Chapter 7

OpenGL ES: A Gentle Introduction

Mr. Nom was a great success. Due to our good initial design and the game framework we wrote, actually implementing Mr. Nom was a breeze. Best of all, the game runs smoothly even on low-end devices. Of course, Mr. Nom is not a very complex or graphically intense game, so using the `Canvas` API for rendering was a good idea.

However, once you want to do something more complex—say, something like Replica Island—you will hit a wall: `Canvas` just can't keep up with the visual complexity of such a game. And if you want to go fancy-pants 3D, `Canvas` won't help you either. So what can we do?

This is where OpenGL ES comes to the rescue. In this chapter we'll first briefly look at what OpenGL ES actually is and does. We'll then focus on using OpenGL ES for 2D graphics, without having to dive into the more mathematically complex realms of using the API for 3D graphics (we'll get to that in a later chapter). We'll take baby steps at first, as OpenGL ES can get quite involved. So, let's get to know OpenGL ES.

What Is OpenGL ES and Why Should I Care?

OpenGL ES is an industry standard for (3D) graphics programming. It is especially targeted at mobile and embedded devices. It is maintained by the Khronos Group, which is a conglomerate of companies including ATI, NVIDIA, and Intel, who together define and extend the standard.

Speaking of standards, there are currently three incremental versions of OpenGL ES: 1.0, 1.1, and 2.0. The first two are the ones we are concerned with in this book. All Android devices support OpenGL ES 1.0, and most also support 1.1, which adds some new features to the 1.0 specification. OpenGL ES 2.0, however, breaks compatibility with the 1.x versions. You can use either 1.x or 2.0, but not both at the same time. The reason for this is that the 1.x versions use a programming model called *fixed-function pipeline*, while 2.0 lets you programmatically define parts of the rendering pipeline via *so-called* shaders.

M. Zechner, *Beginning Android Games*
© Mario Zechner 2011

Many of the second-generation devices already support OpenGL ES 2.0; however, the Java bindings are currently not in a usable state (unless you target the new Android 2.3). OpenGL ES 1.x is more than good enough for most games, though, so we will stick to it here.

NOTE: The emulator only supports OpenGL ES 1.0. The implementation is a little shoddy, though, so never rely on the emulator for testing. Use a real device.

OpenGL ES is an API that comes in the form of a set of C header files provided by the Khronos group, along with a very detailed specification of how the API defined in those headers should behave. This includes things such as how pixels and lines have to be rendered. Hardware manufacturers then take this specification and implement it for their GPU on top of the GPU driver. The quality of these implementations varies a little; some companies strictly adhere to the standard (PowerVR) while others seem to have difficulty sticking to the standard. This can sometimes result in GPU-dependent bugs in the implementation that have nothing to do with Android itself, but with the hardware drivers provided by the manufacturers. I'll point out any device-specific issues along our way into OpenGL ES land.

NOTE: OpenGL ES is more or less a sibling of the more feature-rich desktop OpenGL standard. It deviates from the latter in that some of the functionality is reduced or completely removed. Nevertheless, it is possible to write an application that can run with both specifications, which is great if you want to port your game to the desktop as well.

So what does OpenGL ES actually do? The short answer is that's it's a lean and mean triangle-rendering machine. The long answer is a little bit more involved.

The Programming Model: An Analogy

OpenGL ES is in general a 3D graphics programming API. As such it has a pretty nice and (hopefully) easy-to-understand programming model that we can illustrate with a simple analogy.

Think of OpenGL ES as working like a camera. To take a picture you have to first go to the scene you want to photograph. Your scene is composed of objects—say, a table with more objects on it. They all have a position and orientation relative to your camera, as well as different materials and textures. Glass is translucent and a little reflective, a table is probably made out of wood, a magazine has the latest photo of some politician on it, and so on. Some of the objects might even move around (e.g., a fruit fly you can't get rid of). Your camera also has some properties, such as focal length, field of view, image resolution and size the photo will be taken at, and its own position and orientation within the world (relative to some origin). Even if both objects and the camera are moving, when you press the button to take the photo you catch a still image of the scene (for now we'll neglect the shutter speed, which might cause a blurry image). For that infinitely small moment everything stands still and is well defined, and the picture reflects exactly all those configurations of positions, orientations, textures, materials, and lighting. Figure 7–1 shows an abstract scene with a camera, a light, and three objects with different materials.

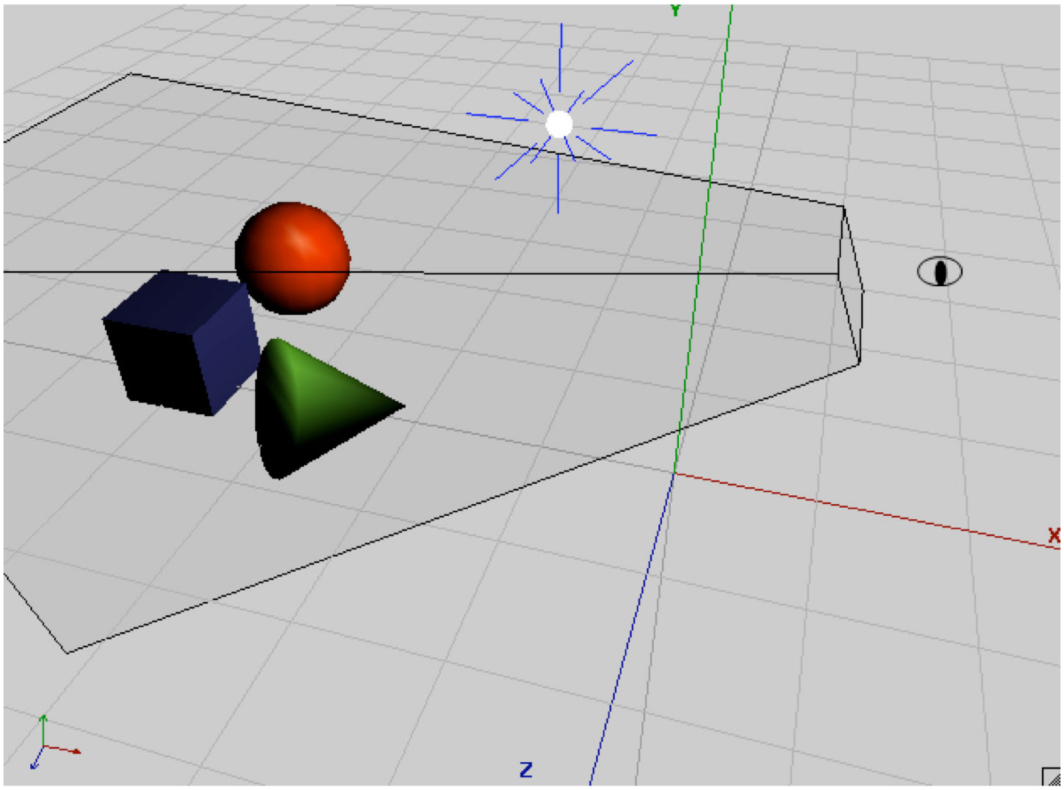

Figure 7–1. *An abstract scene*

Each object has a position and orientation relative to the scene's origin. The camera, indicated by the eye, also has a position in relation to the scene's origin. The pyramid in Figure 7–1 is the so-called *view volume* or *view frustum*, which shows how much of the scene the camera captures and how the camera is oriented. The little white ball with the rays is our light source in the scene, which also has a position relative to the origin.

We can directly map this scene to OpenGL ES, but to do so we need to define a couple of things:

- *Objects (aka models)*: These are generally composed of two four: their geometry, as well as their color, texture, and material. The geometry is specified as a set of triangles. Each triangle is composed of three points in 3D space, so we have x-, y-, and z coordinates defined relative to the coordinate system origin, as in Figure 7–1. Note that the z-axis points toward us. The color is usually specified as an RGB triple, as we are already used to. Textures and materials are little bit more involved. We'll get to those later on.

- *Lights*: OpenGL ES offers us a couple of different light types with various attributes. They are just mathematical objects with a position and/or direction in 3D space, plus attributes such as color.
- *Camera*: This is also a mathematical object that has a position and orientation in 3D space. Additionally it has parameters that govern how much of the image we see, similar to a real camera. All this things together define a view volume, or view frustum (indicated as the pyramid with the top cut off in Figure 7–1). Anything inside this pyramid can be seen by the camera; anything outside will not make it into the final picture.
- *Viewport*: This defines the size and resolution of the final image. Think of it as the type of film you put into your analog camera or the image resolution you get for pictures taken with your digital camera.

Given all this, OpenGL ES can construct a 2D bitmap of our scene from the point of view of the camera. Notice that we define everything in 3D space. So how can OpenGL ES map that to two dimensions?

Projections

This 2D mapping is done via something called *projection*. We already mentioned that OpenGL ES is mainly concerned with triangles. A single triangle has three points defined in 3D space. To render such a triangle to the framebuffer, OpenGL ES needs to know the coordinates of these 3D points within the pixel-based coordinate system of the framebuffer. Once it knows those three corner-point coordinates, it can simply draw the pixels in the framebuffer that are inside the triangle. We could even write our own little OpenGL ES implementation by projecting 3D points to 2D, and simply draw lines between them via the `Canvas`.

There are two kinds of projections that are commonly used in 3D graphics. :

- *Parallel, or orthographic, projection*: If you have ever played with a CAD application you might already know about these. A parallel projection doesn't care how far an object is away from the camera; the object will always have the same size in the final image. This type of projection is typically used for rendering 2D graphics in OpenGL ES.
- *Perspective projections*: These are what we are used to when using our eyes. Objects further away from us will appear smaller on our retina. This type of projection is typically used when we do 3D graphics with OpenGL ES.

In both cases we need something called a *projection plane*. This is nearly exactly the same as the retina of our eyes. It's where the light is actually registered to form the final image. While a mathematical plane is infinite, our retina is limited in area. Our OpenGL

ES "retina" is equal to the rectangle at the top of the view frustum in Figure 7–1. This part of the view frustum is where OpenGL ES will project the points to. It is called the *near clipping plane* and has its own little 2D coordinate system. Figure 7–2 shows that near clipping plane again, from the point of view of the camera, with the coordinate system superimposed.

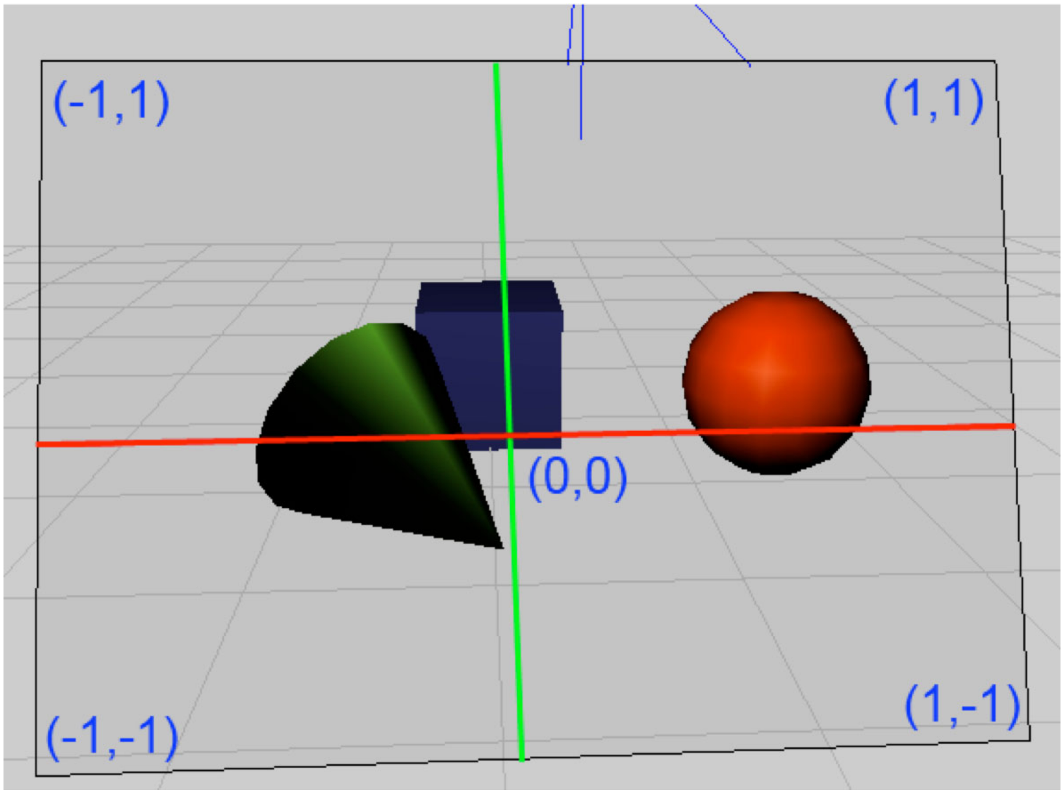

Figure 7–2. *The near clipping plane (also known as the projection plane) and its coordinate system.*

Note that the coordinate system is by no means fixed. We can manipulate it so that we can work in any projected coordinate system we like (e.g., we could instruct OpenGL ESto let the origin be in the bottom-left corner, and let the visible area of the "retina" be 480 units on the x-axis and 320 units on the y-axis). Sounds familiar? Yes, OpenGL ES allows us to specify any coordinate system we want for the projected points.

Once we specify our view frustum, OpenGL ES then takes each point of a triangle and shoots a ray from it through the projection plane. The difference between a parallel and a perspective projection is how the direction of those rays is constructed. Figure 7–3 shows the difference between the two, viewed from above.

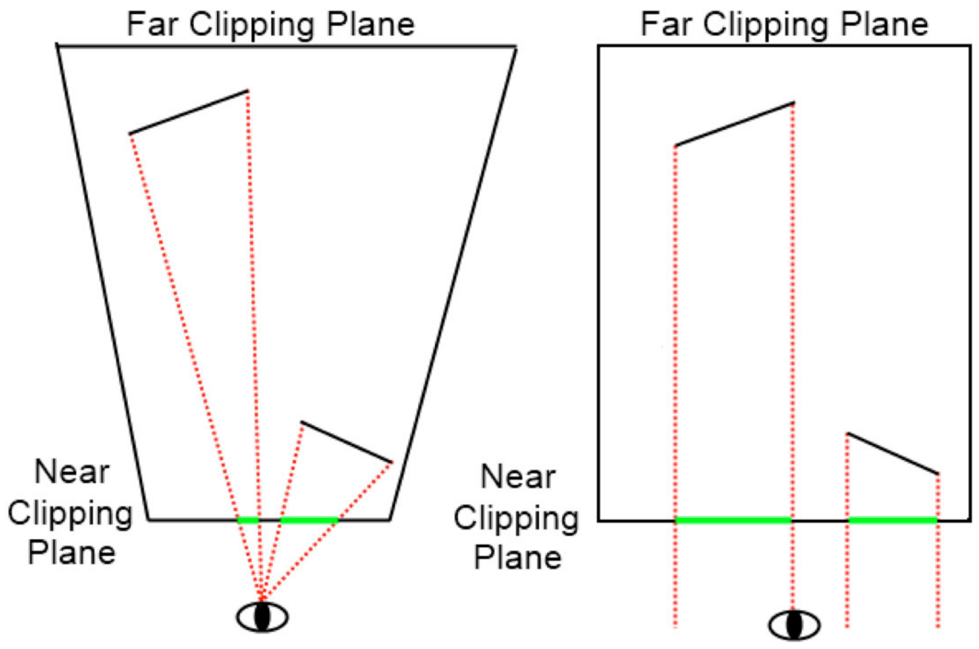

Figure 7–3. *A perspective projection (left) and a parallel projection (right)*

A perspective projection shoots the rays from the triangle points through the camera (or eye, in this case). Objects further away will thus appear smaller on the projection plane. When we use a parallel projection, the rays are shot perpendicular to the projection plane. In this case an object will keep its size on the projection plane no matter how far away it is.

Our projection plane is called a near clipping plane in OpenGL ES lingo, as pointed out earlier. All of the sides of the view frustum have similar names. The one furthest away from the camera is called the far clipping plane. The others are called the left, right, top, and bottom clipping planes. Anything outside or behind those planes will not be rendered. Objects that are partially within the view frustum will be clipped from these planes, meaning that the parts outside the view frustum get cut away. That’s where the name *clipping plane* comes from.

You might be wondering why the view frustum of the parallel projection case in Figure 7–3 is rectangular. It turns out that the projection is actually governed by how we define our clipping planes. In the case of a perspective projection, the left, right, top, and bottom clipping planes are not perpendicular to the near and far planes (see Figure 7–3, which only shows the left and right clipping planes. In the case of the parallel projection, these planes are perpendicular, which tells OpenGL ES to render everything at the same size no matter how far away it is from the camera.

Normalized Device Space and the Viewport

Once OpenGL ES has figured out the projected points of a triangle on the near clipping plane, it can finally translate them to pixel coordinates in the framebuffer. For this, it must first transform the points to so-called *normalized device space*. This equals the coordinate system depicted in Figure 7–2. Based on these normalized device space coordinates OpenGL ES calculates the final framebuffer pixel coordinates via the following simple formulas:

```
pixelX = (norX + 1) / (viewportWidth + 1) + norX
pixelY = (norY + 1) / (viewportHeight +1) + norY
```

where `norX` and `norY` are the normalized device coordinates of a 3D point, and `viewportWidth` and `viewportHeight` are the size of the viewport in pixels on the x- and y-axes. We don't have to worry about the normalized device coordinates all that much, as OpenGL will do the transformation for us automagically. What we do care about, though, are the viewport and the view frustum.

Matrices

Later you will see how to specify a view frustum, and thus a projection. OpenGL ES expresses projections in the form of so-called *matrices*. For our purposes we don't need to know the internals of matrices. We only need to know what they do to the points we define in our scene. Here's the executive summary of matrices:

- A matrix encodes transformations to be applied to a point. A transformation can be a projection, a translation (in which the point is moved around), a rotation around another point and axis, or a scale, among other thing.
- By multiplying such a matrix with a point, we apply the transformation to the point. For example, multiplying a point with a matrix that encodes a translation by 10 units on the x-axis will move the point 10 units on the x-axis and thereby modify its coordinates.
- We can concatenate transformations stored in separate matrices into a single matrix by multiplying the matrices. When we multiply this single concatenated matrix with a point, all the transformations stored in that matrix will be applied to that point. The order in which the transformations are applied is dependent on the order in which we multiplied the matrices with each other.

- There's a special matrix called an *identity matrix*. If we multiply a matrix or a point with it, nothing will happen. Think of multiplying a point or matrix by an identity matrix as multiplying a number by 1. It simply has no effect. The relevance of the identity matrix will become clear once we learn how OpenGL ES handles matrices (see the section "Matrix Modes and Active Matrices"). A classic hen and egg problem.

NOTE: When I talk about points in this context, I actually mean 3D vectors.

OpenGL ES has three different matrices that it applies to the points of our models:

- *Model-view matrix*: We can use this matrix to move, rotate, or scale the points of our triangles around (this is the *model* part of the model-view matrix). This matrix is also used to specify the position and orientation of our camera (this is the *view* part).
- *Projection matrix*: The name says it all—this matrix encodes a projection, and thus the view frustum of our camera.
- *Texture matrix*: This matrix allow us to manipulate so-called texture coordinates (which we'll discuss later). However, we'll avoid using this matrix in this book since this part of OpenGL ES is broken on a couple of devices thanks to buggy drivers.

The Rendering Pipeline

OpenGL ES keeps track of these three matrices. Each time we set one of the matrices, it will remember it until we change the matrix again. In OpenGL ES speak, this is called a state. OpenGL keeps track of more than just the matrix states, though; it also keeps track of whether we want it to alpha-blend triangles, whether we want lighting to be taken into account, which texture should be applied to our geometry, and so on. In fact, OpenGL ES is one huge state machine. We set its current state, feed it the geometries of our objects, and tell it to render an image for us. Let's see how a triangle passes through this mighty triangle-rendering machine. Figure 7–4 shows a very high-level, simplified view of the OpenGL ES pipeline:

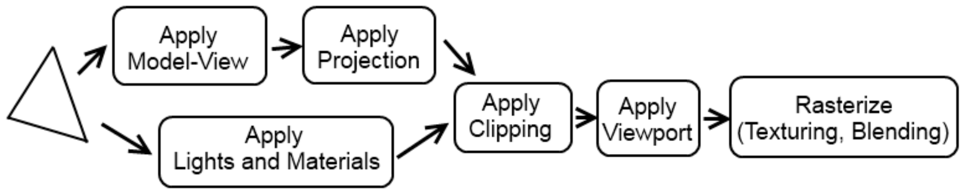

Figure 7–4. *The way of the triangle*

The way of the a triangle through this pipeline looks as follows:

1. Our brave triangle is first transformed by the model-view matrix. This means that all its points are multiplied with this matrix. This multiplication will effectively move the triangle's points around in the world.
2. The output of this is then multiplied by the projection matrix, effectively transforming the 3D points onto the 2D projection plane.
3. In between these two steps (or parallel to them), the currently set lights and materials are also applied to our triangle, giving it its color.
4. Once all that is done, the projected triangle is clipped to our "retina" and transformed to framebuffer coordinates.
5. As a final step, OpenGL fills in the pixels of the triangle based on the colors from the lighting stage, textures to be applied to the triangle, and the blending state, in which each pixel of the triangle might or might not be combined with the pixel in the framebuffer.

All we need to learn is how to throw geometry and textures at OpenGL ES, and set the states used by each of the preceding steps. Before we can do that, though, we need to check out how Android grants us access to OpenGL ES.

NOTE: While the high-level description of the OpenGL ES pipeline is mostly correct, it is heavily simplified and leaves out some details that will become important in a later chapter. Another thing to note is that when OpenGL ES performs projections, it doesn't actually project onto a 2D coordinate system. Instead it projects into something called a *homogenous coordinate system*, which is actually four dimensional. This is a very involved mathematical topic, so for the sake of simplicity, we'll just stick to the simplified belief that OpenGL ES projects to 2D coordinates.

Before We Begin

In the rest of this chapter, we'll write a lot of small examples, as we did in Chapter 4 when discussing the Android API basics. We'll use the same starter class as we did in Chapter 4, which shows us a list of test `Activity`s we can start. The only things that will change are the names of the `Activity`s we instantiate via reflection, and the package they are located in. All the examples of the rest of this chapter will be in the package `com.badlogic.androidgames.glbasics`. The rest of the code will stay the same. Our new starter `Activity` will be called `GLBasicsStarter`. We will also copy over all the source code from Chapter 5, which contains our framework classes, as we of course want to reuse those. Finally, we will write some new framework and helper classes, which will go in the `com.badlogic.androidgames.framework` package and subpackages.

We also have a manifest file again. As each of the following little examples will be an `Activity`, we also have to make sure it has an entry in the manifest. All the examples will use a fixed orientation (either portrait or landscape, depending on the example), and will tell Android that they can handle `keyboard`, `keyboardHidden`, and `orientationChange` events.

With that out of our way, let the fun begin!

GLSurfaceView: Making Things Easy Since 2008

The first thing we need is some type of `View` that will allow us to draw via OpenGL ES. Luckily there's such a `View` in the Android API. It's called `GLSurfaceView`, and it's a descendent of the `SurfaceView` class, which we already used for drawing the world of Mr. Nom.

We also need a separate main loop thread again so that we don't bog down the UI thread. Surprise: `GLSurfaceView` already sets up such a thread for us! All we need to do is implement a listener interface called `GLSurfaceView.Renderer` and register it with the `GLSurfaceView`. The interface has three methods:

```
interface Renderer {
    public void onSurfaceCreated(GL10 gl, EGLConfig config);

    public void onSurfaceChanged(GL10 gl, int width, int height);

    public void onDrawFrame(GL10 gl);
}
```

The `onSurfaceCreated()` method is called each time the `GLSurfaceView` surface is created. This happens the first time we fire up the `Activity` and each time we come back to the `Activity` from a paused state. The method takes two parameters, a `GL10` instance and an `EGLConfig`. The `GL10` instance allows us to issue commands to OpenGL ES. The `EGLConfig` just tells us about the attributes of the surface, such as the color depth and so on. We usually ignore it. We will set up our geometries and textures in the `onSurfaceCreated()` method.

The `onSurfaceChanged()` method is called each time the surface is resized. We get the new width and height of the surface in pixels as parameters, along with a `GL10` instance if we want to issue OpenGL ES commands.

The `onDrawFrame()` method is where the fun happens. It is similar in spirit to our `Screen.render()` method, which gets called as often as possible by the rendering thread that the `GLSurfaceView` sets up for us. In this method we perform all our rendering.

Besides registering a `Renderer` listener, we also have to call `GLSurfaceView.onPause()/onResume()` in our `Activity`'s `onPause()/onResume()` methods. The reason for this is simple. The `GLSurfaceView` will start up the rendering thread in its `onResume()` method and tear it down in its `onPause()` method. This means that our listener will not be called while our `Activity` is paused, since the rendering thread which calls our listener will also be paused.

And here comes the only bummer: each time our Activity is paused, the surface of the GLSurfaceView will be destroyed. When the Activity is resumed again (and GLSurfaceView.onResume() is called by us), the GLSurfaceView instantiates a new OpenGL ES rendering surface for us, and informs us of this by calling our listener's onSurfaceCreated() method. This would all be well if not for a single problem: all the OpenGL ES states we set so far will be lost. This also includes things such as textures and so on, which we'll have to reload in that case. This problem is known as a *context loss*. The word *context* stems from the fact that OpenGL ES associates a so-called context with each surface we create, which holds the current states. When we destroy that surface, the context is lost as well. It's not all that bad, though, given that we design our games properly to handle this context loss.

NOTE: Actually, it's EGL that is responsible for the context and surface creation and destruction. EGL is another Khronos Group standard; it defines how an operating system's UI works together with OpenGL ES, and how the operating system grants OpenGL ES access to the underlying graphics hardware. This includes surface creation as well as context management. Since GLSurfaceView handles all the EGL stuff for us, we can safely ignore it in almost all cases.

Following tradition, let's write a small example that will clear the screen with a random color each frame. Listing 7–1 shows the code.

Listing 7–1. *GLSurfaceViewTest.java; Screen-Clearing Madness*

```
package com.badlogic.androidgames.glbasics;

import java.util.Random;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.opengl.GLSurfaceView.Renderer;
import android.os.Bundle;
import android.util.Log;
import android.view.Window;
import android.view.WindowManager;

public class GLSurfaceViewTest extends Activity {
    GLSurfaceView glView;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                WindowManager.LayoutParams.FLAG_FULLSCREEN);
        glView = new GLSurfaceView(this);
        glView.setRenderer(new SimpleRenderer());
        setContentView(glView);
    }
```

We keep a reference to a GLSurfaceView instance as a member of the class. In the onCreate() method, we make our application go full-screen, create the GLSurfaceView, set our Renderer implementation, and make the GLSurfaceView the content view of our Activity.

```
    @Override
    public void onResume() {
        super.onPause();
        glView.onResume();
    }

    @Override
    public void onPause() {
        super.onPause();
        glView.onPause();
    }
```

In the onResume() and onPause() methods, we call the supermethods as well as the respective GLSurfaceView methods. These will start up and tear down the rendering thread of the GLSurfaceView, which in turn will trigger the callback methods of our Renderer implementation at appropriate times.

```
    static class SimpleRenderer implements Renderer {
        Random rand = new Random();

        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            Log.d("GLSurfaceViewTest", "surface created");
        }

        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
            Log.d("GLSurfaceViewTest", "surface changed: " + width + "x"
                    + height);
        }

        @Override
        public void onDrawFrame(GL10 gl) {
            gl.glClearColor(rand.nextFloat(), rand.nextFloat(),
                    rand.nextFloat(), 1);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        }
    }
}
```

The final piece of the code is our Renderer implementation. It's just logs some information in the onSurfaceCreated() and onSurfaceChanged() methods. The really interesting part is the onDrawFrame() method.

As said earlier, the GL10 instance gives us access to the OpenGL ES API. The 10 in GL10 indicates that it offers us all the functions defined in the OpenGL ES 1.0 standard. For now we can be happy with that. All the methods of that class map to a corresponding C function, as defined in the standard. Each method begins with the prefix gl, an old tradition of OpenGL ES.

The first OpenGL ES method we call is glClearColor(). You probably already know what that will do. It sets the color to be used when we issue a command to clear the screen. Colors in OpenGL ES are almost always RGBA colors where each component has a range between 0 and 1. There are ways to define a color in, say, RGB565, but for now let's stick to the floating-point representation. We could set the color used for clearing only once and OpenGL ES would remember it. The color we set with glClearColor() is one of OpenGL ES's states.

The next call actually clears the screen with the clear color we just specified. The method glClear() takes a single argument that specifies which buffer to clear. OpenGL ES does not only have the notation of a framebuffer that holds pixels, but also other types of buffers. We'll get to know them in Chapter 10, but for now all we care about is the framebuffer that holds our pixels. OpenGL ES calls that the *color buffer*. To tell OpenGL ES that we want to clear that exact buffer, we specify the constant GL10.GL_COLOR_BUFFER_BIT.

OpenGL ES has a lot of constants, which are all defined as static public members of the GL10 interface. Like the methods, each constant has the prefix GL_.

And that was our first OpenGL ES application. I'll spare you the impressive screenshot, since you probably know what it looks like.

NOTE: Thou shall never call OpenGL ES from another thread! First and last commandment! The reason is that OpenGL ES is designed to be used in single threaded environments only and is not thread-safe. It can be made to somewhat work on multiple threads, but many drivers have problems with this and there's no real benefit to doing so.

GLGame: Implementing the Game Interface

In the previous chapter, we implemented the AndroidGame class, which ties together all the submodules for audio, file I/O, graphics, and user input handling. We want to reuse most of this for our upcoming 2D OpenGL ES game, so let's implement a new class called GLGame that implements the Game interface we defined earlier.

The first thing you will notice is that we can't possibly implement the Graphics interface with our current knowledge of OpenGL ES. Here's a surprise: we won't implement it. OpenGL does not lend itself well to the programming model of our Graphics interface. Instead we'll implement a new class, GLGraphics, which will keep track of the GL10 instance we get from the GLSurfaceView. Listing 7–2 shows the code.

Listing 7–2. *GLGraphics.java; Keeping Track of the GLSurfaceView and the GL10 Instance*

```
package com.badlogic.androidgames.framework.impl;

import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView;
```

```
public class GLGraphics {
    GLSurfaceView glView;
    GL10 gl;

    GLGraphics(GLSurfaceView glView) {
        this.glView = glView;
    }

    public GL10 getGL() {
        return gl;
    }

    void setGL(GL10 gl) {
        this.gl = gl;
    }

    public int getWidth() {
        return glView.getWidth();
    }

    public int getHeight() {
        return glView.getHeight();
    }
}
```

This class has just a few getters and setters. Note that we will use this class in the rendering thread set up by the `GLSurfaceView`. As such, it might be problematic to call methods of a `View`, which lives mostly on the UI thread. In this case it's OK, though, as we only query for the `GLSurfaceView`'s width and height, so we get away with it.

The `GLGame` class is a bit more involved. It borrows most of its code from the `AndroidGame` class. The only thing that is a little bit more complex is the synchronization between the rendering and UI threads. Let's have a look at it in Listing 7–3.

Listing 7–3. *GLGame.java, the Mighty OpenGL ES Game Implementation*

```
package com.badlogic.androidgames.framework.impl;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.app.Activity;
import android.content.Context;
import android.opengl.GLSurfaceView;
import android.opengl.GLSurfaceView.Renderer;
import android.os.Bundle;
import android.os.PowerManager;
import android.os.PowerManager.WakeLock;
import android.view.Window;
import android.view.WindowManager;

import com.badlogic.androidgames.framework.Audio;
import com.badlogic.androidgames.framework.FileIO;
import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Input;
import com.badlogic.androidgames.framework.Screen;

public abstract class GLGame extends Activity implements Game, Renderer {
    enum GLGameState {
        Initialized,
        Running,
        Paused,
        Finished,
        Idle
    }

    GLSurfaceView glView;
    GLGraphics glGraphics;
    Audio audio;
    Input input;
    FileIO fileIO;
    Screen screen;
    GLGameState state = GLGameState.Initialized;
    Object stateChanged = new Object();
    long startTime = System.nanoTime();
    WakeLock wakeLock;
```

The class extends the `Activity` class and implements the `Game` and `GLSurfaceView.Renderer` interface. It has an enum called `GLGameState` that keeps track of the state the `GLGame` instance is currently in. We'll see how those are used in a bit.

The members of the class consist of a `GLSurfaceView` and `GLGraphics` instance. The class also has `Audio`, `Input`, `FileIO`, and `Screen` instances, which we need for writing our game, just as in the `AndroidGame` class. The `state` member keeps track of the state via one of the `GLGameState` enums. The `stateChanged` member is an object we'll use to synchronize the UI thread and the rendering thread. Finally we have a member to keep track of the delta time and a `WakeLock` we'll use to keep the screen from dimming.

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                             WindowManager.LayoutParams.FLAG_FULLSCREEN);
        glView = new GLSurfaceView(this);
        glView.setRenderer(this);
        setContentView(glView);

        glGraphics = new GLGraphics(glView);
        fileIO = new AndroidFileIO(getAssets());
        audio = new AndroidAudio(this);
        input = new AndroidInput(this, glView, 1, 1);
        PowerManager powerManager = (PowerManager)
getSystemService(Context.POWER_SERVICE);
        wakeLock = powerManager.newWakeLock(PowerManager.FULL_WAKE_LOCK, "GLGame");
    }
```

In the `onCreate()` we perform the usual setup routine. We make the `Activity` go full-screen and instantiate the `GLSurfaceView`, setting it as the content `View`. We also instantiate all the other classes that implement framework interfaces, such as the AndroidFileIO or AndroidInput classes. Note that we reuse the classes we used in the AndroidGame class, except for AndroidGraphics. Another important point is that we no longer let the AndroidInput class scale the touch coordinates to a target resolution, as in AndroidGame. The scale values are both 1, so we will get the real touch coordinates. It will become clear later on why we do that. The last thing we do is create the WakeLock instance.

```
    public void onResume() {
        super.onResume();
        glView.onResume();
        wakeLock.acquire();
    }
```

In the onResume() method we let the GLSurfaceView start the rendering thread with a call to its onResume() method. We also acquire the WakeLock.

```
    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        glGraphics.setGL(gl);

        synchronized(stateChanged) {
            if(state == GLGameState.Initialized)
                screen = getStartScreen();
            state = GLGameState.Running;
            screen.resume();
            startTime = System.nanoTime();
        }
    }
```

The next thing that will be called is the onSurfaceCreate() method. The method is invoked on the rendering thread, of course. Here we can see how the state enums are used. If the application is started for the first time, the state will be GLGameState.Initialized. In this case we call the getStartScreen() method to return the starting screen of the game. If the game is not in an initialized state, but was already been running, we know that we have just resumed from a paused state. In any case we set the state to GLGameState.Running and call the current Screen's resume() method. We also keep track of the current time so we can calculate the delta time later on.

The synchronization is necessary, since the members we manipulate within the synchronized block could be manipulated in the onPause() method on the UI thread. That's something we have to prevent, so we use an object as a lock. We could have also used the GLGame instance itself here, or a proper lock.

```
    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {
    }
```

The onSurfaceChanged() method is basically just a stub. There's nothing for us to do here.

```
    @Override
    public void onDrawFrame(GL10 gl) {
        GLGameState state = null;
        synchronized(stateChanged) {
            state = this.state;
        }

        if(state == GLGameState.Running) {
            float deltaTime = (System.nanoTime()-startTime) / 1000000000.0f;
            startTime = System.nanoTime();

            screen.update(deltaTime);
            screen.present(deltaTime);
        }

        if(state == GLGameState.Paused) {
            screen.pause();
            synchronized(stateChanged) {
                this.state = GLGameState.Idle;
                stateChanged.notifyAll();
            }
        }

        if(state == GLGameState.Finished) {
            screen.pause();
            screen.dispose();
            synchronized(stateChanged) {
                this.state = GLGameState.Idle;
                stateChanged.notifyAll();
            }
        }
    }
```

The onDrawFrame() method is were the bulk of all the work is performed. It is called by the rendering thread as often as possible. Here we check which state our game is currently in and react accordingly. As the state can be set on the onPause() method on the UI thread, we have to synchronize the access to it.

If the game is running we calculate the delta time and tell the current Screen to update and present itself.

If the game is paused we tell the current Screen to pause itself as well. We then change the state to GLGameState.Idle, indicating that we have received the pause request from the UI thread. Since we wait for this to happen in the onPause() method in the UI thread, we notify the UI thread that it can now really pause the application. This notification is necessary, as we have to make sure that the rendering thread is paused/shut down properly in case our Activity is paused or closed on the UI thread.

If the Activity is being closed (and not paused), we react to GLGameState.Finished. In this case we tell the current Screen to pause and dispose of itself, and then send another notification to the UI thread, which waits for the rendering thread to properly shut things down.

```
    @Override
    public void onPause() {
        synchronized(stateChanged) {
            if(isFinishing())
```

```
                state = GLGameState.Finished;
            else
                state = GLGameState.Paused;
            while(true) {
                try {
                    stateChanged.wait();
                    break;
                } catch(InterruptedException e) {
                }
            }
        }
        wakeLock.release();
        glView.onPause();
        super.onPause();
    }
```

The onPause() method is our usual Activity notificaton method that's called on the UI thread when the Activity is paused. Depending on whether the application is closed or paused, we set the state accordingly and wait for the rendering thread to process the new state. This is achieved with the standard Java wait/notify mechanism.

Finally we release the WakeLock and tell the GLSurfaceView and the Activity to pause themselves, effectivley shutting down the rendering thread and destroying the OpenGL ES surface, which triggers the dreaded OpenGL ES context loss mentioned earlier.

```
    public GLGraphics getGLGraphics() {
        return glGraphics;
    }
```

The getGLGraphics() method is a new method that is only accessible via the GLGame class. It returns the instance of GLGraphics we store so that we can get access to the GL10 interface in our Screen implementations later on.

```
    @Override
    public Input getInput() {
        return input;
    }

    @Override
    public FileIO getFileIO() {
        return fileIO;
    }

    @Override
    public Graphics getGraphics() {
        throw new IllegalStateException("We are using OpenGL!");
    }

    @Override
    public Audio getAudio() {
        return audio;
    }

    @Override
    public void setScreen(Screen screen) {
        if (screen == null)
```

```
            throw new IllegalArgumentException("Screen must not be null");

        this.screen.pause();
        this.screen.dispose();
        screen.resume();
        screen.update(0);
        this.screen = screen;
    }

    @Override
    public Screen getCurrentScreen() {
        return screen;
    }
}
```

The rest of the class works as before. In case we accidentially try to access the standard `Graphics` instance, we throw an exception, though, as it is not supported by `GLGame`. Instead we'll work with the `GLGraphics` method we get via the `GLGame.getGLGraphics()` method.

Why did we go through all the pain of synchronizing with the rendering thread? Well, it will make our `Screen` implementations live entirely on the rendering thread. All the methods of `Screen` will be executed there, which is necessary if we want to access OpenGL ES functionaility. Remeber, we can only access OpenGL ES on the rendering thread.

Let's round this out with an example. Listing 7–4 shows how our first example in this chapter looks when using `GLGame` and `Screen`.

Listing 7–4. *GLGameTest.java; More Screen Clearing, Now with 100 Percent More GLGame*

```
package com.badlogic.androidgames.glbasics;

import java.util.Random;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class GLGameTest extends GLGame {
    @Override
    public Screen getStartScreen() {
        return new TestScreen(this);
    }

    class TestScreen extends Screen {
        GLGraphics glGraphics;
        Random rand = new Random();

        public TestScreen(Game game) {
            super(game);
            glGraphics = ((GLGame) game).getGLGraphics();
        }

        @Override
        public void present(float deltaTime) {
            GL10 gl = glGraphics.getGL();
            gl.glClearColor(rand.nextFloat(), rand.nextFloat(),
                    rand.nextFloat(), 1);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        }

        @Override
        public void update(float deltaTime) {
        }

        @Override
        public void pause() {
        }

        @Override
        public void resume() {
        }

        @Override
        public void dispose() {
        }
    }
}
```

This is the same program as in our last example, except that we now derive from `GLGame` instead of `Activity`, and we provide a `Screen` implementation instead of a `GLSurfaceView.Renderer` implementation.

In the following examples, we'll only have a look at the relevant parts of each example's `Screen` implementation. The overall structure of our examples will stay the same. Of course, we have to add the example `GLGame` implementations to our starter `Activity`, as well as to the manifest file.

With that out of our way, let's render our first triangle.

Look Mom, I Got a Red Triangle!

You already learned that OpenGL ES needs a couple of things set before we can tell it to draw some geometry. The two things we are most concerned about are the projection matrix (and with it our view frustum) and the viewport, which governs the size of our output image and the position of our rendering output in the framebuffer.

Defining the Viewport

OpenGL ES uses the viewport as a way to translate the coordinates of points projected to the near clipping plane to framebuffer pixel coordinates. We can tell OpenGL ES to use only a portion of our framebuffer, or all of it, with the following method:

```
GL10.glViewport(int x, int y, int width, int height)
```

The x- and y-coordinates specify the top-left corner of the viewport in the framebuffer, and `width` and `height` specify the viewport's size in pixels. Note that OpenGL ES assumes the framebuffer coordinate system to have its origin in the lower left of the screen. Usually we set x and y to zero and `width` and `height` to our screen resolution, as we are using full-screen mode. We could instruct OpenGL ES to only use a portion of the framebuffer with this method. It would then take the rendering output and automatically stretch it to that portion.

NOTE: While this method looks like it sets up a 2D coordinate system for us to render to, it actually does not. It only defines the portion of the framebuffer OpenGL ES uses to output the final image. Our coordinate system is defined via the projection and model-view matrices.

Defining the Projection Matrix

The next thing we need to define is the projection matrix. As we are only concerned with 2D graphics in this chapter, we want to use a parallel projection. How do we do that?

Matrix Modes and Active Matrices

We already discussed that OpenGL ES keeps track of three matrices: the projection matrix, the model-view matrix, and the texture matrix (which we'll continue to ignore). OpenGL ES offers us a couple of specific methods to modify these matrices. Before we can use these methods, however, we have to tell OpenGL ES which matrix we want to manipulate. This is done with the following method:

```
GL10.glMatrixMode(int mode)
```

The `mode` parameter can be `GL10.GL_PROJECTION`, `GL10.GL_MODELVIEW`, or `GL10.GL_TEXTURE`. It should be clear which of these constants will make which matrix active. Any subsequent calls to the matrix manipulation methods will target the matrix we set with this method until we change the active matrix again via another call to this method. This matrix mode is one of OpenGL ES's states (which will get lost when we lose the context if our application is paused and resumed). To manipulate the projection matrix with any subsequent calls, we can call the method like this:

```
gl.glMatrixMode(GL10.GL_PROJECTION);
```

Orthographic Projection with glOrthof

OpenGL ES offers us the following method for setting the active matrix to an orthographic (parallel) projection matrix:

```
GL10.glOrthof(int left, int right, int bottom, int top, int near, int far)
```

Hey, that looks a lot like it has something to do with our view frustum's clipping planes. And indeed it does. So what values do we specify here?

OpenGL ES has a standard coordinate system, as depicted in Figure 7–4. The positive x-axis points to the right, the positive y-axis points upward, and the positive z-axis points toward us. With `glOrthof()` we define the view frustum of our parallel projection in this coordinate system. If you look back at Figure 7–3, you can see that the view frustum of a parallel projection is a box. We can interpret the parameters for `glOrthof()` as specifying two of these corners of our view frustum box. Figure 7–5 illustrates this.

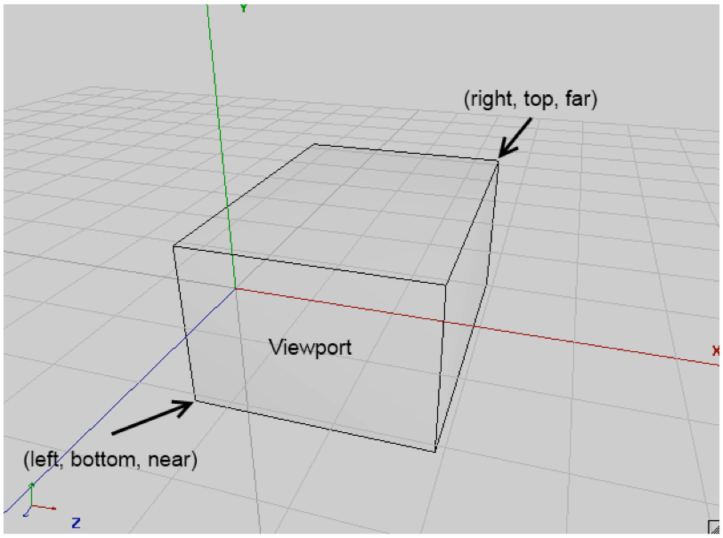

Figure 7–5. *An orthographic view frustum*

The front side of our view frustum will be directly mapped to our viewport. In the case of a full-screen viewport from, say, (0,0) to (480,320) (e.g., landscape mode on a Hero), the bottom-left corner of the front side would map to the bottom-left corner of our screen, and the top-right corner of the front side would map to the top-left corner of our screen. OpenGL will perform the stretching automatically for us.

Since we want to do 2D graphics, we will specify the corner points (left, bottom, near) and (right, top, far) (see figure 7–5) in a way that allows us to work in a sort of pixel coordinate system, as we did with the `Canvas` and Mr. Nom. Here's how we could set up such a coordinate system:

```
gl.glOrthof(0, 480, 0, 320, 1, -1);
```

Figure 7–6 shows the view frustum.

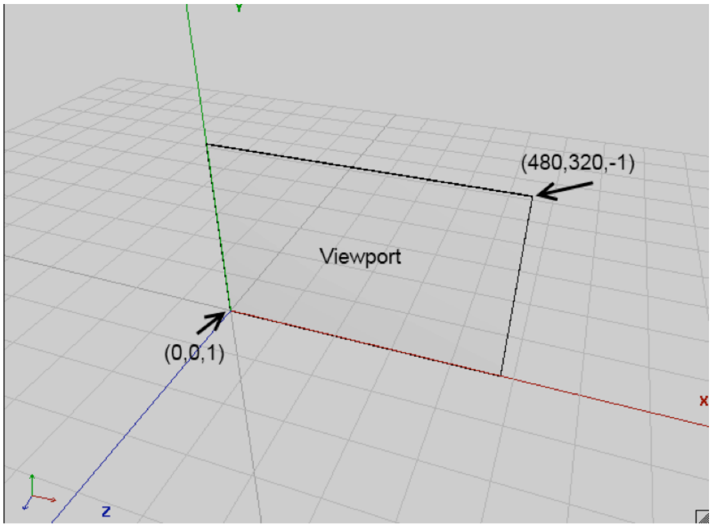

Figure 7–6. *Our parallel projection view frustum for 2D rendering with OpenGL ES*

Our view frustum is pretty thin, but that's OK because we'll only be working in 2D. The visible part of our coordinate system goes from (0,0,1) to (480,320,–1). Any points we specify within this box will be visible on the screen as well. The points will be projected onto the front side of this box, which is our beloved near clipping plane. The projection will then get stretched out onto the viewport, whatever dimensions it has. Say we have a Nexus One with a resolution of 800×480 pixels in landscape mode. When we specify our view frustum as just mentioned, we can work in a 480×320 coordinate system, and OpenGL will stretch it to the 800×480 framebuffer (if we specified that the viewport covers the complete framebuffer). Best of all, there's nothing keeping us from using crazier view frustums. We could also use one with the corners (–1,–1,100) and (2,2,–100). Everything we specify that falls inside this box will be visible and get stretched automatically. Pretty nifty.

Note that we also set the near and far clipping planes. Since we are going to neglect the z-coordinate completely in this chapter, you might be tempted to use zero for both near and far. However, that's a bad idea for various reasons. To play it safe, we grant the view frustum a little buffer in the z-axis. All our geometries' points will be defined in the x-y plane with z set to zero, though—2D all the way.

NOTE: You might have noticed that the y-axis is pointing upward now, and the origin is in the lower-left corner of our screen. While the `Canvas`, the UI framework, and many other 2D-rendering APIs use the y-down, origin-top-left convention, it is actually more convenient to use this "new" coordinate system for game programming. For example, if Super Mario is jumping, wouldn't you expect his y-coordinate to increase instead of decrease while he's on his way up? Want to work in the other coordinate system? Fine, just swap the bottom and top parameters of `glOrthof()`. Also, while the illustration of the view frustum is mostly correct from a geometric point of view, the near and far clipping planes are actually interpreted a little differently by `glOrthof()`. Since that is a little involved, we'll just pretend the preceding illustrations are correct, though.

A Helpful Snippet

Here's a small snippet that will be used in all of our examples in this chapter. It clears the screen with black, sets the viewport to span the whole framebuffer, and sets up the projection matrix (and thereby the view frustum) so we can work in a comfortable coordinate system with the origin in the lower-left corner of the screen and the y-axis pointing upward:

```
gl.glClearColor(0,0,0,1);
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrthof(0, 320, 0, 480, 1, -1);
```

Wait, what does `glLoadIdentity()` do in there? Well, most of the methods OpenGL ES offers us to manipulate the active matrix don't actually set the matrix. Instead they construct a temporary matrix from whatever parameters they take and multiply it with the current matrix. The `glOrthof()` method is no exception. For example, if we called `glOrthof()` each frame, we'd multiply the projection matrix to death with itself. So instead of doing that, we make sure we have a clean identity matrix in place before we multiply the projection matrix. Remember, multiplying a matrix by the identity matrix will output the matrix itself again. And that's what `glLoadIdentity()` is for. Think of it as first loading the value 1 and then multiplying it with whatever we have (in our case, the projection matrix produced by `glOrthof()`).

Note that our coordinate system now goes from (0,0,1) to (320,480,–1)—that's for portrait mode rendering.

Specifying Triangles

Next up we have to figure out how we can tell OpenGL ES about the triangles we want it to render. First let's define what a triangle is made of:

- A triangle is made of three points.
- Each point is called a vertex.
- A vertex has a position in 3D space.
- A position in 3D space is given as three floats, specifying the x-, y-, and z-coordinates.
- A vertex can have additional attributes, such as a color or texture coordinates (which we'll talk about later). These can be represented as floats as well.

OpenGL ES expects to send our triangle definitions in the form of arrays. However, given that OpenGL ES is actually a C API, we can't just use standard Java arrays. Instead we have to use Java NIO buffers, which are just memory blocks of consecutive bytes.

A Small NIO Buffer Digression

To be totally exact, we need to use *direct* NIO buffers. This means that the memory is not allocated in the virtual machine's heap memory, but in native heap memory. To construct such a direct NIO buffer, we can use the following code snippet:

```
ByteBuffer buffer = ByteBuffer.allocateDirect(NUMBER_OF_BYTES);
buffer.order(ByteOrder.nativeOrder());
```

This will allocate a `ByteBuffer` that can hold `NUMBER_OF_BYTES` bytes in total, and make sure that the byte order is equal to the byte order used by the underlying CPU. A NIO buffer has three attributes:

- `Capacity`: The number of elements the buffer can hold in total
- `Position`: The current position to which the next element would be written to or read from
- `Limit`: The index of the last element that has been defined plus one

The `capacity` of a buffer is actually its size. In the case of a `ByteBuffer`, it is given in bytes. The `position` and `limit` attributes can be thought of as defining a segment within the buffer starting at `position` and ending at `limit` (exclusive).

Since we want to specify our vertices as floats, it would be nice not to have to cope with bytes. Luckily we can convert the `ByteBuffer` instance to a `FloatBuffer` instance which allows us just that: working with floats.

```
FloatBuffer floatBuffer = buffer.asFloatBuffer();
```

Capacity, position, and limit are given in floats in the case of a `FloatBuffer`. Our usage pattern of these buffers will be pretty limited—it goes like this:

```
float[] vertices = { ... definitions of vertex positions etc  ...;
floatBuffer.clear();
floatBuffer.put(vertices);
floatBuffer.flip();
```

We first define our data in a standard Java float array. Before we put that float array into the buffer, we tell the buffer to clear itself via the `clear()` method. This doesn't actually erase any data, but sets the position to zero and the limit to the capacity. Next we use the `FloatBuffer.put(float[] array)` method to copy the content of the complete array to the buffer, beginning at the buffer's current position. After the copying, the position of the buffer will be increased by the length of the array. Next, the call to the `put()` method then appends the additional data to the data of the last array we copied to the buffer. The final call to `FloatBuffer.flip()` just swaps the position and limit.

For this example, let's assume that our vertices array is five floats in size and that our `FloatBuffer` has enough capacity to store those five floats. After the call to `FloatBuffer.put()`, the position of the buffer will be 5 (indices 0 to 4 are taken up by the five floats from our array). The limit will still be equal to whatever the capacity of the buffer is. After the call to `FloatBuffer.flip()`, the position will be set to 0 and the limit will be set to 5. Any party interested in reading the data from the buffer will then know that it should read the floats from index 0 to 4 (remember that the limit is exclusive). And that's exactly what OpenGL ES needs to know as well. Note, however, that it will happily ignore the limit. Usually we have to tell it the number of elements to read in addition to passing the buffer to it. There's no error checking done, so watch out.

Sometimes it is useful to set the position of the buffer manually after we have filled it. This can be done via a call to the following method:

```
FloatBuffer.position(int position)
```

This will come in handy later on, when we temporarily set the position of a filled buffer to something other than zero for OpenGL ES to start reading at a specific position.

Sending Vertices to OpenGL ES

So how do we define the positions of the three vertices of our first triangle? Easy—assuming our coordinate system is (0,0,1) to (320,480,–1), as we defined it in the preceding code snippet, we can do the following:

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * 2 * 4);
byteBuffer.order(ByteOrder.nativeOrder());
FloatBuffer vertices = byteBuffer.asFloatBuffer();
vertices.put(new float[] {    0.0f,   0.0f,
                            319.0f,   0.0f,
                            160.0f, 479.0f  });
vertices.flip();
```

The first three lines should be familiar already. The only interesting part is how many bytes we allocate. We have three vertices, each composed of a position given as x- and y-coordinates. Each coordinate is a float, and thus takes up 4 bytes. That's three vertices times two coordinates times four bytes, for a total of 24 bytes for our triangle.

NOTE: We can specify vertices with x- and y-coordinates only, and OpenGL ES will automatically set the z-coordinate to zero for us.

Next we put a float array holding our vertex positions into the buffer. Our triangle starts at the bottom-left corner (0,0), goes to the right edge of the view frustum/screen (319,0), and then goes to the middle of the top edge of the view frustum/screen. Being the good NIO buffer users we are, we also call the `flip()` method on our buffer. Thus, the position will be 0 and the limit will be 6 (remember, `FloatBuffer` limits and positions are given in floats, not bytes).

Once we have our NIO buffer ready, we can tell OpenGL ES to draw it with its current state (e.g., viewport and projection matrix). This can be done with the following snippet:

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glVertexPointer( 2, GL10.GL_FLOAT, 0, vertices);
gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
```

The call to `glEnableClientState()` is a bit of a relic. It tells OpenGL ES that the vertices we are going to draw have a position. This is a bit silly for two reasons:

- The constant is called `GL10.GL_VERTEX_ARRAY`, which is a bit confusing. It would make more sense if it were called `GL10.GL_POSITION_ARRAY`.
- There's no way to draw anything that has no position, so the call to this method is a little bit superfluous. We do it anyway, though, to make OpenGL ES happy.

In the call to `glVertexPointer()` we tell OpenGL ES where it can find the vertex positions, and give it some additional information. The first parameter tells OpenGL ES that each vertex position is composed of two coordinates, x and y. If we would have specified x, y, and z, we would have passed `3` to the method. The second parameter tells OpenGL ES the data type we used to store each coordinate. In this case it's `GL10.GL_FLOAT`, indicating that we used floats encoded as 4 bytes each. The third parameter, `stride`, tells OpenGL how far apart each of our vertex positions are from each other in bytes. In the preceding case, `stride` is zero, as the positions are tightly packed (vertex 1 (x,y), vertex 2(x,y), etc.). The final parameter is our `FloatBuffer`, for which there are two things to remember:

- The `FloatBuffer` represents a memory block in the native heap, and thus has a starting address.
- The position of the `FloatBuffer` is an offset from that starting address.

OpenGL ES will take the buffer's starting address and add the buffer's positions to arrive at the float in the buffer that it will start reading the vertices from when we tell it to draw the contents of the buffer. The vertex pointer (which again should be called the position pointer) is a state of OpenGL ES. As long as we don't change it (and the context isn't lost), OpenGL ES will remember and use it for all subsequent calls that need vertex positions.

Finally there's the call to `glDrawArrays()`. It will draw our triangle. The first parameter specifies what type of primitive we are going to draw. In this case we say that we want to render a list of triangles, which is specified via `GL10.GL_TRIANGLES`. The next parameter is an offset relative to the first vertex the vertex pointer points to. The offset is measured in vertices, not bytes or floats. If we'd have specified more than one triangle, we could use this offset to render only a subset of our triangle list. The final argument tells OpenGL ES how many vertices it should use for rendering. In our case that's three vertices. Note that we always have to specify a multiple of 3 if we draw `GL10.GL_TRIANGLES`. Each triangle is composed of three vertices, so that makes sense. For other primitive types the rules are a little different.

Once we issue the `glVertexPointer()` command, OpenGL ES will transfer the vertex positions to the GPU and store them there for all subsequent rendering commands. Each time we tell OpenGL ES to render vertices, it takes their positions from the data we last specified via `glVertexPointer()`.

Each of our vertices might have more attributes than just its position. One other attribute might be a vertex's color. We usually refer to those attributes as *vertex attributes*.

You might wonder how OpenGL ES knows what color our triangle should have, as we have only specified positions. It turns out that OpenGL ES has sensible defaults for any vertex attribute that we don't specify. Most of these defaults can be set directly. For example, if we want to set a default color for all vertices that we draw, we can use the following method:

```
GL10.glColor4f(float r, float g, float b, float a)
```

This method will set the default color to be used for all vertices for which we didn't specify a color. The color is given as RGBA values in the range 0.0 to 1.0, as was the case for the clear color earlier. The default color OpenGL ES starts with is (1,1,1,1)—that is, fully opaque white.

And that is all the code we need to render a triangle with a custom parallel projection with OpenGL ES. That's a mere 16 lines of code for clearing the screen, setting the viewport and projection matrix, creating an NIO buffer that we store our vertex positions in, and drawing the triangle. Now compare that to the six pages it took me to explain this to you. I could have of course left out the details and used coarser language. The problem is that OpenGL ES is a pretty complex beast at times, and to avoid getting an empty screen, it's best to learn what it is all about rather than just copying and pasting code.

Putting It Together

To round this section out, let's put all this together via a nice `GLGame` and `Screen` implementation. Listing 7–5 shows the complete example.

Listing 7–5. *FirstTriangleTest.java*

```
package com.badlogic.androidgames.glbasics;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class FirstTriangleTest extends GLGame {
    @Override
    public Screen getStartScreen() {
        return new FirstTriangleScreen(this);
    }
```

The `FirstTriangleTest` class derives from `GLGame`, and thus has to implement the `Game.getStartScreen()` method. In that method we create a new `FirstTriangleScreen`, which will then be frequently called to update and present itself by the `GLGame`. Note that when this method is called, we are already in the main loop—or rather the `GLSurfaceView` rendering thread—so we can use OpenGL ES methods in the constructor of the `FirstTriangleScreen` class. Let's have a closer look at that `Screen` implementation:

```
    class FirstTriangleScreen extends Screen {
        GLGraphics glGraphics;
        FloatBuffer vertices;

        public FirstTriangleScreen(Game game) {
            super(game);
            glGraphics = ((GLGame)game).getGLGraphics();

            ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * 2 * 4);
            byteBuffer.order(ByteOrder.nativeOrder());
            vertices = byteBuffer.asFloatBuffer();
            vertices.put( new float[] {    0.0f,   0.0f,
                                         319.0f,   0.0f,
                                         160.0f, 479.0f});
            vertices.flip();
        }
```

The `FirstTriangleScreen` class holds two members: a `GLGraphics` instance and our trusty `FloatBuffer`, which stores the 2D positions of the three vertices of our triangle. In the constructor we fetch the `GLGraphics` instance from the `GLGame` and create and fill the `FloatBuffer` according to our previous code snippet. Since the `Screen` constructor gets a `Game` instance, we have to cast it to a `GLGame` instance so we can use the `GLGame.getGLGraphics()` method.

```
        @Override
        public void present(float deltaTime) {
            GL10 gl = glGraphics.getGL();
            gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            gl.glOrthof(0, 320, 0, 480, 1, -1);

            gl.glColor4f(1, 0, 0, 1);
            gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
            gl.glVertexPointer( 2, GL10.GL_FLOAT, 0, vertices);
            gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
        }
```

The present() method then reflects what we just discussed: we set the viewport, clear the screen, set the projection matrix so that we can work in our custom coordinate system, set the default vertex color (red in this case), specify that our vertices will have positions, tell OpenGL ES where it can find those vertex positions, and finally render our awesome little red triangle.

```
        @Override
        public void update(float deltaTime) {
            game.getInput().getTouchEvents();
            game.getInput().getKeyEvents();
        }

        @Override
        public void pause() {

        }

        @Override
        public void resume() {

        }

        @Override
        public void dispose() {

        }
    }
}
```

The rest of the class is just boilerplate code. In the update() method we make sure that our event buffers don't get filled up. The rest of the code does nothing.

NOTE: From here on we'll only focus on the Screen classes themselves, as the enclosing GLGame derivatives, such as FirstTriangleTest, will always be the same. We'll also reduce the code size a little by leaving out any empty or boilerplate methods of the Screen class. The following examples will all just differ in terms of members, constructors, and present methods.

Figure 7–7 shows the output of the preceding example.

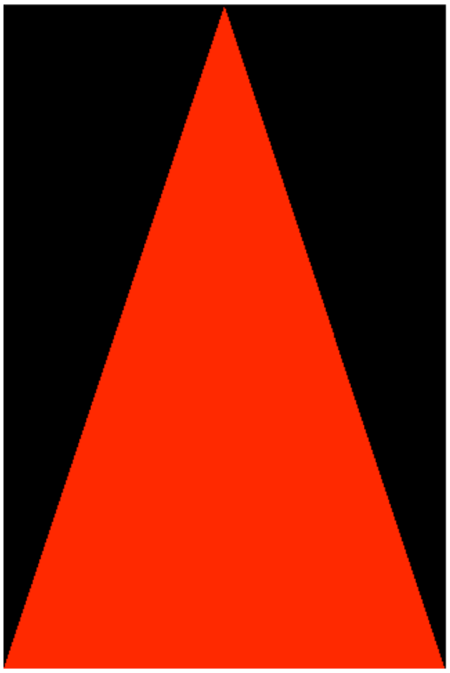

Figure 7–7. *Our first sexy triangle*

So here's what we did wrong in this example in terms of OpenGL ES best practices:

- We set the same states to the same values over and over again without any need. State changes in OpenGL ES are expensive—some a little bit more, some a little bit less. We should always try to reduce the number of state changes we make in a single frame.
- The viewport and projection matrix will never change once we set them. We could move that code to the `resume()` method, which is only called once each time the OpenGL ES surface gets (re)-created, thus also handling OpenGL ES context loss.
- We could also move setting the color used for clearing and setting the default vertex color to the resume() method. These two colors won't change either.
- We could move the `glEnableClientState()` and `glVertexPointer()` methods to the `resume()` method.
- The only things that we need to call each frame are `glClear()` and `glDrawArrays()`. Both use the current OpenGL ES states, which will stay the same as long as we don't change them and as long as we don't lose the context due to the `Activity` being paused and resumed.

If we had put these optimizations into practice, we would have only two OpenGL ES calls in our main loop. For the sake of clarity, we'll refrain from using these kind of minimal state change optimizations for now. When we start writing our first OpenGL ES game, though, we'll have to follow those practices as best as we can to guarantee good performance.

Let's add some more attributes to our triangle's vertices, starting with color.

NOTE: Very, very alert readers might have noticed that the triangle in Figure 7–7 is actually missing a pixel in the bottom-right corner. This may look like a typical off-by-one error, but it's actually due to the way OpenGL ES rasterizes (draws the pixels of) the triangle. There's a specific triangle rasterization rule that is responsible for that artifact. Worry not—we are mostly concerned with rendering 2D rectangles (composed of two triangles), where this effect will vanish.

Specifying Per Vertex Color

In the last example we set a global default color for all vertices we draw via `glColor4f()`. Sometimes we want to have more granular control (e.g., we want to set a color per vertex). OpenGL ES offers us this functionality, and it's really easy to use. All we have to do is add RGBA float components to each vertex and tell OpenGL ES where it can find the color for each vertex, similar to how we told it where it can find the position for each vertex. Let's start by adding the colors to each vertex:

```
int VERTEX_SIZE = (2 + 4) * 4;
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
byteBuffer.order(ByteOrder.nativeOrder());
FloatBuffer vertices = byteBuffer.asFloatBuffer();
vertices.put( new float[] {   0.0f,   0.0f, 1, 0, 0, 1,
                            319.0f,   0.0f, 0, 1, 0, 1,
                            160.0f, 479.0f, 0, 0, 1, 1});
vertices.flip();
```

We first have to allocate a `ByteBuffer` for our three vertices. How big should that `ByteBuffer` be? We have two coordinates and four (RGBA) color components per vertex, so that's six floats in total. Each float value takes up 4 bytes, so a single vertex uses 24 bytes. We store this information in `VERTEX_SIZE`. When we call `ByteBuffer.allocateDirect()`, we just multiply `VERTEX_SIZE` by the number of vertices we want to store in the `ByteBuffer`. The rest is pretty self-explanatory. We get a `FloatBuffer` view to our `ByteBuffer` and `put()` the vertices into the `ByteBuffer`. Each row of the float array holds the x- and y-coordinates, and the R, G, B, and A components of a vertex, in that order.

If we want to render this, we have to tell OpenGL ES that our vertices not only have a position, but also a color attribute. We start off, as before, by calling `glEnableClientState()`:

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
```

Now that OpenGL ES knows that it can expect position and color information for each vertex, we have to tell it where it can find that information:

```
vertices.position(0);
gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
vertices.position(2);
gl.glColorPointer(4, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
```

We start of by setting the position of our `FloatBuffer`, which holds our vertices to 0. The position thus points to the x-coordinate of our first vertex in the buffer. Next we call `glVertexPointer()`. The only difference from the previous example is that we now also specify the vertex size (remember, it's given in bytes). OpenGL ES will then start reading in vertex positions from the position in the buffer we told it to start from. For the second vertex position it will add `VERTEX_SIZE` bytes to the first position's address, and so on.

Next we set the position of the buffer to the R component of the first vertex and call `glColorPointer()`, which tells OpenGL ES where it can find the colors of our vertices. The first argument is the number of components per color. This is always four, as OpenGL ES demands an R, G, B, and A component per vertex from us. The second parameter specifies the type of each component. As with the vertex coordinates, we use `GL10.GL_FLOAT` again to indicate that each color component is a float in the range between 0 and 1. The third parameter is the stride between vertex colors. It's of course the same as the stride between vertex positions. The final parameter is our vertices buffer again.

Since we called `vertices.position(2)` before the `glColorPointer()` call, OpenGL ES knows that the first vertex color can be found starting from the third float in the buffer. If we wouldn't have set the position of the buffer to 2, OpenGL ES would have started reading in the colors from position 0. That would have been bad, as that's where the x-coordinate of our first vertex is. Figure 7–8 shows where OpenGL ES will read our vertex attributes from, and how it jumps from one vertex to the next for each attribute.

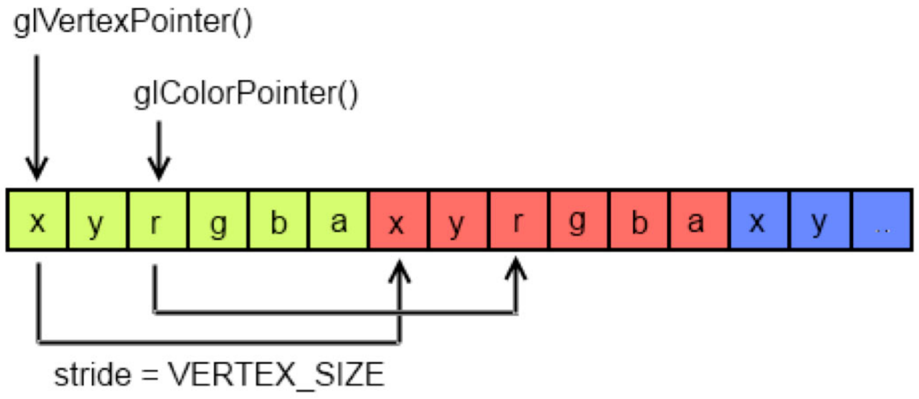

Figure 7–8. *Our vertices FloatBuffer, start addresses for OpenGL ES to read position/color from, and stride to be used to jump to the next position/color.*

To draw our triangle, we again call `glDrawElements()`, which tells OpenGL ES to draw a triangle using the first three vertices of our `FloatBuffer`:

```
gl.glDrawElements(GL10.GL_TRIANGLES, 0, 3);
```

Since we enabled the `GL10.GL_VERTEX_ARRAY` and `GL10.GL_COLOR_ARRAY`, OpenGL ES knows that it should use the attributes specified by `glVertexPointer()` and `glColorPointer()`. It will ignore the default color, as we provide our own per-vertex colors.

> **NOTE:** The way we just specified our vertices' positions and colors is called *interleaving*. This means that we pack the attributes of a vertex in one continuous memory block. There's another way we could have achieved this: *noninterleaved vertex arrays*. We'd have used two `FloatBuffers`, one for the positions and one for the colors. However, interleaving performs a lot better due to memory locality, so we won't discuss noninterleaved vertex arrays here.

Putting it all together into a new `GLGame` and `Screen` implementation should be a breeze by now. Listing 7–6 shows an excerpt from the file `ColoredTriangleTest.java`. I left out the boilerplate code.

Listing 7–6. *Excerpt from ColoredTriangleTest.java; Interleaving Position and Color Attributes*

```
class ColoredTriangleScreen extends Screen {
    final int VERTEX_SIZE = (2 + 4) * 4;
    GLGraphics glGraphics;
    FloatBuffer vertices;

    public ColoredTriangleScreen(Game game) {
        super(game);
        glGraphics = ((GLGame) game).getGLGraphics();

        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertices = byteBuffer.asFloatBuffer();
        vertices.put( new float[] {   0.0f,   0.0f, 1, 0, 0, 1,
                                    319.0f,   0.0f, 0, 1, 0, 1,
                                    160.0f, 479.0f, 0, 0, 1, 1});
        vertices.flip();
    }

    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

        vertices.position(0);
        gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        vertices.position(2);
        gl.glColorPointer(4, GL10.GL_FLOAT, VERTEX_SIZE, vertices);

        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
    }
```

Cool, that still looks pretty straightforward. All we changed compared to the previous example is adding the four color components to each vertex in our `FloatBuffer` and enabling the `GL10.GL_COLOR_ARRAY`. The best thing about it is that any additional vertex attributes we add in the subsequent examples will work the same way. We just tell OpenGL ES to not use the default value for that specific attribute but instead look up the attributes in our `FloatBuffer`, starting at a specific position and moving from vertex to vertex by `VERTEX_SIZE` bytes.

Now, we could also turn off the `GL10.GL_COLOR_ARRAY` so that OpenGL ES uses the default vertex color again, which we can specify via `glColor4f()` as we did previously. For this we can call

```
gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
```

OpenGL ES will just turn off the feature to read the colors from our `FloatBuffer`. If we already set a color pointer via `glColorPointer()`, OpenGL ES will remember the pointer, though. We just told it to not use it.

To round this example out, let's have a look at the output of the preceding program. Figure 7–9 shows a screenshot.

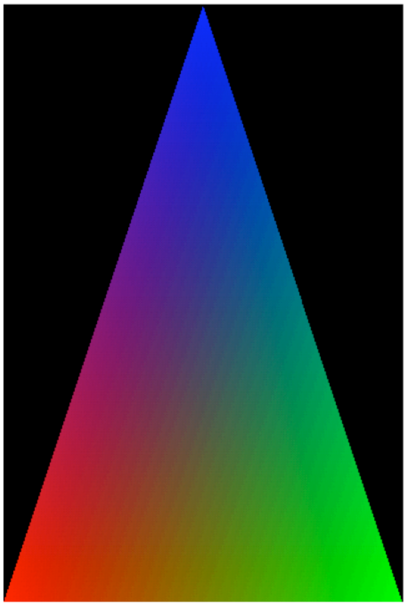

Figure 7–9. *Per-vertex colored triangle*

Woah, this is pretty neat. We didn't make any assumptions about how OpenGL ES will use the three colors we specified (red for the bottom-left vertex, green for the bottom-right vertex, and blue for the top vertex). It turns out that it will interpolate the colors between the vertices for us. With this we can easily create nice gradients. However, colors alone will not make us happy for very long. We want to draw images with OpenGL ES. And that's where so-called texture mapping comes into play.

Texture Mapping: Wallpapering Made Easy

When we wrote Mr. Nom we loaded some bitmaps and directly drew them to the framebuffer—no rotation involved, just a little bit of scaling, which is pretty easy to achieve. In OpenGL ES we are mostly concerned with triangles, which can have any orientation or scale we want them to have. So how can we render bitmaps with OpenGL ES?

Easy, just load up the bitmap to OpenGL ES (and for that matter to the GPU, which has its own dedicated RAM), add a new attribute to each of our triangle's vertices, and tell OpenGL ES to render our triangle and apply the bitmap (also known as *texture* in OpenGL ES speak) to the triangle. Let's first look at what these new vertex attributes actually specify.

Texture Coordinates

To map a bitmap to a triangle we need to add so-called *texture coordinates* to each vertex of the triangle. What is a texture coordinate? It specifies a point within the texture (our uploaded bitmap) to be mapped to one of the triangle's vertices. Texture coordinates are usually 2D.

While we call our positional coordinates x, y, and z, texture coordinates are usually called u and v or s and t, depending on the circle of graphics programmers you are a part of. OpenGL ES calls them s and t, so that's what we'll stick to. If you read resources on the Web that use the u/v nomenclature, don't get confused: it's the same as s and t. So what does the coordinate system look like? Figure 7–10 shows Bob in the texture coordinate system after we uploaded him to OpenGL ES.

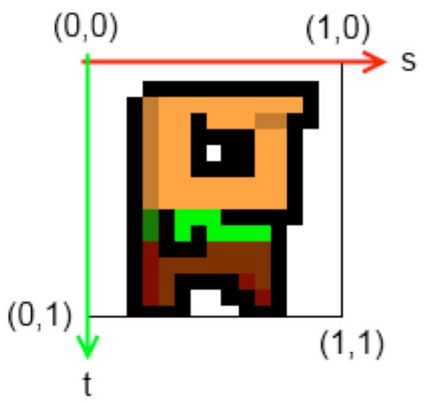

Figure 7–10. *Bob, uploaded to OpenGL ES, shown in the texture coordinate system*

There are a couple of interesting things going on here. First of all, s equals the x-coordinate in a standard coordinate system, and t is equal to the y-coordinate. The s-axis points to the right, and the t-axis points downward. The origin of the coordinate system coincides with the top-left corner of Bob's image. The bottom-right corner of the image maps to (1,1).

So, what happened to pixel coordinates? It turns out that OpenGL ES doesn't like them a lot. Instead, any image we upload, no matter its width and height in pixels, will be embedded into this coordinate system. The top-left corner of the image will always be at (0,0), the bottom-right corner will always be at (1,1)—even if, say, the width is twice as large as the height. We call these *normalized coordinates*, and they actually makes our lives easier at times. So how can we map Bob to our triangle? Easy, we just give each vertex of the triangle a texture coordinate pair in Bob's coordinate system. Figure 7–11 shows a few configurations.

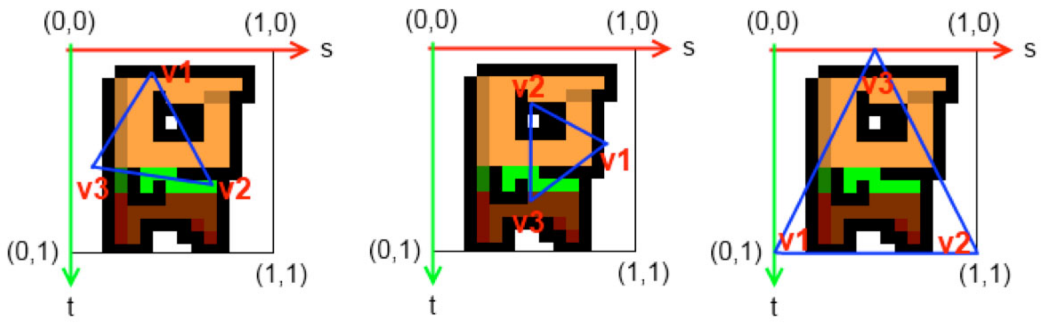

Figure 7–11. *Three different triangles mapped to Bob. The names v1, v2, and v3 each specify a vertex of the triangle.*

We can map our triangle's vertices to the texture coordinate system however we want. Note that the orientation of the triangle in the positional coordinate system does not have to be the same as in the texture coordinate system. The coordinate systems are completely decoupled. So let's see how we can add those texture coordinates to our vertices:

```
Int VERTEX_SIZE = (2 + 2) * 4;
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
byteBuffer.order(ByteOrder.nativeOrder());
vertices = byteBuffer.asFloatBuffer();
vertices.put( new float[] {    0.0f,   0.0f, 0.0f, 1.0f,
                             319.0f,   0.0f, 1.0f, 1.0f,
                             160.0f, 479.0f, 0.5f, 0.0f});
vertices.flip();
```

That was easy. All we have to do is to make sure that we have enough room in our buffer, and then append the texture coordinates to each vertex. The preceding code corresponds to the rightmost mapping in Figure 7–10. Note that our vertex positions are still given in the usual coordinate system we defined via our projection. If we wanted to, we could also add the color attributes to each vertex, as in the previous example. OpenGL ES would then mix the interpolated vertex colors with the colors from the pixels of the texture that the triangle maps to on the fly. Of course, we'd need to adjust the size of our buffer, as well as the `VERTEX_SIZE` constant, accordingly (e.g., (2 + 4 + 2) × 4). To tell OpenGL ES that our vertices have texture coordinates, we again use `glEnableClientState()` together with the `glTexCoordPointer()`method, which behaves exactly the same as `glVertexPointer()` and `glColorPointer()` (can you see a pattern here?).

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

vertices.position(0);
gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
vertices.position(2);
gl.glTexCoordPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
```

Nice, that looks very familiar. So, the remaining question is how we can upload the texture to OpenGL ES and tell it to map it to our triangle. Naturally that's a little bit more involved. But fear not, it's still pretty easy.

Uploading Bitmaps

First we have to load our bitmap. We already know how to do that on Android:

```
Bitmap bitmap = BitmapFactory.decodeStream(game.getFileIO().readAsset("bobrgb888.png"));
```

Here we load Bob in an RGB888 configuration. The next thing we need to do is tell OpenGL ES that we want to create a new texture. OpenGL ES has the notion of objects for a couple of things, such as textures. To create a texture object, we can call the following method:

```
GL10.glGenTextures(int numTextures, int[] ids, int offset)
```

The first parameter specifies how many texture objects we want to create. Usually we only want to create one. The next parameter is an int array to which OpenGL ES will write the IDs of the generated texture objects. The final parameter just tells OpenGL ES where in the array it should start writing the IDs to.

You already learned that OpenGL ES is a C API. Naturally it can't return a Java object to us for a new texture. Instead it gives us an ID, or handle, to that texture. Each time we want OpenGL ES to do something with that specific texture, we specify its ID. So here's a more complete code snippet showing how to generate a single new texture object and get its ID:

```
int textureIds[] = new int[1];
gl.glGenTextures(1, textureIds, 0);
int textureId = textureIds[0];
```

The texture object is still empty, which means it doesn't have any image data yet. Let's upload our bitmap. For this we have to first bind the texture. To bind something in OpenGL ES means that we want OpenGL ES to use that specific object for all subsequent calls until we change the binding again. Here we want to bind a texture object for which the method `glBindTexture()` is available. Once we have bound a texture, we can manipulate its attributes, such as its image data. Here's how we can upload Bob to our new texture object:

```
gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
```

First we bind the texture object with `glBindTexture()`. The first parameter specifies the type of texture we want to bind. Our image of `Bob` is 2D, so we use `GL10.GL_TEXTURE_2D`. There are other texture types, but we don't have a need for them in this book. We'll always specify `GL10.GL_TEXTURE_2D` for the methods that need to know the texture type we want to work with. The second parameter of that method is our texture ID. Once the method returns, all subsequent methods that work with a 2D texture will work with our texture object.

The next method call invokes a method of the `GLUtils` class, a class provided by the Android framework. Usually the task of uploading a texture image is pretty involved; this little helper class eases the pain for us a lot. All we need to do is specify the texture type (`GL10.GL_TEXTURE_2D`) the mip mapping level (we'll look at that in Chapter 11; it defaults to zero), the bitmap we want to upload, and another argument, which has to be set to zero in all cases. After this call our texture object has image data attached to it.

NOTE: The texture object and its image data are actually held in video RAM, not in our usual RAM. The texture object (and the image data) will get lost when the OpenGL ES context is destroyed (e.g., when our activity is paused and resumed). This means that we have to re-create the texture object and reupload our image data every time the OpenGL ES context is (re)-created. If we don't do this, all we'll see is a white triangle.

Texture Filtering

There's one last thing we need to define before we can use the texture object. It has to do with the fact that our triangle might take up more or fewer pixels on the screen than there are pixels in the mapped region of the texture. For example, the image of Bob in Figure 7–10 has a size of 128×128 pixels. Our triangle maps to half that image, so uses (128×128) / 2 pixels from the texture (which are also called *texels*). When we draw the triangle to the screen with the coordinates we defined in the preceding snippet, it will take up (320×480) / 2 pixels. That's a lot more pixels we use on the screen than we fetch from the texture map. It can of course also be the other way around: we use fewer pixels on the screen than from the mapped region of the texture. The first case is called *magnification*, and the second *minification*. For each case we need to tell OpenGL ES how it should upscale or downscale the texture. The up- and downscaling are also referd to as minification and magnification filters in OpenGL ES lingo. These filters are attributes of our texture object, much like the image data itself. To set them we have to first make sure that the texture object is bound via a call to `glBindTexture()`. If that's the case, we can set them like this:

```
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_NEAREST);
```

Both times we use the method `GL10.glTexParameterf()`, which sets an attribute of the texture. In the first call we specify the minification filter; in the second we call the magnification filter. The first parameter to that method is the texture type, which defaults to `GL10.GL_TEXTURE_2D` for us. The second argument tells the method which attributes we want to set—in our case, the `GL10.GL_TEXTURE_MIN_FILTER` and the `GL10.GL_TEXTURE_MAG_FILTER`. The last parameter specifies the type of filter that should be used. We have two options here: `GL10.GL_NEAREST` and `GL10.GL_LINEAR`.

The first filter type will always choose the nearest texel in the texture map to be mapped to a pixel. The second filter type will sample the four nearest texels for a pixel of the triangle and average them to arrive at the final color. We use the first type of filter if we want to have a pixelated look and the second if we want a smoothed look. Figure 7–12 shows the difference between the two types of filters.

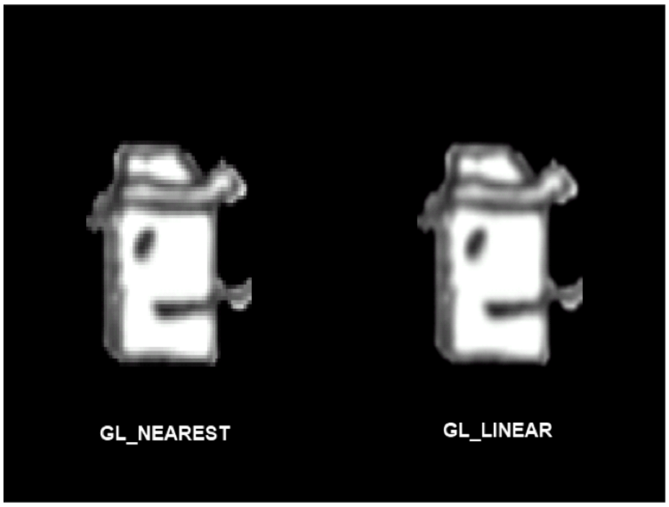

Figure 7–12. *GL10.GL_NEAREST vs. GL10.GL_LINEAR. The first filter type makes for a pixelated look; the second one smoothes things out a little.*

Our texture object is now fully defined: we created an ID, set the image data, and specified the filters to be used in case our rendering is not pixel perfect. It is a common practice to unbind the texture once we are done defining it. We should also recycle the `Bitmap` we loaded, as we no longer need it. Why waste memory? That can be achieved with the following snippet:

```
gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
bitmap.recycle();
```

`0` is a special ID that tells OpenGL ES that it should unbind the currently bound object. If we want to use the texture for drawing our triangles, we need to bind it again, of course.

Disposing of Textures

It is also useful to know how to delete a texture object from video RAM if we no longer need it (like we use `Bitmap.recycle()` to release the memory of a bitmap). This can be achieved with the following snippet:

```
gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
int textureIds = { textureid };
gl.glDeleteTextures(1, textureIds, 0);
```

Note that we first have to make sure that the texture object is not currently bound before we can delete it. The rest is similar to how we used `glGenTextures()` to create a texture object.

A Helpful Snippet

For reference, here's the complete snippet to create a texture object, load image data, and set the filters on Android:

```
Bitmap bitmap = BitmapFactory.decodeStream(game.getFileIO().readAsset("bobrgb888.png"));
int textureIds[] = new int[1];
gl.glGenTextures(1, textureIds, 0);
int textureId = textureIds[0];
gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
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_NEAREST);
gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
bitmap.recycle();
```

Not so bad after all. The most important part of all this is to actually recycle the `Bitmap` once we are done. Otherwise we'd waste memory. Our image data is safely stored in video RAM in the texture object (until the context is lost and we need to reload it again).

Enabling Texturing

There's one more thing before we can draw our triangle with the texture. We need to bind the texture, and we need to tell OpenGL ES that it should actually apply the texture to all triangles we render. Whether texture mapping is performed or not is another state of OpenGL ES, which we can enable and disable with the following methods:

```
GL10.glEnable(GL10.GL_TEXTURE_2D);
GL10.glDisable(GL10.GL_TEXTURE_2D);
```

These look vaguely familiar. When we enabled/disabled vertex attributes in the previous sections, we used `glEnableClientState()/glDisableClientState()`. As I said earlier, those are relics from the infancy of OpenGL itself. There's a reason why those are not merged with `glEnable()/glDisable()`, but we won't go into that here. Just remember to use `glEnableClientState()/glDisableClientState()` to enable and disable vertex attributes, and use `glEnable()/glDisable()` for any other states of OpenGL, such as texturing.

Putting It Together

With that out of our way, we can now write a small example that puts all of this together. Listing 7–7 shows an excerpt of the `TexturedTriangleTest.java` source file, listing only the relevant parts of the `TexturedTriangleScreen` class contained in it.

Listing 7–7. *Excerpt from TexturedTriangleTest.java; Texturing a Triangle*

```
class TexturedTriangleScreen extends Screen {
    final int VERTEX_SIZE = (2 + 2) * 4;
    GLGraphics glGraphics;
    FloatBuffer vertices;
    int textureId;
    public TexturedTriangleScreen(Game game) {
        super(game);
        glGraphics = ((GLGame) game).getGLGraphics();

        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertices = byteBuffer.asFloatBuffer();
        vertices.put( new float[] {    0.0f,   0.0f, 0.0f, 1.0f,
                                     319.0f,   0.0f, 1.0f, 1.0f,
                                     160.0f, 479.0f, 0.5f, 0.0f});
        vertices.flip();
        textureId = loadTexture("bobrgb888.png");
    }

    public int loadTexture(String fileName) {
        try {
            Bitmap bitmap =
BitmapFactory.decodeStream(game.getFileIO().readAsset(fileName));
            GL10 gl = glGraphics.getGL();
            int textureIds[] = new int[1];
            gl.glGenTextures(1, textureIds, 0);
            int textureId = textureIds[0];
            gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
            GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
            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_NEAREST);
            gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
            bitmap.recycle();
            return textureId;
        } catch(IOException e) {
            Log.d("TexturedTriangleTest", "couldn't load asset 'bobrgb888.png'!");
            throw new RuntimeException("couldn't load asset '" + fileName + "'");
        }
    }

    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        vertices.position(0);
        gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        vertices.position(2);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
    }
```

I took the freedom to put the texture loading into a method called `loadTexture()`, which simply takes the filename of a bitmap to be loaded. The method returns the texture object ID generated by OpenGL ES, which we'll use in the `present()` method to bind the texture.

The definition of our triangle shouldn't be a big surprise; we just added texture coordinates to each vertex.

The `present()` method does what it always does: it clears the screen and sets the projection matrix. Next we enable texture mapping via a call to `glEnable()` and bind our texture object. The rest is just what we did before: enabling the vertex attributes we want to use, telling OpenGL ES where it can find them and what strides to use, and finally drawing the triangle with a call to `glDrawArrays()`. Figure 7–13 shows the output of the preceding code.

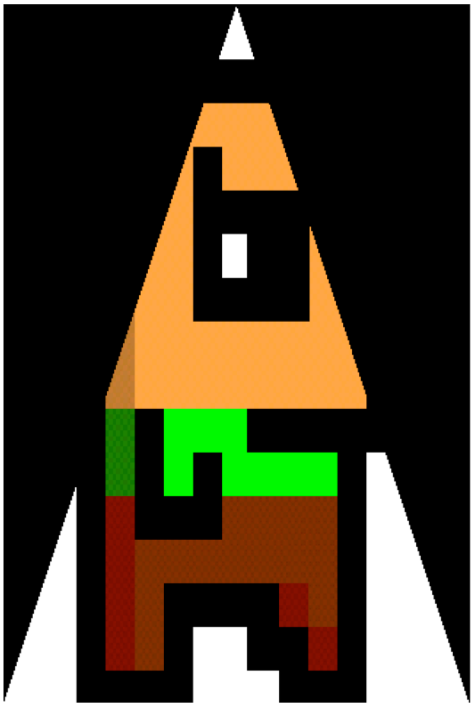

Figure 7–13. *Texture mapping Bob onto our triangle*

There's one last thing I haven't mentioned yet, and it's of great importance*:*

All bitmaps we load must have a width and height that is a power of two.

Stick to it or else things will explode.

So what does this actually mean? The image of Bob that we used in our example has a size of 128×128 pixels. The value 128 is 2 to the power of 7 (2×2×2×2×2×2×2). Other valid image sizes would be 2×8, 32×16, 128×256, and so on. There's also a limit to how big our images can be. Sadly, it varies depending on the hardware our application is running on. The OpenGL ES 1.x standard doesn't specify a minimally supported texture size to my knowledge. However, from my experience it seems that 512×512-pixel textures work on all current Android devices (and most likely will work on all future devices as well). I'd even go so far to say that 1024×1024 is OK as well.

Another issue that we have pretty much ignored so far is the color depth of our textures. Luckily the method `GLUtils.texImage2D()`, which we used to upload our image data to the GPU, handles this for us pretty well. OpenGL ES can cope with color depths like RGBA8888, RGB565, and so on. We should always strive to use the lowest possible color depth to decrease bandwidth. For this we can employ the `BitmapFactory.Options` class, as in previous chapters, to load a RGB888 `Bitmap` to a RGB565 `Bitmap` in memory, for example. Once we have loaded our `Bitmap` instance with the color depth we want it to have, `GLUtils.texImage2D()` takes over and makes sure that OpenGL ES gets the image data in the correct format. Of course, you should always check whether the reduction in color depth has a negative impact on the visual fidelity of your game.

A Texture Class

To reduce the code needed for subsequent examples, I wrote a little helper class called `Texture`. It will load a bitmap from an asset and create a texture object from it. It also has a few convenience methods to bind the texture and dispose of it. Listing 7–8 shows the code.

Listing 7–8. *Texture.java, a Little OpenGL ES Texture Class*

```
package com.badlogic.androidgames.framework.gl;

import java.io.IOException;
import java.io.InputStream;

import javax.microedition.khronos.opengles.GL10;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.opengl.GLUtils;

import com.badlogic.androidgames.framework.FileIO;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Texture {
    GLGraphics glGraphics;
    FileIO fileIO;
    String fileName;
    int textureId;
    int minFilter;
    int magFilter;
    public Texture(GLGame glGame, String fileName) {
        this.glGraphics = glGame.getGLGraphics();
        this.fileIO = glGame.getFileIO();
        this.fileName = fileName;
        load();
    }

    private void load() {
        GL10 gl = glGraphics.getGL();
        int[] textureIds = new int[1];
        gl.glGenTextures(1, textureIds, 0);
        textureId = textureIds[0];

        InputStream in = null;
        try {
            in = fileIO.readAsset(fileName);
            Bitmap bitmap = BitmapFactory.decodeStream(in);
            gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
            GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
            setFilters(GL10.GL_NEAREST, GL10.GL_NEAREST);
            gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
        } catch(IOException e) {
            throw new RuntimeException("Couldn't load texture '" + fileName +"'", e);
        } finally {
            if(in != null)
                try { in.close(); } catch (IOException e) { }
        }
    }

    public void reload() {
        load();
        bind();
        setFilters(minFilter, magFilter);
        glGraphics.getGL().glBindTexture(GL10.GL_TEXTURE_2D, 0);
    }

    public void setFilters(int minFilter, int magFilter) {
        this.minFilter = minFilter;
        this.magFilter = magFilter;
        GL10 gl = glGraphics.getGL();
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER, minFilter);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER, magFilter);
    }

    public void bind() {
        GL10 gl = glGraphics.getGL();
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
    }

    public void dispose() {
        GL10 gl = glGraphics.getGL();
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
        int[] textureIds = { textureId };
        gl.glDeleteTextures(1, textureIds, 0);
    }
}
```

The only interesting thing about this class is the `reload()` method, which we can use when the OpenGL ES context is lost. Also note that the `setFilters()` method will only work if the `Texture` is actually bound. Otherwise it will set the filters of the currently bound texture.

We could also write a little helper method for our vertices buffer. But before we can do this we have to discuss one more thing: indexed vertices.

Indexed Vertices: Because Reuse Is Good for You

Up until this point, we have always defined lists of triangles, where each triangle has its own set of vertices. We have actually only ever drawn a single triangle, but adding more would not have been a big deal.

There are cases, however, where two or more triangles can share some vertices. Let's think about how we'd render a rectangle with our current knowledge. We'd simply define two triangles that would have two vertices with the same positions, colors, and texture coordinates. We can do better. Figure 7–14 shows the old way and the new way of rendering a rectangle.

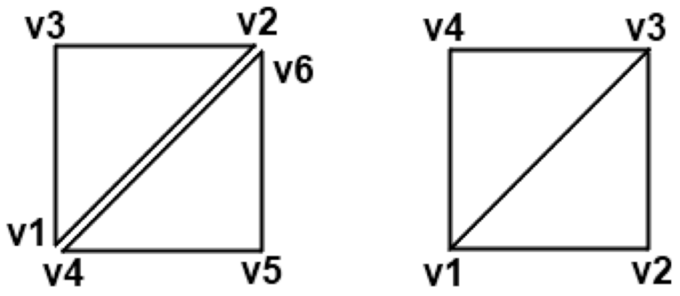

Figure 7–14. *Rendering a rectangle as two triangles with six vertices (left), and rendering it with four vertices (right)*

Instead of duplicating vertex v1 and v2 with vertex v4 and v6, we only define these vertices once. We still render two triangles in this case, but we tell OpenGL ES explicitly which vertices to use for each triangle (e.g., use v1, v2, and v3 for the first triangle and v3, v4, and v1 for the second one). Which vertices to use for each triangle is defined via indices into our vertices array. The first vertex in our array has index 0, the second vertex has index 1, and so on. For the preceding rectangle, we'd have a list of indices like this:

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

Incidentally, OpenGL ES wants us to specify the indices as shorts (which is not entirely correct; we could also use bytes). However, as with the vertex data, we can't just pass a short array to OpenGL ES. It wants a direct `ShortBuffer`. We already know how to handle that:

```
ByteBuffer byteBuffer = ByteBuffer.allocate(indices.length * 2);
```

```
byteBuffer.order(ByteOrder.nativeOrder());
ShortBuffer shortBuffer = byteBuffer.asShortBuffer();
shortBuffer.put(indices);
shortBuffer.flip();
```

A short needs 2 bytes of memory, so we allocate `indices.length` × 2 bytes for our `ShortBuffer`. We set the order to native again and get a `ShortBuffer` view so we can handle the underlying `ByteBuffer` more easily. All that's left is putting our indices into the `ShortBuffer` and flipping it so the limit and position are set correctly.

If we wanted to draw Bob as a rectangle with two indexed triangles, we could define our vertices like this:

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(4 * VERTEX_SIZE);
byteBuffer.order(ByteOrder.nativeOrder());
vertices = byteBuffer.asFloatBuffer();
vertices.put(new float[] {  100.0f, 100.0f, 0.0f, 1.0f,
                            228.0f, 100.0f, 1.0f, 1.0f,
                            228.0f, 229.0f, 1.0f, 0.0f,
                            100.0f, 228.0f, 0.0f, 0.0f });
vertices.flip();
```

The order of the vertices is exactly the same as in the right part of Figure 7–13. We tell OpenGL ES that we have positions and texture coordinates for our vertices and where it can find these vertex attributes via the usual calls to `glEnableClientState()` and `glVertexPointer()`/`glTexCoordPointer()`. The only thing that is different is the method we call to actually draw the two triangles:

```
gl.glDrawElements(GL10.GL_TRIANGLES, 6, GL10.GL_UNSIGNED_SHORT, indices);
```

It is very similar to `glDrawArrays()`, actually. The first parameter specifies the type of primitive we want to render—in this case a list of triangles. The next parameter specifies how many vertices we want to use, which equals six in our case. The third parameter specifies what type the indices have—we specify unsigned short. Note that Java has no unsigned types, though. However, given the one-complement encoding of signed numbers, it's OK to use a `ShortBuffer` that actually holds signed shorts. The last parameter is our `ShortBuffer` holding the six indices.

So, what will OpenGL ES do? It knows that we want to render triangles. It knows that we want to render two triangles, as we specified six vertices to be rendered. But instead of fetching six vertices sequentially from the vertices array, it goes sequentially through the index buffer and uses the vertices indexed by it.

Putting It Together

When we put it all together, we arrive at the code in Listing 7–9.

Listing 7–9. *Excerpt from IndexedTest.java; Drawing Two Indexed Triangles*

```
class IndexedScreen extends Screen {
    final int VERTEX_SIZE = (2 + 2) * 4;
    GLGraphics glGraphics;
    FloatBuffer vertices;
    ShortBuffer indices;
    Texture texture;

    public IndexedScreen(Game game) {
        super(game);
        glGraphics = ((GLGame) game).getGLGraphics();

        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(4 * VERTEX_SIZE);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertices = byteBuffer.asFloatBuffer();
        vertices.put(new float[] {  100.0f, 100.0f, 0.0f, 1.0f,
                                    228.0f, 100.0f, 1.0f, 1.0f,
                                    228.0f, 228.0f, 1.0f, 0.0f,
                                    100.0f, 228.0f, 0.0f, 0.0f });
        vertices.flip();

        byteBuffer = ByteBuffer.allocateDirect(6 * 2);
        byteBuffer.order(ByteOrder.nativeOrder());
        indices = byteBuffer.asShortBuffer();
        indices.put(new short[] { 0, 1, 2,
                                  2, 3, 0 });
        indices.flip();

        texture = new Texture((GLGame)game, "bobrgb888.png");
    }

    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        texture.bind();

        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

        vertices.position(0);
        gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        vertices.position(2);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);

        gl.glDrawElements(GL10.GL_TRIANGLES, 6, GL10.GL_UNSIGNED_SHORT, indices);
    }
```

Note the use of our awesome `Texture` class, which brings down the code size considerably. Figure 7–15 shows the output, and Bob in all his glory.

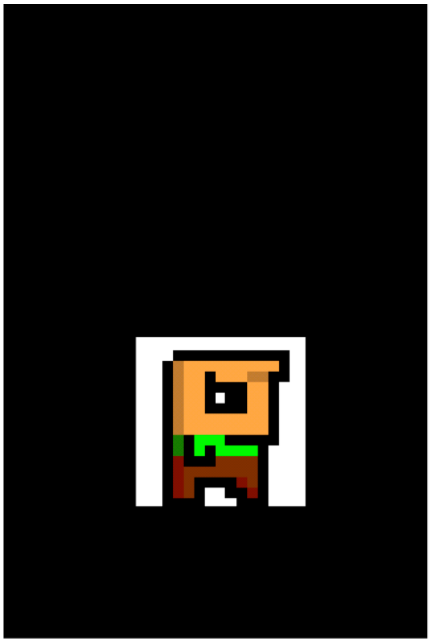

Figure 7–15. *Bob, indexed*

Now, this is pretty close already to how we worked with `Canvas`. We have a lot more flexibility as well, since we are not limited to axis-aligned rectangles anymore.

This example has covered all we need to know about vertices for now. We saw that every vertex must have at least a position, and can have additional attributes, such as a color given as four RGBA float values and texture coordinates. We also saw that we can reuse vertices via indexing in case we want to avoid duplication. This gives us a little performance boost, since OpenGL ES does not have to multiply more vertices by the projection and model-view matrices than absolutely necessary (which is again not entirely correct, but let's stick to this interpretation).

A Vertices Class

Let's make our code easier to write by creating a `Vertices` class that can hold a maximum number of vertices and, optionally, indices to be used for rendering. It should also take care of enabling all the states needed for rendering, as well as cleaning up the states after rendering has finished, so that other code can rely on a clean set of OpenGL ES states. Listing 7–10 shows our easy-to-use `Vertices` class.

Listing 7–10. *Vertices.java; Encapsulating (Indexed) Vertices*

```
package com.badlogic.androidgames.framework.gl;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Vertices {
    final GLGraphics glGraphics;
    final boolean hasColor;
    final boolean hasTexCoords;
    final int vertexSize;
    final FloatBuffer vertices;
    final ShortBuffer indices;
```

The Vertices class has a reference to the GLGraphics instance, so we can get ahold of the GL10 instance when we need it. We also store whether the vertices have colors and texture coordinates. This gives us great flexibility, as we can choose the minimal set of attributes we need for rendering. We also store a FloatBuffer that holds our vertices and a ShortBuffer that holds the optional indices.

```
    public Vertices(GLGraphics glGraphics, int maxVertices, int maxIndices, boolean
hasColor, boolean hasTexCoords) {
        this.glGraphics = glGraphics;
        this.hasColor = hasColor;
        this.hasTexCoords = hasTexCoords;
        this.vertexSize = (2 + (hasColor?4:0) + (hasTexCoords?2:0)) * 4;

        ByteBuffer buffer = ByteBuffer.allocateDirect(maxVertices * vertexSize);
        buffer.order(ByteOrder.nativeOrder());
        vertices = buffer.asFloatBuffer();

        if(maxIndices > 0) {
            buffer = ByteBuffer.allocateDirect(maxIndices * Short.SIZE / 8);
            buffer.order(ByteOrder.nativeOrder());
            indices = buffer.asShortBuffer();
        } else {
            indices = null;
        }
    }
```

In the constructor, we specify how many vertices and indices our Vertices instance can hold maximally, as well as whether the vertices have colors or texture coordinates. Inside the constructor, we then set the members accordingly, and instantiate the buffers. Note that the ShortBuffer will be set to null if maxIndices is zero. Our rendering will be performed nonindexed in that case.

```
    public void setVertices(float[] vertices, int offset, int length) {
        this.vertices.clear();
        this.vertices.put(vertices, offset, length);
        this.vertices.flip();
    }

    public void setIndices(short[] indices, int offset, int length) {
        this.indices.clear();
        this.indices.put(indices, offset, length);
        this.indices.flip();
    }
```

Next up are the setVertices() and setIndices() methods. The latter will throw a NullPointerException in case the Vertices instance does not store indices. All we do is clear the buffers and copy the contents of the arrays.

```
    public void draw(int primitiveType, int offset, int numVertices) {
        GL10 gl = glGraphics.getGL();

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        vertices.position(0);
        gl.glVertexPointer(2, GL10.GL_FLOAT, vertexSize, vertices);

        if(hasColor) {
            gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
            vertices.position(2);
            gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
        }

        if(hasTexCoords) {
            gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
            vertices.position(hasColor?6:2);
            gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
        }

        if(indices!=null) {
            indices.position(offset);
            gl.glDrawElements(primitiveType, numVertices, GL10.GL_UNSIGNED_SHORT,
indices);
        } else {
            gl.glDrawArrays(primitiveType, offset, numVertices);
        }

        if(hasTexCoords)
            gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        if(hasColor)
            gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
    }
}
```

The final method of the Vertices class is draw(). It takes the type of the primitive (e.g., GL10.GL_TRIANGLES), the offset into the vertices buffer (or the indices buffer if we use indices), and the number of vertices to use for rendering. Depending on whether the vertices have colors and texture coordinates, we enable the relevant OpenGL ES states and tell OpenGL ES where to find the data. We do the same for the vertex positions, of course, which are always needed. Depending on whether indices are used or not, we either call glDrawElements() or glDrawArrays() with the parameters passed to the method. Note that the offset parameter can also be used in case of indexed rendering: we simply set the position of the indices buffer accordingly so that OpenGL ES starts reading the indices from that offset instead of the first index of the indices buffer. The last thing we do in the draw() method is clean up the OpenGL ES state a little. We call glDisableClientState() with either GL10.GL_COLOR_ARRAY or

GL10.GL_TEXTURE_COORD_ARRAY in case our vertices have these attributes. We need to do this, as another instance of Vertices might not use those attributes. If we rendered that other Vertices instance, OpenGL ES would still look for colors and/or texture coordinates.

We could replace all the tedious code in the constructor of our preceding example with the following snippet:

```
Vertices vertices = new Vertices(glGraphics, 4, 6, false, true);
vertices.setVertices(new float[] { 100.0f, 100.0f, 0.0f, 1.0f,
                                   228.0f, 100.0f, 1.0f, 1.0f,
                                   228.0f, 228.0f, 1.0f, 0.0f,
                                   100.0f, 228.0f, 0.0f, 0.0f }, 0, 16);
vertices.setIndices(new short[] { 0, 1, 2, 2, 3, 0 }, 0, 6);
```

Likewise, we could replace all the calls for setting up our vertex attribute arrays and rendering with a single call to the following:

```
vertices.draw(GL10.GL_TRIANGLES, 0, 6);
```

Together with our Texture class we have a pretty nice basis for all our 2D OpenGL ES rendering now. One of the things we are still missing to be able to completely reproduce all our Canvas rendering abilities is, though, blending. Let's have a look at that.

Alpha Blending: I Can See Through You

Alpha blending in OpenGL ES is pretty easy to enable. We only need two method calls:

```
gl.glEnable(GL10.GL_BLEND);
gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
```

The first method call should be familiar: it just tells OpenGL ES that it should apply alpha blending to all triangles we render from this point on. The second method is a little bit more involved. It specifies how the source and destination color should be combined. If you remember what we discussed in Chapter 3, the way a source color and a destination color are combined is governed by a simple blending equation. The method glBlendFunc() just tells OpenGL ES which kind of equation to use. The preceding parameters specify that we want the source color to be mixed with the destination color exactly as specified in the blending equation in Chapter 3. This is equal to how the Canvas blended Bitmaps for us.

Blending in OpenGL ES is pretty powerful and complex, and there's a lot more to it. For our purposes, we can ignore all those details, though, and just use the preceding blending function whenever we want to blend our triangles with the framebuffer—the same way we blended Bitmaps with the Canvas.

The second question is where the source and destination colors come from. The latter is easy to explain: it's the color of the pixel in the framebuffer we are going to overwrite with the triangle we draw. The source color is actually a combination of two colors:

The vertex color: This is the color we either specify via glColor4f() for all vertices or on a per-vertex basis by adding a color attribute to each vertex.

The texel color: As mentioned before, a texel is a pixel from a texture. When our triangle is rendered with a texture mapped to it, OpenGL ES will mix the texel colors with the vertex colors for each pixel of a triangle.

So if our triangle is not texture mapped, the source color for blending is equal to the vertex color. If the triangle is texture mapped, the source color for each of the triangle's pixels is a mixture of the vertex color and the texel color. We could specify how the vertex and texel colors are combined by using the `glTexEnv()` method. The default is to *modulate* the vertex color by the texel color, which basically means that the two colors are multiplied with each other component-wise (vertex r × texel r, and so on). For all our use cases in this book, this is exactly what we want, so we won't go into `glTexEnv()`. There are also some very specialized cases where you might want to change how the vertex and texel colors are combined. As with `glBlendFunc()`, we'll ignore the details and just use the default.

When we load a texture image that doesn't have an alpha channel, OpenGL ES will automatically assume an alpha value of 1 for each pixel. If we load an image in RGBA8888 format, OpenGL ES will happily use the supplied alpha values for blending.

For vertex colors we always have to specify an alpha component, either by using `glColor4f()`, where the last argument is the alpha value, or by specifying the four components per vertex, where again the last component is the alpha value.

Let's put this into practice with a little example. We want to draw `Bob` twice: once by using the image `bobrgb888.png`, which does not have an alpha channel per pixel, and a second time by using the image `bobargb8888.png`, which has alpha information. Note that the PNG image actually stores the pixels in ARGB8888 format instead of RGBA8888. Luckily the `GLUtils.texImage2D()` method we use to upload the image data for a texture will do the conversion for us automatically. Listing 7–11 shows the code of our little experiment, using the `Texture` and `Vertices` classes.

Listing 7–11. ***Excerpt from BlendingTest.java; Blending in Action***

```
class BlendingScreen extends Screen {
    GLGraphics glGraphics;
    Vertices vertices;
    Texture textureRgb;
    Texture textureRgba;

    public BlendingScreen(Game game) {
        super(game);
        glGraphics = ((GLGame)game).getGLGraphics();

        textureRgb = new Texture((GLGame)game, "bobrgb888.png");
        textureRgba = new Texture((GLGame)game, "bobargb8888.png");

        vertices = new Vertices(glGraphics, 8, 12, true, true);
        float[] rects = new float[] {
                100, 100, 1, 1, 1, 0.5f, 0, 1,
                228, 100, 1, 1, 1, 0.5f, 1, 1,
                228, 228, 1, 1, 1, 0.5f, 1, 0,
                100, 228, 1, 1, 1, 0.5f, 0, 0,
            100, 300, 1, 1, 1, 1, 0, 1,
            228, 300, 1, 1, 1, 1, 1, 1,
            228, 428, 1, 1, 1, 1, 1, 0,
            100, 428, 1, 1, 1, 1, 0, 0
        };
        vertices.setVertices(rects, 0, rects.length);
        vertices.setIndices(new short[] {0, 1, 2, 2, 3, 0,
                                         4, 5, 6, 6, 7, 4 }, 0, 12);
    }
```

Our little `BlendingScreen` implementation holds a single `Vertices` instance where we'll store the two rectangles, as well as two `Texture` instances—one holding the RGBA8888 image of Bob and the other one storing the RGB888 version of Bob. In the constructor we load both textures from the files `bobrgb888.png` and `bobargb8888.png`, and rely on the `Texture` class and `GLUtils.texImag2D()` to convert the ARGB8888 PNG to RGBA8888, as needed by OpenGL ES. Next up, we define our vertices and indices. The first rectangle, consisting of four vertices, maps to the RGB888 texture of Bob. The second rectangle maps to the RGBA8888 version of Bob and is rendered 200 units above the RGB888 Bob rectangle. Note that the vertices of the first rectangle all have the color (1,1,1,0.5f) while the vertices of the second rectangle have the color (1,1,1,1).

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClearColor(1,0,0,1);
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        textureRgb.bind();
        vertices.draw(GL10.GL_TRIANGLES, 0, 6 );

        textureRgba.bind();
        vertices.draw(GL10.GL_TRIANGLES, 6, 6 );
    }
```

In our `present()` method we clear the screen with red and set the projection matrix, as we are used to doing. Next we enable alpha blending and set the correct blend equation. Finally we enable texture mapping and render the two rectangles. The first rectangle is rendered with the RGB888 texture bound, and the second rectangle is rendered with the RGBA8888 texture bound. We store both rectangles in the same `Vertices` instance and thus use offsets with the `vertices.draw()` methods. Figure 7–16 shows the output of this little gem.

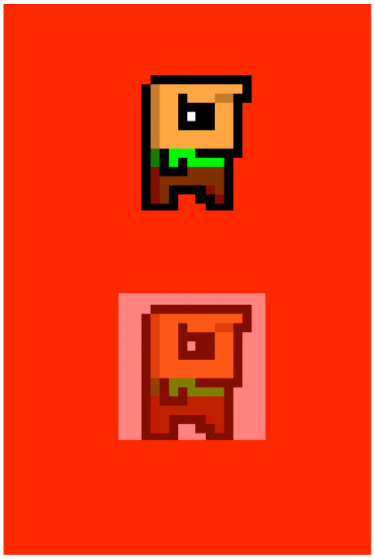

Figure 7–16. *Bob, vertex color blended (bottom) and texture blended (top)*

In the case of RGB888 Bob, the blending is performed via the alpha values in the per-vertex colors. Since we set those to 0.5f, Bob is 50 percent translucent.

In the case of RGBA8888 Bob, the per-vertex colors all have an alpha value of 1. However, since the background pixels of that texture have alpha values of 0, and since the vertex colors and the texel colors are modulated, the background of this version of Bob disappears. If we'd have set the per-vertex colors' alpha values to 0.5f as well, then Bob himself would also have been 50 percent as translucent as his clone in the bottom of the screen. Figure 7–17 shows what that would have looked like.

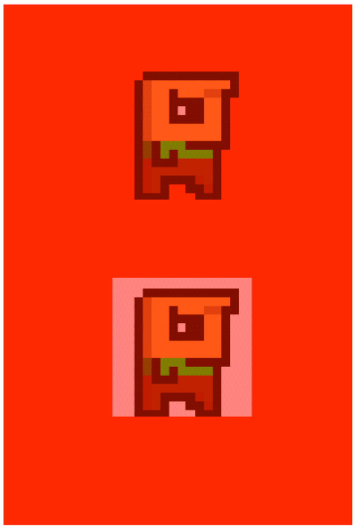

Figure 7–17. *An alternative version of RGBA8888 Bob using per-vertex alpha of 0.5f (top of the screen)*

And that's basically all we need to know about blending with OpenGL ES in 2D.

However, there is one more very important thing I'd like to point out: *Blending is expensive*! Seriously, don't overuse it. Current mobile GPUs are not all that good at blending massive amounts of pixels. You should only use blending if absolutely necessary.

More Primitives: Points, Lines, Strips, and Fans

When I told you that OpenGL ES was a big, nasty triangle-rendering machine, I was not being 100 percent honest. In fact, OpenGL ES can also render points and lines. Best of all: these are also defined via vertices, and thus all of the above also applies to them (texturing, per-vertex colors, etc.). All we need to do to render these primitives is use something other than `GL10.GL_TRIANGLES` when we call `glDrawArrays()/glDrawElements()`. We can also perform indexed rendering with these primitives, although that's a bit redundant (in the case of points at least). Figure 7–18 shows a list of all the primitive types OpenGL ES offers us.

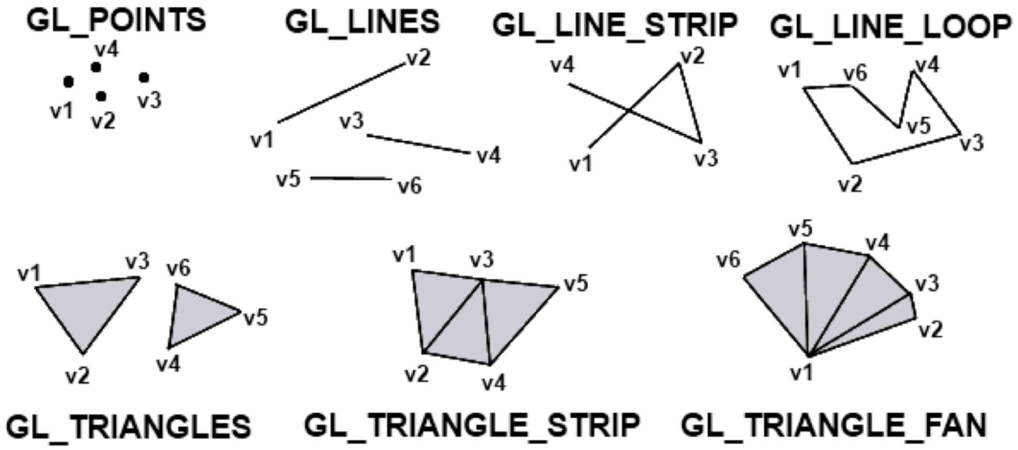

Figure 7–18. *All the primitives OpenGL ES can render*

Let's go through all of these primitives really quickly:

Point: With a point, each vertex is its own primitive.

Line: A line is made up of two vertices. As with triangles, we can just have 2 × *n* vertices to define *n* lines.

Line strip: All the vertices are interpreted as belonging to one long line.

Line loop: This is similar to a line strip, with the difference that OpenGL ES will automatically draw an additional line from the last vertex to the first vertex.

Triangle: This we already know. Each triangle is made up of three vertices.

Triangle strip: Instead of specifying three vertices, we just specify *number of triangles* + 1 vertices. OpenGL ES will then construct the first triangle from vertices (v1,v2,v3), the next triangle from vertices (v2,v3,v4), and so on.

Triangle fan: This has one base vertex (v1) that is shared by all triangles. The first triangle will be (v1,v2,v3), the next triangle (v1,v3,v4), and so on.

Triangle strips and fans are a little bit less flexible than pure triangle lists. But they can give a little performance boost, as fewer vertices have to be multiplied by the projection and model-view matrices. We'll stick to triangle lists in all our code, though, as they are easier to use and can be made to achieve similar performance by using indices.

Points and lines are a little bit strange in OpenGL ES. When we use a pixel-perfect orthographic projection (e.g., our screen resolution is 320×480 pixels and our `glOrthof()` call uses those exact values), we still don't get pixel-perfect rendering in all cases. The positions of the point and line vertices have to be offset by 0.375f due to something called the diamond exit rule. Keep that in mind if you want to render pixel-perfect points and lines. We already saw that something similar applies to triangles. However, given that we usually draw rectangles in 2D, we don't run into that problem.

Given that all you have to do to render primitives other than `GL10.GL_TRIANGLES` is to use one of the other constants in Figure 7–17, I'll spare you an example program. We'll stick to triangle lists for the most part, especially when doing 2D graphics programming.

Let's now dive into one more thing OpenGL ES offers us: the almighty model-view matrix.

2D Transformations: Fun with the Model-View Matrix

All we have done so far is define static geometries in the form of triangle lists. There was nothing moving, rotating, or scaling. Also, even when the vertex data itself stayed the same (e.g., the width and height of a rectangle composed of two triangles along with texture coordinates and color), we still had to duplicate the vertices if we wanted to draw the same rectangle at different places. Look back at Listing 7–11 and ignore the color attributes of the vertices for now. The two rectangles only differ in their y-coordinates by 200 units. If we had a way to move those vertices without actually changing their values we could get away with defining the rectangle of Bob only once, and simply drawing him at different locations. And that's exactly what we can use the model-view matrix for.

World and Model Space

To understand how this works we have to literally think outside of our little orthographic view frustum box. Our view frustum is in a special coordinate system called the *world space*. This is the space where all our vertices are going to end up eventually.

Up until now we have specified all vertex positions in absolute coordinates relative to the origin of this world space (compare with Figure 7–5). What we really want is to make the definition of the positions of our vertices independent from this world space coordinate system. We can achieve this by giving each of our models (e.g., Bob's rectangle, a spaceship, etc.) its own coordinate system.

This is what we usually call *model space*, the coordinate system within which we define the positions of our model's vertices. Figure 7–19 illustrates this concept in 2D, and the same rules apply to 3D as well (just add a z-axis).

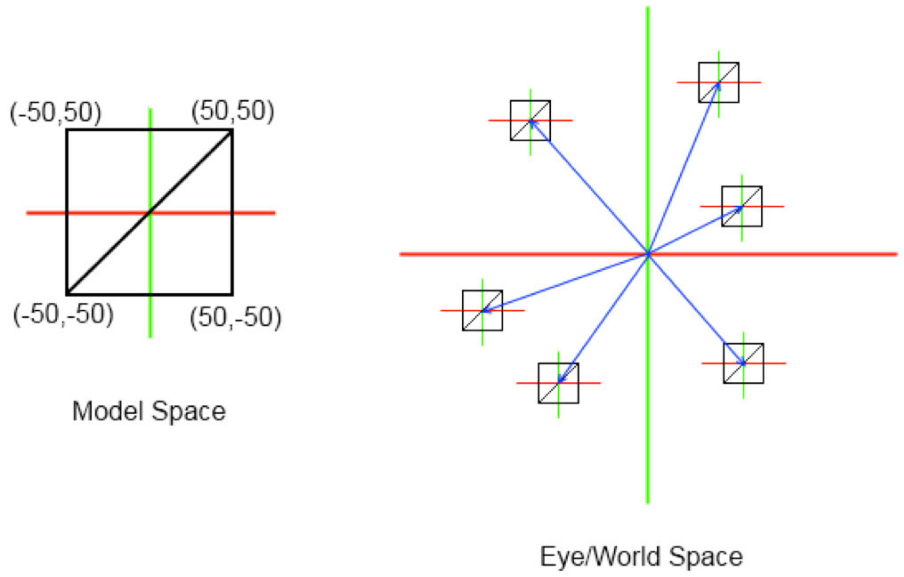

Figure 7–19. *Defining our model in model space, reusing it, and rendering it at different locations in the world space*

In Figure 7–19 we have a single model, defined via a `Vertices` instance—for example, like this:

```
Vertices vertices = new Vertices(glGraphics, 4, 12, false, false);
vertices.setVertices(new float[] { -50, -50,
                                    50, -50,
                                    50,  50,
                                   -50,  50 }, 0, 8);
vertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
```

For our discussion we just leave out any vertex colors or texture coordinates. Now, when we render this model without any further modifications, it will be placed around the origin in the world space in our final image. If we want to render it at a different position—say, its center being at (200,300) in world space—we could redefine the vertex positions like this:

```
vertices.setVertices(new float[] { -50 + 200, -50 + 300,
                                    50 + 200, -50 + 300,
                                    50 + 200,  50 + 300,
                                  -50 + 200,  50 + 300 }, 0, 8);
```

On the next call to `vertices.draw()`, the model would be rendered with its center at (200,300). But this is a tad bit tedious isn't it?

Matrices Again

Remember when we briefly talked about matrices earlier? We discussed how matrices can encode transformations such as translations (moving stuff around), rotations, and scaling. The projection matrix we use to project our vertices onto the projection plane encodes a special type of transformation: a projection.

Matrices are the key to solving our previous problem more elegantly. Instead of manually moving our vertex positions around by redefining them, we simply set a matrix that encodes a translation. Since the projection matrix of OpenGL ES is already occupied by our orthographics projection matrix we specified via `glOrthof()`, we use a different OpenGL ES matrix: the model-view matrix. Here's how we could render our model with its origin moved to a specific location in eye/world space:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(200, 300, 0);
vertices.draw(GL10.GL_TRIANGLES, 0, 6);
```

We have to first tell OpenGL ES which matrix we want to manipulate. In our case that's the model-view matrix, which is specified by the constant `GL10.GL_MODELVIEW`. Next we make sure that the model-view matrix is set to an identity matrix. Basically we just overwrite anything that was in there already—we sort of clear the matrix. The next call is where the magic happens.

The method `glTranslatef()` takes three arguments: the translation on the x-, y-, and z-axes. Since we want the origin of our model to be placed at (200,300) in eye/world space, we specify a translation by 200 units on the x-axis and a translation by 300 units on the y-axis. As we are working in 2D, we simply ignore the z-axis and set the translation component to zero. We didn't specify a z-coordinate for our vertices, so these will default to zero. Adding zero to zero equals zero, so our vertices will stay in the x-y plane.

From this point on, the model-view matrix of OpenGL ES encodes a translation by (200,300,0), which will be applied to all vertices that pass through the OpenGL ES pipeline. If you refer back to Figure 7–4, you'll see that OpenGL ES will multiply each vertex with the model-view matrix first and then apply the projection matrix. Up until this point, the model-view matrix was set to an identity matrix (the default of OpenGL ES). It therefore did not have an effect on our vertices. Our little `glTranslatef()` call changes this, and will move all vertices first before they are projected.

Of course, this is done on the fly; the values in our `Vertices` instance do not change at all. We would have noticed any permanent change to your `Vertices` instance as by that logic the projection matrix would have changed it already.

An First Example Using Translation

What can we use this for? Say we want to render 100 Bobs at different positions in our world. Additionally we want them to move around on the screen and change direction each time they hit an edge of the screen (or rather a plane of our parallel projection view frustum, which coincides with the extents of our screen). We could do this by having one large `Vertices` instance that holds the vertices of the 100 rectangles—one for each Bob—and recalculate the vertex positions each frame. The easier method is to have one small `Vertices` instance that only holds a single rectangle (the model of Bob) and reuse it by translating it with the model-view matrix on the fly. Let's define our Bob model:

```
Vertices bobModel = new Vertices(glGraphics, 4, 12, false, true);
bobModel.setVertices(new float[] { -16, -16, 0, 1,
                                    16, -16, 1, 1,
                                    16,  16, 1, 0,
                                   -16,  16, 0, 0, }, 0, 8);
bobModel.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
```

So, each Bob is 32×32 units in size. We also texture map him (we'll use `bobrgb888.png` to see the extents of each Bob).

Bob Becomes a Class

Let's define a simple `Bob` class. It will be responsible for holding a `Bob`'s position and advancing his position in his current direction based on the delta time, just like we advanced Mr. Nom (with the difference that we don't move in a grid anymore). The `update()` method will also make sure that Bob doesn't escape our view volume bounds. Listing 7–12 shows the `Bob` class.

Listing 7–12. *Bob.java*

```
package com.badlogic.androidgames.glbasics;

import java.util.Random;

class Bob {
    static final Random rand = new Random();
    public float x, y;
    float dirX, dirY;

    public Bob() {
        x = rand.nextFloat() * 320;
        y = rand.nextFloat() * 480;
        dirX = 50;
        dirY = 50;
    }

    public void update(float deltaTime) {
        x = x + dirX * deltaTime;
        y = y + dirY * deltaTime;

        if (x < 0) {
            dirX = -dirX;
            x = 0;
        }

        if (x > 320) {
            dirX = -dirX;
            x = 320;
        }

        if (y < 0) {
            dirY = -dirY;
            y = 0;
        }

        if (y > 480) {
            dirY = -dirY;
            y = 480;
        }
    }
}
```

Every Bob will place himself at a random location in the world when we construct him. All the Bobs will initially move in the same direction: 50 units to the right and 50 units upward per second (as we multiply by the deltaTime). In the update() method we simply advance Bob in his current direction in a time-based manner, and then check if he left the view frustum bounds. If that's the case we invert his direction and make sure he's still in the view frustum.

Now let's assume we are instantiating 100 Bobs, like this:

```
Bob[] bobs = new Bob[100];
for(int i = 0; i < 100; i++) {
    bobs[i] = new Bob();
}
```

To render each of these Bobs, we'd do something like this (assuming we've already cleared the screen, set the projection matrix, and bound the texture):

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
for(int i = 0; i < 100; i++) {
    bob.update(deltaTime);
    gl.glLoadIdentity();
    gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
    bobModel.render(GL10.GL_TRIANGLES, 0, 6);
}
```

That is pretty sweet, isn't it? For each Bob, we call his update() method, which will advance his position and make sure he stays within the bounds of our little world. Next we load an identity matrix into the model-view matrix of OpenGL ES so we have a clean slate. We then use the current Bob's x- and y-coordinates in a call to glTransltef(). When we then render the Bob model in the next call, all the vertices will be offset by the current Bob's position—exactly what we wanted.

Putting It Together

Let's make this a full-blown example. Listing 7–13 shows the code.

Listing 7–13. *BobTest.java; 100 Moving Bobs!*

```
package com.badlogic.androidgames.glbasics;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.gl.FPSCounter;
import com.badlogic.androidgames.framework.gl.Texture;
import com.badlogic.androidgames.framework.gl.Vertices;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class BobTest extends GLGame {

    @Override
    public Screen getStartScreen() {
        return new BobScreen(this);
    }

    class BobScreen extends Screen {
        static final int NUM_BOBS = 100;
        GLGraphics glGraphics;
        Texture bobTexture;
        Vertices bobModel;
        Bob[] bobs;
```

Our BobScreen class holds a Texture (loaded from bobrbg888.png), a Vertices instance holding the model of Bob (a simple textured rectangle), and an array of Bob instances. We also define a little constant named NUM_BOBS so we can modify the number of Bobs we want to have on the screen.

```
        public BobScreen(Game game) {
            super(game);
            glGraphics = ((GLGame)game).getGLGraphics();

            bobTexture = new Texture((GLGame)game, "bobrgb888.png");

            bobModel = new Vertices(glGraphics, 4, 12, false, true);
            bobModel.setVertices(new float[] { -16, -16, 0, 1,
                                                16, -16, 1, 1,
                                                16,  16, 1, 0,
                                               -16,  16, 0, 0, }, 0, 16);
            bobModel.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);


            bobs = new Bob[100];
            for(int i = 0; i < 100; i++) {
                bobs[i] = new Bob();
            }
        }
```

The constructor just loads the texture, creates the model, and instantiates `NUM_BOBS` `Bob` instances.

```
@Override
        public void update(float deltaTime) {
            game.getInput().getTouchEvents();
            game.getInput().getKeyEvents();

            for(int i = 0; i < NUM_BOBS; i++) {
                bobs[i].update(deltaTime);
            }
        }
```

The `update()` method is where we let our `Bobs` update themselves. We also make sure our input event buffers are emptied.

```
@Override
        public void present(float deltaTime) {
            GL10 gl = glGraphics.getGL();
            gl.glClearColor(1,0,0,1);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            gl.glOrthof(0, 320, 0, 480, 1, -1);

            gl.glEnable(GL10.GL_TEXTURE_2D);
            bobTexture.bind();

            gl.glMatrixMode(GL10.GL_MODELVIEW);
            for(int i = 0; i < NUM_BOBS; i++) {
                gl.glLoadIdentity();
                gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
                gl.glRotatef(45, 0, 0, 1);
                gl.glScalef(2, 0.5f, 0);
                bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
            }
        }
```

In the `render()` method we clear the screen, set the projection matrix, enable texturing, and bind the texture of Bob. The last couple of lines are responsible for actually rendering each `Bob` instance. Since OpenGL ES remembers its states, we have to set the active matrix only once (in this case we are going to modify the model-view matrix in the rest of the code). We then loop through all the `Bobs`, set the model-view matrix to a translation matrix based on the position of the current `Bob`, and render the model, which will be translated by the model view-matrix automatically.

```
        @Override
        public void pause() {
        }

        @Override
        public void resume() {
        }

        @Override
        public void dispose() {
        }
    }
}
```

That's it. Best of all, we employed the MVC pattern we used in Mr. Nom again. It really lends itself well to game programming. The logical side of Bob is completely decoupled from his appearance, which is nice, as we can easily replace his appearance with something more complex. Figure 7–20 shows the output of our little program after running for a few seconds.

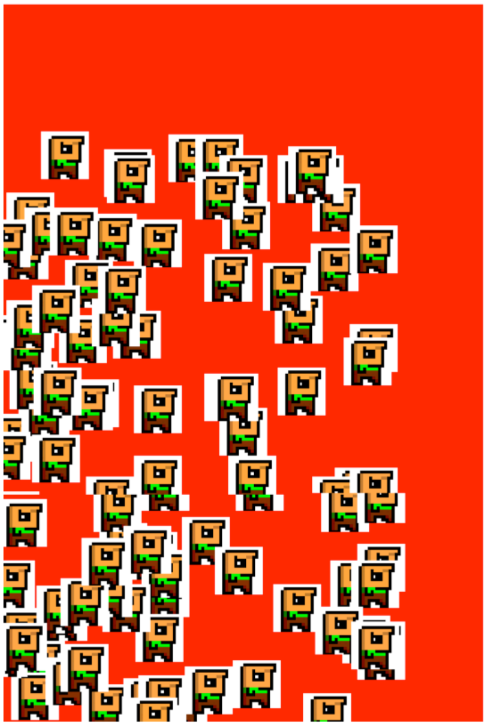

Figure 7–20. *That's a lot of Bobs.*

That's not the end of all our fun with transformations yet. If you remember what I said a couple of pages ago, you'll know what's coming: rotations and scaling.

More Transformations

Besides the `glTranslatef()` method, OpenGL ES also offers us two methods for transformations: `glRotatef()` and `glScalef()`.

Rotation

Here's the signature of `glRotatef()`:

```
GL10.glRotatef(float angle, float axisX, float axisY, float axisZ);
```

The first parameter is the angle in degrees we want to rotate our vertices by. But what do the rest of the parameters mean?

When we rotate something, we rotate it around an axis. What is an axis? Well, we already know three axes: the x-axis, the y-axis, and the z-axis. We can express these three axes as so-called *vectors*. The positive x-axis would be described as (1,0,0), the positive y-axis would be (0,1,0) and the positive z-axis would be (0,0,1). As you can see, a vector actually encodes a direction, in our case in 3D space. Bob's direction is also a vector, but in 2D space. Vectors can also encode positions, like Bob's position in 2D space.

To define the axis around which we want to rotate the model of Bob, we need to go back to 3D space, actually. Figure 7–21 shows the model of Bob (with a texture applied for orientation) as defined in the previous code in 3D space.

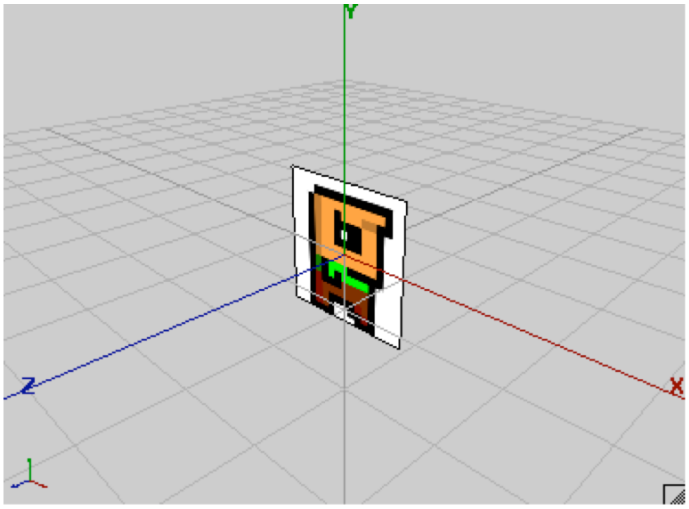

Figure 7–21. *Bob in 3D*

Since we haven't defined z-coordinates for Bob's vertices, he is embedded in the x-y plane of our 3D space (which is actually the model space, remember?). If we want to rotate Bob, we can do it around any axis we can think of: the x-, y-, or z-axis, or even a totally crazy axis like (0.75,0.75,0.75). However, for our 2D graphics programming needs, it makes sense to rotate Bob in the x-y plane. Hence, we'll use the positive z-axis as our rotation axis, which can be defined as (0,0,1). The rotation will be counterclockwise around the z-axis. A call to `glRotatef()`, like this, would cause the vertices of Bob's model to be rotated as shown in Figure 7–22:

```
gl.glRotatef(45, 0, 0, 1);
```

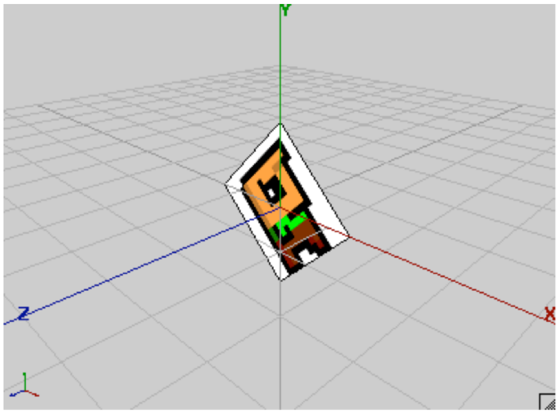

Figure 7–22. *Bob, rotated around the z-axis by 45 degrees*

Scaling

We can also scale Bob's model with `glScalef()`, like this:

```
glScalef(2, 0.5f, 1);
```

which, given Bob's original model pose, would result in the new orientation depicted in Figure 7–23.

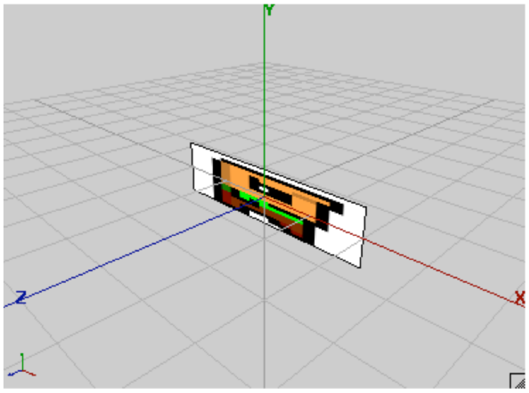

Figure 7–23. *Bob, scaled by a factor of 2 on the x-axis and a factor of 0.5 on the y-axis. Ouch.*

Combining Transformations

Now, we also discussed that we can combine the effect of multiple matrices by multiplying them together to form a new matrix. All the methods—`glTranslatef()`, `glScalef()`, `glRotatef()`, and `glOrthof()`—actually do just that. They multiply the current active matrix by the temporary matrix they create internally based on the parameters we pass to them. So let's combine the rotation and scaling of Bob:

```
gl.glRotatef(45, 0, 0, 1);
gl.glScalef(2, 0.5f, 1);
```

This would make Bob's model look like Figure 7–24 (remember, we are still in model space).

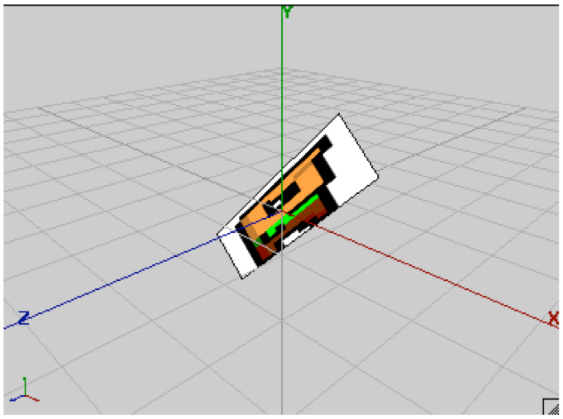

Figure 7–24. *Bob, first scaled and then rotated (still not looking happy)*

What would happen if we applied the transformations the other way around, like this:

```
gl.glScalef(2, 0.5, 0);
gl.glRotatef(45, 0, 0, 1)
```

Figure 7–25 gives you the answer.

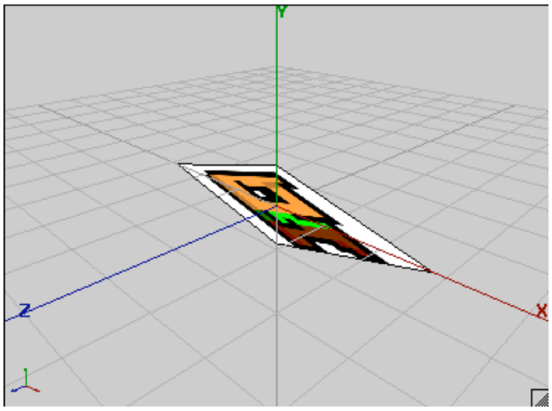

Figure 7–25. *Bob, first rotated, and then scaled*

Wow, this is not the Bob we used to know. What happened here? If you look at the code snippets, you'd actually expect Figure 7–24 to look like Figure 7–25, and Figure 7–25 to look like Figure 7–24. In the first snippet we apply the rotation first, and then scale Bob, right?

Wrong. The way OpenGL ES multiplies matrices with each other dictates the order in which the transformations the matrices encode are applied to a model. The last matrix we multiply the currently active matrix with will be the first that gets applied to the vertices. So if we want to scale, rotate, and translate Bob, in that exact order, we have to call the methods like this:

```
glTranslatef(bobs[i].x, bobs[i].y, 0);
glRotatef(45, 0, 0, 1);
glScalef(2, 0.5f, 1);
```

If we changed the loop in our `BobScreen.present()` method to the following code:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
for(int i = 0; i < NUM_BOBS; i++) {
    gl.glLoadIdentity();
    gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
    gl.glRotatef(45, 0, 0, 1);
    gl.glScalef(2, 0.5f, 0);
    bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
}
```

the output would look like Figure 7–25.

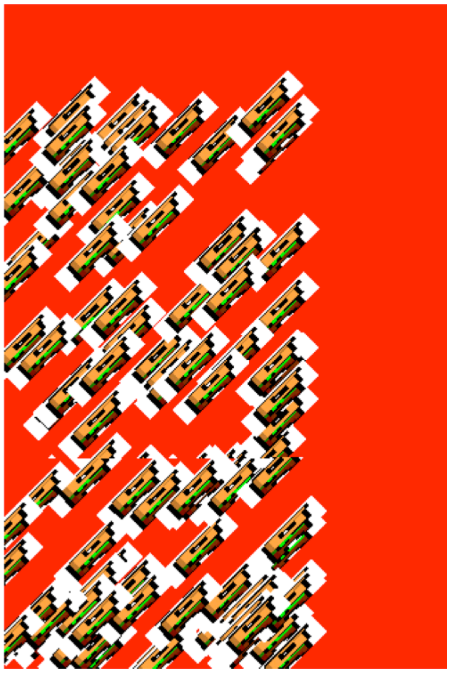

Figure 7–26. *A hundred Bobs, scaled, rotated, and translated (in that order) to their positions in world space*

I always mixed up the order of these matrix operations when I started out with OpenGL on the desktop. To remember how to do it correctly, I eventually arrived at a mnemonic device called it the LASFIA principle: last specified, first applied. (Yeah, my mnemonics aren't all that great huh?)

The easiest way to get comfortable with model-view transformations is to use them heavily. I suggest you take the `BobTest.java` source file, modify the inner loop for some time, and observe the effects. Note that you can specify as many transformations as you want for rendering each model. Add more rotations, translations, and scaling. Go crazy.

With this last example we basically know everything we need to know about OpenGL ES to write 2D games. Or do we?

Optimizing for Performance

When we run this example on a beefy second-generation device like a Droid or a Nexus One, everything will run smooth as silk. If we run it on a Hero, everything will start to stutter and look pretty unpleasant. But hey, didn't I say OpenGL ES was the silver bullet for fast graphics rendering? Well, yes it is. But only if we do things the way OpenGL ES wants us to do them.

Measuring Frame Rate

`BobTest` provides a perfect example to start with some optimizations. Before we can do that, though, we need a way to assess performance. Manual visual inspection ("doh, it looks like it stutters a little") is not precise enough. A better way to measure how fast our program performs is to count the number of frames we render per second. If you remember Chapter 3, we talked about something called the vertical synchronization, or vsync for short. This is enabled on all Android devices that are on the market so far, and limits the maximum frames per second (FPS) we can achieve to 60. We know our code is good enough when we run at that frame rate.

NOTE: While 60 FPS would be nice to have, in reality it is pretty hard to achieve such performance. Devices like the Nexus One and the Droid have a lot of pixels to fill, even if we're just clearing the screen. We'll be happy if our game renders the world at more than 30 FPS in general. More frames don't hurt, though.

Let's write a little helper class that counts the FPS and outputs that value periodically. Listing 7–14 shows the code of a class called `FPSCounter`.

Listing 7–14. *FPSCounter.java; Counting Frames and Logging Them to LogCat Each Second*

```
package com.badlogic.androidgames.framework.gl;

import android.util.Log;

public class FPSCounter {
    long startTime = System.nanoTime();
    int frames = 0;

    public void logFrame() {
        frames++;
        if(System.nanoTime() - startTime >= 1000000000) {
            Log.d("FPSCounter", "fps: " + frames);
            frames = 0;
            startTime = System.nanoTime();
        }
    }
}
```

We can put an instance of this class in our `BobScreen` class and call the `logFrame()` method once in the `BobScreen.present()` method. I just did this, and here is the output for a Hero (running Android 1.5), a Droid (running Android 2.2), and a Nexus One (running Android 2.2.1).

```
Hero:
12-10 03:27:05.230: DEBUG/FPSCounter(17883): fps: 22
12-10 03:27:06.250: DEBUG/FPSCounter(17883): fps: 22
12-10 03:27:06.820: DEBUG/dalvikvm(17883): GC freed 21818 objects / 524280 bytes in
132ms
12-10 03:27:07.270: DEBUG/FPSCounter(17883): fps: 20
12-10 03:27:08.290: DEBUG/FPSCounter(17883): fps: 23

Droid:
12-10 03:29:44.825: DEBUG/FPSCounter(8725): fps: 39
12-10 03:29:45.864: DEBUG/FPSCounter(8725): fps: 38
12-10 03:29:46.879: DEBUG/FPSCounter(8725): fps: 38
12-10 03:29:47.879: DEBUG/FPSCounter(8725): fps: 39
12-10 03:29:48.887: DEBUG/FPSCounter(8725): fps: 40

Nexus One:
12-10 03:28:05.923: DEBUG/FPSCounter(930): fps: 43
12-10 03:28:06.933: DEBUG/FPSCounter(930): fps: 43
12-10 03:28:07.943: DEBUG/FPSCounter(930): fps: 44
12-10 03:28:08.963: DEBUG/FPSCounter(930): fps: 44
12-10 03:28:09.973: DEBUG/FPSCounter(930): fps: 44
12-10 03:28:11.003: DEBUG/FPSCounter(930): fps: 43
12-10 03:28:12.013: DEBUG/FPSCounter(930): fps: 44
```

Upon first inspection we can see the following:

- The Hero is twice as slow as the Droid and the Nexus One.
- The Nexus One is slightly faster than the Droid.
- We generate garbage on the Hero in our process (17883).

Now, the last item on that list is somewhat puzzling. We run the same code on all three devices. And upon further inspection, we do not allocate any temporary objects in either the `present()` or `update()` method. So what's happening on the Hero?

The Curious Case of the Hero on Android 1.5

It turns out that there is a bug in Android 1.5. Well, it's not a bug, it's just some extremely sloppy programming. Remember that we use direct NIO buffers for our vertices and indices. These are actually memory blocks in native heap memory. Each time we call `glVertexPointer()`, `glColorPointer()`, or any other of the `glXXXPointer()` methods, OpenGL ES will try to fetch the native heap memory address of that buffer to look up the vertices to transfer the data to video RAM. The problem on Android 1.5 is that each time we request the memory address from a direct NIO buffer, it will generate a temporary object called `PlatformAddress`. Since we have a lot of calls to the `glXXXPointer()` and `glDrawElements()` methods (remember, the latter fetches the address from a direct `ShortBuffer`), Android allocates a metric ton of temporary `PlatformAdress` instances, and there's nothing we can do about it. (Well, there's actually a workaround, but for now we won't discuss it). Let's just accept the fact that using NIO buffers on Android 1.5 is horribly broken and move on.

What's Making My OpenGL ES Rendering So Slow?

That the Hero is slower than the second-generation devices is no big surprise. However, the PowerVR chip in the Droid is slightly faster than the Adreno chip in the Nexus One, so the preceding results are a little bit strange at first sight. On further inspection we can probably attribute the difference not to the GPU power but to the fact that we call many OpenGL ES methods each frame, which are costly Java Native Interface methods. This means that they actually call into C code, which costs more than calling a Java method on Dalvik. The Nexus One has a JIT compiler and can optimize a little bit there. So let's just assume that the difference stems from the JIT compiler (which is probably not entirely correct).

Now let's examine what's bad for OpenGL ES:

- Changing states a lot per frame (e.g., blending, enabling/disabling texture mapping, etc.)
- Changing matrices a lot per frame.
- Binding textures a lot per frame.
- Changing the vertex, color, and texture coordinate pointers a lot per frame.

It all boils down to changing state really. Why is this costly? GPUs work like an assembly line in a factory. While the front of the line processes new incoming pieces, the end of the line finishes off pieces already processed by previous stages of the line. Let's try it with a little car factory analogy.

The production line has a few states, such as the tools that are available to factory workers, the type of bolts that are used to assemble parts of the cars, the color the cars get painted with, and so on. Yes, real car factories have multiple assembly lines, but let's just pretend there's only one. Now, each stage of the line will be busy as long as we don't change any of the states. As soon as we change a single state, however, the line will stall until all the cars currently being assembled are finished off. Only then can we actually change the state and assemble cars with the new paint/bolts/whatever.

The key insight is that a call to `glDrawElements()` or `glDrawArrays()` is not immediately executed. Instead the command is put into a buffer that is processed asynchronously by the GPU. This means that the calls to the drawing methods will not block. It's therefore a bad idea to measure the time a call to `glDrawElements()` takes, as the actual work might be performed in the future. That's why we measure FPS instead. When the framebuffer is swapped (yes, we use double-buffering with OpenGL ES as well), OpenGL ES will make sure that all pending operations will be executed.

So translating the car factory analogy to OpenGL ES means the following. While new triangles enter the command buffer via a call to `glDrawElements()` or `glDrawArrays()`, the GPU pipeline might finish off the rendering of currently processed triangles from earlier calls to the render methods (e.g., a triangle can be currently processed in the rasterization state of the pipeline). This has the following implications:

- Changing the currently bound texture is expensive. Any triangles in the command buffer that have not been processed yet and that use the texture must be rendered first. The pipeline will stall.
- Changing the vertex, color, and texture coordinate pointers is expensive. Any triangles in the command buffer that haven't been rendered yet and use the old pointers must be rendered first. The pipeline will stall.
- Changing blending state is expensive. Any triangles in the command buffer that need/don't need blending and haven't been rendered yet must be rendered first. The pipeline will stall.
- Changing the model-view or projection matrix is expensive. Any triangles in the command buffer that haven't been processed yet and to which the old matrices should be applied must be rendered first. The pipeline will stall.

The quintessence of all this is *reduce your state changes*—all of them.

Removing Unnecessary State Changes

So let's look at the `present()` method of `BobTest` and see what we can change. Here's the snippet for reference (I added the `FPSCounter` in, and we also use `glRotatef()` and `glScalef()`):

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClearColor(1,0,0,1);
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, 320, 0, 480, 1, -1);

    gl.glEnable(GL10.GL_TEXTURE_2D);
    bobTexture.bind();

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        gl.glRotatef(45, 0, 0, 1);
        gl.glScalef(2, 0.5f, 1);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    fpsCounter.logFrame();
}
```

The first thing we could do is to move the calls to `glViewport()` and `glClearColor()`, as well as the method calls that set the projection matrix to the `BobScreen.resume()` method. The clear color will never change, the viewport and the projection matrix won't change either. Why not put the code to setup all persistent OpenGL states like the viewport or projection matrix in the constructor of `BobScreen`? Well, we need to battle context loss. All OpenGL ES state modifications we perform will get lost, and when our screen's `resume()` method is called, we know that the context has been recreated and is thus missing all the states we might have set before. We can also put the call the `glEnable()` and the texture-binding call into the `resume()` method. After all, we want texturing to be enabled all the time, and we also only want to use that single Bob texture. For good measure we also call `texture.reload()` in the `resume()` method so that our texture image data is also reloaded in the case of a context loss. So here are our modified `present()` and `resume()` methods:

```
@Override
public void resume() {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClearColor(1, 0, 0, 1);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, 320, 0, 480, 1, -1);

    bobTexture.reload();
    gl.glEnable(GL10.GL_TEXTURE_2D);
    bobTexture.bind();
}

@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        gl.glRotatef(45, 0, 0, 1);
        gl.glScalef(2, 0.5f, 0);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
    }

    fpsCounter.logFrame();
}
```

Running this "improved" version gives the following performance on the three devices:

```
Hero:
12-10 04:41:56.750: DEBUG/FPSCounter(467): fps: 23
12-10 04:41:57.770: DEBUG/FPSCounter(467): fps: 23
12-10 04:41:58.500: DEBUG/dalvikvm(467): GC freed 21821 objects / 524288 bytes in 133ms
12-10 04:41:58.790: DEBUG/FPSCounter(467): fps: 19
12-10 04:41:59.830: DEBUG/FPSCounter(467): fps: 23

Droid:
12-10 04:45:26.906: DEBUG/FPSCounter(9116): fps: 39
12-10 04:45:27.914: DEBUG/FPSCounter(9116): fps: 41
12-10 04:45:28.922: DEBUG/FPSCounter(9116): fps: 41
12-10 04:45:29.937: DEBUG/FPSCounter(9116): fps: 40

Nexus One:
12-10 04:37:46.097: DEBUG/FPSCounter(2168): fps: 43
12-10 04:37:47.127: DEBUG/FPSCounter(2168): fps: 45
12-10 04:37:48.147: DEBUG/FPSCounter(2168): fps: 44
12-10 04:37:49.157: DEBUG/FPSCounter(2168): fps: 44
12-10 04:37:50.167: DEBUG/FPSCounter(2168): fps: 44
```

So all the devices have already benefited a tiny bit by our optimizations. Of course, the effect is not exactly huge. This can be attributed to the fact that when we originally called all those methods at the beginning of the frame, there were no triangles in the pipeline yet.

Reducing Texture Size Means Fewer Pixels to Be Fetched

So what else could be changed? Something that is not all that obvious. Our `Bob` instances are 32×32 units in size. We use a projection plane that is 320×480 units in size. On a Hero, that will give us pixel-perfect rendering. On a Nexus One or a Droid, a single unit in our coordinate system would take up a little under a pixel. In any case our texture is actually 128×128 pixels in size. We don't need that much resolution, so let's resize the texture image `bobrgb888.png` to 32×32 pixels. We'll call the new image `bobrgb888-32x32.png`. Using this smaller texture, we get the following FPS for each device:

```
Hero:
12-10 04:48:03.940: DEBUG/FPSCounter(629): fps: 23
12-10 04:48:04.950: DEBUG/FPSCounter(629): fps: 23
12-10 04:48:05.860: DEBUG/dalvikvm(629): GC freed 21812 objects / 524256 bytes in 134ms
12-10 04:48:05.990: DEBUG/FPSCounter(629): fps: 21
12-10 04:48:07.030: DEBUG/FPSCounter(629): fps: 24

Droid:
12-10 04:51:11.601: DEBUG/FPSCounter(9191): fps: 56
12-10 04:51:12.609: DEBUG/FPSCounter(9191): fps: 56
12-10 04:51:13.625: DEBUG/FPSCounter(9191): fps: 55
12-10 04:51:14.641: DEBUG/FPSCounter(9191): fps: 55

Nexus One:
12-10 04:48:18.067: DEBUG/FPSCounter(2238): fps: 53
12-10 04:48:19.077: DEBUG/FPSCounter(2238): fps: 56
12-10 04:48:20.077: DEBUG/FPSCounter(2238): fps: 53
12-10 04:48:21.097: DEBUG/FPSCounter(2238): fps: 54
```

Wow, that makes a huge difference on the second-generation devices. It turns out that the GPUs of those devices hate nothing more than having to scan over a large amount of pixels. This is true for fetching texels from a texture as well as actually rendering triangles to the screen. The rate at which those GPUs can fetch texels and render pixels to the framebuffer is called the *fill rate*. All the second-generation GPUs are heavily fill-rate limited, so we should try to use textures that are as small as possible (or map our triangles only to a small portion of them), and not render extremely huge triangles to the screen. We should also look out for overlap: the fewer overlapping triangles, the better.

NOTE: Actually, overlap is not an extremely big problem with GPUs such as the PowerVR SGX 350 on the Droid. These GPUs have a special mechanism, called *tile-based deferred rendering*, that can eliminate a lot of that overlap under certain conditions. We should still care about pixels that will never be seen on the screen, though.

The Hero did only slightly benefit from the decrease in texture image size. So what could be the culprit here?

Reducing Calls to OpenGL ES/JNI Methods

The first suspects are the many OpenGL ES calls we issue per frame when we render the model for each Bob. First of all, we have four matrix operations per Bob. If we don't need rotation or scaling, we can bring that down to two calls. Here are the FPS numbers for each device when we only use `glLoadIdentity()` and `glTranslatef()` in the inner loop:

```
Hero:
12-10 04:57:49.610: DEBUG/FPSCounter(766): fps: 27
12-10 04:57:49.610: DEBUG/FPSCounter(766): fps: 27
12-10 04:57:50.650: DEBUG/FPSCounter(766): fps: 28
12-10 04:57:50.650: DEBUG/FPSCounter(766): fps: 28
12-10 04:57:51.530: DEBUG/dalvikvm(766): GC freed 22910 objects / 568904 bytes in 128ms

Droid:
12-10 05:08:38.604: DEBUG/FPSCounter(1702): fps: 56
12-10 05:08:39.620: DEBUG/FPSCounter(1702): fps: 57
12-10 05:08:40.628: DEBUG/FPSCounter(1702): fps: 58
12-10 05:08:41.644: DEBUG/FPSCounter(1702): fps: 57

Nexus One:
12-10 04:58:01.277: DEBUG/FPSCounter(2509): fps: 54
12-10 04:58:02.287: DEBUG/FPSCounter(2509): fps: 54
12-10 04:58:03.307: DEBUG/FPSCounter(2509): fps: 55
12-10 04:58:04.317: DEBUG/FPSCounter(2509): fps: 55
```

Well, it improved the performance on the Hero quite a bit, and the Droid and Nexus One also benefited a little from removing the two matrix operations. Of course, there's a little bit of cheating involved: if we need to rotate and scale our Bobs, there's no way around issuing those two additional calls. However, when all we do is 2D rendering, there's a neat little trick we can use that will get rid of all matrix operations (we'll look into this in the next chapter).

OpenGL ES is a C API provided to Java via a JNI wrapper. This means that any OpenGL ES method we call has to cross that JNI wrapper to call the actual C native function. This is somewhat costly on earlier Android versions, but has gotten better with more recent versions. As shown, the impact is not all that huge, especially if the actual operations take up more time than issuing the call itself.

The Concept of Binding Vertices

So is there anything else we can improve? Let's look at our current `present()` method one more time (with removed `glRotatef()` and `glScalef()`):

```
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
    }

    fpsCounter.logFrame();
}
```

That looks pretty much optimal, doesn't it? Well, in fact it is not optimal. First, we can also move the `gl.glMatrixMode()` call to the `resume()` method. But that won't have a huge impact on performance, as we already saw. The second thing that can be optimized is a little subtler.

We use the `Vertices` class to store and render the model of our Bobs. Remember the `Vertices.draw()` method? Here it is one more time:

```
public void draw(int primitiveType, int offset, int numVertices) {
    GL10 gl = glGraphics.getGL();

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    vertices.position(0);
    gl.glVertexPointer(2, GL10.GL_FLOAT, vertexSize, vertices);

    if(hasColor) {
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        vertices.position(2);
        gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
    }

    if(hasTexCoords) {
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        vertices.position(hasColor?6:2);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
    }
    if(indices!=null) {
        indices.position(offset);
        gl.glDrawElements(primitiveType, numVertices, GL10.GL_UNSIGNED_SHORT, indices);
    } else {
        gl.glDrawArrays(primitiveType, offset, numVertices);
    }

    if(hasTexCoords)
        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

    if(hasColor)
        gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
}
```

Now look at preceding the loop again. Notice something? For each Bob, we enable the same vertex attributes over and over again via `glEnableClientState()`. We actually only need to set those once, as each Bob uses the same model, which always uses the same vertex attributes. The next big problems are the calls to `glXXXPointer()` for each Bob. Since those pointers are also OpenGL ES states, we only need to set them once as well, as they will never change once set. So how can we fix that? Let's rewrite the `Vertices.draw()` method a little:

```
public void bind() {
    GL10 gl = glGraphics.getGL();

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    vertices.position(0);
    gl.glVertexPointer(2, GL10.GL_FLOAT, vertexSize, vertices);

    if(hasColor) {
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        vertices.position(2);
        gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
    }

    if(hasTexCoords) {
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        vertices.position(hasColor?6:2);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
    }
}

public void draw(int primitiveType, int offset, int numVertices) {
    GL10 gl = glGraphics.getGL();

    if(indices!=null) {
        indices.position(offset);
        gl.glDrawElements(primitiveType, numVertices, GL10.GL_UNSIGNED_SHORT, indices);
    } else {
        gl.glDrawArrays(primitiveType, offset, numVertices);
    }
}

public void unbind() {
    GL10 gl = glGraphics.getGL();
    if(hasTexCoords)
        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

    if(hasColor)
        gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
}
```

Can you see what we've done here? We can treat our vertices and all those pointers just like we treat a texture. We "bind" the vertex pointers via a single call to `Vertices.bind()`. From this point onward every `Vertices.draw()` call will work with those "bound" vertices, just like the draw call will also use the currently bound texture. Once we are done rendering stuff with that `Vertices` instance, we call `Vertices.unbind()` to disable any vertex attributes that another `Vertices` instance might not need. Keeping our OpenGL ES state clean is a good thing. Here's how our `present()` method looks now (I moved the `glMatrixMode(GL10.GL_MODELVIEW)` call to `resume()` as well):

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    bobModel.bind();
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    bobModel.unbind();

    fpsCounter.logFrame();
}
```

This effectively calls the `glXXXPointer()` and `glEnableClientState()` methods only once per frame. We thus save nearly 100 × 6 calls to OpenGL ES. That should have a huge impact on performance, right? Right.

```
Hero:
12-10 05:16:59.710: DEBUG/FPSCounter(865): fps: 51
12-10 05:17:00.720: DEBUG/FPSCounter(865): fps: 46
12-10 05:17:01.720: DEBUG/FPSCounter(865): fps: 47
12-10 05:17:02.610: DEBUG/dalvikvm(865): GC freed 21815 objects / 524272 bytes in 131ms
12-10 05:17:02.740: DEBUG/FPSCounter(865): fps: 44
12-10 05:17:03.750: DEBUG/FPSCounter(865): fps: 50

Droid:
12-10 05:22:27.519: DEBUG/FPSCounter(2040): fps: 57
12-10 05:22:28.519: DEBUG/FPSCounter(2040): fps: 57
12-10 05:22:29.526: DEBUG/FPSCounter(2040): fps: 57
12-10 05:22:30.526: DEBUG/FPSCounter(2040): fps: 55

Nexus One:
12-10 05:18:31.915: DEBUG/FPSCounter(2509): fps: 56
12-10 05:18:32.935: DEBUG/FPSCounter(2509): fps: 56
12-10 05:18:33.935: DEBUG/FPSCounter(2509): fps: 55
12-10 05:18:34.965: DEBUG/FPSCounter(2509): fps: 54
```

All three devices are nearly on par now. The Droid performs the best, followed by the Nexus One. Our little Hero performs great as well. We are up to 50 FPS from 22 FPS in the nonoptimized case. That's an increase in performance by 100 percent. We can be proud of ourselves. Our optimized Bob test is pretty much optimal.

Our new bindable `Vertices` class has of course a few restrictions now:

- We can only set the vertex and index data when the `Vertices` instance is not bound, as the upload of that information is performed in `Vertices.bind()`.
- We can't bind two `Vertices` instances at once. This means that we can only render with a single `Vertices` instance at any point in time. That's usually not a big problem, though, and given the impressive increase in performance, we will live with it.

In Closing

There's one more optimization we can apply that is suited for 2D graphics programming with flat geometry, such as with rectangles. We'll look into that in the next chapter. The keyword to search for is *batching*, which means reducing the number of `glDrawElements()/glDrawArrays()` calls. An equivalent for 3D graphics exists as well, called *instancing*, but that's not possible to do with OpenGL ES 1.x.

I want to mention two more things before we close this chapter. First of all, when you run either `BobText` or `OptimizedBobTest` (which contains the superoptimized code we just developed), notice that the Bobs wobble around the screen somewhat. This is due to the fact that their positions are passed to `glTranslatef()` as floats. The problem with pixel-perfect rendering is that OpenGL ES is really sensitive to vertex positions with fractional parts in their coordinates. We can't really work around this problem; the effect will be less pronounced or even nonexistent in a real game, as we'll see when we implement our next game. We can hide the effect to some extent by using a more diverse background, among other things.

The second thing I want to point out is how we interpret the FPS measurements. As you can see from the preceding output, the FPS fluctuate a little. This can be attributed to background processes that run alongside our application. We will never have all of the system resources for our game, so we have to learn to live with this issue. When you are optimizing your program, don't fake the environment by killing all background processes. Run the application on a phone that is in a normal state, as you'd use it yourself during the day. This will reflect the same experience that a user will have.

Our nice achievement concludes this chapter. As a word of warning, only start optimizing your rendering code after you have it working, and only then after you actually have a performance problem. Premature optimization is often a cause for having to rewrite your entire rendering code, as it may become unmaintainable.

Summary

OpenGL ES is a huge beast. We managed to boil all that down to a size that makes it easily usable for our game programming needs. We discussed what OpenGL ES is (a lean, mean triangle-rendering machine) and how it works. We then explored how to make use of OpenGL ES functionality by specifying vertices, how to create textures, and how to use states such as blending for some nice effects. We also looked a little bit into projections and how they are connected to matrices. While we didn't discuss what a matrix does internally, we explored how to use them to rotate, scale, and translate reusable models from model space to world space. When we use OpenGL ES for 3D programming later, you'll notice that you've already learned 90 percent of what you need to know. All we'll do is change the projection and add a z-coordinate to our vertices. (Well, there are a few more things, but on a high level that's actually it). Before that, though, we'll write a nice 2D game with OpenGL ES. In the next chapter, you'll get to know some of the 2D programming techniques we might need for that.