangle" />

      <item android:id="@+id/mid_rectangle"
         android:title="Rectangle" />

      <item android:id="@+id/mid_square_polygon"
         android:title="Square polygon" />

      <item android:id="@+id/mid_polygon"
         android:title="Polygon" />

      <item android:id="@+id/mid_textured_square"
         android:title="Textured Square" />

      <item android:id="@+id/mid_textured_polygon"
         android:title="Textured Polygon" />

      <item android:id="@+id/mid_multiple_figures"
         android:title="Multiple Figures" />

      <item android:id="@+id/mid_OpenGL_Current"
         android:title="Current" />

      <item android:id="@+id/mid_es20_triangle"
         android:title="ES20 Triangle" />
   </group>
</menu>
```

By looking at the menu .xml file, we can anticipate the type of OpenGL renderers that will be demonstrated. If we return to the multiview activity in Listing 20–12, we'll notice that the activity is switching the renderer based on the menu IDs defined in this menu .xml file.

How does the multiview activity get the menu ID? This is done by the following code snippet (taken from Listing 20–12):

```
Intent intent = getIntent();
int mid = intent.getIntExtra("com.ai.menuid",
                R.id.mid_OpenGL_Current);
```

This code snippet is asking the intent that is responsible for invoking this activity if there is an extra called "com.ai.menuid." If it's not present, then the code should use a menu id called "mid_opengl_current" as the default menu ID.

Who puts this extra in the intent? Where is the invoking driver activity? This invoking driver activity is presented in Listing 20–14.

Listing 20–14. *TestOpenGLMainDriver Activity*

```
public class TestOpenGLMainDriverActivity extends Activity {
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
   }
   @Override
   public boolean onCreateOptionsMenu(Menu menu){
         super.onCreateOptionsMenu(menu);
         MenuInflater inflater = getMenuInflater(); //from activity
         inflater.inflate(R.menu.main_menu, menu);
         return true;
   }
   @Override
   public boolean onOptionsItemSelected(MenuItem item)
   {
         this.invokeMultiView(item.getItemId());
         return true;
   }
   private void invokeMultiView(int mid)
   {
         Intent intent =
     new Intent(this,MultiViewTestHarnessActivity.class);
         intent.putExtra("com.ai.menuid", mid);
         startActivity(intent);
   }
```

```
}
```

We need a layout file to complete and compile this activity. This layout file is in Listing 20–15.

Listing 20–15. *TestOpenGLMainDriver Activity Layout File (layout/main.xml)*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
<TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="A Simple Main Activity. Click Menu to Proceed"
   />
</LinearLayout>
```

Of course, nothing moves in Android without a manifest file. The manifest file is given in Listing 20–16.

Listing 20–16. *AndroidManifest File*

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
     package="com.androidbook.OpenGL"
     android:versionCode="1"
     android:versionName="1.0.0">
   <application android:icon="@drawable/icon"
           android:label="OpenGL Test Harness"
           android:debuggable="true">
      <activity android:name=".TestOpenGLMainDriverActivity"
              android:label="OpenGL Test Harness">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>

      <activity android:name="MultiViewTestHarnessActivity"
              android:label="OpenGL MultiView Test Harness"/>
   </application>
   <uses-sdk android:minSdkVersion="3" />
</manifest>
```

To summarize, we need the following files to compile and run our program:

- TestOpenGLMainDriverActivity.java (Main driver activty; Listing 20–14)

- AbstractRenderer.java (Listing 20–9)
- SimpleTriangleRenderer.java (Listing 20–10)
- MultiViewTestHarnessActivity.java (Listing 20–12)
- res/menu/main_menu.xml (Menu file; Listing 20–13)
- layout/main.xml (Layout file; Listing 20–15)

Once we compile and run the program, we'll see the driver activity show up. We can click on the menu to see the possible menus, as shown in Figure 20–2.

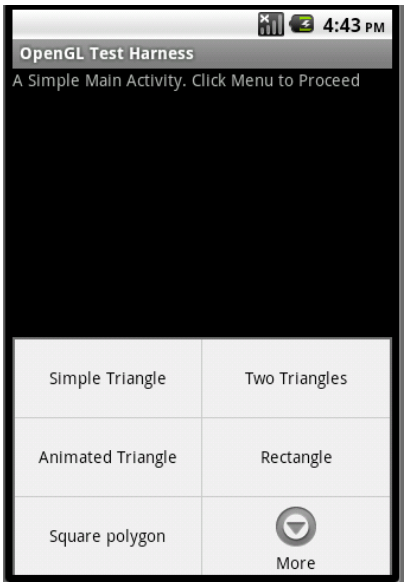

Figure 20–2. *OpenGL test harness driver*

Now if you click the "Simple Triangle" menu item, you will see the triangle like the one in Figure 20–3.

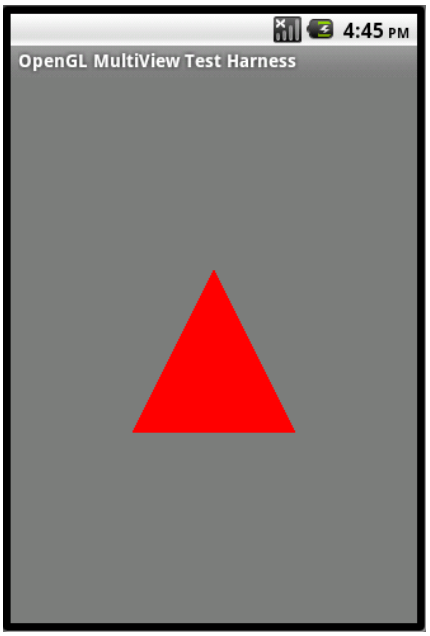

Figure 20–3. *A simple OpenGL triangle*

Changing Camera Settings

To understand the OpenGL coordinates better, let's experiment with the camera-related methods and see how they affect the triangle that we drew in Figure 20–3. Remember that these are the points of our triangle: (-0.5,-0.5,0 0.5,-0.5,0 0,0.5,0). With these points, the following three camera-related methods as used in AbstractRenderer (Listing 20–9) yielded the triangle as it appears in Figure 20–3:

```
//Look at the screen (origin) from 5 units away from the front of the screen
GLU.gluLookAt(gl, 0,0,5,   0,0,0,   0,1,0);

//Set the height to 2 units and depth to 4 units
gl.glFrustumf(-ratio, ratio, -1, 1, 3, 7);

//normal window stuff
gl.glViewport(0, 0, w, h);
```

Now suppose you change the camera's up vector toward the negative y direction, like this:

```
GLU.gluLookAt(gl, 0,0,5,   0,0,0,   0,-1,0);
```

If you do this, you'll see an upside-down triangle (Figure 20–4). If you want to make this change, you can find the method to change in the AbstractRenderer.java file (Listing 20–9).

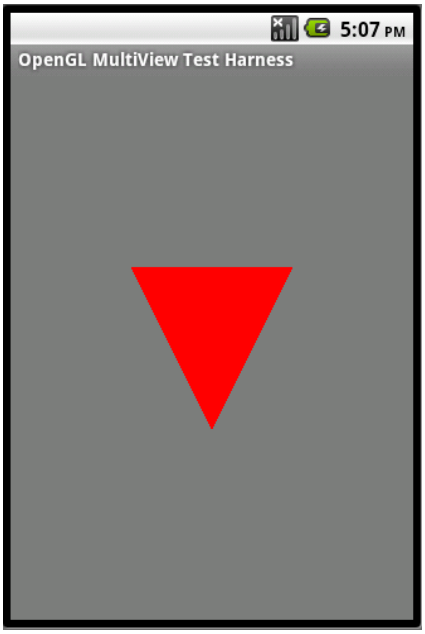

Figure 20–4. *A triangle with the camera upside down*

Now let's see what happens if we change the frustum, (also called the viewing volume or box). The following code increases the viewing box's height and width by a factor of 4 (see Figure 20–1 to understand these dimensions). If you recall, the first four arguments of glFrustum points to the front rectangle of the viewing box. By multiplying each value by 4, we have scaled the viewing box four times, like so:

```
gl.glFrustumf(-ratio * 4, ratio * 4, -1 * 4, 1 *4, 3, 7);
```

With this code, the triangle we see shrinks because the triangle stays at the same units while our viewing box has grown (Figure 20–5). This method call appears in the AbstractRenderer.java class (see Listing 20–9).

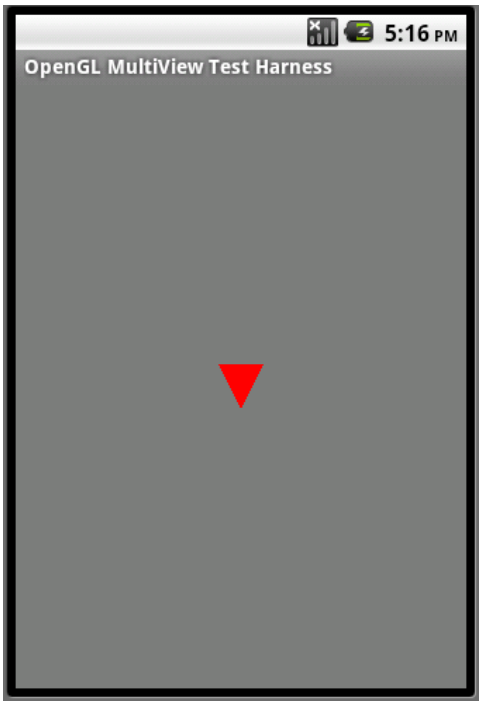

Figure 20–5. *A triangle with a viewing box that is four times bigger*

Using Indices to Add Another Triangle

We'll conclude these simple triangle examples by inheriting from the AbstractRenderer class and creating another triangle simply by adding an additional point and using indices. Conceptually, we'll define the four points as (-1,-1, 1,-1, 0,1, 1,1). And we'll ask OpenGL to draw these as (0,1,2 0,2,3). Listing 20–17 shows the code that does this (notice that we changed the dimensions of the triangle).

Listing 20–17. *The SimpleTriangleRenderer2 Class*

```
//filename: SimpleTriangleRenderer2.java
public class SimpleTriangleRenderer2 extends AbstractRenderer
{
   private final static int VERTS = 4;
   private FloatBuffer mFVertexBuffer;
   private ShortBuffer mIndexBuffer;

   public SimpleTriangleRenderer2(Context context)
   {
      ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
      vbb.order(ByteOrder.nativeOrder());
```

```
        mFVertexBuffer = vbb.asFloatBuffer();

        ByteBuffer ibb = ByteBuffer.allocateDirect(6 * 2);
        ibb.order(ByteOrder.nativeOrder());
        mIndexBuffer = ibb.asShortBuffer();

        float[] coords = {
                -1.0f, -1.0f, 0, // (x1,y1,z1)
                 1.0f, -1.0f, 0,
                 0.0f,  1.0f, 0,
                 1.0f,  1.0f, 0
        };
        for (int i = 0; i < VERTS; i++) {
            for(int j = 0; j < 3; j++) {
                mFVertexBuffer.put(coords[i*3+j]);
            }
        }
    short[] myIndecesArray = {0,1,2,    0,2,3};
        for (int i=0;i<6;i++)
        {
           mIndexBuffer.put(myIndecesArray[i]);
        }
        mFVertexBuffer.position(0);
        mIndexBuffer.position(0);
    }

    protected void draw(GL10 gl)
    {
       gl.glColor4f(1.0f, 0, 0, 0.5f);
       gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
   gl.glDrawElements(GL10.GL_TRIANGLES, 6, GL10.GL_UNSIGNED_SHORT,
                                          mIndexBuffer);
    }
}
```

Once this SimpleTriangleRenderer2 class is in place, we can add the if condition code in Listing 20–18 to the MultiViewTestHarness in Listing 20–12.

Listing 20–18. *Using SimpleTriangleRenderer2*

```
        if (mid == R.id.mid_OpenGL_SimpleTriangle2)
        {
            mTestHarness.setRenderer(new SimpleTriangleRenderer2(this));
            mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
            setContentView(mTestHarness);
            return;
        }
```

After we add this code, we can run the program again and choose the menu option "Two Triangles" to see the two triangles drawn out (see Figure 20–6). Notice how the design of the MultiviewTestHarness saved us from creating a new activity and registering that activity in the manifest file. We will continue this pattern of adding additional if clauses for the subsequent renderers.

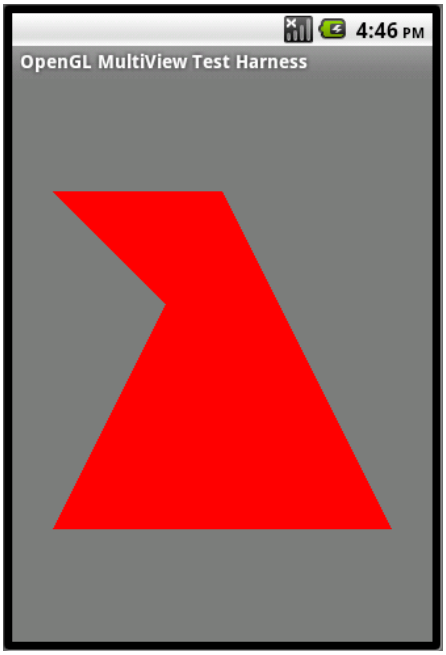

Figure 20–6. *Two triangles with four points*

Animating the Simple OpenGL Triangle

We can easily accommodate OpenGL animation by changing the rendering mode on the GLSurfaceView object. Listing 20–19 shows the sample code.

Listing 20–19. *Specifying Continuous-Rendering Mode*

```
//get a GLSurfaceView
GLSurfaceView openGLView;

//Set the mode to continuous draw mode
openGLView.setRenderingMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
```

Note that we're showing how to change the rendering mode here because we had specified RENDERMODE_WHEN_DIRTY in the previous section (see Listing 20-18). As mentioned, RENDERMODE_CONTINUOUSLY is the default setting, so animation is enabled by default.

Once the rendering mode is continuous, it is up to the renderer's onDraw method to do what's necessary to affect animation. To demonstrate this, let's use the triangle drawn in the previous example (see Listing 20–10 and Figure 20–3) and rotate it in a circular fashion.

AnimatedSimpleTriangleRenderer

The AnimatedSimpleTriangleRenderer class is very similar to the SimpleTriangleRenderer (see Listing 20–10), except for what happens in the onDraw method. In this method, we set a new rotation angle every four seconds. As the image gets drawn repeatedly, we'll see the triangle spinning slowly. Listing 20–20 contains the complete implementation of the AnimatedSimpleTriangleRenderer class.

Listing 20–20. *AnimatedSimpleTriangleRenderer Source Code*

```
//filename: AnimatedSimpleTriangleRenderer.java
public class AnimatedSimpleTriangleRenderer extends AbstractRenderer
{
  private int scale = 1;
  //Number of points or vertices we want to use
   private final static int VERTS = 3;

   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFVertexBuffer;

   //A raw native buffer to hold indices
   //allowing a reuse of points.
   private ShortBuffer mIndexBuffer;

   public AnimatedSimpleTriangleRenderer(Context context)
   {
      ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
      vbb.order(ByteOrder.nativeOrder());
      mFVertexBuffer = vbb.asFloatBuffer();

      ByteBuffer ibb = ByteBuffer.allocateDirect(VERTS * 2);
      ibb.order(ByteOrder.nativeOrder());
      mIndexBuffer = ibb.asShortBuffer();

      float[] coords = {
            -0.5f, -0.5f, 0, // (x1,y1,z1)
             0.5f, -0.5f, 0,
             0.0f,  0.5f, 0
      };
      for (int i = 0; i < VERTS; i++) {
         for(int j = 0; j < 3; j++) {
```

```
                mFVertexBuffer.put(coords[i*3+j]);
            }
        }
        short[] myIndecesArray = {0,1,2};
        for (int i=0;i<3;i++)
        {
          mIndexBuffer.put(myIndecesArray[i]);
        }
        mFVertexBuffer.position(0);
        mIndexBuffer.position(0);
    }

   //overridden method
    protected void draw(GL10 gl)
    {
       long time = SystemClock.uptimeMillis() % 4000L;
       float angle = 0.090f * ((int) time);

       gl.glRotatef(angle, 0, 0, 1.0f);

       gl.glColor4f(1.0f, 0, 0, 0.5f);
       gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glDrawElements(GL10.GL_TRIANGLES, VERTS,
              GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

Once this AnimatedSimpleTriangleRenderer class is in place, we can add the if condition code in Listing 20–21 to the MultiViewTestHarness in Listing 20–12.

Listing 20–21. *Using AnimatedSimpleTriangleRenderer*

```
if (mid == R.id.mid_OpenGL_AnimatedTriangle)
{
      mTestHarness.setRenderer(new AnimatedSimpleTriangleRenderer(this));
      setContentView(mTestHarness);
      return;
}
```

After we add this code, we can run the program again and choose the menu option "Animated Triangle" to see the triangle in Figure 20–3 spinning.

Braving OpenGL: Shapes and Textures

In the examples shown thus far, we have specified the vertices of a triangle explicitly. This approach becomes inconvenient as soon as we start drawing squares, pentagons, hexagons, and the

like. For these, we'll need higher-level object abstractions such as shapes and even scene graphs, where the shapes decide what their coordinates are. Using this approach, we will show you how to draw any polygon with any number of sides anywhere in your geometry.

In this section, we will also cover OpenGL textures. Textures allow you to attach bitmaps and other pictures to surfaces in your drawing. We will take the polygons that we know how to draw now and attach some pictures to them. We will follow this up with another critical need in OpenGL: drawing multiple figures or shapes using the OpenGL drawing pipeline.

These fundamentals should take you a bit closer to starting to create workable 3D figures and scenes.

Drawing a Rectangle

Before going on to the idea of shapes, let's strengthen our understanding of drawing with explicit vertices by drawing a rectangle using two triangles. This will also lay the groundwork for extending a triangle to any polygon.

We already have enough background to understand the basic triangle, so here's the code for drawing a rectangle (Listing 20–22), followed by some brief commentary.

Listing 20–22. *Simple Rectangle Renderer*

```
public class SimpleRectangleRenderer extends AbstractRenderer
{
  //Number of points or vertices we want to use
   private final static int VERTS = 4;

   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFVertexBuffer;

   //A raw native buffer to hold indices
   //allowing a reuse of points.
   private ShortBuffer mIndexBuffer;

   public SimpleRectangleRenderer(Context context)
   {
      ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
      vbb.order(ByteOrder.nativeOrder());
      mFVertexBuffer = vbb.asFloatBuffer();

      ByteBuffer ibb = ByteBuffer.allocateDirect(6 * 2);
      ibb.order(ByteOrder.nativeOrder());
      mIndexBuffer = ibb.asShortBuffer();
```

```
        float[] coords = {
                -0.5f, -0.5f, 0, // (x1,y1,z1)
                 0.5f, -0.5f, 0,
                 0.5f,  0.5f, 0,
                -0.5f,  0.5f, 0,
        };

        for (int i = 0; i < VERTS; i++) {
            for(int j = 0; j < 3; j++) {
                mFVertexBuffer.put(coords[i*3+j]);
            }
        }
        short[] myIndecesArray = {0,1,2,0,2,3};
        for (int i=0;i<6;i++)
        {
          mIndexBuffer.put(myIndecesArray[i]);
        }
        mFVertexBuffer.position(0);
        mIndexBuffer.position(0);
    }

   //overriden method
    protected void draw(GL10 gl)
    {
       gl.glColor4f(1.0f, 0, 0, 0.5f);
       gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
       gl.glDrawElements(GL10.GL_TRIANGLES, 6,
                GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

Notice that the approach for drawing a rectangle is quite similar to that for a triangle. We have specified four vertices instead of three. Then we have used indices as here:

```
short[] myIndecesArray = {0,1,2,0,2,3};
```

We have reused the numbered vertices (0 through 3) twice so that each three vertices make up a triangle. So (0,1,2) makes up the first triangle and (0,2,3) makes up the second triangle. Drawing these two triangles using the GL_TRIANGLES primitives will draw the necessary rectangle.

Once this rectangle renderer class is in place, we can add the if condition code in Listing 20–23 to the MultiViewTestHarness in Listing 20–12.

Listing 20–23. *Using SimpleRectangleRenderer*

```
if (mid == R.id.mid_rectangle)
{
```

```
        mTestHarness.setRenderer(new SimpleRectangleRenderer(this));
        mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
        setContentView(mTestHarness);
        return;
    }
```

After we add this code, we can run the program again and choose the menu option "Rectangle" to see the rectangle in Figure 20–7.

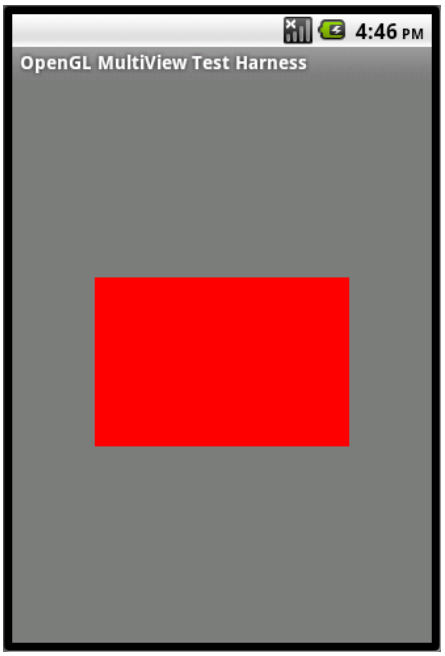

Figure 20–7. *OpenGL rectangle drawn with two triangles*

Working with Shapes

This method of explicitly specifying vertices to draw can be tedious. For example, if you want to draw a polygon of 20 sides, then you need to specify 20 vertices, with each vertex requiring up to three values. That's a total of 60 values. It's just not workable.

A Regular Polygon as a Shape

A better approach to draw figures like triangles or squares is to define an abstract polygon by defining some aspects of it, such as the origin and radius, and then have that polygon give us the vertex array and the index array (so that we can draw individual triangles) in return. We named

this class RegularPolygon. Once we have this kind of object, we can use it as shown in Listing 20–24 to render various regular polygons.

Listing 20–24. *Using a RegularPolygon Object*

```
//A polygon with 4 sides and a radious of 0.5
//and located at (x,y,z) of (0,0,0)
 RegularPolygon square = new RegularPolygon(0,0,0,0.5f,4);

//Let the polygon return the vertices
 mFVertexBuffer = square.getVertexBuffer();

//Let the polygon return the triangles
 mIndexBuffer = square.getIndexBuffer();

//you will need this for glDrawElements
numOfIndices = square.getNumberOfIndices();

//set the buffers to the start
  this.mFVertexBuffer.position(0);
  this.mIndexBuffer.position(0);

//set the vertex pointer
  gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);

//draw it with the given number of Indices
 gl.glDrawElements(GL10.GL_TRIANGLES, numOfIndices,
        GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
```

Notice how we have obtained the necessary vertices and indices from the shape square. Although we haven't abstracted this idea of getting vertices and indices to a basic shape, it is possible that RegularPolygon could be deriving from such a basic shape that defines an interface for this basic contract. Listing 20–25 shows an example.

Listing 20–25. *Shape Interface*

```
public interface Shape
{
  FloatBuffer          getVertexBuffer();
  ShortBuffer          getIndexBuffer();
  int                  getNumberofIndices();
}
```

We will leave this idea of defining a base interface for a shape as food for thought for your own work. For now, we have built these methods out directly into the RegularPolygon.

Implementing the RegularPolygon Shape

As indicated, this RegularPolygon has the responsibility of returning what is needed to draw using OpenGL: vertices. First, we need a mechanism to define what this shape is and where it is in the geometry.

For a regular polygon, there are a number of ways of doing this. In our approach, we have defined the regular polygon using the number of sides and the distance from the center of the regular polygon to one of its vertices. We called this distance the radius, because the vertices of a regular polygon fall on the perimeter of a circle whose center is also the center of the regular polygon. So the radius of such a circle and the number of sides will tell us the polygon we want. By specifying the coordinates of the center, we also know where to draw the polygon in our geometry.

The responsibility of this RegularPolygon class is to give us the coordinates of all the vertices of the polygon, given its center and radius. Again, there may be a number of ways of doing this. Whatever mathematical method you choose to employ (based on middle school or high school math), as long as you return the vertices, you're good to go.

For our approach, we started with the assumption that the radius is 1 unit. We figured out the angles for each line connecting the center to each vertex of the polygon. We kept these angles in an array. For each angle, we calculated the x-axis projection and called this the "x multiplier array." (We used "multiplier array" because we started out with a unit of radius.) When we know the real radius, we will multiply these values with the real radius to get the real x coordinate. These real x coordinates are then stored in an array called "x array." We do the same for the y-axis projections.

Now that you have an idea of what needs to happen in the implementation of the RegularPolygon, we'll give you the source code that addresses these responsibilities. Listing 20–26 shows all the code for the RegularPolygon in one place. (Please note that the source code is several pages long.) To make the process of going through it less cumbersome, we have highlighted the function names and provided inline comments at the beginning of each function.

We define the key functions in a list that follows Listing 20–26. The important thing here is to figure out the vertices and return. If this is too cryptic, it shouldn't be hard to write your own code to get the vertices. You'll also note that this code also has functions that deal with texturing. We'll explain these texture functions in the "Working with Textures" section.

Listing 20–26. *Implementing a RegularPolygon Shape*

```
public class RegularPolygon
{
  //Space to hold (x,y,z) of the center: cx,cy,cz
  //and the radius "r"
  private float      cx, cy, cz, r;
```

```
private int         sides;

//coordinate array: (x,y) vertex points
private float[] xarray = null;
private float[] yarray = null;

//texture arrray: (x,y) also called (s,t) points
//where the figure is going to be mapped to a texture bitmap
private float[] sarray = null;
private float[] tarray = null;

//***********************************************
// Constructor
//***********************************************
public RegularPolygon(float incx, float incy, float incz, // (x,y,z) center
                      float inr, // radius
                      int insides) // number of sides
{
   cx = incx;
   cy = incy;
   cz = incz;
   r = inr;
   sides = insides;

   //allocate memory for the arrays
   xarray = new float[sides];
   yarray = new float[sides];

   //allocate memory for texture point arrays
   sarray = new float[sides];
   tarray = new float[sides];

   //calculate vertex points
   calcArrays();

  //calculate texture points
   calcTextureArrays();
}

//***********************************************
//Get and convert the vertex coordinates
//based on origin and radius.
//Real logic of angles happen inside getMultiplierArray() functions
//***********************************************
private void calcArrays()
{
  //Get the vertex points assuming a circle
```

```
  //with a radius of "1" and located at "origin" zero
   float[] xmarray = this.getXMultiplierArray();
   float[] ymarray = this.getYMultiplierArray();

   //calc xarray: get the vertex
   //by adding the "x" portion of the origin
   //multiply the coordinate with radius (scale)
   for(int i=0;i<sides;i++)
   {
      float curm = xmarray[i];
      float xcoord = cx + r * curm;
      xarray[i] = xcoord;
   }
   this.printArray(xarray, "xarray");

   //calc yarray: do the same for y coordinates
   for(int i=0;i<sides;i++)
   {
      float curm = ymarray[i];
      float ycoord = cy + r * curm;
      yarray[i] = ycoord;
   }
   this.printArray(yarray, "yarray");

}
//*********************************************
//Calculate texture arrays
//See Texture subsection for more discussion on this
//very similar approach.
//Here the polygon has to map into a space
//that is a square
//*********************************************
private void calcTextureArrays()
{
   float[] xmarray = this.getXMultiplierArray();
   float[] ymarray = this.getYMultiplierArray();

   //calc xarray
   for(int i=0;i<sides;i++)
   {
      float curm = xmarray[i];
      float xcoord = 0.5f + 0.5f * curm;
      sarray[i] = xcoord;
   }
   this.printArray(sarray, "sarray");

   //calc yarray
```

```
    for(int i=0;i<sides;i++)
    {
      float curm = ymarray[i];
      float ycoord = 0.5f +  0.5f * curm;
      tarray[i] = ycoord;
    }
    this.printArray(tarray, "tarray");
}

//***********************************************
//Convert the java array of vertices
//into an nio float buffer
//***********************************************
public FloatBuffer getVertexBuffer()
{
    int vertices = sides + 1;
    int coordinates = 3;
    int floatsize = 4;
    int spacePerVertex = coordinates * floatsize;

     ByteBuffer vbb = ByteBuffer.allocateDirect(spacePerVertex * vertices);
     vbb.order(ByteOrder.nativeOrder());
     FloatBuffer mFVertexBuffer = vbb.asFloatBuffer();

     //Put the first coordinate (x,y,z:0,0,0)
     mFVertexBuffer.put(cx); //x
     mFVertexBuffer.put(cy); //y
     mFVertexBuffer.put(0.0f); //z

     int totalPuts = 3;
     for (int i=0;i<sides;i++)
     {
        mFVertexBuffer.put(xarray[i]); //x
        mFVertexBuffer.put(yarray[i]); //y
        mFVertexBuffer.put(0.0f); //z
        totalPuts += 3;
     }
     Log.d("total puts:",Integer.toString(totalPuts));
    return mFVertexBuffer;
}

//***********************************************
//Convert texture buffer to an nio buffer
//***********************************************
public FloatBuffer getTextureBuffer()
{
    int vertices = sides + 1;
```

```
    int coordinates = 2;
    int floatsize = 4;
    int spacePerVertex = coordinates * floatsize;

     ByteBuffer vbb = ByteBuffer.allocateDirect(spacePerVertex * vertices);
     vbb.order(ByteOrder.nativeOrder());
     FloatBuffer mFTextureBuffer = vbb.asFloatBuffer();

     //Put the first coordinate (x,y (s,t):0,0)
     mFTextureBuffer.put(0.5f); //x or s
     mFTextureBuffer.put(0.5f); //y or t

     int totalPuts = 2;
     for (int i=0;i<sides;i++)
     {
        mFTextureBuffer.put(sarray[i]); //x
        mFTextureBuffer.put(tarray[i]); //y
         totalPuts += 2;
     }
     Log.d("total texture puts:",Integer.toString(totalPuts));
   return mFTextureBuffer;
}

//***********************************************
//Calculate indices forming multiple triangles.
//Start with the center vertex which is at 0
//Then count them in a clockwise direction such as
//0,1,2,  0,2,3, 0,3,4 and so on.
//***********************************************
public ShortBuffer getIndexBuffer()
{
   short[] iarray = new short[sides * 3];
     ByteBuffer ibb = ByteBuffer.allocateDirect(sides * 3 * 2);
     ibb.order(ByteOrder.nativeOrder());
     ShortBuffer mIndexBuffer = ibb.asShortBuffer();
     for (int i=0;i<sides;i++)
     {
        short index1 = 0;
        short index2 = (short)(i+1);
        short index3 = (short)(i+2);
        if (index3 == sides+1)
        {
           index3 = 1;
        }
        mIndexBuffer.put(index1);
        mIndexBuffer.put(index2);
        mIndexBuffer.put(index3);
```

```
        iarray[i*3 + 0]=index1;
        iarray[i*3 + 1]=index2;
        iarray[i*3 + 2]=index3;
      }
      this.printShortArray(iarray, "index array");
      return mIndexBuffer;
}
//*********************************************
//This is where you take the angle array
//for each vertex and calculate their projection multiplier
//on the x axis
//*********************************************
private float[] getXMultiplierArray()
{
   float[] angleArray = getAngleArrays();
   float[] xmultiplierArray = new float[sides];
   for(int i=0;i<angleArray.length;i++)
   {
      float curAngle = angleArray[i];
      float sinvalue = (float)Math.cos(Math.toRadians(curAngle));
      float absSinValue = Math.abs(sinvalue);
      if (isXPositiveQuadrant(curAngle))
      {
         sinvalue = absSinValue;
      }
      else
      {
         sinvalue = -absSinValue;
      }
      xmultiplierArray[i] = this.getApproxValue(sinvalue);
   }
   this.printArray(xmultiplierArray, "xmultiplierArray");
   return xmultiplierArray;
}

//*********************************************
//This is where you take the angle array
//for each vertex and calculate their projection multiplier
//on the y axis
//*********************************************
private float[] getYMultiplierArray() {
   float[] angleArray = getAngleArrays();
   float[] ymultiplierArray = new float[sides];
   for(int i=0;i<angleArray.length;i++) {
      float curAngle = angleArray[i];
      float sinvalue = (float)Math.sin(Math.toRadians(curAngle));
```

```
      float absSinValue = Math.abs(sinvalue);
      if (isYPositiveQuadrant(curAngle)) {
        sinvalue = absSinValue;
      }
      else {
        sinvalue = -absSinValue;
      }
      ymultiplierArray[i] = this.getApproxValue(sinvalue);
    }
    this.printArray(ymultiplierArray, "ymultiplierArray");
    return ymultiplierArray;
  }

  //***********************************************
  //This function may not be needed
  //Test it yourself and discard it if you dont need
  //***********************************************
  private boolean isXPositiveQuadrant(float angle) {
    if ((0 <= angle) && (angle <= 90))  { return true;    }
    if ((angle < 0) && (angle >= -90))   { return true;    }
    return false;
  }
  //***********************************************
  //This function may not be needed
  //Test it yourself and discard it if you dont need
  //***********************************************
  private boolean isYPositiveQuadrant(float angle) {
    if ((0 <= angle) && (angle <= 90)) { return true; }
    if ((angle < 180) && (angle >= 90)) {return true;}
    return false;
  }
  //***********************************************
  //This is where you calculate angles
  //for each line going from center to each vertex
  //***********************************************
  private float[] getAngleArrays() {
    float[] angleArray = new float[sides];
    float commonAngle = 360.0f/sides;
    float halfAngle = commonAngle/2.0f;
    float firstAngle = 360.0f - (90+halfAngle);
    angleArray[0] = firstAngle;

    float curAngle = firstAngle;
    for(int i=1;i<sides;i++)
    {
      float newAngle = curAngle - commonAngle;
      angleArray[i] = newAngle;
```

```
            curAngle = newAngle;
        }
        printArray(angleArray, "angleArray");
        return angleArray;
    }

    //*************************************************
    //Some rounding if needed
    //*************************************************
    private float getApproxValue(float f) {
        return (Math.abs(f) < 0.001) ? 0 : f;
    }
    //*************************************************
    //Return how many Indices you will need
    //given the number of sides
    //This is the count of number of triangles needed
    //to make the polygon multiplied by 3
    //It just happens that the number of triangles is
    // same as the number of sides
    //*************************************************
    public int getNumberOfIndices() {
        return sides * 3;
    }
    public static void test() {
        RegularPolygon triangle = new RegularPolygon(0,0,0,1,3);
    }
    private void printArray(float array[], String tag) {
        StringBuilder sb = new StringBuilder(tag);
        for(int i=0;i<array.length;i++) {
            sb.append(";").append(array[i]);
        }
        Log.d("hh",sb.toString());
    }
    private void printShortArray(short array[], String tag) {
        StringBuilder sb = new StringBuilder(tag);
        for(int i=0;i<array.length;i++) {
            sb.append(";").append(array[i]);
        }
        Log.d(tag,sb.toString());
    }
}
```

Here are the key elements in the code:

- *Constructor*: The constructor of a RegularPolygon takes as input the coordinates of the center, the radius, and the number of sides.

- *getAngleArrays*: This method is a key method that is responsible for calculating the angles of each spine of the regular polygon with the assumption that one of the sides of the polygon is parallel to the x-axis.

- *getXMultiplierArray* and *getYMultiplierArray*: These methods take the angles from getAngleArrays and project them to the x-axis and y-axis to get the corresponding coordinates, assuming the spine is a unit in length.

- *calcArrays*: This method uses the getXMultiplierArray and the getYMultiplierArray to take each vertex and scales them to match the specified radius and specified origin. At the end of this method, the RegularPolygon will have the right coordinates, albeit in Java float arrays.

- *getVertexBuffer*: This method then takes the Java float coordinate arrays and populates NIO-based buffers that are needed by the OpenGL draw methods.

- *getIndexBuffer*: This method takes the vertices that are gathered and orders them such that each triangle will contribute to the final polygon.

The other methods that deal with textures follow a very similar pattern and will make more sense when we explain the textures in the next section. We have also included some print functions to print the arrays for debugging purposes.

Rendering a Square Using RegularPolygon

Now that we have looked at the basic building blocks, let's see how we could draw a square using a RegularPolygon of four sides. Listing 20–27 shows the code for the SquareRenderer.

Listing 20–27. *SquareRenderer*

```
public class SquareRenderer extends AbstractRenderer
{
   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFVertexBuffer;

   //A raw native buffer to hold indices
   //allowing a reuse of points.
   private ShortBuffer mIndexBuffer;

   private int numOfIndices = 0;

   private int sides = 4;

   public SquareRenderer(Context context)
   {
```

```
        prepareBuffers(sides);
    }

    private void prepareBuffers(int sides)
    {
        RegularPolygon t = new RegularPolygon(0,0,0,0.5f,sides);
        //RegularPolygon t = new RegularPolygon(1,1,0,1,sides);
        this.mFVertexBuffer = t.getVertexBuffer();
        this.mIndexBuffer = t.getIndexBuffer();
        this.numOfIndices = t.getNumberOfIndices();
        this.mFVertexBuffer.position(0);
        this.mIndexBuffer.position(0);
    }

   //overriden method
    protected void draw(GL10 gl)
    {
        prepareBuffers(sides);
       gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
       gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
              GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

This code should be fairly obvious. We have derived it from the AbstractRenderer (see Listing 20–9) and overrode the draw method and used the RegularPolygon to draw out a square.

Once this square renderer class is in place, we can add the if condition code in Listing 20–28 to the MultiViewTestHarness in Listing 20–12.

Listing 20–28. *Using SimpleRectangleRenderer*

```
if (mid == R.id.mid_square_polygon)
{
    mTestHarness.setRenderer(new SquareRenderer(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(mTestHarness);
    return;
}
```

After we add this code, we can run the program again and choose the menu option "Square Polygon" to see the square in Figure 20–8.

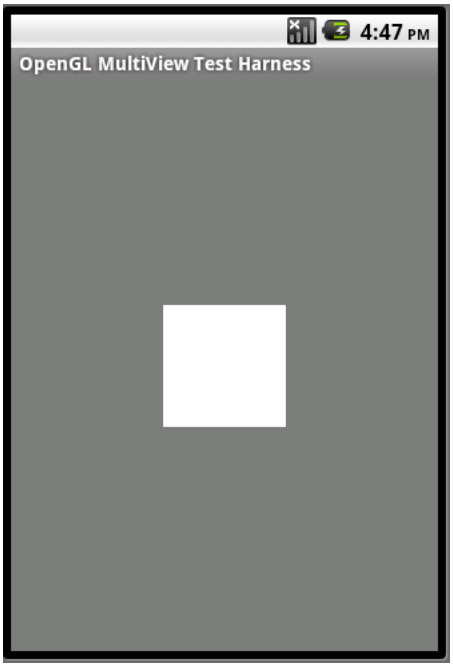

Figure 20–8. *A square drawn as a regular polygon*

Animating RegularPolygons

Now that we have explored the basic idea of drawing a shape generically through RegularPoygon, let's get a bit sophisticated. Let's see if we can use an animation where we start with a triangle and end up with a circle by using a polygon whose sides increase every four seconds or so. The code for this is in Listing 20–29.

Listing 20–29. *PolygonRenderer*

```
public class PolygonRenderer extends AbstractRenderer
{
  //Number of points or vertices we want to use
   private final static int VERTS = 4;

   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFVertexBuffer;

   //A raw native buffer to hold indices
   //allowing a reuse of points.
   private ShortBuffer mIndexBuffer;

   private int numOfIndices = 0;
```

```
    private long prevtime = SystemClock.uptimeMillis();

    private int sides = 3;

    public PolygonRenderer(Context context)
    {
prepareBuffers(sides);
    }

    private void prepareBuffers(int sides)
    {
        RegularPolygon t = new RegularPolygon(0,0,0,1,sides);
this.mFVertexBuffer = t.getVertexBuffer();
        this.mIndexBuffer = t.getIndexBuffer();
        this.numOfIndices = t.getNumberOfIndices();
        this.mFVertexBuffer.position(0);
        this.mIndexBuffer.position(0);
    }

   //overriden method
    protected void draw(GL10 gl)
    {
        long curtime = SystemClock.uptimeMillis();
        if ((curtime - prevtime) > 2000)
        {
          prevtime = curtime;
          sides += 1;
          if (sides > 20)
          {
            sides = 3;
          }
```

```
            this.prepareBuffers(sides);
        }
gl.glColor4f(1.0f, 0, 0, 0.5f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
                GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

All we are doing in this code is changing the sides variable every four seconds. The animation comes from the way the Renderer is registered with the surface view.

Once we have this renderer available, we will need to add the code in Listing 20–30 to the MultiviewTestHarness code.

Listing 20–30. *Menu Item for testing a polygon*

```
if (mid == R.id.mid_polygon)
{
    mTestHarness.setRenderer(new PolygonRenderer(this));
    setContentView(mTestHarness);
    return;
}
```

If we run the program again and choose the menu item "Polygon," we'll see a set of transforming polygons whose sides continue to increase. It is instructive to see the progress of the polygons over time. Figure 20–9 shows a hexagon toward the beginning of the cycle.

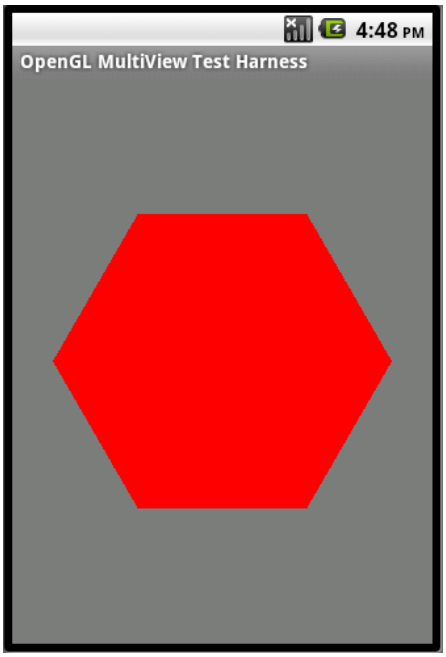

Figure 20–9. *Hexagon at the beginning of the polygon drawing cycle*

Figure 20–10 shows it towards the end of the cycle.

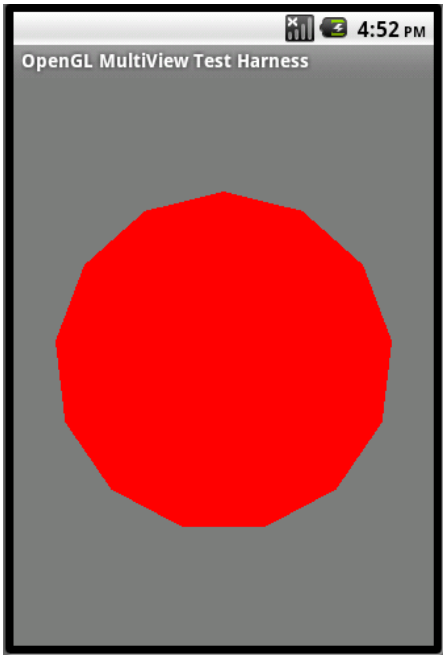

Figure 20–10. *A circle drawn as a regular polygon*

You can extend this idea of abstract shapes to more complex shapes and even to a scene graph where it consists of a number of other objects that are defined through some type of XML and then renders them in OpenGL using those instantiated objects.

Let's now move on to textures to see how we can integrate the idea of sticking wallpapers to the surfaces we have drawn so far, such as squares and polygons.

Working with Textures

Textures are another core topic in OpenGL. OpenGL textures have a number of nuances. We will cover only the fundamentals in this chapter so that you can get started with OpenGL textures. Use the resources provided at the end of this chapter to dig further into textures.

Understanding Textures

An OpenGL Texture is a bitmap that you paste on a surface in OpenGL. (In this chapter, we will cover only surfaces.) For example, you can take the image of a postage stamp and stick it on a square so that the square looks like a postage stamp. Or you can take the bitmap of a brick and paste it on a rectangle and repeat the brick image so that the rectangle looks like a wall of bricks.

The process of attaching a texture bitmap to an OpenGL surface is similar to the process of pasting a piece of wallpaper (in the shape of a square) on the side of a regularly or irregularly shaped object. The shape of the surface doesn't matter as long as you choose a paper that is large enough to cover it.

However, to align the paper so that the image is correctly lined up, you have to take each vertex of the shape and exactly mark it on the wallpaper so that the wallpaper and the object's shape are in lockstep. If the shape is odd and has a number of vertices, each vertex needs to be marked on your paper as well.

Another way of looking at this is to envision that you lay the object on the ground face up and put the wallpaper on top of it and rotate the paper until the image is aligned in the right direction. Now poke holes in the paper at each vertex of the shape. Remove the paper and see where the holes are and note their coordinates on the paper, assuming the paper is calibrated. These coordinates are called *texture coordinates*.

Normalized Texture Coordinates

One unresolved or unstated detail is the size of the object and the paper. OpenGL uses a normalized approach to resolve this. OpenGL assumes that the paper is always a 1 × 1 square with its origin at (0,0) and the top right corner is at (1,1). Then OpenGL wants you to shrink your object surface so that it fits within these 1 × 1 boundaries. So the burden is on the programmer to figure out the vertices of the object surface in a 1 × 1 square.

In the design of our RegularPolygon from Listing 20–26, we drew a polygon using a similar approach where we assumed it was a circle of 1 unit radius. Then we figured out where each vertex is. If we assume that that circle is inside a 1 × 1 square, then that square could be our paper. So figuring out texture coordinates is very similar to figuring out the polygon vertex coordinates. This is why Listing 20–26 has the following function to calculate the texture coordinates:

```
calcTextureArray()
getTextureBuffer()
```

If you notice, every other function is common between calcTextureArrays and calcArrays methods. This commonality between vertex coordinates and texture coordinates is important to note when you are learning OpenGL.

Abstracting Common Texture Handling

Once you understand this mapping between texture coordinates and vertex coordinates and can figure out the coordinates for the texture map, the rest is simple enough. (Nothing in OpenGL can be boldly stated as "quite simple!") Subsequent work involves loading the texture bitmap into memory and giving it a texture ID so that you can reuse this texture again. Then, to allow for multiple textures loaded at the same time, you have a mechanism to set the current texture by specifying an ID. During a drawing pipeline, you will specify the texture coordinates along with the drawing coordinates. Then you draw.

Because the process of loading textures is fairly common, we have abstracted out this process by inventing an abstract class called SingleAbstractTextureRenderer that inherits from AbstractRenderer.

Listing 20–31 shows the source code that abstracts out all the set-up code for a single texture.

Listing 20–31. *Abstracting Single Texturing Support*

```
public abstract class AbstractSingleTexturedRenderer
extends AbstractRenderer
{
  int mTextureID;
```

```
int mImageResourceId;
Context mContext;
public AbstractSingleTexturedRenderer(Context ctx,
                                int imageResourceId) {
  mImageResourceId = imageResourceId;
  mContext = ctx;
}

 public void onSurfaceCreated(GL10 gl, EGLConfig eglConfig) {
    super.onSurfaceCreated(gl, eglConfig);
    gl.glEnable(GL10.GL_TEXTURE_2D);
    prepareTexture(gl);
 }
 private void prepareTexture(GL10 gl)
 {
    int[] textures = new int[1];
    gl.glGenTextures(1, textures, 0);

    mTextureID = textures[0];
    gl.glBindTexture(GL10.GL_TEXTURE_2D, mTextureID);

    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,
          GL10.GL_NEAREST);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D,
          GL10.GL_TEXTURE_MAG_FILTER,
          GL10.GL_LINEAR);

    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,
          GL10.GL_CLAMP_TO_EDGE);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,
          GL10.GL_CLAMP_TO_EDGE);

    gl.glTexEnvf(GL10.GL_TEXTURE_ENV, GL10.GL_TEXTURE_ENV_MODE,
          GL10.GL_REPLACE);

    InputStream is = mContext.getResources()
          .openRawResource(this.mImageResourceId);
    Bitmap bitmap;
    try {
       bitmap = BitmapFactory.decodeStream(is);
    } finally {
       try {
          is.close();
       } catch(IOException e) {
          // Ignore.
       }
    }
```

```
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
        bitmap.recycle();
    }

    public void onDrawFrame(GL10 gl)
    {
        gl.glDisable(GL10.GL_DITHER);
        gl.glTexEnvx(GL10.GL_TEXTURE_ENV, GL10.GL_TEXTURE_ENV_MODE,
                GL10.GL_MODULATE);

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        gl.glActiveTexture(GL10.GL_TEXTURE0);
        gl.glBindTexture(GL10.GL_TEXTURE_2D, mTextureID);
        gl.glTexParameterx(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,
                GL10.GL_REPEAT);
        gl.glTexParameterx(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,
                GL10.GL_REPEAT);

        draw(gl);
    }
}
```

In this code, the single texture (a bitmap) is loaded and prepared in the onSurfaceCreated method. The code for onDrawFrame, just like the AbstractRenderer, sets up the dimensions of our drawing space so that our coordinates make sense. Depending on your situation, you may want to change this code to figure out your own optimal viewing volume.

Note how the constructor takes a texture bitmap which it prepares for later use. Depending on how many textures you have, you can craft your abstract classes accordingly.

As shown in Listing 20–31, the following APIs that revolve around textures are required:

- *glGenTextures*: This OpenGL method is responsible for generating unique IDs for textures so that those textures can be referenced later. Once we load the texture bitmap through GLUtils.texImage2D, we bind that texture to a specific ID. Until a texture is bound to an ID generated by glGenTextures, the ID is just an ID. The OpenGL literature refers to these integer IDs as *texture names*.

- *glBindTexture*: We use this OpenGL method to bind the currently loaded texture to a texture ID obtained from glGenTextures.

- *glTexParameter*: There are many optional parameters we can set when we apply texture. This API allows us to define what these options are. Some examples include GL_REPEAT, GL_CLAMP etc. For example, GL_REPEAT allows us to repeat the bitmap many times if the size of the object is larger. A complete list of these parameters can be found at www.khronos.org/opengles/documentation/opengles1_0/html/glTexParameter.html.

- *glTexEnv*: Some of the other texture-related options are specified through the glTexEnv method. Some example values include GL_DECAL, GL_MODULATE, GL_BLEND, GL_REPLACE, etc. For example, in the case of GL_DECAL, texture covers the underlying object. GL_MODULATE, as the name indicates, modulates the underlying colors instead of replacing them. Refer to the following URL for a complete list of the options for this API: www.khronos.org/opengles/documentation/opengles1_0/html/glTexEnv.html.

- *GLUtils.texImage2D*: This is an Android API that allows us to load the bitmap for texturing purposes. Internally, this API calls the glTexImage2D of the OpenGL.

- *glActiveTexture*: This sets a given texture ID as the active structure.

- *glTexCoordpointer*: This OpenGL method is used to specify the texture coordinates. Each coordinate must match the coordinate specified in the glVertexPointer.

You can read up on most of these APIs from the OpenGL ES reference available at

www.khronos.org/opengles/documentation/opengles1_0/html/index.html

Drawing Using Textures

Once the bitmap is loaded and set up as a texture, we should be able to utilize the RegularPolygon and use the texture coordinates and vertex coordinates to draw a regular polygon along with the texture. Listing 20–32 shows the actual drawing class that draws a textured square.

Listing 20–32. *TexturedSquareRenderer*

```
public class TexturedSquareRenderer extends AbstractSingleTexturedRenderer
{
  //Number of points or vertices we want to use
   private final static int VERTS = 4;

   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFVertexBuffer;

   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFTextureBuffer;

   //A raw native buffer to hold indices
   //allowing a reuse of points.
   private ShortBuffer mIndexBuffer;

   private int numOfIndices = 0;

   private int sides = 4;

   public TexturedSquareRenderer(Context context)
   {
     super(context,com.androidbook.OpenGL.R.drawable.robot);
     prepareBuffers(sides);
   }

   private void prepareBuffers(int sides)
   {
     RegularPolygon t = new RegularPolygon(0,0,0,0.5f,sides);
     this.mFVertexBuffer = t.getVertexBuffer();
     this.mFTextureBuffer = t.getTextureBuffer();
     this.mIndexBuffer = t.getIndexBuffer();
     this.numOfIndices = t.getNumberOfIndices();
     this.mFVertexBuffer.position(0);
     this.mIndexBuffer.position(0);
     this.mFTextureBuffer.position(0);

   }
```

```
  //overriden method
   protected void draw(GL10 gl)
   {
      prepareBuffers(sides);
      gl.glEnable(GL10.GL_TEXTURE_2D);
      gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
      gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mFTextureBuffer);
      gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
            GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
   }
}
```

As you can see, most of the heavy lifting is carried out the by abstract textured renderer class and the RegularPolygon calculated the texture mapping vertices (see Listing 20–26).

Once we have this renderer available, we will need to add the code in Listing 20–33 to MultiviewTestHarness in Listing 20–12 to test the textured square.

Listing 20–33. *Responding to Textured Square Menu Item*

```
if (mid == R.id.mid_textured_square)
{
    mTestHarness.setRenderer(new TexturedSquareRenderer(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(mTestHarness);
    return;
}
```

Now if we run the program again and choose the menu item "Textured Square," we will see the textured square drawn as shown In Figure 20–11

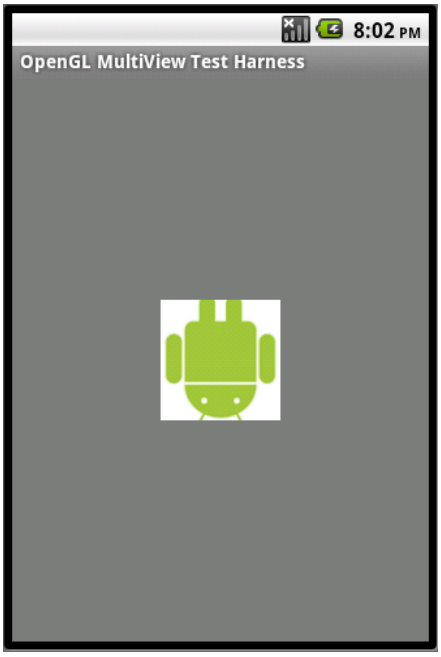

Figure 20–11. *A textured square*

Drawing Multiple Figures

Every example in this chapter so far has involved drawing a simple figure following a standard pattern. The pattern is: set up the vertices, load the texture, set up texture coordinates, and draw a single figure. What happens if we want to draw two figures? What if we want to draw a triangle using traditional means of specifying vertices and then a polygon using shapes such as the RegularPolygon? How do we specify combined vertices? Do we have to specify the vertices one time for both objects and then call the draw method?

As it turns out, between two draw() calls of the Android OpenGL Renderer interface, OpenGL allows us to issue multiple glDraw methods. Between these multiple glDraw methods, we can set up fresh vertices and textures. All of these drawing methods will then go to the screen once the draw() method completes.

There is another trick we can use to draw multiple figures with OpenGL. Consider the polygons we have created so far. These polygons have the capability to render themselves at any origin by taking the origin as an input. As it turns out, OpenGL can do this natively where it allows us to specify a RegularPolygon always at (0,0,0) and have the "translate" mechanism of OpenGL move it off of the origin to the desired position. We can do the same again with another polygon and translate it to a different position, thereby drawing two polygons at two different places on the screen.

Listing 20–34 demonstrates these ideas by drawing the textured polygon multiple times.

Listing 20–34. *Textured Polygon Renderer*

```
public class TexturedPolygonRenderer extends AbstractSingleTexturedRenderer
{
  //Number of points or vertices we want to use
   private final static int VERTS = 4;

   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFVertexBuffer;

   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFTextureBuffer;

   //A raw native buffer to hold indices
   //allowing a reuse of points.
   private ShortBuffer mIndexBuffer;

   private int numOfIndices = 0;

   private long prevtime = SystemClock.uptimeMillis();
  private int sides = 3;

   public TexturedPolygonRenderer(Context context)
   {
     super(context,com.androidbook.OpenGL.R.drawable.robot);
prepareBuffers(sides);
   }

   private void prepareBuffers(int sides)
   {
   RegularPolygon t = new RegularPolygon(0,0,0,0.5f,sides);
this.mFVertexBuffer = t.getVertexBuffer();
   this.mFTextureBuffer = t.getTextureBuffer();
   this.mIndexBuffer = t.getIndexBuffer();
   this.numOfIndices = t.getNumberOfIndices();
   this.mFVertexBuffer.position(0);
   this.mIndexBuffer.position(0);
   this.mFTextureBuffer.position(0);
   }

  //overriden method
   protected void draw(GL10 gl)
   {
      long curtime = SystemClock.uptimeMillis();
      if ((curtime - prevtime) > 2000)
```

```
        {
          prevtime = curtime;
          sides += 1;
          if (sides > 20)
          {
            sides = 3;
          }
           this.prepareBuffers(sides);
        }
        gl.glEnable(GL10.GL_TEXTURE_2D);

        //Draw once to the left
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mFTextureBuffer);

        gl.glPushMatrix();
        gl.glScalef(0.5f, 0.5f, 1.0f);
        gl.glTranslatef(0.5f,0, 0);
        gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
              GL10.GL_UNSIGNED_SHORT, mIndexBuffer);

        //Draw again to the right
        gl.glPopMatrix();
        gl.glPushMatrix();
        gl.glScalef(0.5f, 0.5f, 1.0f);
        gl.glTranslatef(-0.5f,0, 0);
        gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
              GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
        gl.glPopMatrix();
    }
}
```

This example demonstrates the following concepts:

- Drawing using shapes.
- Drawing multiple shapes using transformation matrices.
- Providing textures.
- Providing animation.

The main code in Listing 20–34 responsible for drawing multiple times is in the method draw(). We have highlighted corresponding lines in that method. Note that inside one draw() invocation we have called glDrawElements twice. Each of these times we set up the drawing primitives independent of the other time.

One more point to clarify is the use of transformation matrices. Every time glDrawElements() is called, it uses a specific transformation matrix. If we were to change this to alter the position of the figure (or any other aspect of the figure), we would need to set it back to the original so that the next drawing could correctly draw. This is accomplished through the push and pop operations provided on the OpenGL matrices.

Once we have this renderer available, we will need to add the code in Listing 20–35 to MultiviewTestHarness in Listing 20–12 to test the drawing of multiple figures.

Listing 20–35. *Responding to Multiple Figures Menu Item*

```
if (mid == R.id.mid_multiple_figures)
{
    mTestHarness.setRenderer(new TexturedPolygonRenderer(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
    setContentView(mTestHarness);
    return;
}
```

If we run the program again and choose the menu item "Multiple Figures," we will see two sets of changing polygons drawn (as shown in Figure 20–12) at the beginning of the animation. (Note that we have set the render mode to continuous.)

Figure 20–12. *A pair of textured polygons*

Figure 20–13 shows the same exercise in the middle of the animation.

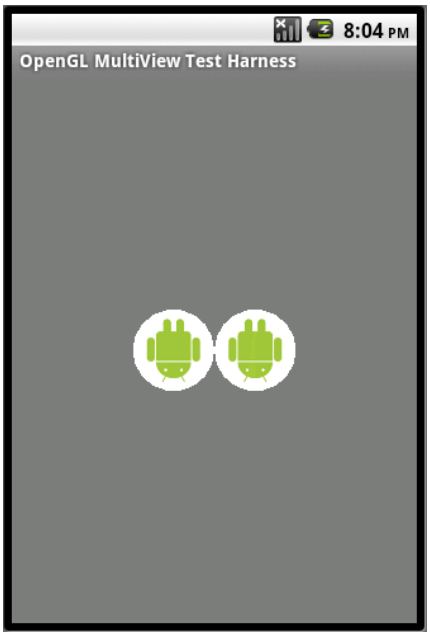

Figure 20–13. *A pair of textured circles*

This concludes another important concept in OpenGL. This section showed how to accumulate a number of different figures or scenes and draw them in tandem so that the end result forms a fairly complex OpenGL scene.

Next, we'll talk about Android support for OpenGL ES 2.0.

OpenGL ES 2.0

The good news is that Android not only supports OpenGL ES 2.0 but also provides Java bindings to the API starting with Android 2.2 (or API Level 8). However, keep the following restrictions in mind:

- OpenGL ES 2.0 is not supported yet on the emulator.
- OpenGL ES 2.0 is significantly different for a beginner, and most OpenGL books are coming out with new editions to cover this aspect of OpenGL. The programmability demanded by OpenGL ES 2.0 on the GPU (Graphics Processing Unit) puts a lot of complexity on the emulator. As a result, it's not even clear when Android will support OpenGL ES 2.0 on the emulator

- The only way to test/learn OpenGL ES 2.0 on Android SDK is to use a real device that runs Android 2.2 or above.

OpenGL ES 2.0 is a very different animal than OpenGL ES 1.x. It is not backward compatible. For beginners, it is most different in its initialization and learning how to draw the simplest of drawings.

It would take many pages to cover OpenGL ES 2.0 thoroughly. Instead, we will present you the basic initialization needed to get started with ES 2.0. Once you have this basic harness, you can consult the references at the end of this chapter to apply the OpenGL ES 2.0 into this framework.

The power of OpenGL ES 2.0 comes from the ability to write programs for the GPU that get compiled at run time and interpret how to draw vertices and fragments. These programs are called shaders. Unfortunately these shaders are necessary even for the simplest of OpenGL ES 2.0 programs. In that sense understanding shaders is mandatory for OpenGL ES 2.0.

Learning the OpenGL Shader Language is necessary to learn OpenGL ES 2.0. We have included a number of references at the end of this chapter to help you with this.

Java Bindings for OpenGL ES 2.0

The Java bindings for this API on Android are available in the package android.opengl.GLES20. All the functions of this class are static and correspond to the respective C APIs in the Khronos spec. (The URL can be found in the references section)

The GLSurfaceView and the corresponding Renderer abstraction introduced in the book for OpenGL ES 1.0 are also applicable to OpenGL ES 2.0. We will cover this soon. The documentation for this aspect is in the API documentation for the function GLSurfaceView.setEGLContextClientVersion.

First, let's see how to figure out if the device or the emulator supports this version of OpenGL ES 2.0 by using the code in Listing 20–36.

Listing 20–36. *Detecting OpenGL ES 2.0 Availability*

```
private boolean detectOpenGLES20() {
    ActivityManager am =
       (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    ConfigurationInfo info = am.getDeviceConfigurationInfo();
    return (info.reqGlEsVersion >= 0x20000);
}
```

Once you have this function (detectOpenGLES20**),** you can start using the GLSurfaceView, as shown in Listing 20–37, in your activity.

Listing 20–37. *Using GLSurfaceView for OpenGL ES 2.0*

```
if (detectOpenGLES20())
{
    GLSurfaceView glview = new GLSurfaceView(this);
    // glview.setEGLConfigChooser(false);
    glview.setEGLContextClientVersion(2);

    glview.setRenderer(new YourGLES20Renderer(this));
    glview.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(glview);
}
```

Notice how the GLSurfaceView is configured to use OpenGL ES 2.0 by setting the client version to "2". Then the class YourGLESRenderer will be similar to the Renderer classes introduced in this chapter. However, in the body of the renderer class, you will be using the GLES20 APIs instead of the GL10 APIs.

In the example we are going to develop, this renderer class is called ES20SimpleTriangleRenderer. We will introduce this class shortly. But let's first look at the activity class in Listing 20–38 that combines code snippets from Listing 20–36 and Listing 20–37.

Listing 20–38. *OpenGL20MultiViewTestHarness Activity*

```
public class OpenGL20MultiViewTestHarnessActivity extends Activity
{
  final String tag="es20";
  private GLSurfaceView mTestHarness;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      if (detectOpenGLES20())
      {
         mTestHarness = new GLSurfaceView(this);
         //DO NOT call the followign function
         //mTestHarness.setEGLConfigChooser(false);
         mTestHarness.setEGLContextClientVersion(2);
      }
      else
      {
         throw new RuntimeException("20 not supported");
      }

      Intent intent = getIntent();
      int mid = intent.getIntExtra("com.ai.menuid", R.id.MenuId_OpenGL15_Current);
      if (mid == R.id.mid_es20_triangle)
```

```
        {
            mTestHarness.setRenderer(new ES20SimpleTriangleRenderer(this));
            mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
            setContentView(mTestHarness);
            return;
        }
        return;
    }
    private boolean detectOpenGLES20() {
        ActivityManager am =
            (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
        ConfigurationInfo info = am.getDeviceConfigurationInfo();
        return (info.reqGlEsVersion >= 0x20000);
    }
    @Override
    protected void onResume()    {
        super.onResume();
        mTestHarness.onResume();
    }
    @Override
    protected void onPause() {
        super.onPause();
        mTestHarness.onPause();
    }
}
```

The ES 2.0 test harness activity in Listing 20–38 is very similar to the test harness we presented for ES 1.x in Listing 20–12. You may be wondering why can't we just use that one and create a different menu option. Two reasons have prompted us to go in this direction.

The first one is we're not sure if we can reuse the SurfaceView between ES 1.x and ES 2.x menu invocations. We just want to be safe.

The second reason is that the way we initialize is different, so we don't want to confuse the code by combining both into one class. For example, for ES 2.0 initialization we check the supported ES version, etc.; such code would have clouded the simpler ES 1.x initialization in Listing 20–12.

Otherwise, the motivation for this ES 2.x test harness is identical to that of the ES1.x test harness.

To be able to use OpenGL ES 2.0 features in our activities such as the one in Listing 20–38, we need to include the following <uses-feature> as a child of the application node (see Listing 20–39).

Listing 20–39. *Using OpenGL ES 2.0 Feature*

```
<application…>
```

```
……other nodes
 <uses-feature android:glEsVersion="0x00020000" />
 </application>
```

As we will be able to test OpenGL ES 2.0 applications only on a real device, we need to specify our application as debuggable using the debuggable attribute of the application node, as shown in Listing 20–40.

Listing 20–40. *Specifying a Debuggable Application*

```
 <application android:icon="@drawable/icon"
          android:label="OpenGL Test Harness"
          android:debuggable="true">
```

To be able to invoke the ES 2.0 test harness activity, we will need to change the driver activity in Listing 20–14 so that it looks like the code in Listing 20–41.

Listing 20–41. *New Main Driver Activity*

```
public class TestOpenGLMainDriverActivity extends Activity {
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
   }
   @Override
   public boolean onCreateOptionsMenu(Menu menu){
      super.onCreateOptionsMenu(menu);
      MenuInflater inflater = getMenuInflater(); //from activity
      inflater.inflate(R.menu.main_menu, menu);
      return true;
   }
   @Override
   public boolean onOptionsItemSelected(MenuItem item)
   {
      if (item.getItemId() >= R.id.mid_es20_triangle)
      {
         this.invoke20MultiView(item.getItemId());
         return true;
      }
      this.invokeMultiView(item.getItemId());
      return true;
   }
   private void invokeMultiView(int mid)
   {
      Intent intent = new Intent(this,MultiViewTestHarnessActivity.class);
```

```
        intent.putExtra("com.ai.menuid", mid);
        startActivity(intent);
    }
    private void invoke20MultiView(int mid)
    {
        Intent intent = new Intent(this,OpenGL20MultiViewTestHarnessActivity.class);
        intent.putExtra("com.ai.menuid", mid);
        startActivity(intent);
    }
}
```

We have inserted two code additions in Listing 20–41: an additional method to invoke the OpenGL20MultiViewTestHarnessActivity that we invoke if the menu ID is above or equal to the "mid_es20_triangle". The thought is that this menu item will start off the demos for ES 2.0. However, we have only one demo of ES 2.0 at this time.

Rendering Steps

Rendering a figure in OpenGL ES 2.0 requires the following steps:

1. Write shader programs that run on the GPU to extract such things as drawing coordinates and model/view/projection matrices from the client memory and draw them. There is no counterpart to this in OpenGL ES 1.0. In a simplistic sense, this is an additional level of indirection before the vertices are drawn and surfaces painted.

2. Compile the source code of shaders from step 1on the GPU.

3. Link the compiled units in step 2 into a program object that can be used at drawing time.

4. Retrieve address handlers from the program in step 3 so that data can be set into those pointers.

5. Define your vertex buffers.

6. Define your model view matrices (this is done through such things as setting the frustum, camera position, etc.; it's very similar to how it's done in OpenGL ES 1.1).

7. Pass the items from step 5 and 6 to the program through the handlers.

8. Finally, draw.

We will examine each of the steps through code snippets and then present a working renderer paralleling the SimpleTriangleRenderer that was presented as part of the OpenGL ES 1.0. Let's start with the key difference of OpenGL ES 2.0, namely shaders.

Understanding Shaders

Even the simplest of drawings in OpenGL ES 2.0 requires program segments called *shaders*. These shaders form the core of OpenGL ES 2.0. We will explain the minimum necessary to accomplish drawing of a simple triangle; we advise you to read the resources listed in the reference section at the end of this chapter.

Any drawing that involves vertices is carried out by *vertex shaders*. Any drawing that involves a fragment, the space between vertices, is carried out by *fragment shaders*. So a vertex shader is concerned with only vertex points. However, a fragment shader deals with every pixel.

Listing 20–42 is an example of a vertex shader program segment.

Listing 20–42. *A Simple Vertex Shader*

```
uniform mat4 uMVPMatrix;
attribute vec4 aPosition;
void main() {
  gl_Position = uMVPMatrix * aPosition;
}
```

This program is written in the shading language. The first line indicates that the variable uMVPMatrix is an input variable to the program and it is of type mat4 (a 4x4 matrix). It is also qualified as a uniform variable because this matrix variable applies to all the vertices and not to any specific vertex.

In contrast, the variable aPosition is a vertex attribute that deals with the position of the vertex (coordinates). It is identified as an attribute of the vertex and is specific to a vertex. The other attributes of a vertex include color, texture, etc. This aPosition variable is a 4 point vector as well. Now the program itself, Listing 20–42, is taking the coordinate position of the vertex and transforming it using a Model View Projection (MVP) matrix (which will be set by the calling program) and multiplying the coordinate position of the vertex to arrive at a final position identified by the reserved gl_Position of the vertex shader.

This vertex shader program is responsible for drawing or positioning the vertices. The calling program, for example, will set the buffer for the vertices of a triangle, for instance, as follows in Listing 20–43.

Listing 20–43. *Setting Data for the Vertices*

```
GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT, false,
            TRIANGLE_VERTICES_DATA_STRIDE_BYTES, mFVertexBuffer);
```

The vertex buffer is the last argument of this GLES 20 method. This looks very much like the glVertexPointer in OpenGL 1.0 except for the first argument, which is identified as

positionHandle. This argument points to the aPostion input attribute variable from the vertex shader program in Listing 20–42. You get this handle using code similar to the following:

```
positionHandle = GLES20.glGetAttribLocation(shaderProgram, "aPosition");
```

Essentially, you are asking the shader program to give a handle to an input variable and go from there. The shaderProgram itself needs to be constructed by passing the shader code segments to the GPU and compiling them and linking them. To make a program where you can start to draw, you also need a fragment shader. Listing 20–44 is an example of a fragment shader.

Listing 20–44. *Example of a Fragment Shader*

```
void main() {
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Again, we take the reserved variable gl_FragColor and hardcode it to the color red. Instead of hardcoding it to red as in Listing 20–44, we can pass these color values all the way from the user program through the vertex shader to the fragment shader. This is a bit out of scope for this chapter but is clearly demonstrated in many of the indicated reference materials on OpenGL ES 2.0

These shader programs are mandatory to start drawing.

Compiling Shaders into a Program

Once we have the shader program segments as seen in Listing 20–42 and 20–44, we can use the code in Listing 20–45 to compile and load a shader program.

Listing 20–45. *Compiling and Loading a Shader Program*

```
private int loadShader(int shaderType, String source) {
   int shader = GLES20.glCreateShader(shaderType);
   if (shader != 0) {
      GLES20.glShaderSource(shader, source);
      GLES20.glCompileShader(shader);
      int[] compiled = new int[1];
      GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS, compiled, 0);
      if (compiled[0] == 0) {
         Log.e(TAG, "Could not compile shader " + shaderType + ":");
         Log.e(TAG, GLES20.glGetShaderInfoLog(shader));
         GLES20.glDeleteShader(shader);
         shader = 0;
      }
   }
   return shader;
}
```

In this code segment, the shadertype is one of GLES20.GL_VERTEX_SHADER or GLES20.GL_FRAGMENT_SHADER. The variable source will need to point to a string containing the source, such as those shown in Listing 20–42 and 20–44.

Listing 20–46 shows how the function loadShader (from Listing 20–45) is utilized in constructing the program object.

Listing 20–46. *Creating a Program and Getting Variable Handles*

```
private int createProgram(String vertexSource, String fragmentSource) {
    int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vertexSource);
    if (vertexShader == 0) {
        return 0;
    }
    Log.d(TAG,"vertex shader created");
    int pixelShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentSource);
    if (pixelShader == 0) {
        return 0;
    }
    Log.d(TAG,"fragment shader created");
    int program = GLES20.glCreateProgram();
    if (program != 0) {
        Log.d(TAG,"program created");
        GLES20.glAttachShader(program, vertexShader);
        checkGlError("glAttachShader");
        GLES20.glAttachShader(program, pixelShader);
        checkGlError("glAttachShader");
        GLES20.glLinkProgram(program);
        int[] linkStatus = new int[1];
        GLES20.glGetProgramiv(program, GLES20.GL_LINK_STATUS, linkStatus, 0);
        if (linkStatus[0] != GLES20.GL_TRUE) {
            Log.e(TAG, "Could not link program: ");
            Log.e(TAG, GLES20.glGetProgramInfoLog(program));
            GLES20.glDeleteProgram(program);
            program = 0;
        }
    }
    return program;
}
```

Getting Access to the Shader Program Variables

Once the program is set up, the program's handle can be used to get handles for the input variables required by the shaders. Listing 20–47 shows how.

Listing 20–47. *Getting Vertex and Uniform Handles*

```
int maPositionHandle =
     GLES20.glGetAttribLocation(mProgram, "aPosition");
int muMVPMatrixHandle =
    GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
```

A Simple ES 2.0 Triangle

We now have covered all the basics necessary to put together a framework similar to the one we created for OpenGL 1.0. We will now put together an abstract renderer which will encapsulate all the initialization work (such as creating shaders, programs, etc.). Listing 20–48 shows the code.

Listing 20–48. *ES20AbstractRenderer*

```
public abstract class ES20AbstractRenderer
implements android.opengl.GLSurfaceView.Renderer
{
  public static String TAG = "ES20AbstractRenderer";

  private float[] mMMatrix = new float[16];
  private float[] mProjMatrix = new float[16];
  private float[] mVMatrix = new float[16];
  private float[] mMVPMatrix = new float[16];

  private int mProgram;
  private int muMVPMatrixHandle;
  private int maPositionHandle;

  public void onSurfaceCreated(GL10 gl, EGLConfig eglConfig)
  {
       prepareSurface(gl,eglConfig);
  }
  public void prepareSurface(GL10 gl, EGLConfig eglConfig)
  {
     Log.d(TAG,"preparing surface");
     mProgram = createProgram(mVertexShader, mFragmentShader);
     if (mProgram == 0) {
        return;
     }
     Log.d(TAG,"Getting position handle:aPosition");
     maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
     checkGlError("glGetAttribLocation aPosition");
     if (maPositionHandle == -1) {
        throw new RuntimeException("Could not get attrib location for aPosition");
     }
     Log.d(TAG,"Getting matrix handle:uMVPMatrix");
     muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
     checkGlError("glGetUniformLocation uMVPMatrix");
     if (muMVPMatrixHandle == -1) {
        throw new RuntimeException("Could not get attrib location for uMVPMatrix");
     }
  }
```

```
public void onSurfaceChanged(GL10 gl, int w, int h)
{
   Log.d(TAG,"surface changed. Setting matrix frustum: projection matrix");
   GLES20.glViewport(0, 0, w, h);
   float ratio = (float) w / h;
   Matrix.frustumM(mProjMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
}
public void onDrawFrame(GL10 gl)
{
   Log.d(TAG,"set look at matrix: view matrix");
   Matrix.setLookAtM(mVMatrix, 0, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

   Log.d(TAG,"base drawframe");
   GLES20.glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
   GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);

   GLES20.glUseProgram(mProgram);
   checkGlError("glUseProgram");

   draw(gl,this.maPositionHandle);
}
private int createProgram(String vertexSource, String fragmentSource) {
   int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vertexSource);
   if (vertexShader == 0) {
      return 0;
   }
     Log.d(TAG,"vertex shader created");
   int pixelShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentSource);
   if (pixelShader == 0) {
      return 0;
   }
   Log.d(TAG,"fragment shader created");
   int program = GLES20.glCreateProgram();
   if (program != 0) {
      Log.d(TAG,"program created");
      GLES20.glAttachShader(program, vertexShader);
      checkGlError("glAttachShader");
      GLES20.glAttachShader(program, pixelShader);
      checkGlError("glAttachShader");
      GLES20.glLinkProgram(program);
      int[] linkStatus = new int[1];
      GLES20.glGetProgramiv(program, GLES20.GL_LINK_STATUS, linkStatus, 0);
      if (linkStatus[0] != GLES20.GL_TRUE) {
         Log.e(TAG, "Could not link program: ");
         Log.e(TAG, GLES20.glGetProgramInfoLog(program));
         GLES20.glDeleteProgram(program);
         program = 0;
```

```
            }
        }
        return program;
    }
    private int loadShader(int shaderType, String source) {
        int shader = GLES20.glCreateShader(shaderType);
        if (shader != 0) {
            GLES20.glShaderSource(shader, source);
            GLES20.glCompileShader(shader);
            int[] compiled = new int[1];
            GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS, compiled, 0);
            if (compiled[0] == 0) {
                Log.e(TAG, "Could not compile shader " + shaderType + ":");
                Log.e(TAG, GLES20.glGetShaderInfoLog(shader));
                GLES20.glDeleteShader(shader);
                shader = 0;
            }
        }
        return shader;
    }
    private final String mVertexShader =
        "uniform mat4 uMVPMatrix;\n" +
        "attribute vec4 aPosition;\n" +
        "void main() {\n" +
        "  gl_Position = uMVPMatrix * aPosition;\n" +
        "}\n";

    private final String mFragmentShader =
        "void main() {\n" +
        "  gl_FragColor = vec4(0.5, 0.25, 0.5, 1.0);\n" +
        "}\n";

    protected void checkGlError(String op) {
        int error;
        while ((error = GLES20.glGetError()) != GLES20.GL_NO_ERROR) {
            Log.e(TAG, op + ": glError " + error);
            throw new RuntimeException(op + ": glError " + error);
        }
    }
    protected void setupMatrices()
    {
        Matrix.setIdentityM(mMMatrix, 0);
        Matrix.multiplyMM(mMVPMatrix, 0, mVMatrix, 0, mMMatrix, 0);
        Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mMVPMatrix, 0);
        GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, mMVPMatrix, 0);
    }
    protected abstract void draw(GL10 gl, int positionHandle);
```

```
}
```

Much of this code is an aggregation of the ideas introduced previously, except for one detail. The function setupMatrices demonstrates how the Matrix class is used to combine multiple matrices into a single matrix called mMVPMatrix by multiplying other matrices, starting with an Identity matrix.

So the variable mMMatrix is an Identity matrix. The variable mVMatrix is obtained by using the eyepoint API or the look-at point of the camera. The Projection matrix mProjMatrix is obtained by using the frustum specification. Both these concepts, the eye point and the frustum, are identical to the concepts covered in OpenGL ES 1.0. The MVP matrix is just a multiplication of these matrices. Finally, the call glUniformMatrix4fv sets this up as a variable in the vertex shader so that the vertex shader can multiply each vertex position with this matrix to get the final position (see Listing 20–42).

Listing 20–49 shows the code for GS20SimpleTriangleRenderer that extends the abstract renderer and the minimum necessary to define the points and draw.

Listing 20–49. *ES20SimpleTriangleRenderer*

```
public class ES20SimpleTriangleRenderer extends ES20AbstractRenderer
{
   //A raw native buffer to hold the point coordinates
   private FloatBuffer mFVertexBuffer;
   private static final int FLOAT_SIZE_BYTES = 4;
   private final float[] mTriangleVerticesData = {
         // X, Y, Z
         -1.0f, -0.5f, 0,
         1.0f, -0.5f, 0,
         0.0f,  1.11803399f, 0 };

   public ES20SimpleTriangleRenderer(Context context)
   {
      ByteBuffer vbb = ByteBuffer.allocateDirect(mTriangleVerticesData.length
            * FLOAT_SIZE_BYTES);
      vbb.order(ByteOrder.nativeOrder());
      mFVertexBuffer = vbb.asFloatBuffer();
      mFVertexBuffer.put(mTriangleVerticesData);
      mFVertexBuffer.position(0);
   }

   protected void draw(GL10 gl, int positionHandle)
   {
      GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT, false,
            0, mFVertexBuffer);
      checkGlError("glVertexAttribPointer maPosition");
```

```
        GLES20.glEnableVertexAttribArray(positionHandle);
        checkGlError("glEnableVertexAttribArray maPositionHandle");

        this.setupMatrices();
        GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, 3);
        checkGlError("glDrawArrays");
    }
}
```

Now if we invoke the activity in Listing 20–38, we will see a triangle drawn to the dimension specified. To do this, we will need the following additional files:

- ES20AbstractRenderer.java (Listing 20–48)
- ES20SimpleTriangleRenderer.java (Listing 20–49)
- OpenGL20MultiveTestHarnessActivity.java (Listing 20–38)

Once we have these files compiled, we can run the program again and chose the menu option "ES20 Triangle." This will display a single triangle very similar to that in Figure 20–3.

However, as suggested, it will not work on the emulator. You have to hook up a real device to eclipse to test this. We tested it using the first Motorola Droid from Verizon. The directions to hook up a device are covered in the second chapter of this book. We have also included a URL in the reference section where we intend to update our notes to cover a variety of devices.

Further Reading on OpenGL ES 2.0

The "References" section can help you with resources on OpenGL ES 2.0. Once you get a feel for these shader programs, the background on OpenGL 1.0, and the few starting points on OpenGL ES 2.0, you can follow the samples in the Android SDK to make headway when you have suitable hardware.

Finally, we have included all of the sample code for the OpenGL ES 20 triangle in the downloadable project. The triangle has all the steps needed for OpenGL ES 2.0.

Instructions for Compiling the Code

The best way to play around with the code listed in this chapter is to download the ZIP file dedicated for this chapter. The URL for this file is listed in the "References" section. Every class file listed here is in the ZIP file. If you want to write your program directly from the listings, we

have included all files here. There may be few resources wanting such as the starting icon, etc. If you are not sure how to hook those up, download the ZIP file.

References

We have found the following resources useful in understanding and working with OpenGL:

- Android's android.opengl package reference URL: http://developer.android.com/reference/android/opengl/GLSurfaceView.html.
- The Khronos Group's OpenGL ES Reference Manual. www.khronos.org/opengles/documentation/opengles1_0/html/index.html.
- OpenGL Programming Guide (the red book). www.glprogramming.com/red/. Although this online reference is handy, it stops at OpenGL 1.1. You will need to buy the 7th edition for information on the recent stuff including OpenGL shaders.
- The following is a very good article on texture mapping from Microsoft: http://msdn.microsoft.com/en-us/library/ms970772(printer).aspx.
- You can find very insightful course material on OpenGL from Wayne O. Cochran from Washington State University at this URL: http://ezekiel.vancouver.wsu.edu/~cs442/.
- Documentation for JSR 239 (Java Binding for the OpenGL ES API) is at http://java.sun.com/javame/reference/apis/jsr239/.
- The man pages at khronos.org for OpenGL ES 2.0 are useful as a reference but not a guide. www.khronos.org/opengles/sdk/docs/man/.
- Understanding shading language is essential to understand the new OpenGL direction including the OpenGL ES 2.0. www.opengl.org/documentation/glsl/
- *OpenGL Shading Language*, 3rd Edition, Randi J Rost, etc. We haven't personally read this book but it seems promising.
- GLES20 API reference from the Android SDK. http://developer.android.com/reference/android/opengl/GLES20.html
- GLSurfaceView Reference. http://developer.android.com/reference/android/opengl/GLSurfaceView.html#setEGLContextClientVersion(int)

- You can find one of the authors of this book's research on OpenGL here: http://www.androidbook.com/akc/display?url=NotesIMPTitlesURL&ownerUserId=satya&folderName=OpenGL&order_by_format=news.
- You can find one of the authors of this book's research on OpenGL textures here: http://www.androidbook.com/item/3190.
- How to run Android applications on the device from Eclipse ADB. http://www.androidbook.com/item/3574
- Download the test project dedicated for this chapter at www.androidbook.com/projects. The name of the ZIP file is ProAndroid3_ch20_TestOpenGL.zip.

Summary

We have covered a lot of ground in OpenGL—especially if you are new to OpenGL programming. We would like to think that this is a great introductory chapter on OpenGL, not only for Android but any other OpenGL system.

In this chapter, you learned the fundamentals of OpenGL. You learned the Android-specific API that allows you to work with OpenGL standard APIs. You played with shapes and textures, and you learned how to use the drawing pipeline to draw multiple figures. You were introduced to OpenGL ES 2.0, its shading language, the basic differences from OpenGL 1.0, and a set of references to further explore OpenGL ES 2.0.