Learn how to solve real-world game development problems

Android Game Recipes

A Problem-Solution Approach

Jerome DiMarzio

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Welcome to *Android Game Recipes*. This book is specifically written to help you with many of the common problems that you may have encountered while in the process of creating a game for the Android platform. Android game development can be a fun, enjoyable, and rewarding process; but it is not without its pitfalls. There always seem to be problems that come up during the development process that are difficult to find solutions to. My hope is that this book can provide you with those solutions.

I have created multiple games for Android, and have encountered many problems while doing so. My experiences, and the solutions I have found, are compiled into 17 chapters, each separated by major topic. Outlined as follows are the chapters in this book and a quick summary of what will be covered in each.

Chapter 1: Getting Started. This chapter covers the skills and software that you need to make the most of this book. Chapter 1 also includes a quick introduction to Android gaming and OpenGL ES versions 1, and 2 / 3.

Chapter 2: Loading an Image. There are different situations that may call for an image to be loaded either with or without OpenGL ES. If you are creating a splash screen you may not want to use OpenGL. The recipes in this chapter help you create a splash screen without using OpenGL.

Chapter 3: The Splash Screen. Here you'll find solutions to common problems in creating splash screens. These problems can include loading the screen image, transitions between multiple images, and loading the game after the splash screen.

Chapter 4: The Menu Screen. In this chapter, you'll learn solutions to common menu screen problems, such as creating buttons, loading options, locking screen rotation, and detecting screen resolution.

Chapter 5: Reading Player Input. The recipes in this chapter solve problems related to reading player input during the game, such as touch screen input, multi-touch, and gestures.

Chapter 6: Loading a SpriteSheet. Being able to load a spritesheet is essential in creating a game. This chapter contains solutions for loading spritesheet images, animating multiple spritesheet images, and storing spritesheets.

Chapter 7: Scrolling a Background. Key to realism, Chapter 7 helps you solve issues related to scrolling a background image on the screen, such as loading the image to the screen and changing the scroll speed.

Chapter 8: Scrolling Multiple Backgrounds. In this chapter you'll encounter recipes for how to scroll multiple background images to give the appearance of a foreground, middleground, and distance.

Chapter 9: Syncing a Background to Character Movement. In this chapter you'll find solutions for changing the direction and speed of the background movement in relationship to the movement of the character.

Chapter 10: Building a Level Using Tiles. You'll learn how to create levels for side-scrolling and platform games from graphic tiles. Using repeatable tiles is a tried and tested way to create game levels.

Chapter 11: Moving a Character. This covers problems that could arise when trying to animate a playable character, everything from walking, to running, to jumping and fighting.

Chapter 12: Moving an Enemy. Like Chapter 11, this chapter also discusses moving characters across the screen. However, this chapter focuses more on the specific problems encountered when creating AI based (non-playable) characters, such as moving on a predetermined path.

Chapter 13: Moving a Character with Obstacles. Most games do not have a smooth surface for which to play. That is, many game levels contain obstacles and inclines that the player needs to navigate. In this chapter you'll encounter recipes for how to let your playable character navigate these obstacles.

Chapter 14: Firing Weapons. In this chapter you'll learn how to fire or throw weapons. There are specific problems that need to be addressed when animating projectiles that include animation and the calculation of trajectories.

Chapter 15: Collision Detection. A key topic in game development, this covers the complex issue of collision detection. You'll find recipes for how to detect and react to interactions between onscreen (in-game) objects.

Chapter 16: Keeping Score. One way for a player to track their process in a game is through a score. The solutions in chapter 16 help you compile a gamer's score and write that score to the screen.

Chapter 17: Keeping Time. Some games are time based, or contain time based levels and challenges. Chapter 17 covers solutions for how to implement and track the expiration of time for marshaling in-game action.

Chapter 1

Getting Started

Welcome to *Android Game Recipes*. This book is very much like a cookbook. It is designed to tackle specific, common problems that could arise while you develop a game for the Android platform. Solutions are provided in a well-tested, thought-out approach that is easy to follow and easy to adapt to multiple situations.

Let's say you know the theory behind what goes into chicken soup, but you are unsure how to turn some chicken and vegetables into soup. Consulting a standard, kitchen cookbook would give you a step-by-step recipe to create the soup. In much the same way, you will be able to use *Android Game Recipes* to find out exactly how to code specific scenarios in a game—from creating a splash screen to using collision detection when destroying an enemy.

Before you move on to the recipes, it's important to establish the proper framework to get the most out of them. In this chapter, we will discuss what skills and tools you will need to get the most out of this book.

What You Will Need

Game programming, as a discipline, is complex and can take years to master. However, the basic concepts of game programming are actually relatively simple to learn and can be reused in many situations. The amount of time that you put into your games and your code will ultimately determine how successful you and your games are. Everyone runs into that one problem when coding which, no matter how long you scratch your head, or how many times you search on Google, you just cannot get an exact solution for. This book is designed to be your solution to many of these problems.

Skills and Experience

This book is not aimed at beginners or people who have no game development experience. You will not learn how to develop an entire game from scratch by reading this book. This is not to say that you need to be a professional game developer to use this book. To the contrary, it is assumed that

by reading this book you are most likely a casual game developer; you are likely to be someone who might have tried to create a game or two (possibly even for Android) and has run into a problem converting some of your development knowledge to the Android platform.

This book is focused on helping you through specific problems or scenarios. Therefore, you should have at least a working knowledge of game development, and at least a basic knowledge of Android-specific development. Neither topic will be covered from the perspective of a "from scratch" primer.

Since Android is developed in Java, you should also possess a good, working knowledge of Java development. There will be no tutorials on how Java works, and it may be implied during certain scenarios that you know the meaning behind the structure of Java.

It is possible however, that you may have some game development experience on another platform—such as Windows—and possibly even some business-level Java experience, and never have used OpenGL ES. Most of the time, developing a game for Android will require use of OpenGL ES. For this reason, the second part of this chapter is dedicated to introducing you to OpenGL ES and explaining why it is important to Android. If you already have experience with OpenGL ES, feel free to skip that part of this chapter, "OpenGL ES at a Glance."

In short, if you have a passion for game development and a passion for Android, but are running into some problems in your development, this book is for you. Whether you have already started to develop a game and are running into problems, or you are in the beginning stages of your development and are unsure what to do next, *Android Games Development Recipes* will guide you through the most common roadblocks and issues.

Software Versions

At this point, you are probably ready to dive right into finding solutions for your Android game scenarios. So what tools do you need to begin your journey?

This book is geared toward Android 4.1 and 4.2 Jelly Bean. If you are not working in Jelly Bean, it is recommended that you upgrade your SDK at `http://developer.android.com/sdk/`. However, the examples should also work on Android 4.0 Ice Cream Sandwich. There are many resources to help you download and install the SDK (and the corresponding Java components that you might need) if you need help doing so; however, this book will not cover installing the SDK.

You will also be using the Kepler version of Eclipse. One of the great features of Eclipse is that it will support multiple versions of Android SDKs. Therefore, you can quickly test your code in Jelly Bean, Ice Cream Sandwich, or even Gingerbread if needed. While you can use almost any Java IDE or text editor to write Android code, I prefer Eclipse because of features such as this and the well-crafted plug-ins that tightly integrate to many of the more tedious manual operations of compiling and debugging Android code. After all, Eclipse is the official Android development IDE recommended by Google, the creator of Android.

If you do not already have Eclipse Kepler, and want to give it a try, it is a free download from `http://eclipse.org`.

This book will not dive into the download or setup of Eclipse. There are many resources, including those on Eclipse's own site and the Android Developer's Forum, that can help you set up your environment should you require assistance.

> **Tip** If you have never installed Eclipse or a similar IDE, follow the installation directions carefully. The last thing you want is an incorrectly installed IDE impeding your ability to write great games.

In the next section, we will explore one of the most used tools in creating games on the Android platform, OpenGL ES.

OpenGL ES at a Glance

OpenGL ES, or OpenGL for Embedded Systems, is an open source graphics API that is packaged with the Android SDK. While there is limited support for working with graphics using core Android calls, it would be extremely difficult—if not impossible—to create an entire game without using OpenGL ES. Core Android graphics calls are slow and clunky, and with few exceptions, should not be used for gaming. This is where OpenGL ES comes in.

OpenGL ES has been included with Android, in one form or another, since the very beginning of the platform. In earlier versions of Android, the implementation of OpenGL ES was a limited version of OpenGL ES 1. As Android grew, and versions of Android matured, more feature-rich implementations of OpenGL ES were added. With Android version Jelly Bean, developers have access to OpenGL ES 2 for game development.

So what exactly does OpenGL ES do for you, and how does it do it? Let's find out.

How OpenGL ES Works with Android

Open GL ES communicates with the graphic hardware in a much more direct manner than a core Android call. This means that you are sending data directly to the hardware that is responsible for processing it. A core Android call would have to go through the core Android processes, threads, and interpreter before getting to the graphics hardware. Games written for the Android platform can only achieve an acceptable level of speed and playability by communicating directly with the GPU (Graphics Processing Unit).

Current versions of Android have the ability to use either OpenGL ES 1 or OpenGL ES 2 / 3 calls. There is a big difference between the two versions, and which one you use will play a role in determining who can run your game, and who will not be able to.

> **Note** All of the examples in this book that include OpenGL ES code are given in both OpenGL ES version 1 and OpenGL ES version 2 / 3.

OpenGL ES facilitates this interaction between your game and the graphics hardware in one of two different ways. The type of GPU employed in the Android device running your game will determine which version of OpenGL ES you use, thus how OpenGL will interact with the hardware. There are two major kinds of graphics hardware in the market, and because they are very different, two different versions of OpenGL ES are required to interact with them.

The two different types of hardware are those with a fixed-function pipeline, and those with shaders. The next few sections quickly review OpenGL ES and fixed-function pipelines, and OpenGL ES and shaders. Keep in mind, OpenGL ES version 1 runs on fixed-function piplelines, while OpenGL ES 2/3 runs on shaders.

Fixed-Function Pipelines

Older devices will have hardware that employs a fixed-function pipeline. In these older GPUs, there was specific dedicated hardware for perform functions. Functions, such as transformations, were performed by dedicated parts of the GPU that you, as a developer, had little to no control over. This means that you would simply hand your vertices to the GPU, tell it to transform the vertices, and that's it.

An example of a transformation can be when you have a set of vertices representing a cube, and you want to move that cube from one location to another. This would be accomplished by putting the vertices into the fixed-function pipeline, and then telling the hardware to perform a transformation on those vertices. The hardware would then do the matrix math for you and determine the placement of the final cube.

In the following code, you will see a very simplified version of what you would do in a fixed-function pipeline. The vertices `myVertices` are sent into the pipeline. The `glTranslatef()` is then used to translate the vertices to new positions. The ensuing matrix math is done for you in the GPU.

```
private float myVertices[] = {
0.0f, 0.0f, 0.0f,
   1.0f, 0.0f, 0.0f,
   1.0f, 1.0f, 0.0f,
   0.0f, 1.0f, 0.0f,
};

//Other OpenGL and game stuff//

gl.glMatrixMode(GL10.GL_MODELVIEW)
gl.glLoadIdentity();
gl.glTranslatef(0f, 1f, 0f);
```

The advantage of this was that in using dedicated hardware, the function could be performed very quickly. Hardware can perform functions at very fast rates, and dedicated hardware—or hardware that has a very limited function set—can perform functions even faster.

The disadvantage to this fixed-function pipeline approach is that hardware cannot be changed or reconfigured like software can. This limits the usefulness of the hardware moving forward. Also, specialized hardware can only perform functions on one queue item at a time. This means that the pipeline can often be slowed down if there are a great amount of items waiting in the queue to be processed.

Newer devices, on the other hand, have GPUs that use shaders. A shader is still a specialized piece of hardware, but it is much more flexible than its fixed-function predecessor. OpenGL ES works with shaders by using a programming language called GLSL or OpenGL Shading Language to perform any number of programmable tasks.

Shaders

A shader is a software program, written in a shader language, that performs all of the functionality that used to be handled by the fixed-function hardware. OpenGL ES 2 / 3 works with two different types of shaders: vertex shaders and fragment shaders.

Vertex Shaders

A vertex shader performs functions on vertices, such as transforming the color, position, and texture of the vertex. The shader will run on every vertex passed into it. This means that if you have a shape made from 256 vertices, the vertex shader will run on each one of them.

Vertices can be small or large. However, in all cases, vertices will consist of many pixels. The vertex shader will work on all of the pixels in a single vertex the same way. All of the pixels within a single vertex are treated as a single entity. When the vertex shader is finished, it passes the vertex downstream to the rasterizer, and then on to the fragment shader.

Following is a basic vertex shader:

```
private final String vertexShaderCode =
        "uniform mat4 uMVPMatrix;" +
        "attribute vec4 vPosition;" +
        "attribute vec2 TexCoordIn;" +
        "varying vec2 TexCoordOut;" +
        "void main() {" +
        "  gl_Position = uMVPMatrix * vPosition;" +
        "  TexCoordOut = TexCoordIn;" +
        "}";
```

Fragment Shaders

Whereas vertex shaders process data for an entire vertex, fragment shaders—sometimes known as pixel shaders—work on each pixel. The fragment shader will make computations for lighting, shading, fog, color, and other things that would affect single pixels within a vertex. Processes for gradients and lighting are performed on the pixel level because they can be applied differently across a vertex.

Following is a basic fragment shader:

```
    private final String fragmentShaderCode =
        "precision mediump float;" +
        "uniform vec4 vColor;" +
        "uniform sampler2D TexCoordIn;" +
        "varying vec2 TexCoordOut;" +
        "void main() {" +
        "  gl_FragColor = texture2D(TexCoordIn, TexCoordOut);" +
        "}";
```

> **Note** There are other types of shaders including Tessellation shaders and Geometry shaders. These can be optional and are handled within the hardware. You will have little to no awareness into their operation.

Most Android devices now can handle a combination of OpenGL ES 1 and OpenGL ES 2 calls. Some developers, if they are uncomfortable with programming shaders, will continue to use fixed-function pipeline calls for the viewport and other dynamics. Be aware that as OpenGL progresses, compatibility with the fixed-function pipeline calls of OpenGL ES is being phased out. There will be a time in the very near future when you will be forced to use only shaders within OpenGL ES. Therefore, if you are at an early point in your career with OpenGL ES, I would suggest making an earnest effort to use shaders whenever possible.

How Games Work

When developing a game or a game loop, the code needs to be executed in a certain order, at certain times. Knowing this execution flow is crucial in understanding how your code should be set up.

The following sections will outline a basic game flow or game loop.

A Basic Game Loop

At the core of every video game is the game engine, and part of that game engine is the game loop. As the name suggests, the game engine is the code that powers the game. Every game, regardless of the type of game—whether it is an RPG, a first-person shooter, a platformer, or even an RTS—requires a fully featured game engine to run.

The game engine typically runs on its own thread, giving it as many resources as possible. All of the tasks that a game needs to run, from graphics to sound, are taken care of in the game engine.

> **Note** The engine of any one game is purposely built to be generic. This allows it to be used and reused in multiple situations, possibly for different games.

One very popular multipurpose game engine is the Unreal engine. The Unreal engine, first developed around 1998 by Epic for its first-person shooter, Unreal, has been used in hundreds of games. The Unreal engine is easily adaptable and works with a variety of game types, not just first-person shooters. This generic structure and flexibility make the Unreal engine popular with not only professions but casual developers as well.

Chances are, in your game development, you might have used a third-party game engine. There are many free and fee-based ones available for Android. This book will be of far greater help to you, though, if you are looking to build your own game engine.

Many of the processes in third-party game engines become obfuscated, and you might not have access to the debugging capability or you might not be able to modify the code within the engine.

When you have a problem, you will generally have to turn to the company that developed the engine, and it could take time for the original developer to fix it—if they even fix it at all. This can be a major drawback if you are thinking about using a third-party game engine.

There is no substitute for the experience of building your own game engine. This book assumes that you are doing just that. Many of the problems that will be tackled in the rest of this book assume you are attempting to write a game engine on Android and are running into some common problems.

So what exactly does the game engine do? The game engine handles all of the grunt work of the game execution, anything from playing the sound effects and background music to rendering graphics onto the screen. The following is a partial list of the functions that a typical game engine will perform.

- Graphics rendering
- Animation
- Sound
- Collision detection
- Artificial intelligence (AI)
- Physics (non-collision)
- Threading and memory management
- Networking
- Command interpreter (IO)

At the core of the game engine is the game loop. While the engine can handle anything from setting up one-time vertices buffers and retrieving images, the game loop serves up the actual code execution of the game.

All games are executed in a code loop. The faster this loop can execute, the better the game will run, the quicker it will react to the player, and the smoother the action will appear on the screen. All of the code necessary to build drawing on the screen, move the game objects, tally the score, detect the collisions, and validate or invalidate items is executed within the game loop.

A game loop is exactly that, a group of code that is executed on a continuous loop. The loop is started when the game begins, and does not stop executing—with some exceptions—until the game is stopped. Let's take a look at all of the things a game loop can be expected to do on every one of its iterations. A typical game loop can do the following:

- Interpret the commands of an input device
- Track the characters and/or the background to make sure none move where they should not be able to move to
- Test for collisions between objects
- Move the background as needed
- Draw a background

- Draw any number of stationary items
- Calculate the physics of any mobile objects
- Move any weapons/bullets/items that have been repositioned
- Draw weapons/bullets/items
- Move the characters independently
- Draw the characters
- Play sound effects
- Spin off threads for continuous background music
- Track the player's score
- Track and manage networked or multiple players

This is not be a comprehensive list, but it is a fairly good list of all of the things expected to be done within the game loop.

It is very important to refine and optimize all of your game code. The more optimized you can make your code in the game loop, the faster it will execute all of the calls it needs to make, thus giving you the best possible gaming experience. In the next section, we will take a look at how Android, as a platform, handles game engines and game loops.

Android and Game Engines

Android is packaged with a powerful, fully featured graphics API called OpenGL ES. But is OpenGL ES absolutely necessary for game development? Rather than go through the trouble of learning a fairly low-level API, such as OpenGL ES, can you just write a game with core Android API calls?

The short answer is that for a game to run efficiently, it cannot rely on the core Android API calls to do this kind of heavy duty work. Yes, most Android does have core calls that could take care of every item on this list. However, the rendering, sound, and memory systems of Android are built for generic tasks and adapt to any number of unpredictable uses, without specializing in any one. Unpredictability means one thing: overhead. The core Android API calls that could take care of the jobs needed to run a game come with a lot of extraneous code. This is acceptable if you are writing business applications, but not if you are writing games. Overhead adds slowness to your code, and games require something with a little more power.

For a game to run smoothly and quickly, the code will need to bypass the overhead that is inherent in core Android API calls; that is, a game should communicate directly with the graphics hardware to perform graphics function, communicate directly with the sound card to play sound effects, and so on. If you were to use the standard memory, graphics, and sound systems that are available to you through core Android API, your game could be threaded with all of the other Android applications that are running on the system. This would make for a choppy looking game that would run very slowly.

For this reason, game engines and game loops are almost always coded in low-level languages or specific API, such as OpenGL ES. As we will touch on in Chapter 2, low-level languages offer a more direct path to the hardware of the system. A game engine needs to be able to take code and commands from the engine and pass them directly to the hardware. This allows the game to run quickly and with all of the control that it needs to be able to provide a rewarding experience.

Summary

In this chapter, we covered what tools you will need to get the most out of this book. Android version Jelly Bean, Eclipse Kepler, and some basic Java and/or game development experience will help you throughout the remainder of this book. We also covered the differences between OpenGL ES versions 1 and 2 / 3, and the difference between fixed pipelines and shaders.

In the next few chapters, we will begin to look at some of the problems in a typical game engine. More specifically, we will look at the problems that could occur with the different ways to load an image. There are many different image formats and a handful of different ways to load these images and display them to the screen. Chances are, if you have tried, you have run into some pretty unexpected results.

Chapter 2

Loading an Image

It should go without saying that if you plan on developing a game, casual or otherwise, you need to work with images. Everything from the backgrounds and characters to the menus and text is made up of images. Android can use different methods to serve up these images to the screen. This chapter will help to solve any problems that you have had retrieving, storing, and serving images within Android.

There are two distinct ways to serve up an image in Android, and each has its place in game development. The first way to serve up an image in Android is to use core Android methods—or those methods that do not involve the direct use of OpenGL ES. These core methods require little to no code to use, but they are slow and definitely not flexible enough to be used in the main, action-oriented parts of the game.

The second way to serve up images within Android is to use OpenGL ES. OpenGL ES is fast, flexible, and perfect for use within a game; however, it requires substantially more code than the core Android methods do. We will look at both of these in this chapter.

So when would you use one method over the other?

Images loaded using the core Android methods are perfect for splash screens, title screens, and even menus. Given the architecture of Android activities, it is very easy to create an activity using core Android methods, that contains a menu system for the game. Menus can include items that are easier accomplished before launching your game thread, such as checking scores, visiting an online shop, or viewing preloaded level information. The menu can then be used to launch the main game thread when the player chooses to enter the game. Once in the main game thread, OpenGL ES can take over the duties of working with the more graphic-intense gameplay. The solutions in this chapter will help you work around many common problems loading images in both OpenGL ES and using core Android methods.

2.1 Loading an Image Using Core Android Methods

Problem

There are times in a game when you might not need to use OpenGL ES for displaying images; for example, the title and menu screens. However, after you have decided to use either core Android methods or OpenGL ES, how do you store the images in your project so that Android can access them?

Solution

Image files are stored in the `res` folder prior to being used within Android. The `res` folder—or resource folder—is where all of the resources for your Android project are stored. There is a set of subdirectories for the `res` folder named `drawable*`. All of your images should be placed in a `drawable` folder. The Android ImageView node is then used to display these images to the screen. This is a perfect solution for game splash screens or any part of your game that displays an image before the actual gameplay starts.

How It Works

One of the good things about this solution is that it can be accomplished with no manual coding whatsoever. Some drag-and-drop action will set this solution up for you in an instant. Since this solution has two parts (storing and displaying images), let's take a look at each part separately.

Storing Imagesin Android

The first part of the problem is where you store images for use within Android. All of the resource files that you use in your Android projects are kept in a project directory named `res`. If you open your project, and expand the file system under the Project Explorer, you will see a root level folder named `res`; this is where all of your in-app resources, such as strings and images, are stored.

> **Note** If you are using Eclipse (the latest version as of the writing of this book is Juno) then you will see the `res` folder in the Package Explorer. However, if you are using a different IDE, or no IDE at all, then locate the file exploring equivalent to see the `res` folder.

If you are using an IDE, open the `res` folder and you should find a number of subfolders. Some of these subfolders should start with the word `drawable-`. All of the subfolders that are meant for storing images within your app will start with this word. You will also notice a notation at the end of the name of each folder, from `-ldpi` to `-xhdpi`. What does this mean?

Android supports a number of different screen sizes and pixel densities. Because you might want to have different resolution images for different screen sizes or pixel densities, Android provides different subfolders for these images. The notation in the folder name indicates screen size from small (`drawable-small`)to extra large(`drawable-xlarge`), and indicates pixel densities from low density(`drawable-ldpi`) to extra-high density(`drawable-xhdpi`).

> **Tip** If you do not care about the pixel density of the screen used to display your images, then you can put all of your files in the default `drawable` folder. If your IDE did not create this folder by default, feel free to add it. Android will look here when you have not specified a pixel density to use.

The image that we will use in this example is the splash screen to our fictitious game Super Bandit Guy, as shown in Figure 2-1.

Figure 2-1. *Super Bandit Guy splash screen image*

Simply drag and drop this image from your working folder, wherever that may be, to the correct `drawable` dpi folder, as shown in Figure 2-2. In this case, I used the `drawable-xhdpi` to test on a tablet.

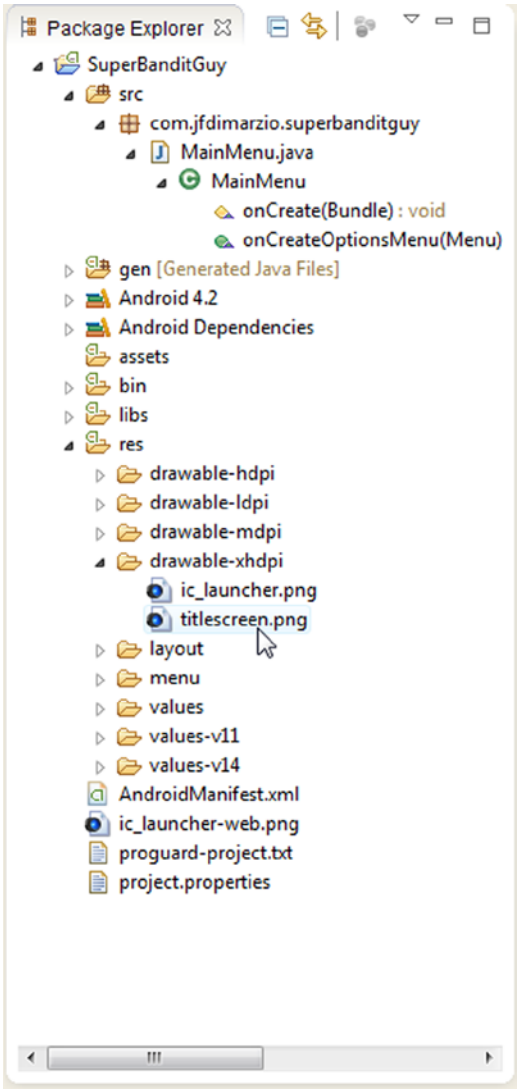

Figure 2-2. *Dragging an image to the* `res/drawable-xhdpi` *folder*

That is all there is to getting the image into Android.

> **Caution** All image files names must begin with a lowercase letter to be used in Android.

Loading and Displaying Images

The image is now ready to use. To display this image to the screen, you need to create an ImageView.

> **Note** Again, if you are using Eclipse, a generic layout should have been created for you just for this purpose. If you are not using Eclipse, please follow your IDE's instructions for creating a *main screen layout*.

Expand the layout root folder, and open the `activity_main_menu.xml` file. With the layout open, expand the Images & Media palette and locate the ImageView, as shown in Figure 2-3.

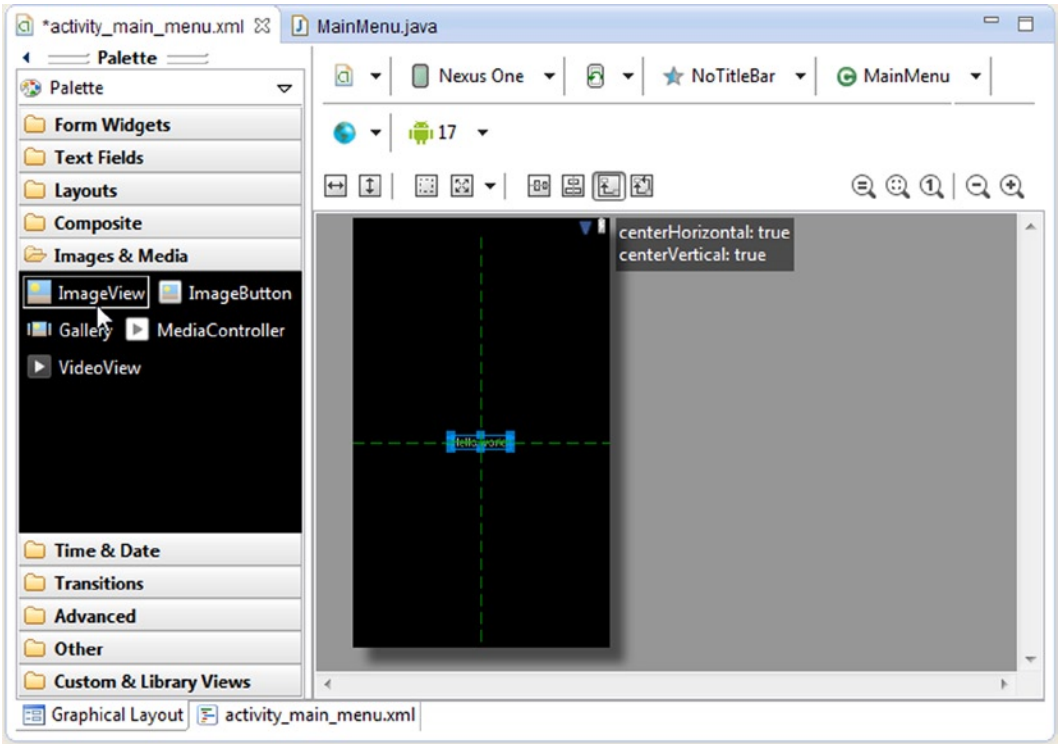

***Figure 2-3.** Locating the ImageView*

Now drag the image from the palette to the layout in the work area. At the top of the working area (again reference Figure 2-3), you will see a row of menu icons. Selecting the state menu icon will allow you to change the orientation of the screen layout from portrait to landscape. I have seen games played in either orientation; however, for this example, Super Bandit Guy is played in landscape. Therefore, a change in orientation will be noticeable in future screen shots.With the ImageView added to your layout, expand the ImageView properties and select the Src property. Clicking on the ellipsis next to the Src property will bring up a list of drawable resources.

Select the correct image, as shown in Figure 2-4.

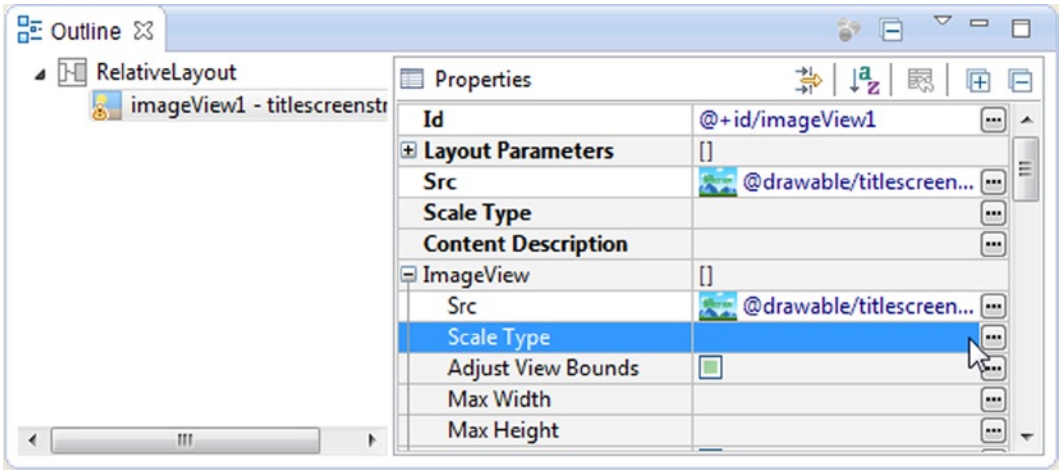

Figure 2-4. Selecting the correct image using the ImageView properties

Compile and run your project. The result should appear as shown in Figure 2-5.

Figure 2-5. Displaying the splash image

There is one piece of business that you might want to take care of before calling this one finished. Notice in Figure 2-5 that there is an action bar menu above the image. This is added by default (in Android version 3.0 and higher) in some IDEs depending on the Android theme that is selected when creating your project. Getting rid of this action bar is easy.

Returning to the Project Explorer, within the `res` folder, you should be able to locate a folder named `values`. Inside this folder is a file named `styles.xml`. Add the following line to the `styles.xml` file, between the style tags of the style that your app is using.

```
<item name="android:windowActionBar">false</item>
```

2.2 Loading an Image Using OpenGL ES

In this recipe, I present two problems and two solutions. You'll first correct ImageView image calls to run properly in a game. Then you'll see how to ensure OpenGL ES displays the correct image when using an Android device.

Problem 1

ImageView image calls are too slow for use in a game.

Solution 1

Use OpenGL ES to write your images to the screen. You must create an OpenGL ES renderer, a `GLSurfaceView`, and a set of vertices and textures. Although this solution might sound like a lot of work, you will only need to do much of the work once, and then you can reuse the same classes throughout your project.

That is, the renderer and the `GLSurfaceView` need to be created only once for your game. They are reused over and over again. The only parts of the solution that you will need to re-create for every image you want to display are the vertices and textures that define the image.

How It Works

We are going to break this solution up into three parts: creating the vertices and textures, creating the renderer, and finally creating the `GLSurfaceView`. Let's start with creating the vertices and textures.

Create Vertices and Textures

This is the most complicated part of the process, and the one that requires the most code. But if you take it slow, it should be no problem. Also, given that creating the vertices and textures is the one part that will be repeated throughout your game in some form, you will get a lot of practice with the code. It will get easier the more you use it.

As far as OpenGL ES is concerned, all images are textures. Textures are meant to be mapped onto a shape. You will be creating a primitive square to map your image (or texture) onto and display it to the screen through the renderer and the `GLSurfaceView`.

To do so, you need to create a new class, `SBGSplash`, which involves the following steps, all of which will be described shortly:

1. Create some buffers.
2. Create the constructor.
3. Create the `loadTexture()` method.
4. Create the `draw()` method.

The constructor for the `SBGSplash` class is going to set up all of the variables that you need to interact with OpenGL ES (see Listing 2-1). You need an array to hold the mapping coordinates of your texture, an array to hold the coordinates of your vertices, and an array to hold the indices of the vertices. Finally, you create an array of resource identifiers that refer to your textures.

Listing 2-1. SBGSplash (OpenGL ES 1)

```
public class SBGSplash {
private int[] textures = new int[1];

private float[]vertices = {
0f, 1f, 0f,
0f, 0f, 0f,
1f, 0f, 0f,
1f, 1f, 0f,
};
private float[] texture = {
1f, 0f,
1f, 1f,
0f, 1f,
0f, 0f,
};
private byte[] indices = {
0,1,2,
0,2,3,
};
public SBGSplash() {
//empty constructor
}
}
```

The `textures` array holds an identifier to each texture that you are loading. You are hard-coding this to 1 because you will only be loading one image, but we are leaving this flexible enough for you to reuse in the future without much rewriting.

The `vertices` array lists a series of points. Each row here represents the x, y, and z value of a corner of a square. This square is the primitive shape that the image will be textured to in order to be displayed. In this case, you are making a square that is the full size of the screen, ensuring that the image covers the entire screen.

The `texture` array represents where the corners of the image (or texture) will line up with the corners of the square you created. Again, in this case, you want the texture to cover the entire square, which in turn is covering the entire background.

Finally, the `indices` array holds the definition for the face of the square. The face of the square is broken into two triangles. The values in this array are the corners of those triangles in counterclockwise order. Notice that one line (two points) overlap (0 and 3).

If you are using OpenGL ES 3, you need to add your shader code here, as shown in Listing 2-2.

Listing 2-2. SBGSplash (OpenGL ES 2/3)

```
public class SBGSplash {
private final String vertexShaderCode =
"uniform mat4 uMVPMatrix;" +
"attribute vec4 vPosition;" +
"attribute vec2 TexCoordIn;" +
"varying vec2 TexCoordOut;" +
"void main() {" +
"  gl_Position = uMVPMatrix * vPosition;" +
"  TexCoordOut = TexCoordIn;" +
"}";
private final String fragmentShaderCode =
"precision mediump float;" +
"uniform vec4 vColor;" +
"uniform sampler2D TexCoordIn;" +
"uniform float scroll;" +
"varying vec2 TexCoordOut;" +
"void main() {" +
" gl_FragColor = texture2D(TexCoordIn, vec2(TexCoordOut.x + scroll,TexCoordOut.y));"+
"}";
private int[] textures = new int[1];

private float[]vertices = {
0f, 1f, 0f,
0f, 0f, 0f,
1f, 0f, 0f,
1f, 1f, 0f,
};
private float[] texture = {
1f, 0f,
1f, 1f,
0f, 1f,
0f, 0f,
};
private byte[] indices = {
0,1,2,
0,2,3,
};
public SBGSplash() {
//empty constructor
}
}
```