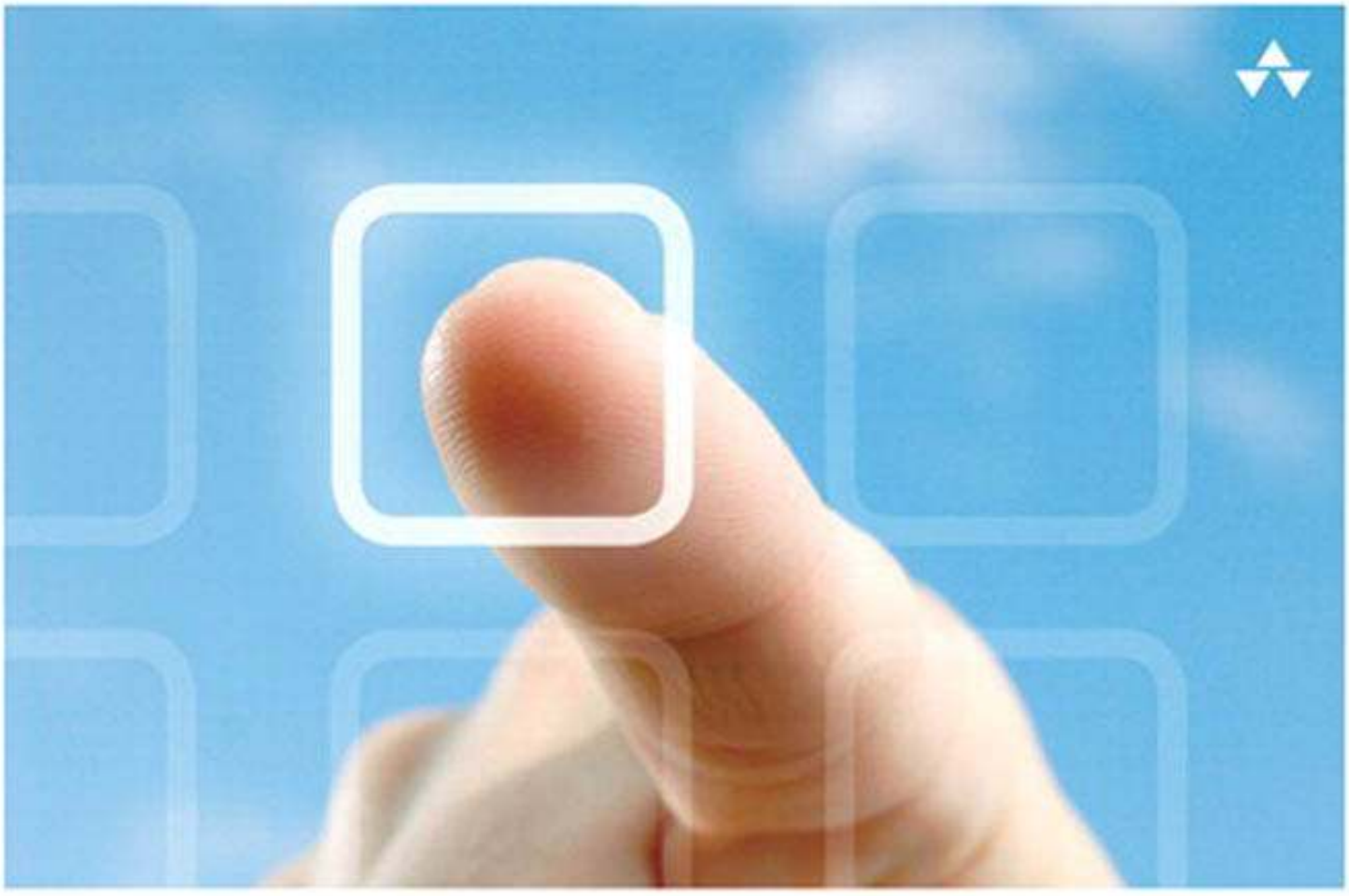


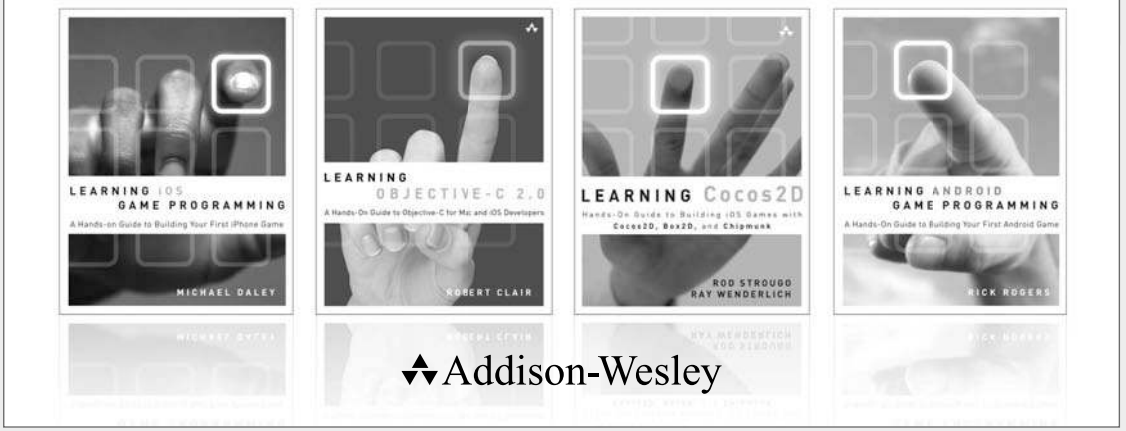
LEARNING Android GAME ROGRAMMING

A Hands-On Guide to Building Your First Android Game

RICK ROGERS

Learning Android Game Programming

Addison-Wesley Learning Series



Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning Android Game Programming

A Hands-On Guide to Building Your First Android Game

Rick Rogers

Addison-Wesley

Upper Saddle River, NJ · Boston · Indianapolis · San Francisco
New York · Toronto · Montreal · London · Munich · Paris · Madrid
Capetown · Sydney · Tokyo · Singapore · Mexico City

Library of Congress Cataloging-in-Publication data is on file.

ISBN-13: 978-0-321-76962-6
ISBN-10: 0-321-76962-7

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2011

Editor-in-Chief
Mark L. Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Jill E. Hobbs

Indexer
Ted Laux

Proofreader
Rebecca Rider

Technical Reviewers
James Becwar
Stephan Branczyk
Jason Wei

Cover Designer
Chuti Prasertsith

Compositor
LaurelTech

❖

For Susie, my muse and my partner

"Let us be grateful to people who make us happy, they are the charming gardeners who make our souls blossom."
—Marcel Proust

❖

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

In early 2010 the availability of powerful and free 2D game engines for the Android platform was an almost empty field. Today, developers can pick from a few engines that best fit the purpose of unleashing their individual creativity.

With currently more than 500,000 Android devices being activated daily, every single one of those is reachable from the minute the device is turned on. Literally, every day counts. This market is shifting the world of successful business models away from big companies toward individual developers, where any developer could create the "Next Angry Birds" in just one night.

I created AndEngine to fulfill the need for a free, easy-to-use game development framework, one capable of allowing even inexperienced game developers quick access to this incredibly fast-growing market without limiting the creativity of expert game developers.

Today more than two hundred games powered by AndEngine have been shipped and the AndEngine code has been executed over one million times. AndEngine has allowed developers to create games that successfully reach millions of customers and provide steady income for the developer. And since Zynga hired me mid-2011, AndEngine has been brought to a whole new level of professionalism.

More and more developers are demanding knowledge about game development on the Android platform, which means there is, and will continue to be, a strong need for solid instructional literature. Rick Rogers has written an excellent book covering general game development topics in simple language, using AndEngine as the powerful back end that brings game development to life. Rick guides the reader through the construction of a complete game example, covering all essential topics for beginners while providing useful tips and hints even for experienced game developers. Enjoy the book!

—Nicolas Gramlich
Creator, AndEngine

This page intentionally left blank

Preface

Key Features of This Book

This is a book about writing games for Android mobile devices. If you have at least some experience developing applications for Android, this book will tell you how to use that experience, combined with an open-source game engine called AndEngine, to write your own 2D mobile games. Whatever genre of game you want to write, examples are provided and explained step by step. The goal is for you to become familiar with AndEngine and publish your game as quickly as possible. Many of the examples support the development of an example game, "Virgins Versus Vampires" (V3).

The book begins by presenting an overview of mobile games, their popularity, the types of games, and an example of planning a game in Chapter 1. The following chapters then expand on a single topic related to developing your game:

- Chapter 2, Game Elements and Tools, describes the tools that are used to develop games, including code development, artwork, and sound.
- Chapter 3, The Game Loop and Menus, introduces the concept of a game loop and shows you how to start development with AndEngine.
- Chapter 4, Scenes, Layers, Transitions, and Modifiers, dives into graphics and uncovers the scene transitions and entity modifiers that AndEngine provides to make a game come alive.
- Chapter 5, Drawing and Sprites, goes deeper into developing bitmap and vector graphics for your game, and shows you how to display sprites.
- Chapter 6, Animation, introduces easy ways to build animated sprites for your game, and really get things moving.
- Chapter 7, Text, gives examples of ways to use AndEngine to display text in your game.
- Chapter 8, User Input, explores the many user input options available for Android games, including touch, multi-touch, keyboard, voice recognition, accelerometer, location, and compass.
- Chapter 9, Tile Maps, describes how AndEngine loads and works with tile maps and their tile sets to build virtual worlds that can be of infinite size.
- Chapter 10, Particle Systems, demonstrates the particle system built into AndEngine and shows how to define and save particle effects as XML files.
- Chapter 11, Sound, shows you how to find, acquire, modify, and use background music and sound effects with AndEngine.

- Chapter 12, Physics, explores the physics engine, Box2D, which works with AndEngine to facilitate building games based on the physical interaction of objects.
- Chapter 13, Artificial Intelligence, examines some of the artificial intelligence techniques you can use to make your game smarter and more fun to play.
- Chapter 14, Scoring and Collisions, builds a scoring framework based on collisions between elements of your game.
- Chapter 15, Multimedia Extensions, investigates some of the extensions that are available for AndEngine to perform tasks such as creating Android live wallpapers, playing MOD music files, creating augmented reality games, and communicating among players in multiplayer games.
- Chapter 16, Game Integration, finishes off the example game by completing or adding features to make it playable.
- Chapter 17, Testing and Publishing, describes what you need to do to ensure your game is ready for publication, and then tells you how to publish and promote your game.
- The Appendix, Exercise Solutions, provides the solutions to the end-of-chapter exercises.

This book is best read in order, but if skipping around suits you better, that will work as well. Each topic is presented mostly as a stand-alone concept, but if references to other chapters are needed, they are provided.

Mostly, the goal of this book for readers is a simple one: Have fun. The book was written in the spirit that games should be fun to play and that developing games should be fun in itself. May your game top the Android Market "Most Frequently Downloaded" list.

Target Audience for This Book

If you have a burning desire to create your own 2D game for Android devices, and at least a little background in developing Android applications using the Android SDK and Java, this is your book. It introduces basic topics in mobile games, and shows how those topics are implemented using the AndEngine game engine. You don't need to be an expert Android developer to follow the examples, but you do need to be familiar with the basic Android concepts (e.g., Activity, Service, Intent), and need to be comfortable with reading and writing Java and with using the Android SDK.

Code Examples for This Book

The code listings in this book are available through this book's website:

`http://www.informit.com/title/9780321769626`

They are also available from the companion github site:

`https://github.com/portmobile/LAGP-Example-Code`

Acknowledgments

The list is long of people I'm indebted to for helping me create this book.

- **Nicolas Gramlich** created the AndEngine game engine on his own, just because he wanted a world-class game engine for Android. He then shared his hard work with the world as an open-source project, and now he's sharing it with you. Nicolas graciously allowed us to use AndEngine as the basis of this book and also volunteered to review the drafts. He continues to improve and extend AndEngine and make those enhancements available to all of us.
- **Trina MacDonald** has been the Acquisitions Editor for this book, and she was the one who suggested the idea of a book on developing Android games. Trina is an awesome manager of projects, and this book could never have been completed without her tireless efforts to bring it all together.
- **James Becwar, Stephan Branczyk,** and **Jason Wei** were the Technical Editors for the book. You won't find a better group of technical reviewers anywhere. This trio of folks kept me honest as I was writing the book, making sure that the technical content was accurate and that the example code really worked.
- **Songlin Qiu** was the fantastic Development Editor for the book. If you find the book clear and easy to read, it is due to the many valuable suggestions Songlin made as she reviewed the drafts. If you find it difficult to read, it is likely due to those few suggestions that I declined.
- The book would not have made it into print without the diligent and tireless efforts of **Julie Nahil,** our Production Manager, and **Jill Hobbs,** the copy editor. They both deserve a lot of credit for suffering through the author's short attention span to persist in getting the project completed.
- As the Editorial Assistant, **Olivia Basegio** is the one who actually gets things done. It was she who made sure the drafts got to the right reviewers and to Rough Cuts, she who organized the illustrations and licenses, and she who remembered to do the things that I had forgotten. Without Olivia, we wouldn't have a book—we'd have a bunch of loose ends.
- I don't have space to list the large collection of friends and family who have encouraged me through the sometimes difficult process of writing a book. I'd especially like to thank our daughters, **Allison Jackson** and **Katie Kehrl,** for their unfailing optimism that I'd actually get the book done someday, and the examples they set for me with their own lives.

- **Susie Jackson,** my wife, is the inspiration for everything I do, including this book. She is an incredible person and I am very lucky to be married to her. The confidence and positive attitude she brings to our lives are what give me the strength to sit down in my office and create. Thanks, Susie, again.

About the Author

Rick Rogers has been developing software for more than thirty years, and has focused on software for mobile devices for the last twelve years. He is the author of numerous technical magazine articles and a previous book on introductory Android application development. He has developed mobile device software for large and small companies, and participated in international consortia that have shaped the evolution of mobile devices.

He lives with his wife in the bucolic town of Harvard, Massachusetts, and on Cape Cod.

This page intentionally left blank

1

Mobile Games

Perhaps nothing is as universal as the spirit of play—almost everyone likes to play games of some sort. Furthermore, if—as the cliché goes—everyone has at least one good novel in them, it's fair to say that everyone has at least one good game idea as well. You probably have an idea for a mobile game, or you wouldn't have picked up this book. The aim of the book is to show you how to write your own game to run on Android mobile phones. Whether your game is very similar to the example game or quite different from it, this book will show you how to use the popular AndEngine game engine[1] to produce your very own 2D mobile game and publish it on Android Market.

For many of us, writing software itself is also a game—an endless puzzle in which we try to figure out the best way to implement application ideas, and more puzzles in which we debug what we wrote initially. When the application is itself a game, we enjoy the process at multiple levels. Come and play the software game, and develop that idea that's been burning in the back of your mind all this time.

The Mobile Game Market

Games are the killer applications for smartphones today. According to one analyst,[2] more than 23% of all mobile phone users older than 13 years of age in the United States play games on their phones—and that percentage is increasing, especially for the 60 million-plus smartphone users. According to another analyst,[3] 65% of smartphone users have played a mobile game on their phones at some point. Doing the math, that means approximately 40 million people today play games on their smartphones.

Creating mobile games can be a very profitable business. It's very difficult to predict which games will be hits, but a quick scan of Android Market shows that hundreds of thousands of users have downloaded certain games. Even at a few dollars per download, that adds up to serious money. People also tend to get tired of games once they've played them for awhile, opening up opportunities for new games.

---

1. The AndEngine website can be found at `http://www.andengine.org`.
2. comScore (`http://www.comscore.com/Press_Events/Press_Releases/2010/12/comScore_Reports_October_2010_U.S._Mobile_Subscriber_Market_Share`).
3. nielsenwire (`http://blog.nielsen.com/nielsenwire/online_mobile/the-state-of-mobile-apps/`).

I'm part of the games-loving public: Games are some of my favorite mobile applications. Whether I'm killing time waiting to see someone, riding public transportation, or just in the mood to escape for a few minutes, playing a game on my mobile phone can be an enjoyable way to pass the time.

I think every game should be fun, but that doesn't mean games cannot be instructive as well. Games are often used as instructional or advertising vehicles—and why not? If students or potential customers have a good time playing a game that teaches them something valuable, that's a good thing.

The World of Computer Games

People have been playing games on computers for almost as long as electronic computers have existed, and a rich variety of games has been invented. In her book *Reality Is Broken*, Jane McGonigal says that most games have four attributes:

- A goal: Games clearly define a goal for the players to achieve. It's important that the goals be challenging, yet achievable. Ideally, players are always playing at the leading edge of their ability. Goals give the players a sense of purpose in playing the game.
- Rules: Games have rules that all the players agree to follow. The rules often make achievement of the goal difficult, which in turn encourages players to be creative.
- Feedback: A game has to tell the players how they are doing. Indeed, an interesting, creative feedback system is key to making a game enjoyable.
- Voluntary participation: It just isn't a game unless you really want to play. This aspect of games implies the players' acceptance of the goal, rules, and feedback system.

Before we create a new game, we want to think about which types of games exist, as well as which types work well on mobile devices and which don't. We also want to take a look at the components that are common to all computer games.

Game Genres

Game developers didn't start out categorizing their games, and there is no standard list of categories. Nevertheless, over time games have been grouped into classes by different people in different ways. The categories identified in this section are not meant to be canonical, and they admittedly overlap in a number of areas. The exact categorization really isn't important—the point is that numerous types of games can be developed.

Skill or Action Games

Action game players typically have to use some real-time skill (e.g., jump a barrel at the right time, shoot at a moving target) to be successful. Subtypes with some examples include the following:

- Maze games
- Platform games where the player moves platforms around either to get somewhere or to stop adversaries

- Tower defense games: the player defends something (the tower) from an oncoming horde of bad guys
- Shooters: with the playing field either fixed, sliding, or scrolling
- One-on-one fighting games: where two opponents battle it out
- One-to-many fighting games: where the player fights through a gang of opponents (often martial arts related)
- First-person shooters (FPS): where the player's view is that of the shooter
- Third-person shooters: same as FPS, but the point of view is that of a third person

Strategy Games

Strategy games are less about reacting to real-time events, and more about devising and implementing a strategic plan to overcome obstacles. They include the following types of games:

- Turn-based games: including traditional board games
- Timed strategy games: where each move occurs in a fixed time
- Massively multiplayer online role-playing games (MMORPs): an extension of the old Dungeons and Dragons genre, in which players assume roles and play against others online

Adventure or Storytelling Games

Adventure and storytelling games are built around a rich storyline, with well-developed characters and a story that defines the player's purpose in playing the game.

- Simpler 2D story games often involve mazes and interactions with other game entities.
- Complex 3D story games can show different points of view as the game is played and the story spun. Some have been turned into Hollywood movies.

Simulation Games

Typically, simulation games depict some real situation, such as a vehicle that the player can operate. The games reproduce the physics of the real situation and can be good enough to use for instruction as well as for just playing a game. They include the following types of games:

- Sports simulators
- Flight or space simulators
- Driving or racing simulators
- Boat or submarine simulators
- Life simulators (overlap with strategy games)

Puzzle Games

Many puzzle games are direct translations of printed puzzles (e.g., crosswords), but the genre also includes matching and hidden object games. Complex games often include smaller puzzle games to solve as part of the larger game. Examples of puzzle games include those based on the following concepts:

- Word based (e.g., crosswords)
- Number/math based (e.g., Sudoku)
- Visual matching
- Hidden object (e.g., Minesweeper)
- Construction from a set of pieces

Augmented-Reality Games

It's fine to play games just for the fun of it, but sometimes there's a bigger motive. As Jane McGonigal's *Reality Is Broken* points out, some games are intended to augment reality in such a way that our real lives are made easier. Examples of augmented-reality games (ARGs) include the following games:

- Jetset: a game that simulates the security line at an airport (to help you pass the time while you wait in the real line)
- Chore Wars: a game that turns household chores into creative competition
- World Without Oil: a game that encourages energy conservation by simulating a world where oil products are in very short supply

Games for Mobile Phones

With this rich variety of game types to choose from, we need to focus on those that are most appropriate for mobile platforms such as phones and tablets. We also need to focus on those games whose development by a small group of people is feasible.

Given the potential size of the mobile device games market, it's not surprising that a substantial amount of research and thought have been put into what makes a good mobile game. The usual principles of good computer game design still apply, along with special characteristics of good mobile games:

- Don't waste the player's time.
- Provide help on playing the game.
- Make the game goals easy to understand.
- Show game status clearly.
- Mobile users typically play games in short sessions.
- Players need to easily pause and resume a game, and the phone should be able to pause and resume games when necessary (e.g., for an incoming call).
- Players should be able to make game progress in a short period of time.

- Mobile devices have physical constraints that affect games:
  - Small screen size and a variety of screen sizes, resolutions, and pixel densities
  - Variety of user input methods (e.g., one- and two-handed operation, touch, keypad, multi-touch, keyboard, Dpad, trackball)
  - Limited computational power
  - Limited battery (a factor that limits power-intensive graphics and computing)

Even if you had the development resources to create a really snazzy 3D first-person shooter game like Halo, players are unlikely to sit with their smartphone and play it for hours the way they might with the XBox version. Users are much more likely to play mobile games in short sessions, pausing and resuming the game perhaps days later.

Speaking of resources, what does it take to create a commercial game? A typical console game for a single console can easily take $10 million to develop, and two or three times that amount for multiple-console development (it has been estimated that some complex games cost as much as $100 million to create). The software development kit (SDK) and license to create a console game alone can cost thousands of dollars. If you think about what goes into a professional 3D console game, it's easy to see where the costs mount up—3D artwork, motion capture, animation, game play, user testing, and software development are all both time consuming and expensive.

This book is about you and maybe one or two friends creating your own mobile game for the Android platform. The Android SDK is free, and as of this writing, it costs only $25 to sign up for Android Market and sell your game to anyone with an Android device. We'll stick to 2D (two-dimensional) games, which makes the artwork and the programming simpler. As you'll see, the basic game structure and components of any 2D game are pretty much the same no matter what the genre, but we need to pick one as an example.

Components of a Typical Game

Before we look at the specifics of the example game, let's examine the general components that we need to work into the game and implement in the code. Here are some components that will be part of our game.

Opening (Splash) Screen

To maximize performance as the game is being played, the graphics needed for a game level are often loaded before the level is started. During the loading process, which can take several seconds, you don't want to leave the user with a blank screen, so you display a splash screen. It lets the user know that the game is working as it should. Splash screens are optional, but we'll include one in our game, just to show how it's done.

Menu Screen

Once the game is ready to run, we'll need a place for the user to enter various options (e.g., turn the sound on/off, get help in playing the game). This is typically done with a graphical menu screen that presents the options and either implements the option or calls another screen (such as Help) to do so.

Music

For most of us, music has strong emotional influence. Background music is very important for setting the mood of your game, and helping with the transitions between parts of the game.

Sound Effects

Sound effects can make a game a lot more fun. When two objects collide, players expect to hear a sound of some kind—whether it's a clang, a thud, or a boing. Our example game also incorporates sound effects for each of the game characters. Each villain has a characteristic sound effect accompanying his or her presence in a scene.

Time

Most games will incorporate time—either clock time (scoring completion of a puzzle based on the time taken to solve the puzzle) or playing against moves the computer (or computer-driven adversaries) makes in real time. In our game Virgins Versus Vampires (V3), this factor takes the form of killing the villains before they can reach the virgins.

Lives

Games have to be challenging to be fun, so the player has to fail every once in a while. Killing the player off (in a virtual way) is a convenient way to give failure a consequence. Some games give the player multiple lives per session, whereas others (and V3) give the player only one life.

Obstacles

Obstacles are used in different ways in different games. In many games, the player is trying to achieve some goal, and obstacles are thrown in the player's path. In tower defense games (and V3), it's the adversaries who are trying to reach a goal, so the player throws obstacles in their paths.

Levels

Challenging games are fun, but it's important to provide a range of challenges, so that players can start with easy challenges and gradually ramp up to higher challenges as their game-playing skills and experience improve. Levels are a proven way to achieve this effect—the player learns how to play the game in the first few levels, and his or her skills have to continue to improve as new levels are presented. This is also a great way to add some variety to the game.

Adversaries

The adversaries in a game are sometimes referred to as entities (although AndEngine uses that word to mean something else). These characters are the villains (or other players) that the player must overcome to win. They are distinct from obstacles in that they take action against the player—obstacles are more passive. We've listed the entities for V3 later in this chapter, along with an outline of their behavior.

Player

Of course, the player is the most important component of any game. The whole point is to keep the player engaged and interested so he or she will keep playing the game. The player has to be challenged by the game, but not too challenged to give up in frustration. The game has to include enough variety to maintain the player's interest, and rewards have to be doled out to recognize success in playing the game.

Scenes

If you think of the game as something like a movie, each screen that is displayed to the player is something like a movie scene. Each scene has background graphics that don't change much (although the player's point of view might change). Animated graphics are then added to the scene to implement the entities and obstacles that interact to make the game.

Virgins Versus Vampires

A popular genre on mobile devices is the tower defense. These games are fairly simple to understand (stop the bad guys), they lend themselves to interrupted playing (pause/resume), they fit well on a small screen, and they don't require a lot of computer horsepower. On the production side, the artwork for a tower defense game is relatively simple, and it makes good use of the major elements of computer game programming. We also want the game to be fun to play, of course, so we'll try to inject some humor and challenge into the genre.

We need a "tower" to protect. Offhand, I can't think of anything that's been protected more vigilantly over the course of history than virginity, so we'll make that the target of the bad guys. Vampires are the trendy bad guys these days, so we'll incorporate them as well. Maybe we can even find a way to fit in the theme of the "vampire with a heart of gold"—ambivalence always adds interest.

Figure 1.1 shows what the screen will look like during a session of our game, which we'll call Virgins Versus Vampires.

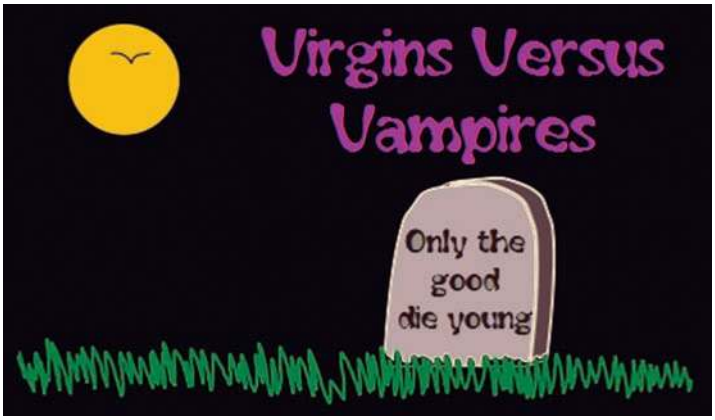


Figure 1.1 Screenshot of Virgins Versus Vampires game

The V3 game is available for free on Android Market. Take a few minutes right now to download it to an Android device and play with it for awhile. At least finish Level 1 of the game, which you should do fairly quickly, to get an idea of the flavor of the game.

We need a variety of obstacles that we can use to impede the vampires' progress. We'll use the items described next for this purpose.

Bullets

We'll have a bullet weapon that players can fire by placing it on the playing field and letting it go:

- Kills: anything it hits
- Life: from where it is launched until it goes off the screen
- Scoring: not so high (per vampire), because it's easy to kill a whole line of vampires

Hatchets

The hatchet weapon is also placed by the player and thrown when they let go of it:

- Kills: the first vampire it hits
- Life: from launch until it hits the first vampire
- Scoring: higher, as it kills only one vampire

Crucifix

The crucifix just stays where the player puts it, and waits for a vampire to trip over it.

- Kills: the first vampire who runs into it
- Life: from placement until it kills a vampire
- Scoring: highest, as it depends on a vampire stumbling into it

The virgins will be held in Miss B's Girls' School on the left of the screen, with bad guys coming from the right. The game player's task is to throw obstacles in the way of the marauding bad guys to keep them from reaching the castle. We need to give the player a way of earning obstacles, placing obstacles, and watching the progress of the bad guys. We want multiple levels, so players can start off with easy games and progress as their strategies and talents improve. And, of course, we want to be able to assign scores and track them.

Design of V3

Once you have a game concept outlined, the next step in designing a game is to envision the scenes needed and describe the flow among them. Screenwriters and many creative writers do this by making a storyboard with pencil and paper, using an index card or drawing a rectangle for each scene, and creating a very rough sketch of the scene and a few words to describe what's going on there. You can show the transitions between scenes with an arrow and a brief description of when the transition takes place.

Figure 1.2 shows the storyboard I drew for V3, which is intentionally a very short game. The storyboard for a complete game will likely spread to multiple pages.

I also created a separate index card for each scene of the game, with a rough sketch of the graphics to be included. If you are creating your storyboard on a large piece of paper, you can just include the sketches right on the flow diagram. Figure 1.3 shows the index card for Level 1 of V3.

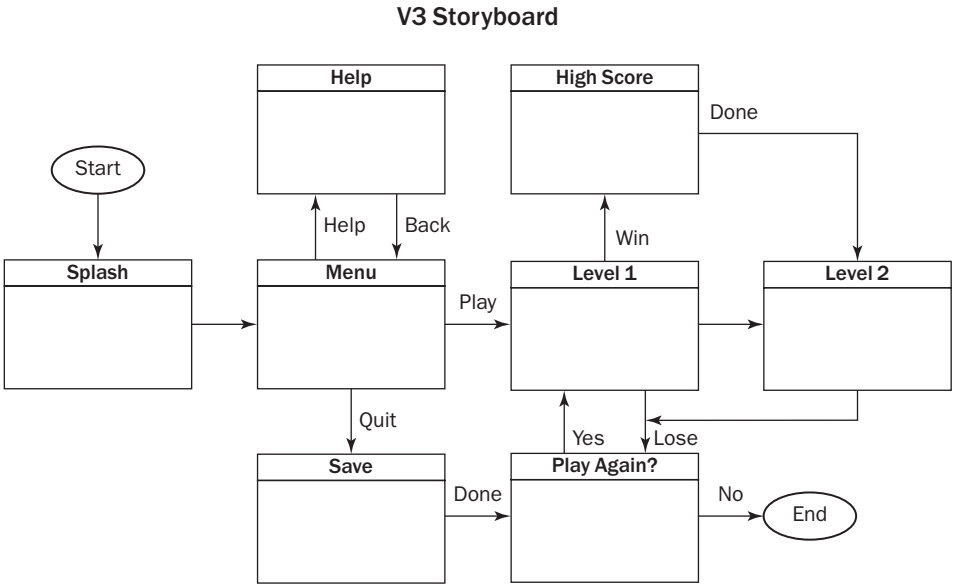


Figure 1.2 Preliminary game storyboard flow diagram

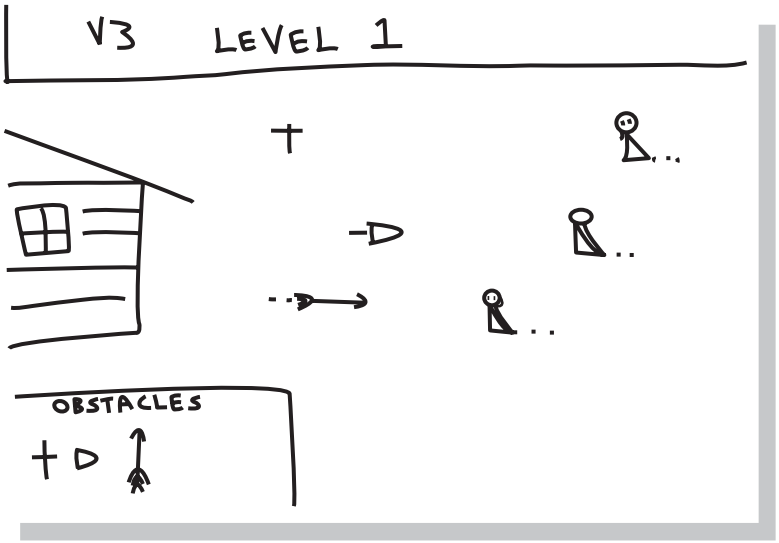


Figure 1.3 Index card for Level 1 of the storyboard

AndEngine Examples

We will be using the AndEngine game platform in the rest of this book, so now would be a good time for you to get a taste of what that game engine can do. Scores of games built using AndEngine are available on Android Market, but instead of downloading another game, let's download an example program that demonstrates many of the features of AndEngine.

Nicolas Gramlich, the lead developer for AndEngine, created the example program, and has made it available on Android Market for free. Go to Android Market from an Android device, and search for "AndEngine Example." You should get the screen shown in Figure 1.4.

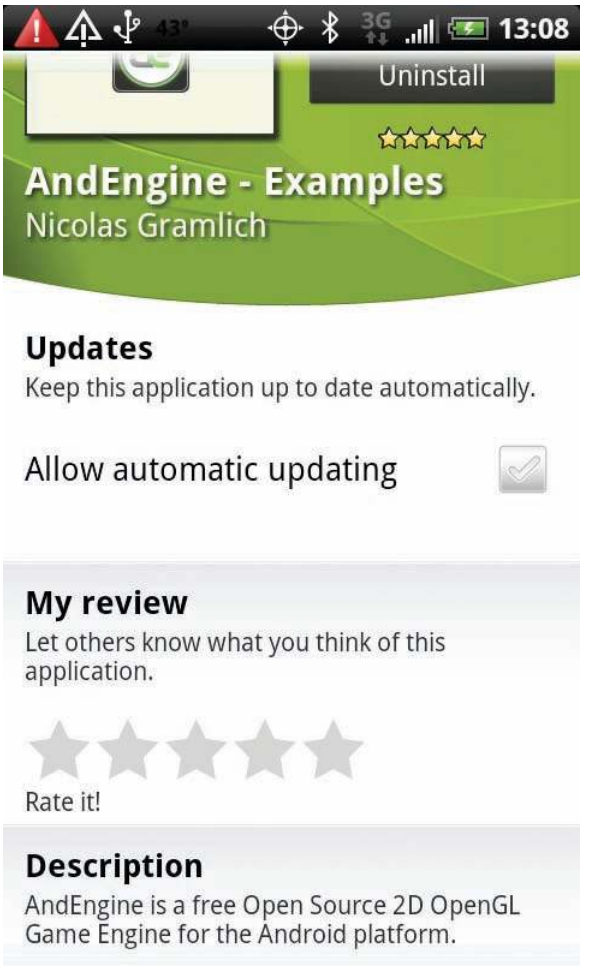


Figure 1.4 AndEngine download from Android Market

Nicolas has generously made the source for AndEngine Examples available as well (at `http://code.google.com/p/andengineexamples/`). These resources are excellent references for how features can be used. If you prefer (or if you don't have access to Android Market for some reason), you can download the `.apk` installation file from that site, and load it onto your Android device (or the emulator) using adb (Android Debug Bridge). We'll get into building source in more detail in Chapter 12. For now, just install the app on your phone, and start it up. You will see a menu of features, as shown in Figure 1.5.

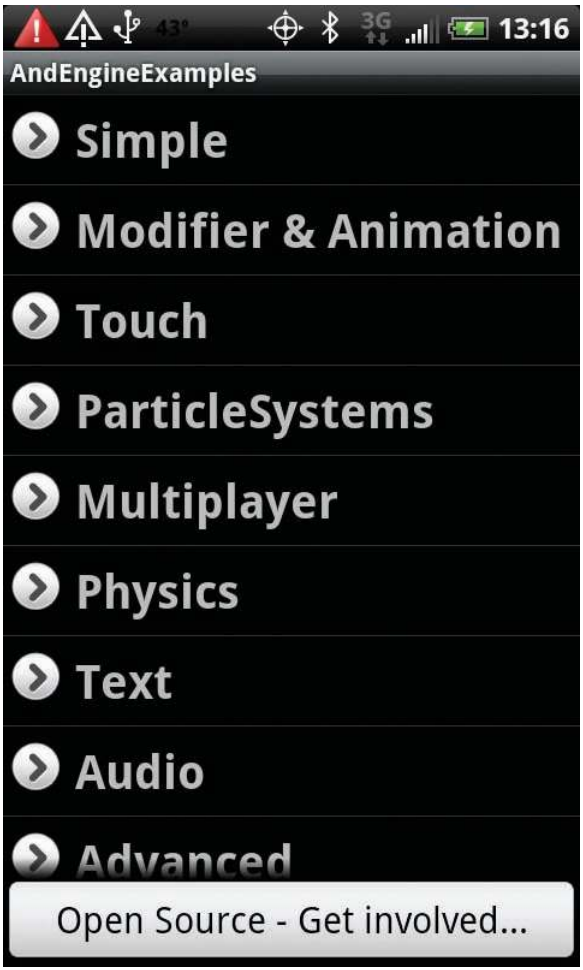


Figure 1.5 AndEngine start-up screen

The menu items form a hierarchy of options, each of which demonstrates one aspect of the AndEngine platform. Take some time now to just play with the examples to get a taste of what AndEngine can enable your game to do.

Summary

This chapter was all about introductions. We covered the basics that we'll need to talk about games and the fundamental concepts that are part of all mobile games:

- We talked about the various types of games, or genres, and surveyed the types that have been invented so far. Of course, you can invent a new genre of your own, but there are already a lot of options to consider for inspiration.
- We looked at what makes a mobile game successful (and fun!). Throughout the rest of this book, we'll try to keep in mind that the point is for the player to have fun playing the game, and we'll try to build on the experiences passed on by previous game inventors regarding what works and what doesn't.
- We started looking at an example tower defense game that we will use to illustrate the tools and techniques discussed in this book. The game concept is quite simple at this point, but it incorporates most of the elements of a typical mobile game.
- We defined some basic game terminology, so we can talk about the typical components of a game.
- We laid the groundwork to talk about the development of your own Android mobile game. We have the terminology necessary to discuss the different components of the game, and we can move on to investigate the tools needed to create them and the techniques needed to implement them.

Exercises

1. Write a description of the game you'd like to build. Don't be too concerned with getting all the details right (you'll think of new details as you implement and test your game), but write down the important elements of the game. Pretend you are writing a proposal aimed at a game publishing company, suggesting development of a new game.
2. Get some friends to review your game proposal. Do they think the game would be fun to play? Which changes or suggestions do they have to make it better? Be prepared for a range of responses, depending on the mood of the group and the beverages available: Some of the suggestions will be practical and some will be "creative." After the review, see how many of the suggestions you can incorporate into your game proposal.

3. Develop a storyboard for your own game. There are no real standards for storyboards, so you can use whatever conventions seem natural to you. Try to include the following elements:
   - Scenes you think the game requires
   - The way the player transitions from scene to scene
   - Any special characteristics of each scene
   - A rough graphical layout of each scene
4. Start a list of artwork that you will need for your game. Some games don't need elaborate artwork; just geometric drawings will suffice. Other games need an entire staff of artists to create complicated virtual worlds.

This page intentionally left blank

2

Game Elements and Tools

From the overview in the last chapter, it's obvious that creating a good mobile game involves writing some rather complex software, and also requires the creation of other components, such as graphics, animation, sound effects, and music. To be able to offer your game for others to play, you need to have commercial rights to all of this intellectual property, and the easiest way to get the rights is to create the game all yourself.

Fortunately, tools are available for all of these components, and many of them are available for free. If you've got an Internet connection and a development machine, you've got access to just about all you need.

The example game used in this book, Virgins Versus Vampires, was written using the Java programming language, which runs in the Dalvik virtual machine on Android devices. The game makes use of an open-source game engine called AndEngine and a physics engine called Box2D, both of which have been ported to Android. The game and all of its intellectual property were created using freely available software development tools, graphics tools, and audio. All of these items are described in this chapter, which also offers our first exposure to actually writing some code—specifically, a splash screen for the game.

It's important to realize that the games we are writing are normal Android applications. They are written in Dalvik/Java, with full access to the Android application programming interfaces (APIs), and their activities have all the characteristics that we expect of Android activities (e.g., pause, resume). Our games will link to the AndEngine library, and each will include a copy of the library in the Android application package file (`.apk`) for the game.

Software Development Tools

We need software development tools to write software. As luck would have it, excellent tools are available for writing mobile software in general and games in particular. Even luckier for us, many of these tools are free to download and use, even if we're creating a game we plan to sell.

Android Software Development Kit

If you are not already familiar with the Android SDK, stop right here and take the time needed to become familiar with it. The first step is to go through the download and installation instructions at the following website: `http://developer.android.com`.

The Android SDK uses Eclipse for its integrated development environment (IDE) and tools that come with Oracle's Java Development Kit (JDK). The installation instructions on the Android website will lead you through their installation (if that step is required). As of this book's writing, the SDK is componentized. The examples and figures you see in the book were all built using the following versions of its components:

- Android SDK
- Android SDK Platform Components through 4.0
- Android SDK Tools, r14
- ADT Plugins for Eclipse 14.0.0
- Eclipse Helios (the version recommended for this version of the SDK)
- Oracle/Sun Java Development Kit (JDK 6, also called JDK 1.6)

You may be using later versions than the ones identified here, but the example code is relatively independent of version. If something doesn't work, take a look at the book's companion website at `https://github.com/portmobile/LAGP-Example-Code` and see if updates have been published for later versions. As this is being written, the current version of Android is 4.0, also known as *Ice Cream Sandwich*.

You should also have created Android Virtual Devices (AVDs) for each type of device you intend to support, using the Android SDK and AVD Manager that comes with the SDK. For the examples shown in the book, we created an AVD that is a lot like the HTC EVO smartphone:

- Name: EVO
- Target: Android 2.2 (API level 8)
- Skin: HVGA
- SD Card: 128M
- SdCard: yes
- Accelerometer: yes
- LCD.density: 160
- AudioOutput: yes
- Camera: no (not supported on this version of the emulator, and not needed)
- Battery: yes

You should have also at least gone through the tutorials that are provided for the SDK and be familiar with the processes of creating Android projects, editing code,

building projects, running them on the Android emulator, and debugging projects using the Eclipse debugger, LogCat, and the other tools provided in the SDK. If you plan to publish your game to Android Market, you need to make sure it runs well on actual phones, so you should also have some experience loading and running `.apk` files on an Android phone.

The documentation on the Android developer site is excellent and very thorough. If you need a gentler introduction or more examples, many excellent Android programming books are also available, including *Sam's Teach Yourself Android Application Development in 24 Hours* by Lauren Darcey and Shane Conder.

AndEngine Game Engine Library

AndEngine is a game engine library that makes it easier to write two-dimensional games for Android devices. Nicolas Gramlich led the effort to create AndEngine and wrote much of its code. The project is open source, so you are encouraged to go to the project website and join in the development effort for AndEngine yourself.

We could, of course, write our own routines in Java, using the Android APIs to implement the components of a game. Nevertheless, there are good reasons for leveraging a game engine that is already written:

- We leverage the work of others. As an extreme example, we could also write our own IDE for Android if we really wanted to, or even our own Java compiler, but unless we really need some special functionality, doing so makes no sense.
- With open-source code, like that used in AndEngine, we always have the option of extending the engine functionality any way we like. If the extension might be useful to others, we can contribute the changes back into the open-source repository and improve the engine for everyone.
- When we run into problems, we can access a community of developers using the same technology. It's likely that someone else has already dealt with our problem and knows how to solve or work around it.
- We get the benefit of many developers' optimizations. Games use a fair amount of computer resources to draw graphics, animate the graphics, compute object physics, render sounds, and keep up with the user input. By using a game engine, we have ready access to optimizations that have already been tuned.

Other game engines are currently under development for Android, but this book will focus on AndEngine. Important websites for AndEngine are listed here:

- The source code repository for AndEngine: `http://code.google.com/p/andengine/`
- The source code repository for examples: `http://code.google.com/p/andengineexamples/`
- The AndEngine community forum: `http://www.andengine.org/forums/`
- The AndEngine wiki: `http://wiki.andengine.org/AndEngine`

These locations may have changed by the time you read this book. If they have, just use your browser to search for "AndEngine Android," and you should be able to find the current locations.

AndEngine comes as a `.jar` file—that is, a Java archive. You'll see how to use that archive with your Android SDK project when we start our game coding later in this chapter. The engine is provided under the GNU Lesser GPL License, which allows you to use the source code and link to the binaries for just about any reasonable purpose. (*Note:* I am not a lawyer—you or your lawyer can read the text of the license in the file that is referenced on the repository website.)

AndEngine Game Concepts

The movie analogy we referred to earlier is a good way to approach AndEngine. Your game is like a movie, and the game engine includes concepts that are analogous to those involved in making a movie.

Camera

The "camera" of the game determines the view of the game that is presented to players. It is very much like a movie camera in two-dimensional space. The camera can pan and zoom across the scene to change the view presented. The panning and zooming can either be under the player's control or be driven programmatically.

Scene

A game, like a movie, consists of a series of scenes where the action takes place. In a movie, the scenes are edited together in a fixed way. In games, the sequence of scenes is driven by the play of the game. Games are like movies edited on the fly.

Layer

Scenes are composed of layers of graphics. The layers are superimposed on one another, much like the animation cels used to create cartoons in the old days. Layers can also be used to introduce 2½D effects, where, as the camera pans, closer layers move faster than more distant layers.

Sprite

Sprites are the visual representation of the actors in our movie, whether those actors are people or objects. Sprites can be animated or not, but they often move about the scene during the course of game play. Sprite textures are often loaded from one large image that comprises a collection of sprite images, called a sprite sheet.

Entity

In AndEngine, entities are just about anything that's drawn to the screen. Sprites are entities, as are tiles, geometric shapes, and lines drawn on the screen. All entities have properties, such as color, rotation, scale and position, that can be changed by modifiers.

Modifier

Modifiers change the properties of an entity, and they are very powerful in AndEngine. They can be used on any entity, and the change they cause can either be immediate or occur gradually over a specified duration. In our game, we'll use modifiers frequently to create effects with sprites and other entities.

Texture

A texture is a 2D, generally bitmapped graphic that can be applied to objects to give them, well, texture. Textures define the way entities look, and much of the OpenGL graphics environment is built around the use of textures.

Texture Region

A texture defines a complete bitmap graphic, and a texture region defines a subset of that region. We'll talk a lot about performance optimizations of 2D graphics later, and using texture regions to map small pieces of a large combined bitmap is one of the key tricks used to create these optimizations.

Engine

An engine runs a scene. It takes care of letting animations and modifiers know when to update the presented graphics, coordinates the actual drawing, handles user input events (touch, keys, sensors), and generally manages the progress of the game. The engine is a lot like the producer/director of our movie, telling everyone what they need to do.

BaseGameActivity

This class, which extends the Android Activity class, will be the basis of each scene in our game. BaseGameActivity does all the work common to all scenes, setting up the game engine, conforming to the Android Activity Lifecycle requirements, and enabling sensors. We'll explore this class in more depth in Chapter 3.

Physics Connector

AndEngine includes basic physics capabilities in the base engine, but the Box2D physics engine expands greatly on those capabilities. We connect AndEngine objects with Box2D through a physics connector. If your game doesn't use Box2D physics, you won't have a physics connector.

Box2D Physics Engine

AndEngine includes the open-source JBox2D port of the Box2D physics engine. It can be used to realistically simulate the interaction of physical objects in the following ways (among others):

- Simulation of the physics of rigid bodies
- Stable stacking
- Gravity

- User-defined units
- Efficient solving for collisions/contacts
- Sliding friction
- Boxes, circles, and polygons
- Several joint types: distance, revolute, prismatic, pulley, gear, mouse
- Sleeping (removes motionless bodies from simulation until touched)

Graphics Tools

Creation of any video game requires the generation of quite a bit of graphics. Backgrounds have to be drawn, sprites rendered, and animations of some sprites created. Many tools are available to manage computer graphics, and the professional tools are incredibly sophisticated. If you are a graphic designer and know how to use Adobe Illustrator or other professional tools, you can safely skip this section.

In contrast, if you are like me, with little graphics experience (or talent) and less money to devote to game creation, this section will describe the (mostly free) tools that I used to create the graphics for V3. These tools are all very good and widely used by professionals. A wide variety of online support for using them is available as well. If you do end up using these "free" tools, please contribute what you can to the projects that create and maintain them. If you can help with coding, testing, bug fixes, support, or documentation, that's great. If you can't, please consider contributing money to help keep the projects going.

You may be tempted to acquire your game graphics directly from the wide variety of graphics available on the Internet. If you do, make sure you obtain the rights to use those graphics from their owners. Just because you can download a graphic doesn't mean you have the rights to use it. If you plan to charge money for your game, "commercial" use of someone else's graphics is a particularly thorny issue.

As you probably know, there are two main classes of graphic programs for drawing pictures: those that allow you to draw and manipulate graphical objects as vectors (sometimes referred to as "draw programs"), and those that let you create a bitmap of colors on a canvas (sometimes called "paint programs"). Each class has its uses, and both were used in creating V3.

Vector Graphics: Inkscape

I find drawing with vector graphics convenient primarily for two reasons:

- Each component of a drawing is treated as an object. Objects can be individually repositioned, scaled, rotated, and edited as you create the final drawing.
- Components and the whole drawing can be easily scaled without losing any resolution. This capability is particularly important for sprites, which tend to be

small in their final form. Scaling vector graphics is not without its pitfalls, so you should draw components as close to their final size as you can, but it's nice to have the flexibility to scale them when necessary.

Inkscape (`http://www.inkscape.org`) is a very popular vector drawing package. It is included with many Linux distributions and also runs on all variants of Windows and Mac OS X as well. The download includes extensive Help and tutorials. If you already know what you're doing, you'll be up and running very quickly. If you are closer to standing in my shoes (no experience, no talent), it will take you a little longer, but it's much quicker than trying to learn something like Adobe Illustrator. And did I mention that it's free? Figure 2.1 shows Inkscape open with a drawing of a bat.

The basic AndEngine game engine doesn't know how to render vector graphics, although we will explore an extension that does. In V3, my practice was to create graphics using Inkscape, edit them to my heart's content, scale them to an appropriate pixel size, save the vector (`.svg`) version, export them as a `.png` (Portable Network Graphics) file, and, if necessary, use GIMP (discussed in the next section) to create a transparent background. The resulting bitmap image (still in `.png` format) is just what basic AndEngine wants for image files.

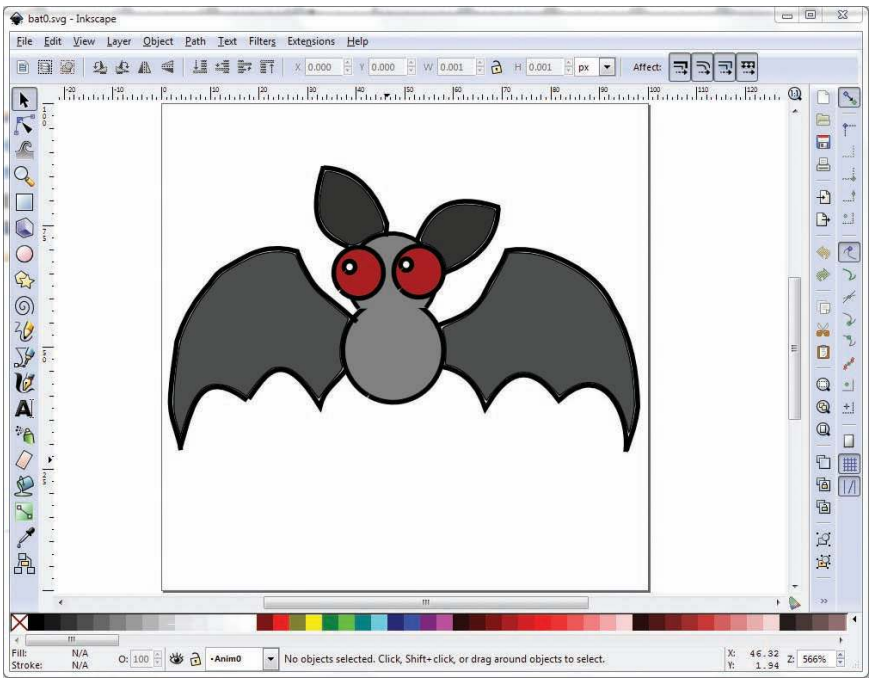


Figure 2.1 Inkscape vector drawing editor

Bitmap Graphics: GIMP

GIMP (GNU Image Manipulation Program; `http://www.gimp.org`) is a venerable cross-platform bitmap paint program used by many people worldwide. It ships with most Linux distributions, and it's free. The GIMP group itself does not support Windows and Mac OS X, but download packages are readily available for both of those operating systems.

Figure 2.2 shows the same image of a bat as appears in Figure 2.1. Here the bat is rendered as a bitmap (the rendering was done by saving the vector drawing as a `.png` file in Inkscape). The result, as you see, is a bit grainier at this size, with a transparent background, which is just what we want for a sprite animation cel. As we display the sprite, the background will show through the transparent (checkerboard) areas.

Animation Capture: AnimGet

Generating animations can be tedious, time-consuming work. You need to create a drawing for each cel of animation, for each viewpoint, and for each pose that you will be using in the game.

A neat shortcut is to create 3D animations, using one of the many 3D tools available (Blender is a widely used open-source tool; 3Ds Max, Poser, and Maya are some of the more popular commercial tools). You can then render 2D animations in each of the views you need. You'll end up with an AVI or animated GIF file, which you then

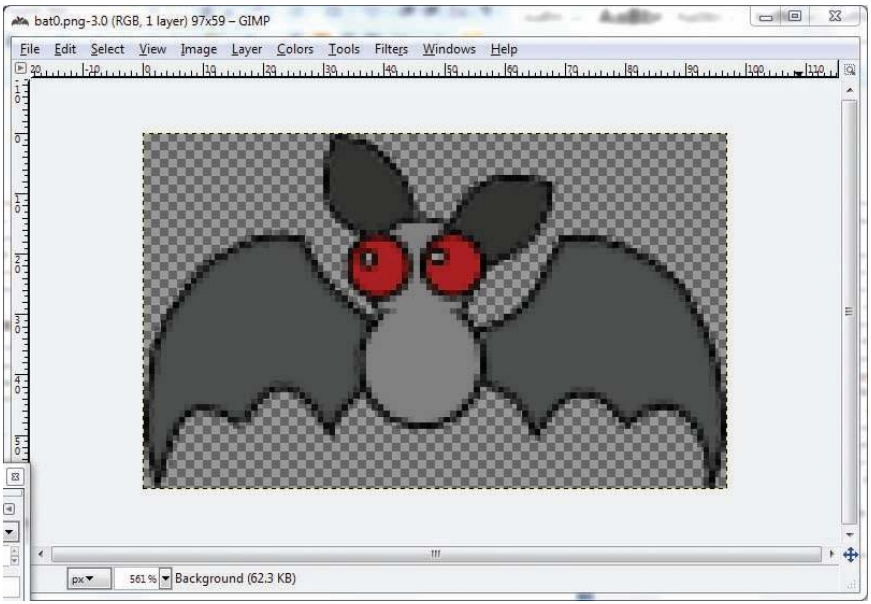


Figure 2.2 GIMP bit map graphics editor

need to break into separate frames (so you can reassemble them into a sprite sheet). Michael Menne created a utility called AnimGet that does the breakdown for you. The utility is available at several places on the Internet (just Google "AnimGet"), and it runs only under Windows.

The concept underlying AnimGet is ingenious. Rather than doing something complicated, such as parsing the animation file, this utility simply watches a defined area of your screen. It snaps a copy of the starting pixels and then comes back every 10 milliseconds to look again. If the pixels in the defined area have changed, AnimGet snaps a new copy of the area, and keeps it for a new file. The snapshots are all taken in memory, so the process can be fast, but you do need to be careful to choose the area so it contains only the animation you're interested in. When you tell AnimGet to stop capturing views, the images are written out to individual files.

TileMap Creation: Tiled

Tiles are often used in computer games to create a regular array of graphics, such as a map or, in the case of V3, the field for the invading vampires and obstacles. The Tiled map editor (available at `http://www.mapeditor.org/`), which is available for free, was written using the Qt cross-platform library, so it runs on Windows, Linux, and Mac OS X. We'll talk more about the way tiles are used in Chapter 9 on tiles, but Figure 2.3 shows what Tiled looks like in the middle of editing a tile map.

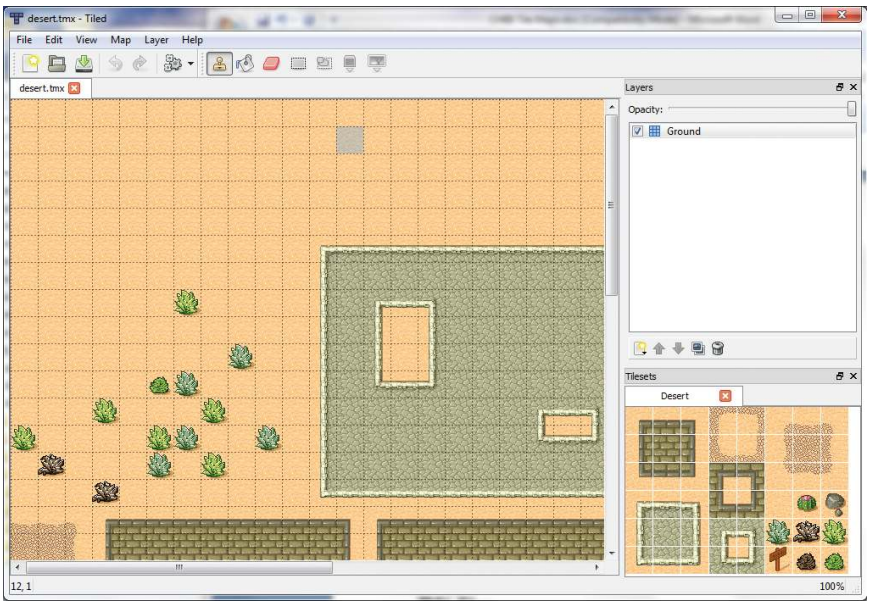


Figure 2.3 Tiled tile map editor

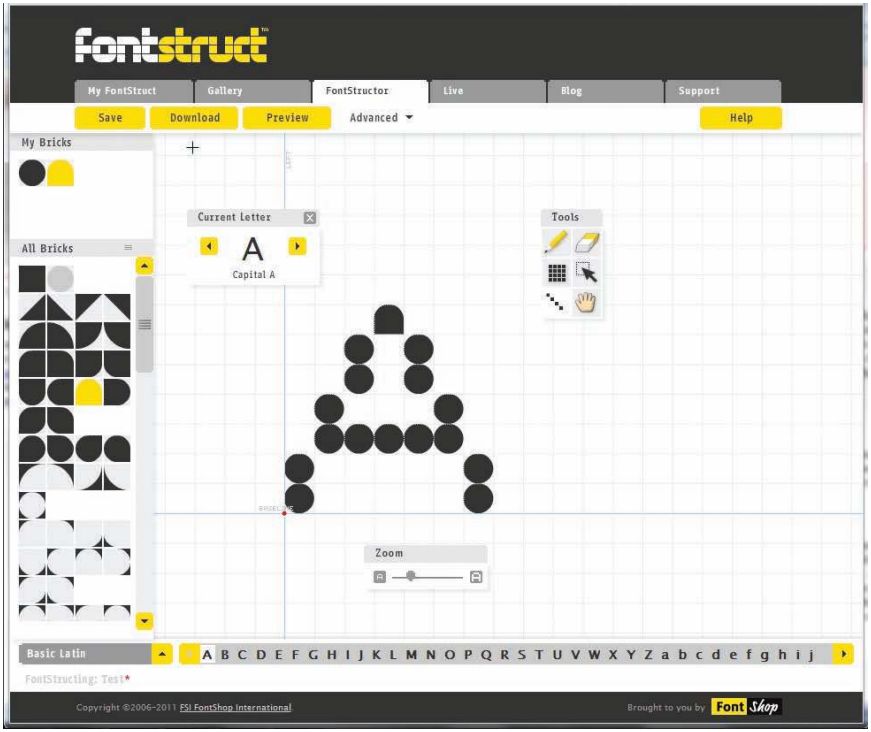


Figure 2.4 FontStruct TrueType font editor

TrueType Font Creation and Editing: FontStruct

Android supports the use of TrueType fonts, which we will use in our game. Many fonts are available (either for free or for purchase), along with tools to create and edit TrueType fonts. Notably, FontShop's web-based tool called FontStruct is offered through an open community for sharing the fonts you've created. Fonts are available for download, and the associated license (often Creative Commons) is shown for each font.

Creating your own font is a lot of work. I wouldn't recommend it unless you can't find a readymade font that will work for your game—or you like creating fonts. Figure 2.4 shows the FontStructor window of the website, where you create and edit your fonts.

Audio Tools

The V3 game will include two types of audio assets, and we need tools to create and edit both of them. Effects are short-duration sounds (5 seconds maximum, but typically less than 3 seconds) that accompany some event in the game. Music files are used to produce longer-duration sounds and are typically played in the background as a scene is being displayed and played.

Sound Effects: Audacity

What would a game be without sound effects? You'll want to create stunning sound effects for your game, and Audacity (`http://audacity.sourceforge.net/`) is a world-class sound editing tool that is freely available for Windows, Mac OS X, and Linux. Audacity lets you import or capture sound files, edit them, and output them in various audio formats. Android prefers AAC, MP3, MIDI, Ogg Vorbis, or WAV files, and Audacity can deal with all of them. Figure 2.5 shows the Audacity console editing a sound effect.

Background Music: MuseScore

Your game will also need background music, in the form of either an MP3 file, an OGG file, or a MIDI file. You could create the needed music in many ways, of course. For example, you might be talented enough to play the music on an instrument and record your performance. Alternatively, you might be able to find music files online that are free for commercial use.

Having minimal musical talent, I elected to use an open-source music creation package called MuseScore (`http://www.musescore.org`) for the V3 game. This software lets you create or edit music using either the computer keyboard or an attached MIDI keyboard. Figure 2.6 shows the main screen of MuseScore in the middle of editing a piece of music.

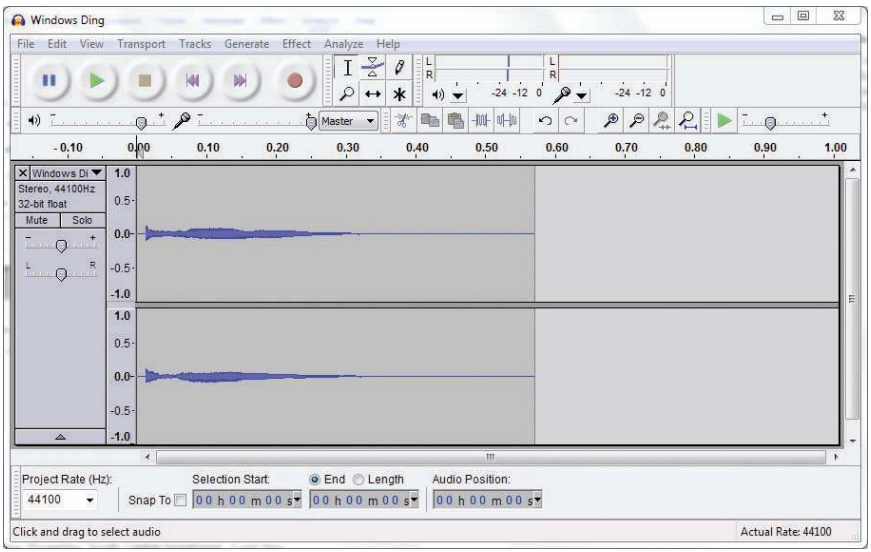


Figure 2.5 Audacity sound editor

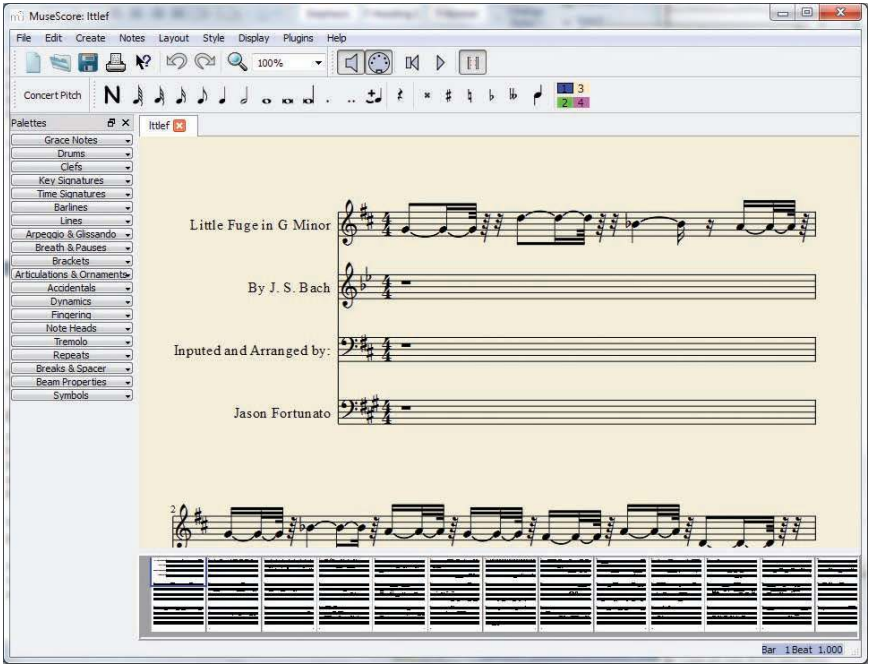


Figure 2.6 MuseScore

Once you've composed your opus, MuseScore lets you render and save the music as a MIDI, MP3, OGG, or WAV file, any of which Android knows how to play.

Getting Our Feet Wet: The Splash Screen

We've devoted a lot of space to game concepts and tools, but now it's time to start writing some code, so we can get into the development groove. The model will be that we create a basic Android project here and add to it in each of the following chapters until we've developed the complete game.

To get going, let's display the splash screen for the program (Figure 2.7). This screen appears for 5 seconds and then transitions to a blank screen.

Creating the Game Project

We use the Android SDK throughout this book. To begin working on the V3 game using the SDK, create a new Android Project. The New > Android Project dialog asks you for the following information:

1. Project Name: V3
2. Select "Create New Project in Workspace"
3. Use default workspace location (or any place you want)

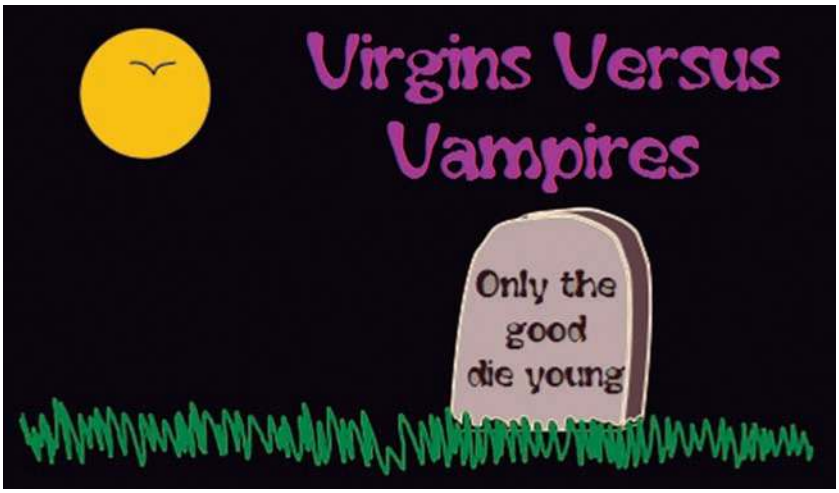


Figure 2.7 Splash screen for the example game, Virgins Versus Vampires

4. Select "Android 1.6" Build Target
5. Application Name: V3
6. Package Name: com.pearson.lagp.v3
7. Create Activity: StartActivity
8. Min SDK Version: 4

These steps create the project for the example game, including the StartActivity, which for now is a dummy "Hello, World" activity. We'll fix its content when we begin to add code in a later section in this chapter.

Adding the AndEngine Library

If you haven't already downloaded the AndEngine library from the AndEngine website, do so now. As of this writing, the best place to get the AndEngine `.jar` file was from the AndEngineExamples repository on Google:

`http://code.google.com/p/andengineexamples/`

Under the source tab, you can either browse for the `.jar` file (in the `lib` folder) or clone the entire source tree to your local machine. The latter approach is recommended, as that way you'll have ready access to the example code. AndEngine uses the Mercurial source code control system. If you don't already have Mercurial installed, instructions on the website explain how to do so. Once you have the `.jar` file on your machine, you then need to import the source:

- In the Eclipse Package Explorer, expand the V3 project (if it's not already expanded). Right-click on the V3 project, and create a `lib` folder by selecting New Folder and filling in the dialog.

- Right-click the `lib` folder and select "Import…" from the pop-up menu.
- Choose General > File System from the next dialog box, and use the Browse button to navigate to the directory where you downloaded the AndEngine `.jar` file. Click on the directory name (e.g., `…/AndEngineExamples/lib`).
- The Import dialog box should now show the directory with an empty check box. The right pane shows the files in the directory, which should include `andengine.jar`. Click on the check box next to the filename and then click Finish.
- To include the `.jar` file in your build path, right-click on the `.jar` file in the Eclipse Project Explorer pane, and then select Build Path > Add to Build Path. The `.jar` file will now also appear directly under the project folder.

Adding the Splash Screen Code

We need to modify the automatically generated `StartActivity.java` file so it displays the splash screen while game components are loaded. Eventually we'll want to move on to the game menu once everything has loaded, but for now we'll just leave the splash screen visible until the user returns to the Home screen. Listing 2.1 shows the modified version of `StartActivity.java`.

Listing 2.1 **StartActivity.java**

```
package com.pearson.lagp.v3;
import org.anddev.andengine.engine.Engine;
import org.anddev.andengine.engine.camera.Camera;
import org.anddev.andengine.engine.options.EngineOptions;
import org.anddev.andengine.engine.options.EngineOptions.
   ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy.
   RatioResolutionPolicy;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.entity.util.FPSLogger;
import org.anddev.andengine.opengl.texture.Texture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.ui.activity.BaseGameActivity;

public class StartActivity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;
```

```
// ===========================================================
// Fields
// ===========================================================

private Camera mCamera;
private Texture mTexture;
private TextureRegion mSplashTextureRegion;

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT),
      this.mCamera));
}

@Override
public void onLoadResources() {
   this.mTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   this.mSplashTextureRegion = TextureRegionFactory
      .createFromAsset(this.mTexture,
      this, "gfx/Splashscreen.png", 0, 0);
   this.mEngine.getTextureManager().loadTexture(this.mTexture);
}

@Override
public Scene onLoadScene() {
   this.mEngine.registerUpdateHandler(new FPSLogger());
   final Scene scene = new Scene(1);
   /* Center the splash on the camera. */
   final int centerX =
     (CAMERA_WIDTH - this.mSplashTextureRegion.getWidth()) / 2;
   final int centerY =
        (CAMERA_HEIGHT -
         this.mSplashTextureRegion.getHeight()) / 2;
   /* Create the sprite and add it to the scene. */
   final Sprite splash = new Sprite(centerX,
         centerY, this.mSplashTextureRegion);
   scene.getLastChild().attachChild(splash);
```

```
        return scene;
    }

    @Override
    public void onLoadComplete() {

    }

}
```

This may seem like a lot of code to display on one screen, but in AndEngine a splash screen is like any other sprite. Thus what we've really done is to set up the initialization of the entire game. Let's look at the code section by section:

- After the imports are complete, we immediately declare our class to be a subclass of BaseGameActivity. This class performs all of the common AndEngine initialization and provides us with some methods that we'll override later.
- We then set up the dimensions of the camera viewpoint. Because this is just a splash screen, we want to display the whole scene. The splash screen image, `Splashscreen.png`, is 480 × 320 pixels in size, corresponding to the default Android display dimensions.
- We next define some fields for the camera, the texture we will use, and the texture region.
- We now override three methods from the superclass BaseGameActivity. Each method corresponds to a point in the loading process:
  - The `onLoadEngine()` method is called when the game engine is loaded for this activity. We initialize the camera and the engine here. There are two things to note about the engine initialization. First, we set the screen orientation to LANDSCAPE. Most mobile games run in landscape orientation. Second, we asked for a Resolution Policy termed RatioResolutionPolicy. With this policy, the engine will adjust for different screen geometries by scaling features while maintaining the original aspect ratio.
  - The `onLoadResources()` method is called to load required resources. Here we create the Texture we will use and load it from the splash screen image, `Splashscreen.png`. That file must be found in the folder named `assets/gfx` in our project workspace, or the application will not run. Don't worry about the TextureOptions settings for now; we'll talk about them more in Chapter 5 on drawing and sprites. Finally, we load the texture into the engine's TextureManager.
  - The `onLoadScene()` method is called when the engine is ready for the scene. We initialize the frames per second logger, create the scene, center the camera on the scene, create the sprite for the splash, attach it to the scene, and return the scene to the engine. AndEngine actually provides a special type of scene for splash screens (called SplashScene), but the way we've handled the matter also gets the job done.

Once we've filled in the code needed in `StartActivity.java`, we need to import the graphics file for the splash into `assets/gfx`. This takes just a few steps:

- Right-click on the `assets` folder under the V3 project in Project Explorer. Select New > Folder, and create the subfolder `gfx`.
- Right-click on the new `gfx` subfolder and import the file `Splashscreen.png`. Having the image under the `assets` folder makes it available for the `createFromAsset()` method we used to load it into a Texture.

Running the Game in the Emulator

Running the game in the Android Emulator is as simple as choosing Run > Run from the Eclipse menu. If you're prompted to choose an AVD, do so; you should then see the splash screen displayed in an emulator window. If you need to rotate the emulator to landscape orientation, pressing the key combination Ctrl + F12 will achieve that effect. If the application died with a forced close, you are probably missing one of the needed assets. LogCat will tell you exactly what was not found.

Running the Game on an Android Device

You'll need to consult your device manufacturer's developer site to get it properly connected for Android development work. Just make sure that the version of Android running on your device is at least as high as the one you picked when creating the project (in the example, we picked API Level 4, which is equivalent to Android 1.6).

Now when you choose Run > Run from the Eclipse menu, you should have the option of running on the device itself. The Android SDK will install the application on the device and start it running. You can use the Back or Home buttons to return to the home screen, but as with all Android applications, your game will keep running in the background. It's not actually doing much, because the display is not visible, and Android is smart enough to not spend cycles updating an invisible screen, but the game is still burning some cycles. To actually stop or uninstall the application, use the Settings > Application > Manage Applications dialog on your device.

Summary

This chapter covered a lot of ground. We took a quick look at the Android SDK and were introduced to Android and the components of a game:

- Camera
- Scene
- Layer
- Sprite
- Entity
- Modifier

- Texture
- Texture region
- Engine
- BaseGameActivity
- Physics connector

We looked at a collection of tools that can be used to generate the components of a mobile game, including the following:

- Inkscape: for scalable vector graphics
- GIMP: for bitmap graphics
- AnimGet: for animation capture
- Tiled: for editing tile maps
- Audacity: for audio editing
- MuseScore: for creating music

Finally, we started putting together the V3 game. We created an Android Project using the Android SDK Wizard, added the AndEngine library, added code to perform appropriate AndEngine initialization, and displayed a splash screen for the game.

Exercises

1. Use Inkscape, GIMP, or your favorite graphics tool to design your own splash screen for your own game. You can start with vector graphics or a bitmap image (whichever you prefer), but you want to end up with a rectangular image in PNG format, 480 pixels wide by 320 pixels high. Substitute your splash screen in `StartActivity.java` and run the "game."
2. Following the instructions on the repository, clone the AndEngine and AndEngine-Examples repositories onto your development computer. You'll need the Mercurial source code management system (which itself is open source) to handle the cloning process. Together the two repositories will take around 40MB of space.
3. In Chapter 1, you drew a storyboard for your game. For each scene in your storyboard, write down a list of the elements:
   - Layers
   - Sprites, noting animations
   - Textures
   - Sounds
4. Find a musical theme for your game. Decide whether you will pay someone for music they've created or whether you will create your own music.

3

The Game Loop and Menus

The game loop is often called the heartbeat of a computer game, and indeed it is what gives life to our games. As a game is being played, inputs to the game, status, and outputs all need to proceed in a regular cycle. The game loop is the implementation of that cycle, so an efficient, high-performance game loop is essential for a playable computer game.

Game Loops in General

Computer games are all based on a cycle of operations. The typical steps in such a cycle include the following actions:

- Acquire input from the player.
- Update the game status as a result of user input.
- Update the game status as a result of time.
- Check for collisions among objects.
- Update objects based on game physics.
- Update the user display based on the updated game status.
- Render the updated display.
- Play appropriate sounds.

In this list of steps, "game status" includes not only variables such as the level the player is on and the current score, but also the position and animation of every object being displayed. For games with many objects, getting through this loop more quickly than the desired animation frame rate can be a daunting task.

Depending on the nature of the game, time may or may not be a factor in the cycle. Board and puzzle games may or may not have a time element, for example. If time is used in these types of games, it often takes the form of something like the time needed to solve the puzzle, which then becomes the player's score. The player uses the user interface (UI) to make a move in the game; in turn, the computer calculates a response and updates the game status to reflect the computer's move. The result is then

displayed to the user, and the cycle continues. The computer is really concerned only with tracking the total time taken to achieve the goal.

In games like V3, in which animations and entities other than the player are active in the game, time is more of a factor. The game must now regularly update the user interface, including the positions of entities and the animations shown, based on time. It has to accomplish this feat predictably on a variety of hardware platforms characterized by different clock speeds, different computing capabilities, and different UI hardware. It also has to perform the updates on schedule despite the completion of occasional garbage collection operations. And if the game is going to be fun to play, it has to be efficient in performing the updates, such that many objects can be updated each animation cycle.

Getting everything right is hard to do, but luckily for us, AndEngine is ready to perform the work for us. All we need to do is use the game loop that comes embedded in the AndEngine library.

The Game Loop in AndEngine

AndEngine includes a component called Engine, which works with the Android runtime to implement the game loop for our games. The Engine is the center of activity from the moment we start the game until the moment we shut it down. One of the first things that happens in an AndEngine game is the override of the `onLoadEngine()` method, as shown in Listing 3.1. In our override, we create the Engine for our game and set the optional parameters of that Engine.

Listing 3.1 **Engine Initialization**

```
. . .
   @Override
   public Engine onLoadEngine() {
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
            CAMERA_HEIGHT), this.mCamera));
. . .
```

The Engine does a number of things for game developers:

- It directs the initialization and ongoing maintenance of the OpenGL SurfaceView that we use to draw to the screen.
- It manages user input from the keyboard and touch screen.
- It manages sensor inputs from the accelerometer and orientation (compass direction) sensors.
- It manages the library of text fonts.

- It creates and manages the loading and playing of both sound effects and music.
- It manages the interface to the device's vibrator.
- It updates the rest of AndEngine periodically to advance the state of the game that is being played.

Engine Initialization

The default constructor for the Engine object is

```
public Engine(final EngineOptions pEngineOptions)
```

The EngineOptions class has the following constructor:

```
public EngineOptions(final boolean pFullscreen,
        final ScreenOrientation pScreenOrientation,
        final IResolutionPolicy pResolutionPolicy, final Camera pCamera)
```

The parameters passed to new `EngineOptions()` tell AndEngine how we want to set up the Engine for this game:

- `pFullscreen`: we will almost always set true, as we want the game to take the full device screen.
- `ScreenOrientation` can take one of two values:
  - LANDSCAPE: which is the preferred orientation for mobile games
  - PORTRAIT: in case we need that orientation
- `pResolutionPolicy` tells Engine how you want to deal with different screen geometries:
  - `RatioResolutionPolicy` says to expand the graphics to fill as much of the screen as possible, while maintaining the original aspect ratios of the graphics. This is the approach that we will always use in this book.
  - `FillResolutionPolicy` says to fill the screen, ignoring aspect ratios.
  - `FixedResolutionPolicy` says to use the original fixed dimensions, and not try to scale the graphics to fit different screen sizes.
  - `RelativeResolutionPolicy` says to use a fixed scale on the original dimensions. You could use this policy if you wanted to optimize for specific devices.

`pCamera` is the camera object that determines the view the player sees of the scenes. The constructor for Camera is

```
public Camera(final float pX, final float pY, final float pWidth,
        final float pHeight)
```

where `pX` and `pY` are the coordinates for the origin, and `pWidth` and `pHeight` are the dimensions of the viewable scene. All of the dimensions are in pixels.

Other Engines

In this book, we will always use the default Engine described previously, but you could also use several other Engines for your own games if you wish. All of these are subclasses of the default Engine, so you inherit all the base characteristics of that class.

FixedStepEngine

```
FixedStepEngine(EngineOptions pEngineOptions, int pStepsPerSecond)
```

The default Engine executes the game loop as fast as it can and then starts the loop again immediately. This strategy provides the highest step and frame rates possible, and the Engine keeps track of elapsed time so it can progress translations and other operations fluidly and consistently. You might still want to use a fixed-step engine if, for some reason, you needed the game to progress in fixed time steps, no matter which device was running it.

LimitedFPSEngine

```
LimitedFPSEngine(EngineOptions pEngineOptions, int pFramesPerSecond)
```

As mentioned previously, the default Engine will run as fast as it can, resulting in a variable frame rate. You can have AndEngine log the frame rate to LogCat (details appear in a later example in this chapter) if you like, and the Engine takes account of the actual elapsed time between frames when computing the positions of moving objects and the like. If for some reason you want to use a fixed frame rate, no matter which device the game is running on, LimitedFPSEngine will try to meet that requirement. This subclass is called "Limited" instead of "Fixed" because it might not be able to achieve the desired frame rate on a given device, so it is really setting an upper limit on frame rate.

SingleSceneSplitScreenEngine

```
SingleSceneSplitScreenEngine(EngineOptions pEngineOptions,
    Camera pSecondCamera)
```

The default Engine displays one scene at a time, and has one camera view of that scene. Some games may benefit from having two views of the current scene displayed in separate windows on a split screen, and that's what this Engine does for you. For example, a two-player, first-person shooter game might show two scenes, each assuming one of the players' current point of view.

DoubleSceneSplitScreenEngine

```
DoubleSceneSplitScreenEngine(EngineOptions pEngineOptions,
    Camera pSecondCamera)
```

If your game needs a split screen showing separate scenes, then this is the Engine you want to use. For example, suppose you are developing a first-person shooter game where one scene shows the current player's view and another shows a global aerial view of the whole battle scene. The "first" scene is shown in the left-hand part of the split, while the "second" is shown in the right-hand split.

Adding a Menu Screen to V3

Next, we would like to add an opening menu screen to the game that we've been developing. Two menu architectures are available to us. Android has a MenuView, of course, and a whole set of APIs around building and displaying menus, responding to the Menu button, and accepting menu inputs. AndEngine provides an alternative menu system, which integrates the menus into the game. This system includes text menus and graphical menus, both using the same OpenGL interfaces that are used to draw the rest of a game. We'll look at both of those options in this section, as well as ways to capture the Menu button keypresses to trigger a pop-up menu presentation. The Android MenuView system is effectively disabled while a scene is being shown. You actually could still use the Android MenuView layered on the SurfaceView that AndEngine uses, but that would be pretty confusing for your players.

Menus in AndEngine

Menus are a special type of Scene in AndEngine. They're special because they arrange text or graphics in an ordered list, and they accept touch inputs from the player to select one of the items on the list. They also provide for animation of the menu items as the menu is being displayed.

MenuScene.java

The `MenuScene.java` class is a subclass of Scene, and the one we use when we want to instantiate a menu scene. There are four constructors for the class:

- `MenuScene()`
- `MenuScene(final Camera pCamera)`
- `MenuScene(final IOnMenuItemClickListener pOnMenuItemClickListener)`
- `MenuScene(final Camera pCamera, final IOnMenuItemClick-Listener pOnMenuItemClickListener)`

The parameters are as follows:

- `pCamera`: the camera to use when displaying the scene. This is usually the same Camera we declared in `onLoadEngine()`, but it can be any Camera.
- `pOnMenuItemClickListener()`: a method to receive notification when the user clicks on an item.

The values for `pCamera` and `pOnMenuItemClickListener()` can also be set via getter/setter methods. The class also includes methods that work with the class instance variables:

- `mMenuItems`: an `<Array List>` of menu items
- `mMenuAnimator`: an animator to use when the scene is displayed

Menu Items and Scenes Attached to MenuScene

The methods are as follows:

- `addMenuItem(final IMenuItem pMenuItem)`

  This method adds `pMenuItem` to the list of menu items to be displayed and clicked on.

- `getMenuItemCount()`

  This method returns the number of menu items on the list.

- `setChildScene(final Scene pChildScene, final boolean pModalDraw, final boolean pModalUpdate, final boolean pModalTouch)`, `getChildScene()`, `clearChildScene()`

  This method addresses a scene that can be attached to the menu scene, as a child of the scene. We'll go into more detail about these methods when we talk about scenes in Chapter 4.

- `setMenuAnimator(final IMenuAnimator pMenuAnimator)`

  This method sets the menu animator to be used with the menu scene.

TextMenuItem.java

This class defines menu items composed of text. It has a single constructor:

TextMenuItem(final int pID, final Font pFont, final String pText)

The parameters to the constructor are as follows:

- `pID`: a unique integer ID that you can use to identify the menu item in the `onClick()` callback, or anywhere else required.
- `pFont`: the font you want to use when displaying the text. We'll delve further into fonts in Chapter 7. For now, simply recognize that this parameter identifies the typeface (e.g., Courier), the size, and any style.
- `pText`: the text to be displayed for this menu item.

SpriteMenuItem.java

This class defines graphical menu items, and the sprite that is displayed for them. It also has a single constructor:

SpriteMenuItem(final int pID, final TextureRegion pTextureRegion)

This constructor has the following parameters:

- `pID`: unique integer ID; the same as for TextMenu items.
- `pTextureRegion`: the sprite texture to be used to display the menu item. We talk a lot more about textures and texture regions in Chapter 5, where we cover sprites.

AnimatedSpriteMenuItem.java

The third class encompasses menu items that are displayed as animated sprites. It has a single constructor:

```
AnimatedSpriteMenuItem(final int pID,
        final TiledTextureRegion pTiledTextureRegion)
```

The parameters are similar to those for `SpriteMenuItem.java`:

- `pID`: unique integer ID.
- `pTiledTextureRegion`. We'll see in Chapter 6 that tiled texture regions are used for animated sprites; this parameter tells AndEngine what to display for this menu item.

ColorMenuItemDecorator.java

Menu item decorators change the appearance of a menu item briefly when it is selected. As you might expect, this class changes the item's color. It has one constructor:

```
ColorMenuItemDecorator(final IMenuItem pMenuItem,
        final float pSelectedRed, final float pSelectedGreen,
        final float pSelectedBlue, final float pUnselectedRed,
        final float pUnselectedGreen, final float pUnselectedBlue)
```

That long string of parameters looks pretty horrible until you realize it's just the menu item to be modified and two colors—one for the item when it is selected by the user and one for the same item when it is unselected. Color values are usually represented in AndEngine as a series of three floating-point numbers, one each for red, blue, and green. The floating-point values may vary from 0.0f (for the color not present) to 1.0f for color fully present. See the example code later in this chapter to see a demonstration of how ColorMenuItemDecorator is used.

ScaleMenuItemDecorator.java

Another way to indicate on the display that a menu item has been selected is to change the item's size briefly. That's what this class does, and as you've probably guessed, it has one constructor:

```
ScaleMenuItemDecorator(final IMenuItem pMenuItem,
        final float pSelectedScale, final float pUnselectedScale)
```

We pass three parameters to this constructor:

- `pMenuItem`: the menu item to be decorated
- `pSelectedScale`: the size multiplier when the item is selected
- `pUnSelectedScale`: the size multiplier when the item is unselected (usually 1.0f)

Building the V3 Opening Menu

We'll start with a text version of the main menu for V3 and will include a small graphical pop-up menu attached to the Menu button. The main (static) menu will provide options for the main functions the game player would want. The pop-up menu will display the About page, and the pop-up will include a Quit option (although the Back button does exactly the same thing). To demonstrate menu animation, we'll slide the pop-up menu in from the left. Figure 3.1 shows a screenshot of the main menu, without the pop-up.

Creating the Menu

We need to do two things to add the main menu:

1. Create the menu scene that we want to show.
2. Modify StartActivity so the splash screen is displayed for a few seconds, followed by the menu scene.

To create the menu scene, we will create a new Activity called MainMenuActivity. In that Activity, we'll make use of a subclass of the AndEngine Scene class called MenuScene. The code for the scene is shown in Listing 3.2.

Figure 3.1 V3 opening menu

Listing 3.2 **MainMenuActivity.java**

```
package com.pearson.lagp.v3;


+imports

public class MainMenuActivity extends BaseGameActivity implements
IOnMenuItemClickListener {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;

   protected static final int MENU_ABOUT = 0;
   protected static final int MENU_QUIT = MENU_ABOUT + 1;
   protected static final int MENU_PLAY = 100;
   protected static final int MENU_SCORES = MENU_PLAY + 1;
   protected static final int MENU_OPTIONS = MENU_SCORES + 1;
   protected static final int MENU_HELP = MENU_OPTIONS + 1;

   // ===========================================================
   // Fields
   // ===========================================================

   protected Camera mCamera;

   protected Scene mMainScene;

   private Texture mMenuBackTexture;
   private TextureRegion mMenuBackTextureRegion;

   protected MenuScene mStaticMenuScene, mPopUpMenuScene;

   private Texture mPopUpTexture;
   private Texture mFontTexture;
   private Font mFont;
   protected TextureRegion mPopUpAboutTextureRegion;
   protected TextureRegion mPopUpQuitTextureRegion;
   protected TextureRegion mMenuPlayTextureRegion;
   protected TextureRegion mMenuScoresTextureRegion;
   protected TextureRegion mMenuOptionsTextureRegion;
   protected TextureRegion mMenuHelpTextureRegion;
   private boolean popupDisplayed;
```

```
// =============================================================
// Constructors
// =============================================================

// =============================================================
// Getter and Setter
// =============================================================

// =============================================================
// Methods for/from SuperClass/Interfaces
// =============================================================

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
      CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
   /* Load Font/Textures. */
   this.mFontTexture = new Texture(256, 256,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);

   FontFactory.setAssetBasePath("font/");
   this.mFont = FontFactory.createFromAsset(this.mFontTexture,
      this, "Flubber.ttf", 32, true, Color.RED);
   this.mEngine.getTextureManager().loadTexture(this.mFontTexture);
   this.mEngine.getFontManager().loadFont(this.mFont);

   this.mMenuBackTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   this.mMenuBackTextureRegion =
      TextureRegionFactory.createFromAsset( this.mMenuBackTexture,
      this, "gfx/MainMenu/MainMenuBk.png", 0, 0);
   this.mEngine.getTextureManager().loadTexture(this.mMenuBackTexture);

   this.mPopUpTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   this.mPopUpAboutTextureRegion =
      TextureRegionFactory.createFromAsset(this.mPopUpTexture,
      this, "gfx/MainMenu/About_button.png", 0, 0);
```

```
    this.mPopUpQuitTextureRegion =
       TextureRegionFactory.createFromAsset(this.mPopUpTexture,
       this, "gfx/MainMenu/Quit_button.png", 0, 50);
    this.mEngine.getTextureManager().loadTexture(this.mPopUpTexture);
    popupDisplayed = false;
    }

@Override
public Scene onLoadScene() {
    this.mEngine.registerUpdateHandler(new FPSLogger());

    this.createStaticMenuScene();
    this.createPopUpMenuScene();

    /* Center the background on the camera. */
    final int centerX = ( CAMERA_WIDTH -
          this.mMenuBackTextureRegion.getWidth()) / 2;
    final int centerY = (CAMERA_HEIGHT -
          this.mMenuBackTextureRegion.getHeight()) /
             2;

    this.mMainScene = new Scene(1);
    /* Add the background and static menu */
    final Sprite menuBack = new Sprite(centerX,
       centerY, this.mMenuBackTextureRegion);
    mMainScene.getLastChild().attachChild(menuBack);
    mMainScene.setChildScene(mStaticMenuScene);

    return this.mMainScene;
}

@Override
public void onLoadComplete() {
}

@Override
public boolean onKeyDown(final int pKeyCode,
    final KeyEvent pEvent) {
    if(pKeyCode == KeyEvent.KEYCODE_MENU &&
       pEvent.getAction() == KeyEvent.ACTION_DOWN) {
       if(popupDisplayed) {
          /* Remove the menu and reset it. */
          this.mPopUpMenuScene.back();
          mMainScene.setChildScene(mStaticMenuScene);
          popupDisplayed = false;
       } else {
```

```
            /* Attach the menu. */
            this.mMainScene.setChildScene(
               this.mPopUpMenuScene, false, true, true);
               popupDisplayed = true;
            }
            return true;
         } else {
            return super.onKeyDown(pKeyCode, pEvent);
         }
      }

      @Override
      public boolean onMenuItemClicked(final MenuScene pMenuScene,
            final IMenuItem pMenuItem,
            final float pMenuItemLocalX,
            final float pMenuItemLocalY) {
         switch(pMenuItem.getID()) {
            case MENU_ABOUT:
               Toast.makeText(MainMenuActivity.this,
               "About selected",
               Toast.LENGTH_SHORT).show();
               return true;
            case MENU_QUIT:
               /* End Activity. */
               this.finish();
               return true;
            case MENU_PLAY:
               Toast.makeText(MainMenuActivity.this,
               "Play selected", Toast.LENGTH_SHORT).show();
               return true;
            case MENU_SCORES:
               Toast.makeText(MainMenuActivity.this,
               "Scores selected",
               Toast.LENGTH_SHORT).show();
               return true;
            case MENU_OPTIONS:
               Toast.makeText(MainMenuActivity.this,
               "Options selected",
               Toast.LENGTH_SHORT).show();
               return true;
            case MENU_HELP:
               Toast.makeText(MainMenuActivity.this,
               "Help selected", Toast.LENGTH_SHORT).show();
               return true;
            default:
               return false;
         }
      }
```

```
// ===========================================================
// Methods
// ===========================================================

protected void createStaticMenuScene() {
   this.mStaticMenuScene = new MenuScene(this.mCamera);
   final IMenuItem playMenuItem = new ColorMenuItemDecorator(
      new TextMenuItem(MENU_PLAY, mFont, "Play Game"),
      0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
   playMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
       GL10.GL_ONE_MINUS_SRC_ALPHA);
   this.mStaticMenuScene.addMenuItem(playMenuItem);

   final IMenuItem scoresMenuItem =
      new ColorMenuItemDecorator(
      new TextMenuItem(MENU_SCORES, mFont, "Scores"),
      0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
   scoresMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
       GL10.GL_ONE_MINUS_SRC_ALPHA);
   this.mStaticMenuScene.addMenuItem(scoresMenuItem);

   final IMenuItem optionsMenuItem =
      new ColorMenuItemDecorator(
      new TextMenuItem(MENU_OPTIONS, mFont, "Options"),
      0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
   optionsMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
       GL10.GL_ONE_MINUS_SRC_ALPHA);
   this.mStaticMenuScene.addMenuItem(optionsMenuItem);

   final IMenuItem helpMenuItem = new ColorMenuItemDecorator(
      new TextMenuItem(MENU_HELP, mFont, "Help"),
      0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
   helpMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
       GL10.GL_ONE_MINUS_SRC_ALPHA);
   this.mStaticMenuScene.addMenuItem(helpMenuItem);
   this.mStaticMenuScene.buildAnimations();

   this.mStaticMenuScene.setBackgroundEnabled(false);

   this.mStaticMenuScene.setOnMenuItemClickListener(this);
}

protected void createPopUpMenuScene() {
   this.mPopUpMenuScene = new MenuScene(this.mCamera);

   final SpriteMenuItem aboutMenuItem =
      new SpriteMenuItem(MENU_ABOUT,
         this.mPopUpAboutTextureRegion);
```

```
        aboutMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
           GL10.GL_ONE_MINUS_SRC_ALPHA);
        this.mPopUpMenuScene.addMenuItem(aboutMenuItem);
        final SpriteMenuItem quitMenuItem = new SpriteMenuItem(
              MENU_QUIT, this.mPopUpQuitTextureRegion);
        quitMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
           GL10.GL_ONE_MINUS_SRC_ALPHA);
        this.mPopUpMenuScene.addMenuItem(quitMenuItem);
        this.mPopUpMenuScene.setMenuAnimator(
           new SlideMenuAnimator());

        this.mPopUpMenuScene.buildAnimations();

        this.mPopUpMenuScene.setBackgroundEnabled(false);

        this.mPopUpMenuScene.setOnMenuItemClickListener(this);
    }

    // ===========================================================
    // Inner and Anonymous Classes
    // ===========================================================
}
```

This is a lot of code at one whack, but we'll break it down into sections to see what's going on. Many of the topics are covered in more detail later in the book, so we'll just introduce them here.

MainMenuActivity

After the imports, we declare the MainMenu Activity class to be a subclass of BaseGameActivity, as usual. We also say that it implements the IOnMenuItemClickListener interface, which we need to respond to user clicks on the menu.

Constants and Fields

We then define a bunch of constants. The `MENU_xxx` constants are IDs that we'll use later to identify the different menu items, both when we create them and in the click listener. We've separated them into two groups (one for the static menu and one for the pop-up), but that's really an arbitrary decision.

onLoadResources()

After declaring the fields we'll use in the Activity, we specify the usual set of override methods. This version of `onLoadEngine()` looks exactly like the version we used with StartActivity, demonstrating the pattern that we will use for most of the Activities we create in the book.

In `onLoadResources()`, we want to load the textures and anything else we need in the way of resources. Given that we plan to display a text menu, we'll need a font for the text; thus the first thing we do is load a font. We create a Texture to hold the font texture, and use `FontFactory.createFromAsset()` to actually load the font into a Font object. Before compiling the game, we create a subfolder of assets called font and import a TrueType font file there, `Flubber.ttf`. In `createFromAsset()`, we tell the method that we want to load the Flubber font and have it render text that is 32 pixels high and colored in red. Chapter 7 provides a lot more information on fonts and text in general.

Still in `onLoadResources()`, we create a Texture and TextureRegion for the background of the static menu scene. As shown in Figure 3.1, our menu includes a black background, the words "Choose Your Poison," and a strip of grass on the bottom. We created a PNG image of that screen separately and stored it in a file, `MainMenuBk.png`, which we then imported into `assets/gfx`. We load the Texture into the Texture Manager. We'll explore Textures and the Texture Manager in depth in Chapters 4 and 5.

We also create and load a Texture and TextureRegions for the two items in the pop-up menu. Because our game will use a graphical menu, we load the Textures from PNG files we've prepared and imported for each of the menu buttons.

onLoadScene()

We're also overriding `onLoadScene()`. To do so, we first enable the frames per second logger, just as we did in StartActivity. We then call two methods to create the two menu scenes—one for the static menu that appears when the scene is first displayed, and one for the scene that we'll superimpose if the user presses the Menu key. We create the MainScene that we'll hang everything off of and center the camera on it. Finally, we create a sprite for the background of the scene, attach the sprite to the scene, and set the static menu scene as a child scene of MainScene.

createMenuScene() and createPopUpScene()

Before we look at `onKeyDown()` or `onMenuItemClicked()`, let's skip down to the menu scene creation methods—that is, `createStaticMenuScene()` and `createPopUpScene()`. For `createStaticMenuScene()`, we begin by creating the scene and then create four TextMenu Item objects for the four menu options. We pass the constructors an ID that we can use to identify the item when clicked, a font to use, and a string to display. At this point, the font that we loaded in `onLoadResources()` comes in handy. We tell OpenGL how to blend the rendered text menu item with the rest of the scene, and add each item to the menu scene. We call two methods—`buildAnimations()` and `setBackgroundEnabled()`—that make the scene display the way we want it, and add `onMenuItemClicked()` as the callback method to be used when the user clicks on the menu item.

The `createPopUpMenuScene()` method is very similar, except that now we're creating a graphical menu. We create SpriteMenu Items and load them with Textures that we loaded earlier under `onLoadResources()`.

onKeyDown() and onMenuItemClicked()

Now that our menu scenes are set up, let's look at the two methods that respond to user actions—`onKeyDown()` and `onMenuItemClicked()`.

`onKeyDown()` checks whether the user pressed the Menu key. If not, it passes the key back to Android for further processing. The user uses the Menu key to both display and remove the pop-up menu. Thus, if the Menu key was pressed, `onKeyDown()` determines whether the pop-up menu is being displayed. If it is being displayed, this menu is removed and the static menu is again set as the child scene of the main scene. If the pop-up is not being displayed, it is added to the main scene.

`onMenuItemClicked()` is a switch statement using the IDs that we created for each of the menu items. We can use one method and one switch statement for both menus, as the IDs are unique. For now, we cannot take the player to any other scenes, so we just display a Toast for each menu item. The exception is Quit, which doesn't really quit the application; instead, it just ends this Activity and takes us back to the splash screen.

Splash to Menu

The next thing we need to do is modify the game so that the splash screen displays for 3 seconds and moves on to the menu scene. Eventually we'll want to use the splash display time to load graphics and sounds, but we don't have any to load yet. Listing 3.3 shows the modified version of `StartActivity.java`.

Listing 3.3 **StartActivity.java**

```
package com.pearson.lagp.v3;

+imports

public class StartActivity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;

   // ===========================================================
   // Fields
   // ===========================================================
```

```
private Camera mCamera;
private Texture mTexture;
private TextureRegion mSplashTextureRegion;
private Handler mHandler;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   mHandler = new Handler();
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
   this.mTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   this.mSplashTextureRegion =
      TextureRegionFactory.createFromAsset(this.mTexture,
         this, "gfx/Splashscreen.png", 0, 0);
   this.mEngine.getTextureManager().loadTexture(this.mTexture);
}

@Override
public Scene onLoadScene() {
   this.mEngine.registerUpdateHandler(new FPSLogger());

   final Scene scene = new Scene(1);

   /* Center the splash on the camera. */
   final int centerX = (CAMERA_WIDTH -
          this.mSplashTextureRegion.getWidth()) / 2;
```

```
        final int centerY = (CAMERA_HEIGHT -
                this.mSplashTextureRegion.getHeight()) / 2;

        /* Create the sprite and add it to the scene. */
        final Sprite splash = new Sprite(centerX, centerY,
                  this.mSplashTextureRegion);
        scene.getLastChild().attachChild(splash);
        return scene;
    }

    @Override
    public void onLoadComplete() {
        mHandler.postDelayed(mLaunchTask,3000);
    }

     private Runnable mLaunchTask = new Runnable() {
          public void run() {
            Intent myIntent = new Intent(StartActivity.this,
                 MainMenuActivity.class);
            StartActivity.this.startActivity(myIntent);
          }
       };
    // ===========================================================
    // Methods
    // ===========================================================

    // ===========================================================
    // Inner and Anonymous Classes
    // ===========================================================
}
```

All we've done is ask Android to post a delayed Intent to start MainMenuActivity after 3 seconds. AndEngine treats scenes as normal Android Activities—one of the advantages of doing so is that we can invoke the scenes like we would any other Activity.

If you now run the application (either in the emulator or on a real phone), the splash screen will display for 3 seconds, and then the static menu will appear. If you touch one of the menu items, a Toast will appear confirming your selection. If you press the Menu key, the About and Quit buttons will be displayed. If you click on About, you will get another Toast. If you click on Quit, you will be taken back to the splash screen, where you will remain forever; `onLoadComplete()` is not called because the Activity is already loaded.

Memory Usage

This is a good opportunity to take a minute and mention a topic that is important to any mobile software design—namely, memory usage and garbage collection. If

you've developed any mobile software at all, you know that one key issue differentiates desktop and mobile applications: Resources are limited on a mobile device. Resources include battery power, memory, permanent storage, screen size, and processor cycles, among other things. As we create our mobile game, we need to constantly be thinking of ways to conserve those resources.

For example, memory usage and garbage collection are a big deal for Android mobile games. As you know, our Java code executes on the Dalvik virtual machine. When it runs out of available memory, the virtual machine calls a garbage collector to recycle objects that are no longer referenced. That action solves most of our potential memory leak problems, but it also introduces a delay in the execution of the game while the garbage collector does its thing. Naturally, we want to minimize garbage collection as much as possible, which we can do by reusing as many objects as possible. If you look at the source code for AndEngine, you'll see that it makes very good use of pools of objects that it assigns for use and then recycles. That approach isn't a panacea for reducing garbage collection, but when used wisely it can be very beneficial.

The Quit Option

Now I have to share a dirty little secret. Well, maybe not so dirty, but it's about the "Quit" option that we just put in that last menu. As you may know, Android applications never have a true "Quit" option. When a user wants to stop (really suspend) an application, the Android standard calls for the user to press the "Back" button, at which point Android displays the user's home screen. If this all sounds like Greek to you, take a look at the Android Developer documentation (`http://developer.android.com`). Search for "Activity Lifecycle." The application gets a chance to shut down any services used and stop any timers in the current Activity's `onPause()` method.

In keeping with Android practices, moving forward we will eliminate the "Quit" option from the menu.

Summary

The chapter began by describing game loops in general, then focused on the way the game loop is implemented in AndEngine. For the most part, the AndEngine game loop is hidden—the code that we write tells the game loop what we'd like to happen, and it executes that plan for us automatically.

AndEngine provides several classes related to menus. We looked at the methods provided for creating and managing text menus, graphical (sprite) menus, and animated sprite menus.

The next step in creating our example game was to create a main menu that the player would see once the display of the splash screen ends. Along the way, we saw the AndEngine menu classes in action. We created a text menu that lists the menu items, and we saw the default behavior for text menus. We also created a pop-up graphical

menu with a few options, and showed how its creation is both similar to and different from the creation of a text menu.

We now have the basic structure in place to create our AndEngine-based game. We have a way to start the game, a way to choose different menu items, and a way to end the game. We know the pattern for creating new scenes and new avenues for the players. It's time we got into the game itself, and the specifics of the elements that make it up. That's where we're headed, beginning in Chapter 4.

Exercises

1. We used a SlideAnimator on the pop-up menu in the MainMenuActivity example. What happens if you attach a SlideAnimator to the static menu in the example? Explain why the static menu doesn't slide in as you might expect.
2. Modify `MainMenuActivity.java` so each line of the menu appears in a different color.
3. Modify `MainMenuActivity.java` so the items on the main menu "bloom" slightly in size when they are selected (instead of changing to a gray color).

4

Scenes, Layers, Transitions, and Modifiers

In a movie or play, a scene is a setting where some action takes place over a constrained period of time. Computer games are also organized around scenes, and scenes are composed of layers of graphics. If you've ever edited your own video, you know that there are many ways to transition from scene to scene, to make the flow more interesting for the viewer. Similarly, in a computer game, we want to transition between scenes, and we want to be able to modify graphical elements in a uniform way. We'd like to be able to modify their position, scale, color, rotation, transparency, and perhaps path. This chapter looks at how AndEngine lets us create and modify scenes and graphical objects.

Scenes in AndEngine

In AndEngine, virtually all of the visible elements in a game are subclasses of the class Entity. Entities can be modified with Modifiers, and in particular scenes can be modified to produce transitions. We're focused on scenes in this chapter, but given that the other visible elements of the game (e.g., sprites, tiles, shapes, particles, and text) are all Entities, the Modifiers we discuss here apply to them as well.

We will go through the characteristics of the Entity class and take an in-depth look at all the Modifiers and Ease functions that are available to change Entities. I apologize in advance if this material gets a bit tedious. There are a lot of Modifiers available, with a lot of options, and a number of Ease functions. If the explanations become too repetitive, feel free to skip ahead and use these sections as a reference to the Modifiers and Ease functions in the future.

The Entity/Component Model

We could take many different approaches when designing a game engine such as AndEngine. Because we know Android application development is mostly done in Java, our first inclination might be to stick to an object-oriented approach. With this

strategy, each character in the game would have its own class, and an inheritance hierarchy would exist among those classes, with abstract classes used to group functionality. For example, in V3 there would likely be a Vampire class, maybe a Virgin class, and classes for each of the things that interact during the game (e.g., Bullet, Hatchet, Cross, Tombstones). There might be an abstract Character class that both Vampire and Virgin inherit from. Bullet, Hatchet, and Cross might be subclasses of an abstract Weapons class. Each class would contain the representation and behaviors for that class.

That approach certainly works, but it can quickly grow difficult to manage. If we never changed our minds about what certain characters should do, we could design an object-oriented hierarchy where everything is logical and just implement that hierarchy. Of course, if you've done any software development at all, you know it doesn't work that way. New ideas emerge as the code is being developed, and changes continue to occur in maintenance releases of the game. The strictly object-oriented approach can become quite brittle to ad hoc changes and become difficult to keep refactoring.

One way to build some flexibility into game design is through entity/component design, which is the way AndEngine games are built. Instead of developing a class for each thing in the game, all things are considered Entities. Entities are then dynamically assigned attributes (or components) that describe how the Entity should be displayed (its TextureRegion), how it should move over time (Modifiers), and how its appearance should change. Interactions between the Entities (e.g., collisions) and with the player (e.g., touch events) are built into the Activity where the interaction takes place.

Both approaches (object-oriented and entity/component) have some drawbacks, and AndEngine strikes a fair compromise between them. AndEngine games are designed around Entities, but, as we'll see in this chapter, the Entity class defines a lot of instance variables that are common to visible elements of the game.

If this discussion seems a bit abstract right now, it will be clearer as we develop more details of the V3 example game. You can also search the Internet for "entity component game design"; you should get quite a few hits to help explain the concepts.

Entity

As the superclass of virtually all things visible, including Scene, Entity contains the properties common to visible things. These properties are accessed via getter and setter methods, and are the means by which Entities are modified by Modifier classes. These properties include the following:

- `float mX, mY`: the current position of the Entity
- `float mInitialX, mInitialY`: the initial position of the Entity
- `boolean mVisible`: whether the Entity is currently visible
- `boolean mIgnoreUpdate`: whether the Entity should pay attention to updates
- `int mZindex`: where the Entity appears in the stack of Entities to be displayed

- `IEntity mParent`: the parent of this Entity
- `SmartList<IEntity> mChildren`: a list of Entities who are children of this Entity
- `EntityModifierList mEntityModifiers`: a list of Modifiers to apply to this Entity
- `UpdateHandlerList mUpdateHandlers`: a list of Update Handlers to apply to this Entity
- `float mRed, mGreen, mBlue`: the color or tint of this Entity
- `float mAlpha`: the transparency of this Entity
- `float mX, mY`: the current position of the Entity
- `float mRotation`: the current rotation of the Entity
- `float mRotationCenterX, mRotationCenterY`: the point about which the Entity rotates
- `float mScale`: the current scaling factor for the Entity's size
- `float mScaleCenterX, mScaleCenterY`: the point about which to scale the Entity

Constructor

You will rarely have to use a constructor for Entity (you typically use the constructor for one of the subclasses), but two exist:

Entity()
Entity(final float pX, final float pY)

The second constructor provides a position that serves as both the initial and current positions of the Entity.

Position

The position of an Entity in AndEngine is the position of its center. Entities keep track of their initial position as well as their current position. The following methods are provided:

float getX()
float getY()

Return the components of the current position of the Entity.

float getInitialX()
float getInitialY()

Return the components of the initial position of the Entity.

void setPosition(final float pX, final float pY)

Sets the current position.

setPosition(final IEntity pOtherEntity)

Sets the current position to be the same as that of the passed Entity.

void setInitialPosition()

Sets the current position to the initial position. (*Note:* This method does *not* change the initial position, as you might expect.)

Scale

Entities have a scale factor that's multiplied by the Entity dimensions when each Entity is displayed. They also have a scale position, which serves as the center point for the scaling; the center point is the only point that doesn't move between the unscaled and scaled versions of the Entity. A scale of 2.0f means the Entity should be displayed at twice its "normal" size. Methods related to scaling are summarized here:

boolean isScaled()

Returns true if the current scale is not 1.0f.

float getScaleX()
float getScaleY()

Return the scale multiplier in each dimension.

void setScaleX(final float pScaleX)
void setScaleY(final float pScaleY)

Set the scale in the indicated direction separately.

void setScale(final float pScale)

Sets the same scale in both the *X* and *Y* directions.

void setScale(final float pScaleX, final float pScaleY)

Sets the scales to (possibly) different values, but both at once.

float getScaleCenterX()
float getScaleCenterY()

Return the scale center point positions.

void setScaleCenterX(final float pScaleCenterX)
void setScaleCenterY(final float pScaleCenterY)

Set the scale center point position values separately.

void setScaleCenter(final float pScaleCenterX, final float pScaleCenterY)

Sets the scale center point position.

Color

In AndEngine, the color of an Entity is a multiplier that is applied to any color that the underlying graphics might include. If the Entity is a sprite, for example, and it has a texture applied to it, these colors are multiplied by the sprite colors to result in a combination. You can think of these colors as tinting the Entity.

Colors in AndEngine are most commonly separated into Red, Green, Blue, and Alpha, and have values ranging from 0.0f, for no intensity, to 1.0f, for full intensity.

```
float getRed()
float getGreen()
float getBlue()
float getAlpha()
```

Each method returns the corresponding component of the Entity color.

```
void setColor(final float pRed, final float pGreen, final float pBlue)
```

Sets all three components of the current color.

```
void setColor(final float pRed, final float pGreen, final float pBlue, final float
        pAlpha)
```

Sets the current color and transparency all at once.

Rotation

Much as with Scale, Entities maintain a value and a center position for Rotation. The Rotation value is always in degrees, and can be less than 0 or greater than 360. Positive rotation moves in a counterclockwise direction.

```
float getRotation()
```

Returns the current Rotation value.

```
setRotation(final float pRotation)
```

Sets the current Rotation value.

```
float getRotationCenterX()
float getRotationCenterY()
```

Return the components of the current Rotation center position.

```
void setRotationCenterX(final float pRotationCenterX)
void setRotationCenterY(final float pRotationCenterY)
```

Set the components of the Rotation center position separately.

```
void setRotationCenter(final float pRotationCenterX, final float pRotationCenterY)
```

Sets the complete Rotation center point all at once.

Managing Children

Many a book has been written on the topic of managing children—but here we're talking about the children of Entities. Entities are often arranged in a hierarchy so they can be modified collectively. Then if a parent Entity is modified, the modification passes automatically to all of its children. Conversely, modifications to the child do not automatically pass to the parent. The methods provided for managing children include the following options:

IEntity IEntity getFirstChild()
IEntity getLastChild()

Get the first or last child attached to this Entity. The last child is also the point where you want to attach new children [see `attachChild()`].

void attachChild(final IEntity pEntity)

Adds a child to this Entity, which is normally the last child added. The pattern is as follows:

```
parent.getLastChild().attachChild(newchild);
```

void detachChildren()
boolean detachChild(final IEntity pEntity)

Detach all the children or a specific child of this Entity.

int getChildCount()

Returns the number of children currently attached.

IEntity getChild(final int pIndex)

Returns the *n*th child attached.

void sortChildren()

Sorts the Entity's children by *z*-index [see `setZindex()`].

Manage Modifiers

Modifiers are used to programmatically change the key parameters for an Entity. You use Modifiers to change the position, scale, color, rotation, or transparency of an Entity, either all at once or gradually over a specific duration. The major Modifiers are described in detail later in this chapter. The Entity methods for managing Modifiers include the following:

void registerEntityModifier(final IEntityModifier pEntityModifier)

Registers a Modifier with the Entity so it will apply when the Entity is displayed.

boolean unregisterEntityModifier(final IEntityModifier pEntityModifier)

Removes the specified Modifier so it is no longer applied (the return value is true if this operation is successful).

void clearEntityModifiers()

Removes all the Modifiers registered with this Entity.

Other Useful Entity Methods

Some other Entity methods don't fall neatly into the categories listed previously. They control the visibility of the Entity, the order in which it is rendered, and a wildcard, where you can attach any Object you like to the Entity.

void setVisible(final boolean pVisible)
boolean isVisible()

Set and return whether the Entity is visible or invisible. This property is independent of the Alpha value, so if you set an Entity to be invisible and then set it to be visible, the Alpha value remains unchanged.

void setZIndex(final int pZIndex)
int getZIndex()

Sets or gets the *z*-index value for the Entity. Entities with a higher *z*-index are displayed on top of those with a lower *z*-index. This method sets the *z*-index, but you have to call `sortChildren()` on the Entity's parent to actually rearrange the drawing order.

void setUserData(final Object pUserData)
Object getUserData()

Sets or retrieves an arbitrary object defined by the user. You can use this technique to attach anything you like to an Entity, for whatever reason.

Layers

A Layer in AndEngine is a subclass of Entity. It adds almost nothing to its superclass, but it allows you to layer the graphics that make up a Scene and to assign *z*-indexes and other properties to the Layers. There are two constructors:

Layer()
Layer(final float pX, final float pY)

The first is equivalent to the second, with `pX` and `pY` both being 0.0f. Both constructors call the superclass constructor—and that's all there is to Layers.

Scenes

AndEngine provides a Scene class that we can instantiate to create each scene of our game. Scenes have the following properties:

- `Scene mParentScene`: Every Scene can have an optional parent Scene.
- `Scene mChildScene`: Every Scene can have an optional child Scene.
- `SmartList<ITouchArea> mTouchAreas`: Scenes know how to accept user touches and have a list of touch areas.
- `IOnSceneTouchListener mOnSceneTouchListener`: A listener can be registered to be called when the Scene is touched.
- `IOnAreaTouchListener mOnAreaTouchListener`: A separate listener can be registered to be called when a touch area is touched.
- `RunnableHandler mRunnableHandler`: A Scene can have a RunnableHandler.
- `IBackground mBackground`: Every Scene has a background, which is a solid black color by default.
- `boolean mOnAreaTouchTraversalBackToFront`: This flag indicates the order in which touches should be processed.
- Layers of graphics can be attached to a Scene as children of the Scene.

Constructor

There is one constructor for Scene:

```
Scene(final int pLayerCount)
```

We can't change the layer count once the Scene is created, so if you edit your game to add a Layer at some point, you may have to go back and change the count given to the constructor.

Background Management

Several methods are used to set and get the Scene's background, and to set and test whether the background is enabled.

```
IBackground getBackground()
void setBackground(final IBackground pBackground)
void setBackgroundEnabled(final boolean pEnabled)
boolean isBackgroundEnabled()
```

As mentioned earlier, the background defaults to a black ColorBackBackground until you set it to something else.

Child Scene Management

Scenes can have one special child that is not a Layer. This child Scene has several methods associated with it:

```
void setChildScene(final Scene pChildScene)
void setChildScene(final Scene pChildScene, final boolean pModalDraw,
        final boolean pModalUpdate, final boolean pModalTouch)
void setChildSceneModal(final Scene pChildScene)
Scene getChildScene()
boolean hasChildScene()
void clearChildScene()
```

If the Boolean values returned by the second `setChildScene()` method are true, they make the child Scene modal, meaning the parent Scene will pause while the child has focus. The third method is equivalent to the second with the Boolean values all set to true.

Layer Management

Layers are managed using the child management methods inherited from Entity. Only one additional method is defined in Scene:

```
void sortLayers()
```

It sorts the Layers who are children of the Scene according to their *z*-index.

Parent Management

Scenes can have a parent Scene, which is set automatically when a child Scene is added to a Scene. Alternatively, it can be set through the following method:

```
void setParentScene(final Scene pParentScene)
```

There is no getter equivalent, and the instance variable is protected, but the value is used in managing the back Scene transition.

Touch Area Management

Scenes know how to respond to touch events, and there are methods to create and respond to touch areas in the Scene:

```
void registerTouchArea(final ITouchArea pTouchArea)
boolean unregisterTouchArea(final ITouchArea pTouchArea)
boolean unregisterTouchAreas(final ITouchAreaMatcher pTouchAreaMatcher)
void clearTouchAreas()
ArrayList<ITouchArea> getTouchAreas()
```

```
boolean onSceneTouchEvent(final TouchEvent pSceneTouchEvent)
boolean onAreaTouchEvent(final TouchEvent pSceneTouchEvent,
        final float sceneTouchEventX, final float sceneTouchEventY,
        final ITouchArea touchArea)
boolean onChildSceneTouchEvent(final TouchEvent pSceneTouchEvent)
```

We'll cover the details of TouchAreas and response to touches in Chapter 8, on user input, but for now recognize that you can set up TouchAreas for a Scene and override the onXTouchEvent methods to respond to touches.

Specialized Scenes

There are subclasses of Scene that define specialized types of Scenes. We'll use the constructors to list them:

```
CameraScene(final int pLayerCount)
CameraScene(final int pLayerCount, final Camera pCamera)
```

Add a Camera for the Scene.

```
SplashScene(final Camera pCamera, final TextureRegion pTextureRegion)
SplashScene(final Camera pCamera, final TextureRegion pTextureRegion,
        final float pDuration, final float pScaleFrom, final float pScaleTo)
```

Create a splash screen (simplifying all that we did to create our splash screen in Chapter 2).

```
MenuScene()
MenuScene(final IOnMenuItemClickListener pOnMenuItemClickListener)
MenuScene(final Camera pCamera)
MenuScene(final Camera pCamera, final IOnMenuItemClickListener
        pOnMenuItemClickListener)
```

Create a MenuScene, like the one we used in Chapter 3.

```
PopupScene(final Camera pCamera, final Scene pParentScene,
        final float pDurationSeconds)
PopupScene(final Camera pCamera, final Scene pParentScene,
        final float pDurationSeconds, final Runnable pRunnable)
TextPopupScene(final Camera pCamera, final Scene pParentScene,
        final Font pFont, final String pText, final float pDurationSeconds)
TextPopupScene(final Camera pCamera, final Scene pParentScene,
        final Font pFont, final String pText, final float pDurationSeconds,
        final IEntityModifier pShapeModifier)
TextPopupScene(final Camera pCamera, final Scene pParentScene,
        final Font pFont, final String pText, final float pDurationSeconds,
        final Runnable pRunnable)
```

TextPopupScene(final Camera pCamera, final Scene pParentScene, final Font pFont, final String pText, final float pDurationSeconds, final IEntityModifier pShapeModifier, final Runnable pRunnable)

Create a variety of Scenes that pop up temporarily over the current Scene.

Entity Modifiers

In the previous section, we said you could attach Modifiers to Entities to control parameters such as position, scale, color, rotation, and transparency. This section explores the Modifiers themselves, with descriptions of the ones available and the ways they can be combined into sequences of Modifiers.

Modifiers for Entities are subclasses of EntityModifier and are created and registered with the Entity they affect using the `registerEntityModifier()` method described earlier. A typical pattern would be

```
entity.registerEntityModifier(new XxxModifier(p1, p2, …);
```

where `entity` is some previously defined Entity, and `XxxModifier` is one of the Modifier classes listed in the next subsection. If the Modifier has a duration parameter, it is usually the first parameter passed. The other parameters vary by Modifier.

Common Methods

Some methods that are common to the Entity Modifiers are especially useful.

XxxModifier clone()

Returns a copy of the Modifier. This method is useful if you need two Entities to behave the same way, but not be driven by the identical Modifier. Each Modifier executes its own animation sequence during the duration, and you might not want the Entities' animations linked, for example.

boolean isFinished()

Returns true if the Modifier has finished its action.

void setRemoveWhenFinished(final boolean pRemoveWhenFinished)
boolean isRemoveWhenFinished()

Sets or gets a flag saying whether the Modifier should remove itself once it's finished.

void setModifierListener(final IModifierListener<T> pModifierListener)
IModifierListener<T> getModifierListener()

Sets or gets a listener routine that is called when the Modifier is finished. Only one such ModifierListener can be registered.

Position

These Modifiers change the current position of the Entity. They are not methods, but rather are classes that are instantiated as shown in the pattern earlier, under "Entity Modifiers."

MoveModifier(final float pDuration, final float pFromX, final float pToX, final float pFromY, final float pToY, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)MoveModifier(final float pDuration, final float pFromX, final float pToX, final float pFromY, final float pToY, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)
MoveModifier(final float pDuration, final float pFromX, final float pToX, final float pFromY, final float pToY, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)
MoveModifier(final float pDuration, final float pFromX, final float pToX, final float pFromY, final float pToY, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)

All of these methods cause the Entity to move in both the *X* and *Y* directions at the same time. In each case pDuration is the length of time, in seconds, that you want the movement to last. We will talk about EaseFunctions in the next section, but for the moment think of them as "modifiers to the Modifiers." EntityModifierListeners are callbacks that are initiated when the Modifier is finished with its action.

MoveXModifier(final float pDuration, final float pFromX, final float pToX)
MoveXModifier(final float pDuration, final float pFromX, final float pToX, final IEaseFunction pEaseFunction)
MoveXModifier(final float pDuration, final float pFromX, final float pToX, final IEntityModifierListener pEntityModifierListener)
MoveXModifier(final float pDuration, final float pFromX, final float pToX, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)
MoveXModifier(final MoveXModifier pMoveXModifier)
MoveYModifier(final float pDuration, final float pFromY, final float pToY)
MoveYModifier(final float pDuration, final float pFromY, final float pToY, final IEaseFunction pEaseFunction)
MoveYModifier(final float pDuration, final float pFromY, final float pToY, final IEntityModifierListener pEntityModifierListener)

MoveYModifier(final float pDuration, final float pFromY, final float pToY, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)
MoveYModifier(final MoveYModifier pMoveYModifier)

These Modifiers move the Entity in only the *X* direction or the *Y* direction, as indicated. Otherwise, they are the same.

Path

Moving from one point to another is all well and good, but AndEngine also allows us to specify a multipoint path and have an Entity follow that Path. What's more, we can register a callback to be initiated whenever the Entity reaches each waypoint in the Path.

There are three constructors for Paths:

Path(final int pLength)
Path(final float[] pCoordinatesX, final float[] pCoordinatesY)
Path(final Path pPath)

The first constructor says that the Path will consist of `pLength` points. The actual Path is then constructed using the `to` method to add each segment:

```
to(final float pX, final float pY)
```

The second constructor uses arrays of *x* and *y* coordinates to construct the Path, and the third simply uses a Path that already exists.

The constructors for PathModifier are as follows:

PathModifier(final float pDuration, final Path pPath)
PathModifier(final float pDuration, final Path pPath, final IEaseFunction pEaseFunction)
PathModifier(final float pDuration, final Path pPath, final IEntityModifierListener pEntityModiferListener)
PathModifier(final float pDuration, final Path pPath, final IEntityModifierListener pEntityModiferListener, final IEaseFunction pEaseFunction)
PathModifier(final float pDuration, final Path pPath, final IEntityModifierListener pEntityModiferListener, final IPathModifierListener pPathModifierListener)
PathModifier(final float pDuration, final Path pPath, final IEntityModifierListener pEntityModiferListener, final IPathModifierListener pPathModifierListener, final IEaseFunction pEaseFunction)

Listing 4.1 shows a brief example of creating a Path and registering a PathModifier with a callback method.

Listing 4.1 **PathModifier Example**

```
. . .
   final Path path = new Path(5).to(10, 10).to(10, 50).to(50, 50).
      to(50, 10).to(10, 10);
   entity.registerEntityModifier(new LoopEntityModifier(new PathModifier(30,
       path, null, new IPathModifierListener() {
      @Override
      public void onWaypointPassed(final PathModifier pPathModifier,
      final IEntity pEntity, final int pWaypointIndex) {
         switch(pWaypointIndex) {
            case 0:
               entity.setColor(1.0f, 0.0f, 0.0f);
               break;
            case 1:
               entity.setColor(0.0f, 1.0f, 0.0f);
               break;
            case 2:
               entity.setColor(0.0f, 0.0f, 1.0f);
               break;
            case 3:
               entity.setColor(1.0f, 0.0f, 0.0f);
               break;
         }
      }
      ));
   scene.getLastChild().attachChild(entity);
. . .
```

Scale

ScaleModifiers modify the displayed scale of the Entity. The set of constructors is similar to the set we saw for Position:

ScaleModifier(final float pDuration, final float pFromScale, final float pToScale)
ScaleModifier(final float pDuration, final float pFromScale, final float pToScale, final IEaseFunction pEaseFunction)
ScaleModifier(final float pDuration, final float pFromScale, final float pToScale, final IEntityModifierListener pEntityModifierListener)
ScaleModifier(final float pDuration, final float pFromScale, final float pToScale, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)

All of these methods cause the Entity to be scaled for display around its scale position over some specific duration. The parameters are straightforward. A scale of 1.0f is the same size as the original.

ScaleModifier(final float pDuration, final float pFromScaleX, final float pToScaleX, final float pFromScaleY, final float pToScaleY)

This Modifier scales the Entity immediately and allows the scaling to be different in the *X* and *Y* directions.

```
ScaleAtModifier(final float pDuration, final float pFromScale, final
        float pToScale, final float pScaleCenterX, final float pScaleCenterY)
ScaleAtModifier(final float pDuration, final float pFromScale, final
        float pToScale, final float pScaleCenterX, final float pScaleCenterY,
        final IEaseFunction pEaseFunction)
ScaleAtModifier(final float pDuration, final float pFromScale, final
        float pToScale, final float pScaleCenterX, final float pScaleCenterY,
        final IEntityModifierListener pEntityModifierListener)
ScaleAtModifier(final float pDuration, final float pFromScale, final
        float pToScale, final float pScaleCenterX, final float pScaleCenterY,
        final IEntityModifierListener pEntityModifierListener,
        final IEaseFunction pEaseFunction)
ScaleAtModifier(final float pDuration, final float pFromScaleX, final
        float pToScaleX, final float pFromScaleY, final float pToScaleY,
        final float pScaleCenterX, final float pScaleCenterY)
ScaleAtModifier(final float pDuration, final float pFromScaleX, final
        float pToScaleX, final float pFromScaleY, final float pToScaleY,
        final float pScaleCenterX, final float pScaleCenterY,
        final IEaseFunction pEaseFunction)
ScaleAtModifier(final float pDuration, final float pFromScaleX, final
        float pToScaleX, final float pFromScaleY, final float pToScaleY,
        final float pScaleCenterX, final float pScaleCenterY,
        final IEntityModifierListener pEntityModifierListener)
ScaleAtModifier(final float pDuration, final float pFromScaleX, final
        float pToScaleX, final float pFromScaleY, final float pToScaleY,
        final float pScaleCenterX, final float pScaleCenterY,
        final IEntityModifierListener pEntityModifierListener,
        final IEaseFunction pEaseFunction)
```

These Modifiers are the same as the ScaleModifiers given previously, but they scale around a point different from the current ScaleCenter position.

Color

The ColorModifiers change the color, as you would expect. All color values range from 0.0f (zero intensity) to 1.0f (full intensity).

```
ColorModifier(final float pDuration, final float pFromRed, final float pToRed,
        final float pFromGreen, final float pToGreen, final float pFromBlue,
        final float pToBlue)
ColorModifier(final float pDuration, final float pFromRed, final float pToRed,
        final float pFromGreen, final float pToGreen, final float pFromBlue,
        final float pToBlue, final IEaseFunction pEaseFunction)
```

```
ColorModifier(final float pDuration, final float pFromRed, final float pToRed,
        final float pFromGreen, final float pToGreen, final float pFromBlue,
        final float pToBlue, final IEntityModifierListener
        pEntityModifierListener, final IEaseFunction pEaseFunction)
ColorModifier(final float pDuration, final float pFromRed, final float pToRed,
        final float pFromGreen, final float pToGreen, final float pFromBlue,
        final float pToBlue, final IEntityModifierListener
        pEntityModifierListener, final IEaseFunction pEaseFunction)
```

All of these methods cause the Entity color to change over the duration. The EaseFunction and EntityModifierListener function just as with the other Entity Modifiers.

Rotation

RotationModifiers rotate an Entity about a point—either the current RotationCenter position or some other point. Rotations are always given in degrees.

```
RotationModifier(final float pDuration, final float pFromRotation,
        final float pToRotation)
RotationModifier(final float pDuration, final float pFromRotation,
        final float pToRotation, final IEntityModifierListener
        pEntityModifierListener)
RotationModifier(final float pDuration, final float pFromRotation,
        final float pToRotation final IEaseFunction pEaseFunction)
RotationModifier(final float pDuration, final float pFromRotation,
        final float pToRotation, final IEntityModifierListener
        pEntityModifierListener, final IEaseFunction pEaseFunction)
```

These Modifiers are the ones to use when you want the Entity to rotate around the current RotationCenter over a given duration.

```
RotationAtModifier(final float pDuration, final float pFromRotation,
        final float pToRotation, final float pRotationCenterX,
        final float pRotationCenterY)
RotationAtModifier(final float pDuration, final float pFromRotation,
        final float pToRotation, final float pRotationCenterX,
        final float pRotationCenterY, final IEntityModifierListener
        pEntityModifierListener)
RotationAtModifier(final float pDuration, final float pFromRotation,
        final float pToRotation, final float pRotationCenterX,
        final float pRotationCenterY, final IEaseFunction pEaseFunction)
RotationAtModifier(final float pDuration, final float pFromRotation,
        final float pToRotation, final float pRotationCenterX,
        final float pRotationCenterY, final IEntityModifierListener
        pEntityModifierListener, final IEaseFunction pEaseFunction)
```

These Modifiers are used when you want rotation to occur about a point that is not the current RotationCenter for a duration. This technique avoids changing the RotationCenter, which you might not want to do.

RotationByModifier(final float pDuration, final float pRotation)
RotationByModifier(final RotationByModifier pRotationByModifier)

These Modifiers rotate an Entity about the current RotationCenter immediately. You could do the same thing with RotationModifier and a pDuration of 0.0f, but these methods are clearer (and more efficient).

Transparency

Transparency of an Entity is controlled through its Alpha value. These Modifiers include the ones used to cause an Entity to fade in or out.

AlphaModifier(final float pDuration, final float pFromAlpha, final float pToAlpha)
AlphaModifier(final float pDuration, final float pFromAlpha, final float pToAlpha, final IEntityModifierListener)
AlphaModifier(final float pDuration, final float pFromAlpha, final float pToAlpha, final IEaseFunction pEaseFunction)
AlphaModifier(final float pDuration, final float pFromAlpha, final float pToAlpha, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)

The parameters are straightforward. Alpha values range from 0.0f (invisible, or completely transparent) to 1.0f (opaque).

FadeInModifier(final float pDuration)
FadeInModifier(final float pDuration, final IEaseFunction pEaseFunction)
FadeInModifier(final float pDuration, final IEntityModifierListener pEntityModifierListener)
FadeInModifier(final float pDuration, final IEntityModifierListener pEntityModifierListener, final IEaseFunction pEaseFunction)

These Modifiers are used to fade Entities in and out. They are convenience classes that wrap the AlphaModifier classes given earlier with Alpha values of 0.0f and 1.0f.

Delay

It may not be obvious why you would ever want a DelayModifier, but in the next section we will talk about combining Modifiers into sequences. In those cases, the ability to delay for a given time is a requirement.

DelayModifier(final float pDuration)
DelayModifier(final float pDuration, final IEntityModifierListener pEntityModifierListener)

These Modifiers simply delay their action for the specified duration, and optionally call the EntityModifierListener when finished.

Modifier Combinations

We often want to combine Modifiers to create an effect with an Entity. For example, we might want to move the Entity while we change its scale and color all at the same time. Alternatively, we might want to move an Entity, then change its scale, and then change its color. We can easily generate complex sequences of Modifiers with AndEngine using some special Modifiers. The pattern for combining Modifiers using one of these is shown in Listing 4.2.

Listing 4.2 **Modifier Combinations**

```
. . .
entity.registerEntityModifier(
   new SequenceEntityModifier(
      new ParallelEntityModifier(
         new MoveYModifier(3, 0.0f, CAMERA_HEIGHT - 40.0f),
         new AlphaModifier(3, 0.0f, 1.0f),
         new ScaleModifier(3, 0.5f, 1.0f)              ),
      new RotationModifier(3, 0, 720)
   )
);
. . .
```

Any of the combination Modifiers will throw an IllegalArgumentException if you pass them anything other than a Modifier where a Modifier is required.

ParallelEntityModifier

When we want to apply two or more Modifiers to an Entity at the same time, we use the following Modifier:

ParallelEntityModifier(final IEntityModifier... pEntityModifiers)
ParallelEntityModifier(final IEntityModifierListener pEntityModifierListener,
final IEntityModifier... pEntityModifiers)

This combination Modifier runs its parameters in parallel on the Entity where it is registered. It calls the EntityModifierListener when the last Modifier is finished.

SequenceEntityModifier

When we want to run several Modifiers in sequence, one after the other, we use this Modifier:

SequenceEntityModifier(final IEntityModifier... pEntityModifiers)
SequenceEntityModifier(final IEntityModifierListener pEntityModifierListener,
final IEntityModifier... pEntityModifiers)

SequenceEntityModifier(final IEntityModifierListener pEntityModifierListener, final ISubSequenceShapeModifierListener pSubSequenceShapeModifierListener, final IEntityModifier... pEntityModifiers)

The first two SequenceEntityModifiers are similar to their Parallel counterparts. The final one is a little different, and it allows the creation of SubSequence.

Ease Functions

So what about those EaseFunctions we've been allowing for throughout the descriptions of the Modifiers? What exactly do they do, and which ones are available? There is a great example application for EaseFunctions that is part of AndEngine Examples, available from the main AndEngine website (`http://www.andengine.org`). If you haven't already done so, download it now and play with the EaseFunctions a bit to get a feel for them.

EaseFunctions change the way a Modifier's action progresses. The default (i.e., where no EaseFunction is assigned) is for a Modifier to progress linearly. For example, if a MoveModifier is moving an Entity from point A to point B, the velocity of that movement will be the same at the beginning, middle, and end of the movement.

EaseFunctions allow you to change the Modifiers so they progress at different rates (maybe even backward) at different times in the duration of the Modifier. Think of those fancy elevators in high-rise hotels that move quickly between floors, but ease themselves to a stop when they arrive so you don't feel a thing.

The naming convention for EaseFunctions is something like

```
Ease<Type><End>
```

where `<Type>` describes the easing function (e.g., back, bounce, circular, cubic, quadratic), and `<End>` says which end of the duration is affected. "In" functions affect the beginning of the duration, "Out" functions affect the end, and "InOut" functions affect both ends. Let's look at each of the types in turn.

In the following descriptions, the diagram in each case shows the progress for the "InOut" case.

EaseBack Functions

EaseBack functions go below the starting point and/or beyond the endpoint, as shown in Figure 4.1.

EaseBackIn
EaseBackOut
EaseBackInOut

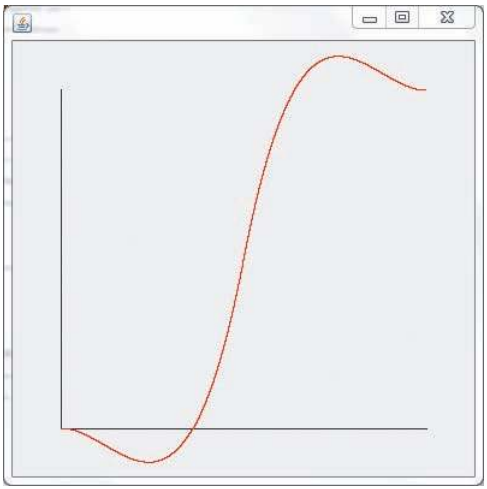


Figure 4.1 EaseBackInOut

EaseBounce Functions

EaseBounce functions have a bouncing start and/or finish, as shown in Figure 4.2. An obvious use for this ease function is when you want to simulate something bouncing on the ground (without using the Physics extension discussed in Chapter 12). There are three options:

EaseBounceIn
EaseBounceOut
EaseBounceInOut

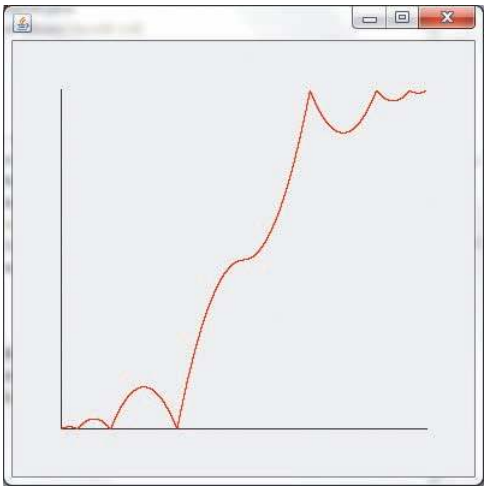


Figure 4.2 EaseBounceInOut

EaseCircular Functions

EaseCircular functions produce a slow start and/or finish following circular acceleration, as shown in Figure 4.3:

EaseCircularIn
EaseCircularOut
EaseCircularInOut

EaseCubic Functions

EaseCubic functions create a slow start and/or finish using a cubic equation, as shown in Figure 4.4. If you are simulating the elevator deceleration movement we talked about earlier, you might use one of these functions:

EaseCubicIn
EaseCubicOut
EaseCubicInOut

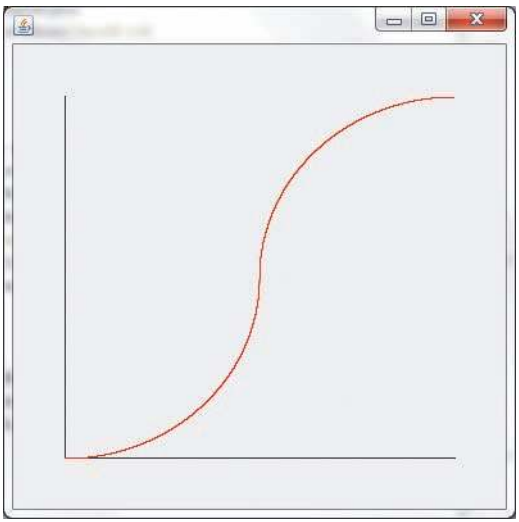


Figure 4.3 EaseCircularInOut

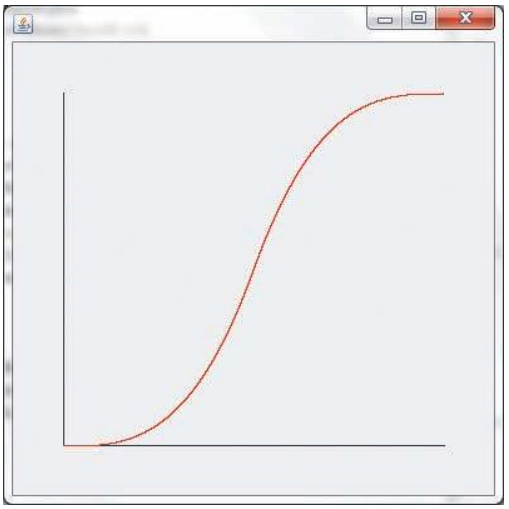


Figure 4.4 EaseCubicInOut

EaseElastic Functions

EaseElastic functions simulate the action of a rubber band or spring, as shown in Figure 4.5:

EaseElasticIn
EaseElasticOut
EaseElasticInOut

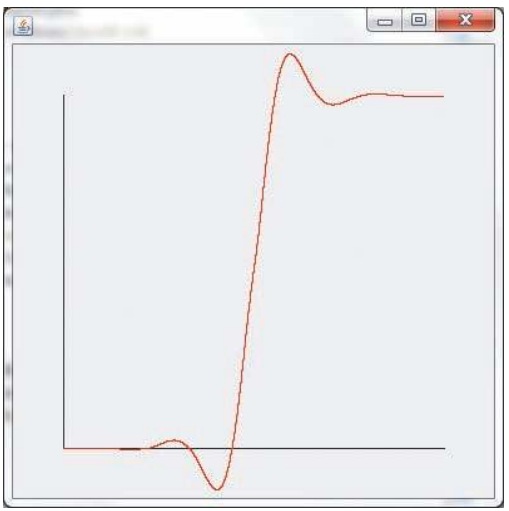


Figure 4.5 EaseElasticInOut

EaseExponential Functions

EaseExponential functions produce an exponentially slow start and/or finish, as shown in Figure 4.6:

EaseExponentialIn
EaseExponentialOut
EaseExponentialInOut

EaseLinear Function

Yes, there is an EaseLinear function, which duplicates the default action (with no EaseFunction), as shown in Figure 4.7. This function can be useful when sequencing through EaseFunctions.

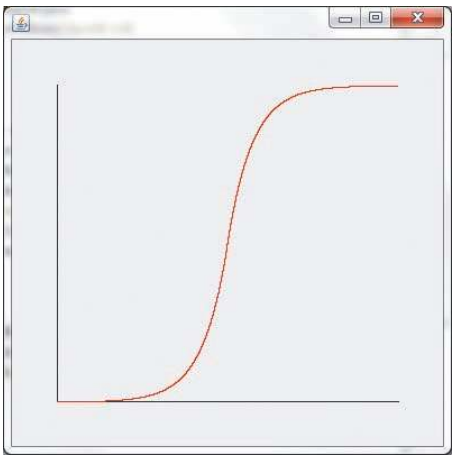


Figure 4.6 EaseExponentialInOut

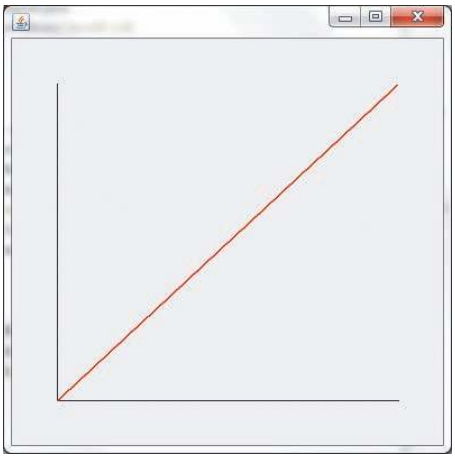


Figure 4.7 EaseLinearInOut

EaseQuad Functions

EaseQuad functions create a slow start and/or stop using a quadratic equation, as shown in Figure 4.8:

EaseQuadIn
EaseQuadOut
EaseQuadInOut

EaseQuart Functions

EaseQuart functions produce a slow start and/or stop using a quartic equation, as shown in Figure 4.9:

EaseQuartIn
EaseQuartOut
EaseQuartInOut

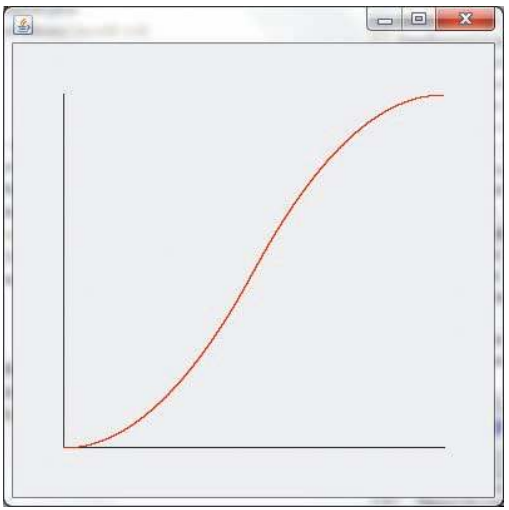


Figure 4.8 EaseQuadInOut

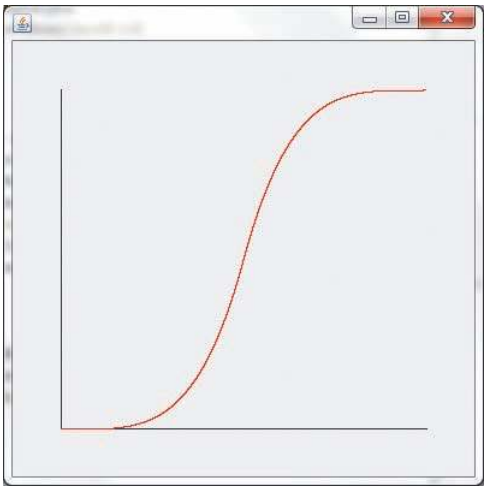


Figure 4.9 EaseQuartInOut

EaseQuint Functions

EaseQuint functions create a slow start and/or stop using a quintic equation, as shown in Figure 4.10:

EaseQuintIn
EaseQuintOut
EaseQuintInOut

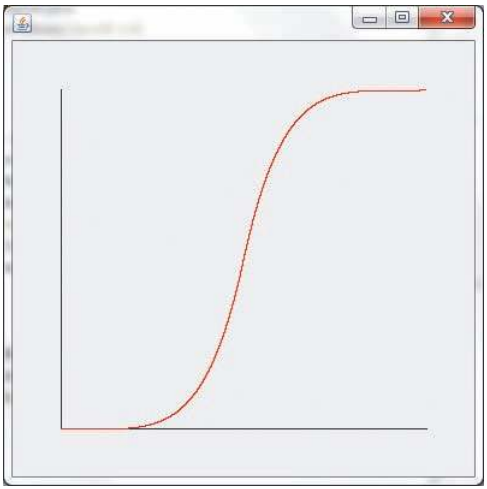


Figure 4.10 EaseQuintInOut

EaseSine Functions

EaseSine functions produce a slow start and/or stop using a sine function, as shown in Figure 4.11:

EaseSineIn
EaseSineOut
EaseSineInOut

EaseStrong Functions

Although they are separately implemented, mathematically EaseStrong functions are the same as the EaseQuint functions, as shown in Figure 4.12:

EaseStrongIn
EaseStrongOut
EaseStrongInOut

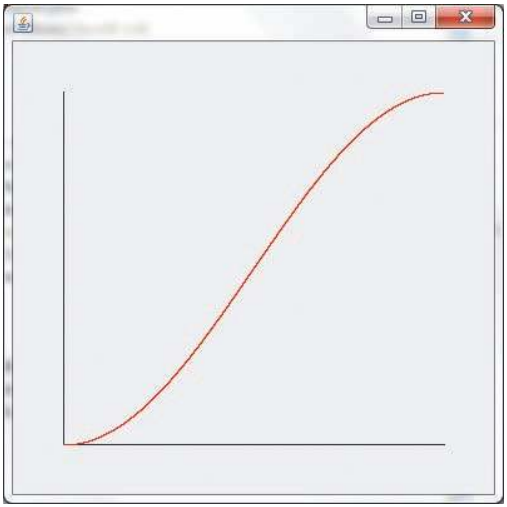


Figure 4.11 EaseSineInOut

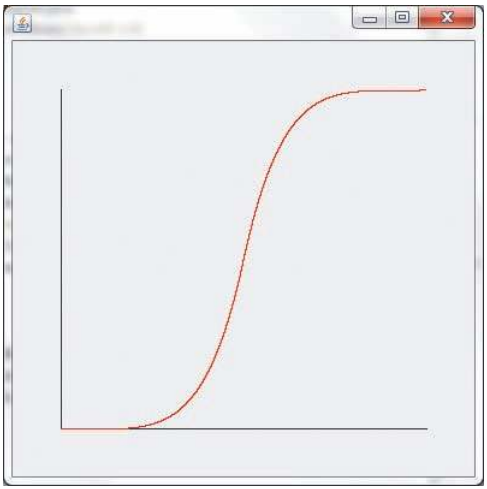


Figure 4.12 EaseStrongInOut

Creating the Game Level 1 Scene

Now that we have this vast, detailed knowledge of scenes, layers, transitions, modifier, and ease functions, it's time to make use of a little of our new knowledge to move our game development forward. When the player selects the menu option to start the game, Virgins Versus Vampires will open on the scene shown in Figure 4.13, which represents the graveyard next to Miss Blossom's Home for Wayward Virgins.

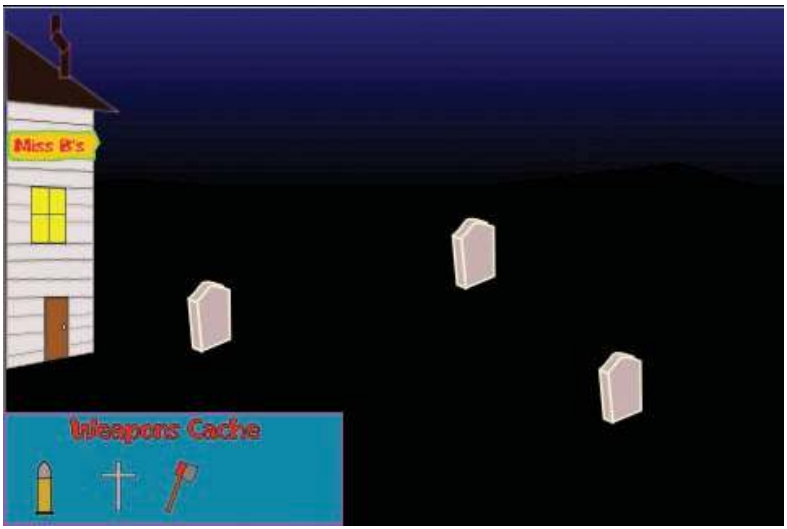


Figure 4.13 V3 opening game scene

The scene will eventually need to receive touch inputs to select obstacles (from the list in the lower-left corner of the scene) and to place objects in the graveyard. We won't start implementing the touch part yet—just the user interface to this scene.

Listing 4.3 shows the changes to `MainMenuActivity.java`, and Listing 4.4 shows `Level1Activity.java`, which implements the first level of the game. For now, when the player touches "Play Game" on the main menu, the menu shrinks, and Level 1 grows into view. Each of the initial weapons then drops into the Weapons Cache and does a brief spin as it settles into place.

At this point, this code should be pretty easy to follow. We create the scene and layer as usual, and then it's just a matter of adding the different children to the layer. The appearance of the weapons is orchestrated through a sequence of modifiers, and the MoveModifiers are eased with EaseQuadOut, so the weapons fall quickly and then slow down as they reach the Weapons Cache.

Listing 4.3 **MainMenuActivity.java**

```
. . .
   protected Handler mHandler;
. . .
   @Override
   public Engine onLoadEngine() {
      mHandler = new Handler();
. . .
   @Override
   public void onResumeGame() {
      super.onResumeGame();
      mMainScene.registerEntityModifier(new ScaleAtModifier(0.5f,
         0.0f, 1.0f, CAMERA_WIDTH/2, CAMERA_HEIGHT/2));
      mStaticMenuScene.registerEntityModifier(
         new ScaleAtModifier(0.5f, 0.0f, 1.0f,
         CAMERA_WIDTH/2, CAMERA_HEIGHT/2));
   }
. . .
   @Override
   public boolean onMenuItemClicked(final MenuScene pMenuScene,
      final IMenuItem pMenuItem, final float pMenuItemLocalX,
      final float pMenuItemLocalY) {
      switch(pMenuItem.getID()) {
. . .
         case MENU_PLAY:
            mMainScene.registerEntityModifier(
               new ScaleModifier(1.0f, 1.0f,
               0.0f));
            mHandler.postDelayed(
               mLaunchLevel1Task,1000);
            return true;
```

```
        case MENU_SCORES:
. . .
    private Runnable mLaunchLevel1Task = new Runnable() {
        public void run() {
          Intent myIntent = new Intent(MainMenuActivity.this,
         Level1Activity.class);
          MainMenuActivity.this.startActivity(myIntent);
        }
     };
. . .
```

The two changes to the `MENU_PLAY` case in the switch statement drive the other changes to `MainMenuActivity.java`:

- We've added a Modifier that will shrink the two displayed scenes (mMainScene and mMainMenuScene) in 1 second.
- To give that Modifier time to play, we've asked Android to start the Level1Activity after a delay of 1 second. The mechanism is the same one we used in StartActivity—namely, posting a delayed Runnable to actually send the Intent requesting the start.
- We've overridden the `onGameResume()` method, so we can grow the main menu scene back to full size if the player backs up from Level1Activity.

Listing 4.4 **Level1Activity.java**

```
package com.pearson.lagp.v3;
. . .
public class Level1Activity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;
   private String tag = "Level1Activity";

   // ===========================================================
   // Fields
   // ===========================================================

   protected Camera mCamera;

   protected Scene mMainScene;

   private Texture mLevel1BackTexture;
   private BuildableTexture mObstacleBoxTexture;
```

```
private TextureRegion mBoxTextureRegion;
private TextureRegion mLevel1BackTextureRegion;
private TextureRegion mBulletTextureRegion;
private TextureRegion mCrossTextureRegion;
private TextureRegion mHatchetTextureRegion;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT),
      this.mCamera));
}

@Override
public void onLoadResources() {
   /* Load Textures. */
   TextureRegionFactory.setAssetBasePath("gfx/Level1/");
   mLevel1BackTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mLevel1BackTextureRegion =
      TextureRegionFactory.createFromAsset(
         this.mLevel1BackTexture,
         this, "Level1Bk.png", 0, 0);
   mEngine.getTextureManager().loadTexture(
       this.mLevel1BackTexture);

   mObstacleBoxTexture = new BuildableTexture(512, 256,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mBoxTextureRegion =
      TextureRegionFactory.createFromAsset(
         mObstacleBoxTexture,
         this, "Obstaclebox.png");
```

```
      mBulletTextureRegion =
         TextureRegionFactory.createFromAsset(
            mObstacleBoxTexture,
          this, "Bullet.png");
      mCrossTextureRegion =
         TextureRegionFactory.createFromAsset(
            mObstacleBoxTexture,
          this, "Cross.png");
      mHatchetTextureRegion =
         TextureRegionFactory.createFromAsset(
            mObstacleBoxTexture,
          this, "Hatchet.png");
         try {
               mObstacleBoxTexture.build(
               new BlackPawnTextureBuilder(2));
            } catch (final TextureSourcePackingException e) {
               Log.d(tag, "Sprites won't fit ");
            }
      this.mEngine.getTextureManager().loadTexture(
this.mObstacleBoxTexture);
      }

   @Override
   public Scene onLoadScene() {
      this.mEngine.registerUpdateHandler(new FPSLogger());

      final Scene scene = new Scene(1);

      /* Center the camera. */
      final int centerX = (CAMERA_WIDTH -
         mLevel1BackTextureRegion.getWidth()) / 2;
      final int centerY = (CAMERA_HEIGHT -
         mLevel1BackTextureRegion.getHeight()) / 2;

      /* Create the sprites and add them to the scene. */
      final Sprite background = new Sprite(centerX, centerY,
         mLevel1BackTextureRegion);
      scene.getLastChild().attachChild(background);
      final Sprite obstacleBox = new Sprite(0.0f, CAMERA_HEIGHT -
         mBoxTextureRegion.getHeight(), mBoxTextureRegion);
      scene.getLastChild().attachChild(obstacleBox);
      final Sprite bullet = new Sprite(20.0f,
         CAMERA_HEIGHT - 40.0f,
          mBulletTextureRegion);
      bullet.registerEntityModifier(
         new SequenceEntityModifier(
            new ParallelEntityModifier(
               new MoveYModifier(3, 0.0f,
```

```
                CAMERA_HEIGHT - 40.0f,
                EaseQuadOut.getInstance() ),
            new AlphaModifier(3, 0.0f, 1.0f),
            new ScaleModifier(3, 0.5f, 1.0f)
        ),
        new RotationModifier(3, 0, 360)
        )
);
scene.getLastChild().attachChild(bullet);
final Sprite cross = new Sprite(bullet.getInitialX() +
   40.0f, CAMERA_HEIGHT - 40.0f, mCrossTextureRegion);
cross.registerEntityModifier(
   new SequenceEntityModifier(
      new ParallelEntityModifier(
         new MoveYModifier(4, 0.0f,
            CAMERA_HEIGHT - 40.0f,
            EaseQuadOut.getInstance() ),
         new AlphaModifier(4, 0.0f, 1.0f),
         new ScaleModifier(4, 0.5f, 1.0f)
      ),
      new RotationModifier(2, 0, 360)
   )
);
cross.registerEntityModifier(new AlphaModifier(
               10.0f, 0.0f, 1.0f));
scene.getLastChild().attachChild(cross);
final Sprite hatchet = new Sprite(cross.getInitialX() +
   40.0f,
    CAMERA_HEIGHT - 40.0f, mHatchetTextureRegion);
hatchet.registerEntityModifier(
   new SequenceEntityModifier(
      new ParallelEntityModifier(
         new MoveYModifier(5, 0.0f,
            CAMERA_HEIGHT - 40.0f,
            EaseQuadOut.getInstance() ),
         new AlphaModifier(5, 0.0f, 1.0f),
         new ScaleModifier(5, 0.5f, 1.0f)
      ),
      new RotationModifier(2, 0, 360)
   )
);
hatchet.registerEntityModifier(new AlphaModifier(
               15.0f, 0.0f, 1.0f));
scene.getLastChild().attachChild(hatchet);
```

```
        scene.registerEntityModifier(new AlphaModifier(10, 0.0f,
            1.0f));
        return scene;
    }

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

Summary

We covered a lot of ground in this chapter, and there's no way you could remember all the Scenes, Modifiers, and EaseFunctions discussed here. Don't worry about it: You'll always have this chapter as a reference when you want to find just the right transition or modifier to make something happen in your game.

- Scenes are the basic building blocks of an AndEngine game.
- Because Scenes are Entities in AndEngine, we took a deep-dive look at `Entity.java` and the constructors and methods it provides. We looked at each of the key properties of an Entity, and described the way they are used by AndEngine.
- A variety of Modifiers can be applied to any Entity (e.g., Scenes, Sprites) to change the way an Entity is displayed. We looked at each of the Modifiers that AndEngine provides to see how we might use them to change the Entities in our game. Modifiers can be combined if necessary, either running in parallel or in sequence.
- AndEngine provides a variety of EaseFunctions that can be used to change the progress of the Modifiers.
- We used some of our new knowledge to make a little progress in creating the Virgins Versus Vampires game. We created and loaded the main game screen and started the action, which, for the moment, is not very active.

In the next chapter, we'll look at Sprites in detail. That will start to give us the tools we need to make V3 come alive—or undead, anyway.

Exercises

1. Write a small Activity that you can use to test Modifier combinations. Display a white background with one Entity, a Sprite (you can use the face in the `mathead.png` file, which is included in the downloaded code for this chapter in the Modifiers project).

2. Construct a combination of Modifiers that change an entity according to the following timeline:

	0 second	1 second	2 seconds	3 seconds	4 seconds	5 seconds
Move	From origin to center of screen			No change		
Color	White to blue		Blue to red		Red to green	
Scale	No change	Full size to half size		Half size to full size		No change
Rotate	No change		720 degrees		No change	

3. Right now when MainMenu shrinks, it shrinks into the location given by the coordinates (0.0f, 0.0f). Change the code so it shrinks into the middle of the screen. [*Hint:* Use the `setScaleCenter()` method.]
4. If none of the 34 EaseFunctions provided by AndEngine meets the needs for your game, write your own! Create a new EaseFunction—call it `EaseWiggle.java`—that has a profile something like that shown in Figure 4.14. [*Hint:* Start with `EaseLinear.java` from the AndEngine sources, and modify the `getPercentageDone()` method, using the `FloatMath.sin()` function.]

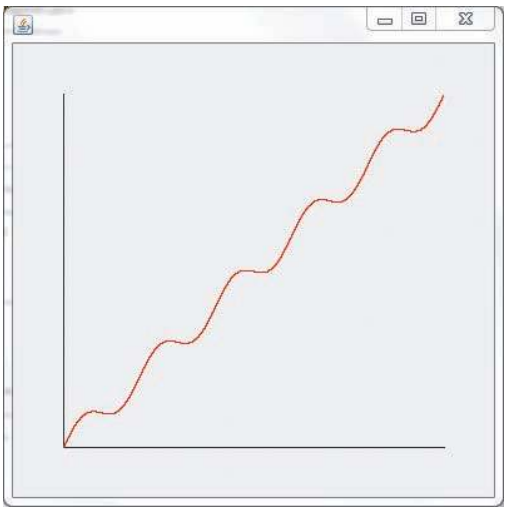


Figure 4.14 EaseWiggle

5

Drawing and Sprites

We've been using Sprites all along in our examples and in the development of V3, but now we'll spend some time exploring them in much more detail. As this chapter explains, AndEngine has methods for creating Sprites, Animated Sprites, and Tiled Sprites. We also want to take a brief look at the methods for drawing primitive graphic objects, such as lines and rectangles.

Most of the objects that are drawn on the screen during your game are Sprites. The term "sprite" originally referred to independent graphical objects that were positioned and animated on top of a game background. As computer games evolved, however, the term came to be used for just about any graphical object. For example, the backgrounds for our AndEngine scenes are loaded into Sprite objects to be displayed.

AndEngine uses OpenGL to render objects to the screen. On most Android devices, OpenGL is implemented with hardware accelerators, so that's the setup that we'll assume in our discussion. Using AndEngine frees you from most of the details of working with OpenGL, but we'll also cover some of what's going on under the covers.

Quick Look Back at Entity

In AndEngine, Sprite is a subclass of the abstract class BaseSprite, which in turn is a subclass of the abstract class BaseRectangle, which in turn is a subclass of the abstract class RectangularShape, which in turn is a subclass of the abstract class Shape, which in turn is a subclass of the class Entity. If that seems confusing, perhaps a picture will help. A simplified class diagram of the Entity class and some subclasses is shown in Figure 5.1.

We could go through each class and determine which capabilities are added at each level, but we don't really need to understand that level of detail to write a game. It's good to know how everything fits together, however, and as we describe drawing and Sprites, we'll cover the important relationships and methods we need to get our game built.

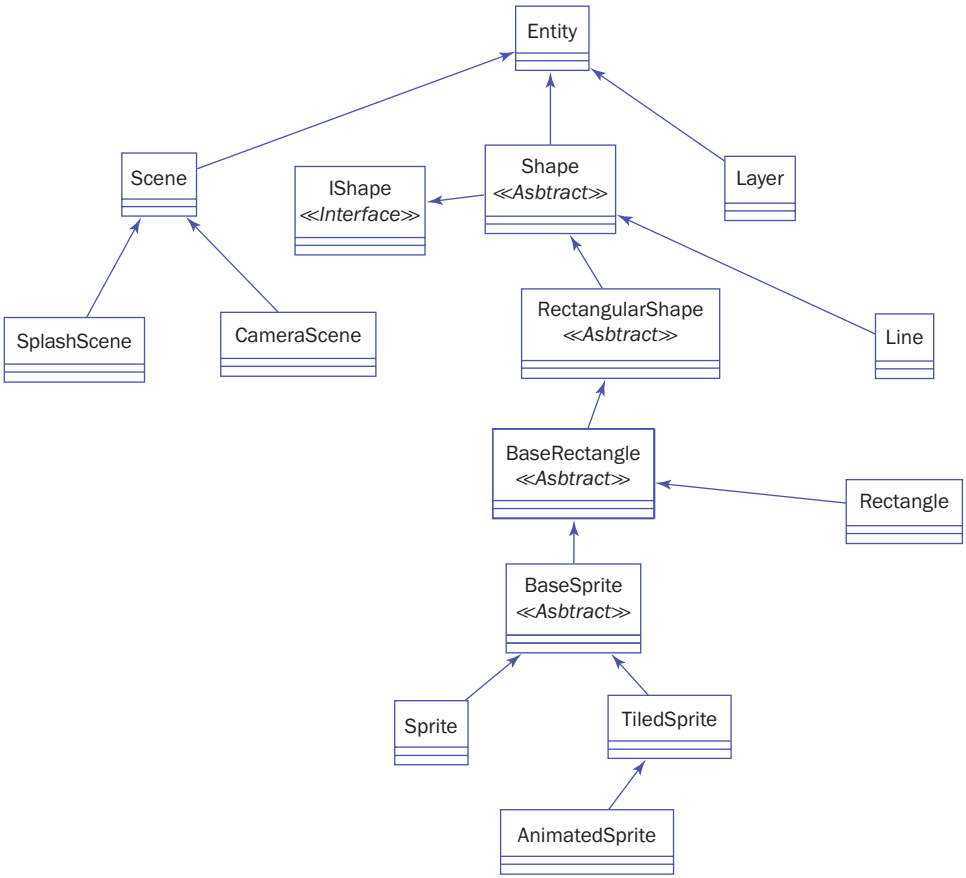


Figure 5.1 Entity simplified class diagram

Drawing Lines and Rectangles

AndEngine doesn't provide a lot of drawing features. For most games, the artwork is created as a series of bitmaps, and drawing complicated graphics at runtime is rarely necessary. AndEngine gives us a way to draw both Lines and Rectangles.

Line

In addition to the variables that Line inherits from Entity, it has a second position (the end-point of the line) and a line width. These parameters can be seen in the Line constructors:

```
Line(final float pX1, final float pY1, final float pX2, final float pY2)
Line(final float pX1, final float pY1, final float pX2, final float pY2,
        final float pLineWidth)
```

If you don't pass in a `pLineWidth`, the value defaults to 1 pixel. The first set of coordinates is taken as the "position" of the Line, and the last set of coordinates is the "other end" of the Line. Attributes such as color and transparency are handled by the Entity superclass.

Rectangle

AndEngine also provides a Rectangle drawing primitive that has the following constructors:

```
Rectangle(final float pX, final float pY, final float pWidth, final float pHeight)
Rectangle(final float pX, fina float pY, final float pWidth, final float pHeight,
        final RectangleVertexBuffer pRectangleVertexBuffer)
```

Here, the *X* and *Y* values mark the upper-left corner of the Rectangle; the width and height are self-explanatory. Once again, color and transparency are handled by the superclass, and the Rectangle is drawn with a fill that corresponds to them (to draw an outline rectangle, you draw four lines).

The optional `RectangleVertexBuffer` parameter can be used to improve the drawing speed if a bunch of Rectangles are needed, or it can be used to distort the displayed rectangle. These capabilities are an OpenGL topic that is beyond the scope of this book, but if you're interested, Google "OpenGL vertex buffer"; you should find plenty of online reference material.

Sprites

Looking toward the bottom of Figure 5.1, we see AndEngine provides three different kinds of Sprites:

- A Sprite uses a single texture, extracted from a TextureRegion.
- A TiledSprite chooses a texture, taken from a regular array of textures in a TiledTextureRegion.
- An AnimatedSprite is a subclass of TiledSprite, whose texture changes at a regular frame rate to show an animation.

Textures

Before we get into Sprites and all their variations, we need to develop some background about the way AndEngine treats textures. If you haven't done much graphics programming, it's helpful to understand that textures are just bitmaps that are "painted" onto objects like Sprites as they are displayed. AndEngine stores collections of textures in memory as instances of the Texture class. There is a class Texture, and a singleton TextureManager manages all the Textures for your game.

Each Texture can have multiple TextureRegions inside it, which identify bitmaps in the Texture. AndEngine has two fundamental types of TextureRegions:

- A TextureRegion usually contains one image, with a unique height and width. An example TextureRegion image is shown in Figure 5.2.
- A TiledTextureRegion usually contains more than one image, where each image has the same height and width. The images are organized as an array of tiles that can be referenced by their position in the array. An example TiledTextureRegion image is shown in Figure 5.3.

Texture

A few constructors are supplied to create a new Texture:

```
Texture(final int pWidth, final int pHeight)
Texture(final int pWidth, final int pHeight, final ITextureStateListener
        pTextureStateListener)
Texture(final int pWidth, final int pHeight, final TextureOptions
        pTextureOptions)
Texture(final int pWidth, final int pHeight, final TextureOptions
        pTextureOptions, final ITextureStateListener pTextureStateListener)
```

All of them create a blank canvas with the given dimensions into which you can load textures for your Sprites. The dimensions of the Texture—that is, `pWidth` and `pHeight`—must be powers of 2 (32, 64, 128, …), and the dimensions must be large enough to hold all of the textures you intend to load into the Texture. If these values

Figure 5.2 TextureRegion image

Figure 5.3 TiledTextureRegion image

are not powers of 2, AndEngine will throw an IllegalArgumentException. If you try to load a bitmap that doesn't fit in your Texture, you will also get an exception. Either type of exception will force your game to close, if it is not caught, with a log message being entered in LogCat.

The optional `TextureStateListener` parameter, in the second and fourth constructors, is triggered when a texture is loaded into the Texture, or unloaded, or when an error occurs during loading. We won't use that feature in our work here, but if you're interested, there is an example for using it, `ImageFormatsExample.java`, included in the AndEnginesExample source.

The optional `TextureOptions` parameter controls the way OpenGL displays the textures. We will often use the default, but the following options are also available:

- `TextureOptions.NEAREST`
- `TextureOptions.BILINEAR`
- `TextureOptions.REPEATING`
- `TextureOptions.REPEATING_BILINEAR`
- `TextureOptions.NEAREST_PREMULTIPLYALPHA` (the default)
- `TextureOptions.BILINEAR_PREMULTIPLYALPHA`
- `TextureOptions.REPEATING_PREMULTIPLYALPHA`
- `TextureOptions.REPEATING_BILINEAR_PREMULTIPLYALPHA`

Each of these options sets the OpenGL texture filters according to patterns defined in `TextureOptions.java`, which is part of the AndEngine sources. Describing the effect of each option is an OpenGL topic beyond the scope of this book, but the differences are real, especially as you scale or rotate textures. If you are concerned about the details of texture rendering, I encourage you to read up on the topic. In particular, an excellent discussion of texture rendering can be found at `http://www.opengl.org/wiki/Texture`.

For most of the examples in the book we will use `BILINEAR_PREMULTIPLYALPHA`, which asks OpenGL to use the following properties:

- `GL_LINEAR`: use linear interpolation in each (x, y) direction to determine the color of pixels when magnifying or minimizing a texture.
- `GL_CLAMP_TO_EDGE`: textures don't wrap around, so clamp-normalize texture coordinates to [0.1].
- `GL_MODULATE`: combine textures by multiplying them.
- Use premultiplied alpha blending, which usually creates more realistic combinations of textures when they overlap.

TextureRegionFactory

So far, all we've done is to create a blank storage area for textures. To load images into the Texture, we use TextureRegionFactory. We've been using this approach all

along to create textures for the Sprites in V3, but now we want to look at all of the capabilities that are possible. TextureRegionFactory knows how to create TextureRegions and TiledTextureRegions from three types of sources:

- Assets: bitmap files stored under the `assets` folder in your game project. This is the method we've been using so far.
- Resources: drawable files stored as resources under the `res` folder of your game project.
- TextureSources: a more generic name for drawable resources and assets. The factory methods for resources and assets are implemented with these methods, and they might also be useful if you're reusing a TextureSource.

A method is available for each of these sources, and for each type of texture region (tiled or not). Another set is available for BuildableTextures, which we'll discuss in the next section. Here are the basic methods for the previously mentioned sources:

```
TextureRegion createFromAsset(final Texture pTexture,
        final Context pContext, final String pAssetPath,
        final int pTexturePositionX, final int pTexturePositionY)
TiledTextureRegion createTiledFromAsset(final Texture pTexture,
        final Context pContext, final String pAssetPath,
        final int pTexturePositionX, final int pTexturePositionY,
        final int pTileColumns, final int pTileRows)
TextureRegion createFromResource(final Texture pTexture,
        final Context pContext, final int pDrawableResourceID,
        final int pTexturePositionX, final int pTexturePositionY)
TiledTextureRegion createTiledFromResource(final Texture pTexture,
        final Context pContext, final int pDrawableResourceID,
        final int pTexturePositionX, final int pTexturePositionY,
        final int pTileColumns, final int pTileRows)
TextureRegion createFromSource(final Texture pTexture,
        final ITextureSource pTextureSource, final int pTexturePositionX,
        final int pTexturePositionY)
TiledTextureRegion createTiledFromSource(final Texture pTexture,
        final ITextureSource pTextureSource, final int pTexturePositionX,
        final int pTexturePositionY, final int pTileColumns, final int pTileRows)
```

The parameters for these methods are defined as follows:

- `Texture pTexture`: the Texture you're loading into.
- `Context pContext`: the Activity Context for the current Activity.
- `String pAssetPath`: the filename for the asset, referenced to the `assetPath`, which is the assets folder by default. (The `setAssetPath()` method is discussed later in this chapter.) AndEngine currently knows how to load PNG, JPG, and BMP images.

- `pDrawableResourceID`: the integer assigned to the resource in R.java, usually referenced as `R.drawable.<resourcename>`.
- `ITextureSource pTextureSource`: the TextureSource to be loaded.
- `int pTexturePositionX, int pTexturePositionY`: the location on the Texture where the image should be loaded. It identifies the position for the upper-left corner of the image, and the Texture must be large enough to hold the image at that position. Images should not overlap, unless you're doing something strange.
- `int pTileColumns, int pTileRows`: for TiledTextureRegions, the number of columns and rows in the tiled image.

TextureRegionFactory also contains a useful method for setting the base asset path to something beneath the `assets` folder. This method is helpful if you keep your images in separate subfolders, and you want to avoid typing the whole path name for every image:

```
void setAssetBasePath(final String pAssetBasePath)
```

The parameter is the partial path name, and it must end in "/". If it doesn't, or if it's zero length, an exception will be thrown.

Example: Creating a TextureRegion from an Asset

We used `setAssetBasePath()` and `TextureRegionFactory.createFromAsset()` in Chapter 4 when we set up the TextureRegions for Level 1 of the V3 game. Listing 5.1 is an excerpt from that code showing the normal pattern for creating a Sprite from an image file in the `assets` folder.

Listing 5.1 **Level1Activity.java Excerpt Using TextureRegionFactory.createFromAsset()**

```
. . .
   @Override
   public void onLoadResources() {
      /* Load Textures. */
      TextureRegionFactory.setAssetBasePath("gfx/Level1/");
      mLevel1BackTexture = new Texture(512, 512,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      mLevel1BackTextureRegion =
      TextureRegionFactory.createFromAsset(
         this.mLevel1BackTexture,
         this, "Level1Bk.png", 0, 0);
      mEngine.getTextureManager().loadTexture(
this.mLevel1BackTexture);
. . .
}
```

We start out by setting the asset base path to `"gfx/Level1/"` because that's the way we've chosen to structure the `assets` subfolder. All of the graphics are located in subfolder `gfx`, and the graphics for each level will placed in a separate subfolder under `gfx`. Remember to include the final "/"; if you omit it, AndEngine will throw an exception.

We then create a Texture big enough to hold the background for Level 1. The background is a PNG file, 480 × 320 pixels in size. The next highest power of 2 is 512 in each case, so we create the Texture as a 512 × 512 pixel space. I've chosen a TextureOption of `BILINEAR_PREMULTIPLYALPHA` in imitation of the examples.

We then create the TextureRegion from the PNG file stored under `assets` and add it to the Texture at position (0, 0). If we were loading other TextureRegions into this Texture, we would load them at positions that did not conflict with the background Texture.

Finally, we ask the singleton TextureManager to load the Texture into its cache of Textures. Now our new Texture will be available for our game.

Example: Creating a TextureRegion from a Resource

There is one big advantage to creating TextureRegions from resources, rather than assets. As you may know, Android defines three screen resolutions (hdpi, mdpi, and ldpi) and manages graphic resources for them, so as to manage the problem of working with different screen resolutions. When you place images in `res/drawable-hdpi`, `res/drawable-mdpi`, and `res/drawable-ldpi` and subsequently reference them in an Android application (as `R.drawable.<filename>`), Android will use the image that most closely matches the screen geometry of the device your application is running on. That capability can be a big help in managing screen geometries.

Listing 5.2 shows an example of using resources instead of assets for the obstacle textures in `Level1Activity.java`.

Listing 5.2 **Level1Activity.java Excerpt Modified to Use createFromResource()**

```
package com.pearson.lagp.v3;
. . .
    @Override
    public void onLoadResources() {
       /* Load Textures. */
       mLevel1BackTexture = new Texture(512, 512,
          TextureOptions.BILINEAR_PREMULTIPLYALPHA);
       mLevel1BackTextureRegion =
       TextureRegionFactory.createFromResource(
          this.mLevel1BackTexture, this, R.drawable.level1bk,
          0, 0);
       mEngine.getTextureManager().loadTexture(
this.mLevel1BackTexture);
. . .
}
```

To make this process work, you first have to populate the `res/drawable-xdpi` folders with images suitable for those resolutions. The filenames have to be all lowercase ("level1bk.png" in this case), and the method takes a reference to the resource—that is, "R.drawable" plus the filename with no extension. The images need to be PNGs or JPGs (Android can deal with GIFs, but AndEngine cannot—PNG should be your first choice).

Example: Creating a TextureRegion from a Vector (SVG) Source

An extension to AndEngine allows you to render SVG vector graphics files at runtime. This is a very useful addition to the basic game engine, particularly for games that may be run on high-definition (HD) screens. When the vector file is rendered at runtime, the rendering can be optimized for the available screen resolution without any compromise in the graphics' appearance. Typically, if you render SVGs at development time, you create at least three bitmap images—one for low-resolution screens, one for mid-resolution screens, and one for high-resolution screens.

To use the extension, you need to load its `.jar` (Java archive) file from the following website:

`http://code.google.com/p/andenginesvgtextureregionextension`

Place the `.jar` file in the `lib` folder in your AndEngine project, right-click on the `.jar` filename, and choose Build Path > Add to Build Path from the pop-up menu. Once the addition is available, you can load textures from SVG graphics as shown in Listing 5.3.

Listing 5.3 **Level1Activity.java Excerpt Using SVG Graphics**

```
. . .
    @Override
    public void onLoadResources() {
       /* Load Textures. */
       mLevel1BackTexture = new Texture(512, 512,
          TextureOptions.BILINEAR_PREMULTIPLYALPHA);
       mLevel1BackTextureRegion =
       mLevel1BackTextureSource = new SVGAssetTextureSource(
          this, "svg/hatchet40.svg", 1.0f);
       TextureRegionFactory.createFromSource(
          this.mLevel1BackTexture,
          this.mLevel1BackTextureSource, 0, 0);
       mEngine.getTextureManager().loadTexture(
this.mLevel1BackTexture);
. . .
```

BuildableTexture

The preceding set of methods works well for building TextureRegions and loading them into Textures. If you've actually used these methods, however, you probably noticed one drawback: You have to tell TextureRegionFactory specifically where you want each image loaded into the Texture.

Suppose you have three images to load:

- `One.png`: 50 × 121 pixels
- `Two.png`: 78 × 324 pixels
- `Three.png`: 233 × 43 pixels

You have to figure out how you want to pack these images into a Texture, and then carefully compute coordinates and dimensions to come up with the parameters for the TextureRegionFactory calls. That's kind of a pain—so AndEngine provides a set of methods that do the math for us. We used a BuildableTexture and these methods in Chapter 4, within `Level1Activity.java`. Listing 5.4 is a part of that file that illustrates how these methods are used.

Listing 5.4 **Level1Activity.java Excerpt Using BuildableTexture**

```
package com.pearson.lagp.v3;
. . .
    @Override
    public void onLoadResources() {
       /* Load Textures. */
. . .
       mObstacleBoxTexture = new BuildableTexture(512, 256,
          TextureOptions.BILINEAR_PREMULTIPLYALPHA);
       mBoxTextureRegion =
          TextureRegionFactory.createFromAsset(mObstacleBoxTexture,
              this, "Obstaclebox.png");
       mBulletTextureRegion =
          TextureRegionFactory.createFromAsset(mObstacleBoxTexture,
              this, "Bullet.png");
       mCrossTextureRegion =
          TextureRegionFactory.createFromAsset(mObstacleBoxTexture,
this, "Cross.png");
       mHatchetTextureRegion =
          TextureRegionFactory.createFromAsset(mObstacleBoxTexture,
              this, "Hatchet.png");
          try   {
           mObstacleBoxTexture.build(
            new BlackPawnTextureBuilder(2));
          } catch (final TextureSourcePackingException e) {
                Log.d(tag,
            "Sprites won't fit in mObstacleBoxTexture");
          }
       this.mEngine.getTextureManager().loadTexture(this
.mObstacleBoxTexture);
    }
. . .
}
```

The constructors for BuildableTexture are exact analogs of the constructors listed earlier for Texture, and they have the same constraints (i.e., must have power of 2 dimensions, must be big enough to hold all the textures). The createFrom … methods are also exact analogs to their previously mentioned counterparts, with the Texture parameter replaced with a BuildableTexture, and no coordinate parameters.

The main difference becomes apparent after you've created all of the TextureRegions. Before you make use of the BuildableTexture, you must build the TextureRegions into it by calling the BuildableTexture's build method, using a builder class such as BlackPawnTextureBuilder (the only builder provided right now, but that could change). If the build succeeds, the TextureRegions are all packed into the BuildableTexture and you can access them by name. If the build does not succeed (e.g., the textures don't all fit in the allocated space), you can catch the exception, as shown in the excerpt in Listing 5.3.

Another Way to Build Textures

Perhaps you don't want to build your Textures at runtime. If so, you might want to use the popular tool called Zwoptex for building Textures (or sprite sheets, as they are sometimes called). You can't import a Zwoptex sprite sheet directly into AndEngine, but you can use it to arrange images and identify what those images' coordinates should be. Zwoptex comes in two flavors:

- An older, web-based (Adobe Flash) version is still freely available but is no longer supported (`http://zwoptexapp.com/flashversion`).
- The supported, more fully featured version, which runs on only Mac OS X (10.6 or later), is available for a small fee (`http://zwoptexapp.com/buy`).

Either version can be used to automatically combine images into a single, larger sprite sheet, or Texture. The result is a sprite sheet image file and an XML list of coordinates. AndEngine does not yet know how to import the image and list (perhaps someone will have added that capability by the time you read these words), but you can look at the XML list and easily find the coordinates needed for `createFromAsset()` without having to do the calculations yourself.

As an example, I've used Zwoptex Flash to combine the images for the Obstacle Box in Level 1 of V3. Figure 5.4 shows the resulting sprite sheet. Listing 5.5 provides the relevant part of the resulting XML file.

Figure 5.4 Texture for obstacle box

Listing 5.5 **Level1Activity.java Excerpt Using BuildableTexture**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
      "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>texture</key>
   <dict>
      <key>width</key>
      <integer>256</integer>
      <key>height</key>
      <integer>128</integer>
   </dict>
   <key>frames</key>
   <dict>
      <key>Bullet.png</key>
      <dict>
         <key>x</key>
         <integer>45</integer>
         <key>y</key>
         <integer>1</integer>
         <key>width</key>
         <integer>11</integer>
         <key>height</key>
         <integer>33</integer>
         <key>offsetX</key>
         <real>0</real>
         <key>offsetY</key>
         <real>0</real>
         <key>originalWidth</key>
         <integer>11</integer>
         <key>originalHeight</key>
         <integer>33</integer>
      </dict>
      <key>Cross.png</key>
      <dict>
         <key>x</key>
         <integer>1</integer>
         <key>y</key>
         <integer>1</integer>
. . .
      </dict>
   </dict>
</dict>
</plist>
```

For each image, the `pX` and `pY` coordinates are given as the values of the keys x and y, respectively. You can easily use Zwoptex to illustrate what the assembled Texture will look like, to identify the coordinates for each image, and to ensure the Texture is large enough to hold all the images.

The Sprite Class

Now that we have TextureRegions with the images we'd like to use for our Sprites, we can create the Sprites themselves. The Sprite class has four constructors:

Sprite(final float pX, final float pY, final TextureRegion pTextureRegion)
Sprite(final float pX, final float pY, final float pWidth, final float pHeight, final TextureRegion pTextureRegion)
Sprite(final float pX, final float pY, final TextureRegion pTextureRegion, final RectangleVertexBuffer pRectangleVertexBuffer)
Sprite(final float pX, final float pY, final float pWidth, final float pHeight, final TextureRegion pTextureRegion, final RectangleVertexBuffer pRectangleVertexBuffer)

All of the constructors require an initial position for the Sprite, given by the parameters `pX` and `pY`. All of them also require a TextureRegion that tells AndEngine and OpenGL which texture to paint for the Sprite. Optionally, you can specify a width and height for the Sprite (otherwise, it will default to the width and height of the TextureRegion). You can also specify a `RectangleVertexBuffer`, just as you could for Rectangle.

Because Sprite is a subclass of Entity, all of the Entity methods are part of Sprites as well. One new method is also introduced:

```
TextureRegion getTextureRegion()
```

This method returns the Sprite's TextureRegion, as you might expect. Note that there is no method to set the Sprite's TextureRegion. That task must be completed when the Sprite is created, unless the Sprite is associated with a tiled map of textures (as described in the next section).

TiledSprites

We can use a TiledSprite to create a sprite that is associated with an array of textures stored in a TiledTextureRegion. A TiledTextureRegion usually has more than one texture, all of the same size, stored in one big array. We saw how TiledTextureRegions are created earlier, in the section on TextureRegionFactory. We can now create the TiledSprites themselves with constructors analogous to the Sprite constructors:

TiledSprite(final float pX, final float pY, final TiledTextureRegion pTiledTextureRegion)
TiledSprite(final float pX, final float pY, final float pTileWidth, final float pTileHeight, final TiledTextureRegion pTiledTextureRegion)

```
TiledSprite(final float pX, final float pY, final TiledTextureRegion
        pTiledTextureRegion, final RectangleVertexBuffer
        pRectangleVertexBuffer)
TiledSprite(final float pX, final float pY, final float pTileWidth,
        final float pTileHeight, final TiledTextureRegion pTiledTextureRegion,
        final RectangleVertexBuffer pRectangleVertexBuffer)
```

The parameters here are very similar to those for the Sprite constructors. Instead of a TextureRegion, however, we pass a TiledTextureRegion, as you would expect. Instead of a Sprite width or height, we have the option to pass tile dimensions.

TiledSprites also add some unique methods to BaseSprite, including these four:

```
int getCurrentTileIndex()
void setCurrentTileIndex(final int pTileIndex)
void setCurrentTileIndex(final int pTileColumn, final int pTileRow)
void nextTile()
```

The first three methods get and set the current tile index for the TiledSprite. They enable you to determine which texture is being shown now, or to change it if you like. The last method advances to the "next" tile. AndEngine orders the tiles as shown in Figure 5.5.

AnimatedSprites

If we want a Sprite to be animated, we must supply all the animation textures that we want AndEngine to apply to the Sprite. As shown in the simplified class diagram in Figure 5.1, AnimatedSprite is a subclass of TiledSprite—an arrangement that enables us to get the needed animation textures from a TiledTextureRegion. The constructors for AnimatedSprite are exactly analogous to those for TiledSprite:

```
AnimatedSprite(final float pX, final float pY, final TiledTextureRegion
        pTiledTextureRegion)
AnimatedSprite(final float pX, final float pY, final float pTileWidth,
        final float pTileHeight, final TiledTextureRegion pTiledTextureRegion)
AnimatedSprite(final float pX, final float pY, final TiledTextureRegion
        pTiledTextureRegion, final RectangleVertexBuffer
        pRectangleVertexBuffer)
```

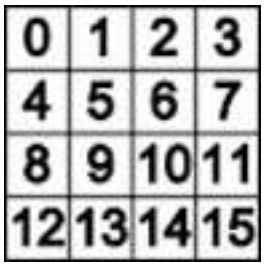


Figure 5.5 Tile order

AnimatedSprite(final float pX, final float pY, final float pTileWidth, final float pTileHeight, final TiledTextureRegion pTiledTextureRegion, final RectangleVertexBuffer pRectangleVertexBuffer)

We'll see a lot more of AnimatedSprites in Chapter 6, where we talk about animation in more depth.

A Word about Performance

As you develop your game, and you begin asking the Android device to do more and more, performance may get to be an issue. Displaying a lot of graphics, running modifiers on them, and implementing the game logic all take computer cycles and affect the rate at which your game runs.

In each of the code examples provided so far in this book, we've included the AndEngine standard frame rate counter FPSLogger, which logs the frame rate to LogCat. The log messages look like this:

```
D/AndEngine( 448): FPS: 9.73 (MIN: 96 ms | MAX: 134 ms)
```

This example message was made with the Android emulator, so the frame rate is quite low—less than 10 frames per second (at this point, the game is simply displaying a bitmap). We'd like the frame rate to be as high as possible, so play can be as natural as possible.

The way a Sprite is created affects the total game performance. When Sprites are bound to individual Textures, a series of things must happen for AndEngine to render them. In pseudocode:

```
<loop over all Sprites to be displayed>
   <initialize OpenGL rendering>
   <render a Sprite>
   <clean up rendering engine>
<end loop>
```

When multiple Sprites are bound to images contained within a single Texture, the initialization and cleanup occur outside the loop. That is, they happen only once for all the Sprites to be displayed, and the game can perform much better:

```
<initialize OpenGL rendering>
<loop over all Sprites to be displayed>
   <render a Sprite>
<end loop>
<clean up rendering engine>
```

Games typically display a lot of Sprites, so using combined Textures speeds up rendering and, therefore, increases the frame rate of your game.

Compound Sprites

All of the Sprites we've seen so far have been childless. As with any Entity, however, you can attach children to a Sprite, and treat the whole collection as a group. Such an arrangement can be essential if, for example, you want to attach a weapon to an actor in your game. If you then move or rotate the actor, the weapon should move with

him, of course. The method for attaching a child node to a Sprite is the same as it is for any other Entity:

```
Entity.attachChild(final IEntity pEntity)
```

where `pEntity` is the child Sprite. Now if a Modifier is registered with the parent Sprite, it will also apply to any children of that Sprite.

To illustrate this point, the example code for this chapter consists of one Activity, called SpriteTestActivity. It features a new character, Mad Mat, who wields a hatchet. Figure 5.6 shows a screenshot of the application as it is running.

Mad Mat on the left—we'll call him Mad Mat0, because that's what he's called in the code—does not move. Mad Mat in the middle (also known as Mad Mat 1) rotates in place, and his hatchet also rotates in place. Mad Mat on the right (also known as Mad Mat 2) rotates *with* his hatchet, which is what you'd normally want to happen. Listing 5.6 shows the relevant code.

Listing 5.6 **SpriteTestActivity.java and Compound Sprites**

```
package com.pearson.lagp.v3;

import org.anddev.andengine.engine.Enginatq e;
import org.anddev.andengine.engine.camera.Camera;
```

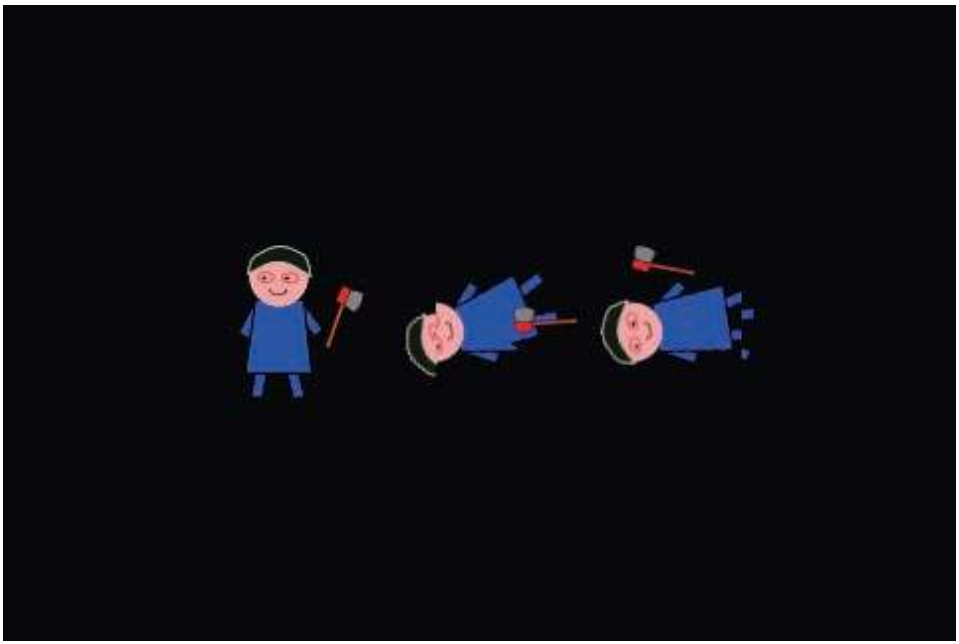


Figure 5.6 SpriteTestActivity screenshot

```
import org.anddev.andengine.engine.options.EngineOptions;
import org.anddev.andengine.engine.options.EngineOptions
   .ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy
   .RatioResolutionPolicy;
import org.anddev.andengine.entity.modifier.RotationModifier;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.scene.background.ColorBackground;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.entity.util.FPSLogger;
import org.anddev.andengine.opengl.texture.BuildableTexture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.builder
   .BlackPawnTextureBuilder;
import org.anddev.andengine.opengl.texture.builder.ITextureBuilder
   .TextureSourcePackingException;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.ui.activity.BaseGameActivity;

import android.util.Log;

public class SpriteTestActivity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;
   private String tag = "SpriteTestActivity";

   // ===========================================================
   // Fields
   // ===========================================================

   protected Camera mCamera;

   protected Scene mMainScene;

   private BuildableTexture mTestTexture;
   private TextureRegion mMadMatTextureRegion;
   private TextureRegion mHatchetTextureRegion;

   // ===========================================================
   // Constructors
   // ===========================================================
```

```
// ============================================================
// Getter and Setter
// ============================================================

// ============================================================
// Methods for/from SuperClass/Interfaces
// ============================================================

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT),
      this.mCamera));
}

@Override
public void onLoadResources() {
   /* Load Textures. */
   TextureRegionFactory.setAssetBasePath("gfx/SpriteTest/");
   mTestTexture = new BuildableTexture(512, 256,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mMadMatTextureRegion =
      TextureRegionFactory.createFromAsset(mTestTexture,
         this, "madmat.png");
   mHatchetTextureRegion =
      TextureRegionFactory.createFromAsset(mTestTexture,
         this, "hatchet40.png");
      try {
      mTestTexture.build(
      new BlackPawnTextureBuilder(2));
       } catch (final TextureSourcePackingException e) {
            Log.d(tag,
         "Sprites won't fit in mTestTexture");
         }
   this.mEngine.getTextureManager().loadTexture(
      this.mTestTexture);
   }

@Override
public Scene onLoadScene() {
   this.mEngine.registerUpdateHandler(new FPSLogger());

   final Scene scene = new Scene(1);
   scene.setBackground(new ColorBackground(0.0f, 0.0f, 0.0f));
```

```
/* Center the camera. */
final int centerX = CAMERA_WIDTH / 2;
final int centerY = CAMERA_HEIGHT / 2;

/* Create the sprites and add them to the scene. */
final Sprite madMat0 = new Sprite(centerX -
      (mMadMatTextureRegion.getWidth() / 2) -
         100.0f,
      centerY - (mMadMatTextureRegion.getHeight()
         / 2),
      mMadMatTextureRegion);
scene.getLastChild().attachChild(madMat0);
final Sprite hatchet0 = new Sprite(madMat0.getInitialX() +
   44.0f,
   madMat0.getInitialY() + 20.0f,
   mHatchetTextureRegion);
scene.getLastChild().attachChild(hatchet0);

final Sprite madMat1 = new Sprite(centerX -
      (mMadMatTextureRegion.getWidth() / 2),
      centerY - (mMadMatTextureRegion.getHeight()
         / 2),
      mMadMatTextureRegion);
scene.getLastChild().attachChild(madMat1);
final Sprite hatchet1 = new Sprite(madMat1.getInitialX() +
      44.0f,
   madMat1.getInitialY() + 20.0f,
   mHatchetTextureRegion);
madMat1.registerEntityModifier(
      new RotationModifier(3, 0, 360)
      );
hatchet1.registerEntityModifier(
      new RotationModifier(3, 0, 360)
      );
scene.getLastChild().attachChild(hatchet1);

final Sprite madMat2 = new Sprite(centerX -
      (mMadMatTextureRegion.getWidth() / 2) +
         100.0f,
      centerY - (mMadMatTextureRegion.getHeight()
         / 2),
      mMadMatTextureRegion);
final Sprite hatchet2 = new Sprite( 44.0f, 20.0f,
      mHatchetTextureRegion);
madMat2.attachChild(hatchet2);
madMat2.registerEntityModifier(
      new RotationModifier(3, 0, 360)
      );
```

```
        scene.getLastChild().attachChild(madMat2);
        return scene;
    }

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

The bits of interest are all found in `onLoadScene()`. For each of the three Mad Mats, we create a Sprite for Mat and a Sprite for the hatchet. For Mad Mat 1, we add an EntityModifier to each of the Sprites, so they rotate. They rotate in their individual positions, however—which is not the behavior normally desired for a character carrying something.

In Mad Mat 2, we add the hatchet Sprite as a child of the Mad Mat 2 Sprite. Then we add the Modifier only to Mad Mat, and we add only the Mad Mat Sprite to the Scene (it brings its children with it). When that Modifier executes, it rotates the whole compound Sprite as a unit. Note that the positions we specified when creating the Sprite for hatchet2 are relative to the Parent, not relative to the Scene, like the positions for hatchet0 and hatchet1. In fact, Sprite positions are always relative to the Parent; we emphasize this point because all of the other Sprites we've seen have the Scene as their Parent.

Summary

This chapter provided an in-depth look at Sprites and the Entity class. We didn't advance the V3 game in this chapter but instead took a closer look at the way we've been creating and displaying Sprites for the game.

Now that we know more about how Sprites fit into the class hierarchy of AndEngine, and the constructors and methods used to create them, let's summarize the steps needed to place a Sprite on the game screen:

1. All Sprites need a TextureRegion, and TextureRegions are loaded into Textures. In the `onLoadResources()` method, we create a Texture large enough to hold all our TextureRegions. If we want AndEngine to build the Texture for us as we add TextureRegions, we create a BuildableTexture.
2. The TextureRegions we need for our Sprite are loaded into the Texture using `TextureRegionFactory.createFrom . . . ()` methods. TextureRegions can be loaded from image files under the asset folder, from images that have been created as Android resources, or from other TextureRegions.
3. If we used a BuildableTexture, we call `BuildableTexture.build()` to complete the process of building the Texture.
4. We ask the TextureManager to load our new Texture into its cache, using the `Engine.getTextureManager().loadTexture()` method.

5. In the `onLoadScene()` method, we actually create the Scene, including the Sprites that are part of the Scene. Sprites are created using the TextureRegions we loaded into the Texture, any needed Modifiers are created and registered with the Sprites, and the Sprites are attached to the Scene as children.
6. When we return the Scene at the end of the method, AndEngine will display it.

Exercises

1. Create a simple program that draws a red, five-pointed star (in outline). Make the star spin around its center point. You'll need to use the idea of compounding to make the star spin properly.
2. Change `SpriteTestActivity.java` so that it creates all of its TextureRegions from SVG files instead of PNGs.
3. Try using Zwoptex to create a Texture image from your own set of game images. Zwoptex includes a number of options we didn't explore in this chapter (e.g., spacing, different packing strategies). Use Zwoptex to create a TiledTexture image by assembling images that are all the same size.

This page intentionally left blank

6

Animation

Now the fun begins in earnest as we take a look at animating our sprites. Although some games don't use animations, almost any game can be enhanced by introducing animated figures.

Requirements for Animation

When we talk about animation in games, we're talking about sprites that change form as they move during the game's action. In the context of AndEngine, animations are distinct from simple translation and rotation of objects, where the form of the object doesn't change, even though it's moving. Rotation and translation were covered in the discussion of Modifiers in Chapter 4.

To animate an object, we need to display frames that represent the object's form—one frame at a time, with each frame shown for a set time. Our eyes and brain blend this series of pictures into a moving scene. The higher the frame rate, the better the illusion. Television in the United States produces images at a speed of 30 frames per second; movies deliver 24 frames per second. While some people claim they can detect variations in higher frame rates, others can't see any difference between 30 fps and 60 fps.

We'll use an animated bat for the first example in this chapter. The frames in the animation sequence are shown in Figures 6.1, 6.2, and 6.3.

These images were created by hand, using Inkscape (okay, so I'm not Andy Warhol), but they could have been created with just about any drawing package. If you're an experienced animation artist, you probably know more about the image creation process than I do. If you're not, you can get a lot of insight into effective animations by looking at sprite sheets from other games. If you investigate the resources available online, you'll note that many of the sprite sheets for games have been stripped from the game images and made available for study. You are not free to use these sprite sheets in your own game, of course, but you are free to look at them and use them to better understand how to animate your own sprites.

Figure 6.1 bat0.png

Figure 6.2 bat1.png

Figure 6.3 bat2.png

You can also create images (frames) with animation software, such as Anime Studio for Windows, or Pixen for Mac OS X. These packages can make you much more productive by generating the "tween" frames that appear between key frames of animation. In the bat example, we have only three frames, but in general you may have more for your games.

Some of the animation packages allow you to export the individual frames of animation to separate files (as we've done here). Others don't have that ability, but produce animated GIF files or Adobe Flash SWF files. To capture frames from one of these programs, you can use the AnimGet utility described in Chapter 2. It watches an area of the screen and records a snapshot of that area whenever pixels change, which, of course, happens for every animation frame.

Animation Tiled Textures

From our discussion of sprite performance in Chapter 5, we know that it's a good idea to collect all of our animation frames on a common sheet. From what we've learned, we recognize that it will be faster to load the single sheet, and the resulting images can be displayed more rapidly.

I've used GIMP (the open-source bitmap editor) to put together the tiled bat Texture for the animation in Figure 6.4. AndEngine prefers a complete matrix of tiles, each the same size, so I placed each bat image on a 100 × 100 pixel transparent canvas, and stuck the images together on the larger (200 × 200 pixel) image. I duplicated `bat0.png` to fill out the matrix, although I could also have gone with a 1 × 3 set of tiles and not introduced the duplicated frame. We'll see how to do that in the second animation example.

Now is a good time to point out the differences between TiledTextureRegions and sprite sheets. You may be familiar with other game engines that use sprite sheets, and perhaps you have already looked at them as animation examples. Images on a sprite sheet can be placed in any position, in any orientation, and the engine will extract and use them. Images in a TiledTextureRegion are all the same size and are placed in order of the animation sequence. As of this writing, AndEngine doesn't make use of sprite sheets. Animation frames are all taken from TiledTextureRegions.

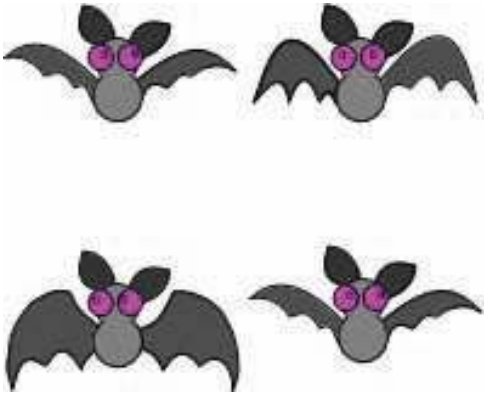


Figure 6.4 bat_tiled.png

Animation in AndEngine

AndEngine includes a special class for animated sprites, called AnimatedSprite. It expects to receive animation frames for the sprite from a TiledTextureRegion. Otherwise, an AnimatedSprite behaves like any other Sprite, and all the Modifiers we talked about in Chapter 4 can be applied to transform an AnimatedSprite.

AnimatedSprite

Four constructors for AnimatedSprite are available:

```
AnimatedSprite(final float pX, final float pY,
        final TiledTextureRegion pTiledTextureRegion)
AnimatedSprite(final float pX, final float pY, final float pTileWidth,
        final float pTileHeight, final TiledTextureRegion pTiledTextureRegion)
AnimatedSprite(final float pX, final float pY,
        final TiledTextureRegion pTiledTextureRegion,
        final RectangleVertexBuffer pRectangleVertexBuffer)
AnimatedSprite(final float pX, final float pY, final float pTileWidth,
        final float pTileHeight, final TiledTextureRegion pTiledTextureRegion,
        final RectangleVertexBuffer pRectangleVertexBuffer)
```

The parameters used are as follows:

- `float pX` and `float pY`: the position of the Sprite. As we saw in Chapter 5, this is the position of the Sprite with respect to its Parent.

- `TileTextureRegion pTiledTextureRegion`: The texture region holding the animation frames for the Sprite. As with Sprites, the TextureRegion was created from an image using TextureRegionFactory and then loaded into a Texture; in turn, the Texture was loaded using the TextureManager.
- `RectangleVertexBuffer pRectangleVertexBuffer`: As we saw with Sprites in Chapter 5, this OpenGL structure can be used to modify the way the Sprite is displayed.
- `float pTileWidth` and `float pTileHeight`: Optionally, the width and height, respectively, of each tile in the TiledTextureRegion. If these values are not given explicitly, AndEngine assumes that the matrix of tiles fills the TiledTextureRegion.

Animate Methods

AnimatedSprite adds a number of methods you can use to control the way it animates the Sprite. The usage pattern is to create an AnimatedSprite using one of the constructors above, and then use its `animate()` method to start the animation. For convenience, let's separate the `animate()` methods into those where the frame duration is the same for all frames, and those for which the frame duration varies by frame. First we present the constant-duration animates:

AnimatedSprite animate(final long pFrameDurationEach)
AnimatedSprite animate(final long pFrameDurationEach, final boolean pLoop)
AnimatedSprite animate(final long pFrameDurationEach, final int pLoopCount)
AnimatedSprite animate(final long pFrameDurationEach, final boolean pLoop, final IAnimationListener pAnimationListener)
AnimatedSprite animate(final long pFrameDurationEach, final int pLoopCount, final IAnimationListener pAnimationListener)

The common parameters are as follows:

- `long pFrameDurationEach`: This parameter identifies the duration, in milliseconds, for which each frame will be displayed. The duration is a goal. Note that we're asking for this frame rate, but given the resources on the runtime platform, we may or may not be able to achieve it.
- `boolean pLoop`: If this parameter's value is `true`, the animation will loop continuously; if it is `false`, the animation plays only once. The default is `true`.
- `int pLoopCount`: If you want the animation to play a specific number of times, use this constructor, and pass the number of animation loops here.
- `IAnimationListener pAnimationListener`: This parameter is a class that implements `IOnAnimationListener`, which has a single callback method called at the end of the animation:

```
void onAnimationEnd(final AnimatedSprite pAnimatedSprite)
```

If you want the frame durations to be different for different frames, you can use similar constructors that pass an array of frame durations. The following methods also give you more flexibility with the array of tiles. If the length of `pFrameDurations` is not equal to the number of frames in the animation, an exception is thrown.

```
AnimatedSprite animate(final long[] pFrameDurations)
AnimatedSprite animate(final long[] pFrameDurations, final boolean pLoop)
AnimatedSprite animate(final long[] pFrameDurations, final int pLoopCount)
AnimatedSprite animate(final long[] pFrameDurations, final boolean pLoop,
        final IAnimationListener pAnimationListener)
AnimatedSprite animate(final long[] pFrameDurations, final int pLoopCount,
        final IAnimationListener pAnimationListener)
AnimatedSprite animate(final long[] pFrameDurations, final int pFirstTileIndex,
        final int pLastTileIndex, final boolean pLoop)
AnimatedSprite animate(final long[] pFrameDurations, final int pFirstTileIndex,
        final int pLastTileIndex, final int pLoopCount)
AnimatedSprite animate(final long[] pFrameDurations, final int[] pFrames,
        final int pLoopCount)
```

These constructors also allow you to specify both the first tile in the TiledTextureRegion to be used in the animation and the last tile. This ability can be convenient, both to display a shortened version of an animation and in case the tiles don't completely fill the TiledTextureRegion. Frame numbers start at 0 and end with the frame count minus 1.

Other Methods

AnimatedSprite also includes two methods you may use to stop the animation (which also call the `pAnimationListener` if you've set one):

```
void stopAnimation()
void stopAnimation(final int pTileIndex)
```

The first method stops the animation wherever it is and leaves the current frame displayed. The second method stops the animation and displays the indicated tile.

Animation Example

To demonstrate how to animate the bat, we'll add it to the splash screen that is shown whenever the game starts. The new splash screen is shown in Figure 6.5. The figure is a static screenshot, but in the running application the bat flaps its wings and bobs up and down.

Figure 6.5 Splash screen with flying bat

The new version of `StartActivity.java` that adds the flying bat is shown in Listing 6.1.

Listing 6.1 **StartActivity.java with Animated Bat**

```
package com.pearson.lagp.v3;


import org.anddev.andengine.engine.Engine;
import org.anddev.andengine.engine.camera.Camera;
import org.anddev.andengine.engine.options.EngineOptions;
import org.anddev.andengine.engine.options.EngineOptions.ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy
  .RatioResolutionPolicy;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.sprite.AnimatedSprite;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.entity.util.FPSLogger;
import org.anddev.andengine.opengl.texture.Texture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.opengl.texture.region.TiledTextureRegion;
import org.anddev.andengine.ui.activity.BaseGameActivity;
```

```
import android.content.Intent;
import android.os.Handler;

public class StartActivity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;

   // ===========================================================
   // Fields
   // ===========================================================

   private Camera mCamera;
   private Texture mTexture, mBatTexture;
   private TextureRegion mSplashTextureRegion;
   private TiledTextureRegion mBatTextureRegion;
   private Handler mHandler;

   // ===========================================================
   // Constructors
   // ===========================================================

   // ===========================================================
   // Getter and Setter
   // ===========================================================

   // ===========================================================
   // Methods for/from SuperClass/Interfaces
   // ===========================================================

   @Override
   public Engine onLoadEngine() {
      mHandler = new Handler();
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH, CAMERA_HEIGHT);
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH, CAMERA_HEIGHT),
         this.mCamera));
   }

   @Override
   public void onLoadResources() {
      TextureRegionFactory.setAssetBasePath("gfx/Splash/");
      this.mTexture = new Texture(512, 1024,
```

```
        TextureOptions.BILINEAR_PREMULTIPLYALPHA);
    this.mSplashTextureRegion =
        TextureRegionFactory.createFromAsset(this.mTexture,
        this, "Splashscreen.png", 0, 0);
    this.mBatTexture = new Texture(256, 256,
        TextureOptions.DEFAULT);
    this.mBatTextureRegion =
        TextureRegionFactory.createTiledFromAsset(
        this.mBatTexture, this, "bat_tiled.png", 0, 0, 2,
            2);
    this.mEngine.getTextureManager().loadTexture(this.mTexture);
    this.mEngine.getTextureManager().loadTexture(this.mBatTexture);
}

@Override
public Scene onLoadScene() {
    this.mEngine.registerUpdateHandler(new FPSLogger());

    final Scene scene = new Scene(1);

    /* Center the splash on the camera. */
    final int centerX = (CAMERA_WIDTH -
          this.mSplashTextureRegion.getWidth()) / 2;
    final int centerY = ( CAMERA_HEIGHT -
          this.mSplashTextureRegion.getHeight()) / 2;

    /* Create the background sprite and add it to the scene. */
    final Sprite splash = new Sprite(centerX, centerY,
        this.mSplashTextureRegion);
    scene.getLastChild().attachChild(splash);

    /* Create the animated bat sprite and add to scene */
    final AnimatedSprite bat = new AnimatedSprite(350, 100,
        this.mBatTextureRegion);
    bat.animate(100);
    scene.getLastChild().attachChild(bat);
    return scene;
}

@Override
public void onLoadComplete() {
    mHandler.postDelayed(mLaunchTask,5000);
}

 private Runnable mLaunchTask = new Runnable() {
     public void run() {
       Intent myIntent = new Intent(StartActivity.this,
```

```
            MainMenuActivity.class);
             StartActivity.this.startActivity(myIntent);
           }
       };
    // ===========================================================
    // Methods
    // ===========================================================

    // ===========================================================
    // Inner and Anonymous Classes
    // ===========================================================
}
```

The important changes are in `onLoadResources()`, `onLoadScene()`, and `onLoadComplete()`.

onLoadResources()

- `mBatTexture`: We've introduced another Texture to hold the TiledTextureRegion that contains the bat animation frames. Our tile image is 100 × 100 pixels, so we choose the next larger power of 2 for the dimensions of the Texture.
- `mBatTiledTextureRegion`: We create this TextureRegion from the tiled image of animation frames, noting that there are two columns and two rows of tiles.
- Finally, we load the Textures using TextureManager as before.

onLoadScene()

- `bat`: We create the AnimatedSprite for the bat, positioning it by the tombstone in the background image.
- We animate the bat, asking for 100 milliseconds between frames (10 frames per second).
- We attach the bat to the current Scene so it will appear.

onLoadComplete()

- We've extended the viewing time of the splash from 3 to 5 seconds, so you have a little longer to view the bat.

The way AndEngine uses TiledTextureRegions for animation images, all images have to be the same size, and it's best if they completely fill the region. When AndEngine displays the animation, it takes the images in sequence, from left to right, top to bottom, starting over after the last frame in the sequence. In the simplest `createTiledFromAsset()` method (and in the related Resource and Source methods), you don't tell AndEngine the dimensions of the tiles but simply indicate how many

columns and how many rows of tiles are present in the region. AndEngine then divides the region into that many subimages and assumes those are the dimensions for the tiles. With the more complicated `createTileFromXX()` methods, you can specify a tile width and height—but that's just more calculation needed at runtime.

Adding Animation to Level1Activity

Back to Level 1 of our game: Let's add some animations that bring some bad guys in from the right of the screen, moving toward the house. Because we don't have a way to detect collisions yet, we'll just have the villains keep walking until they reach the left side of the screen, and have them then pile up there. We'll start 10 of these characters at random times and have them take random paths through the graveyard.

First we need the animation frames for a bad guy. Here I confess I took the easy way out: I purchased an animation from a third party that provides animations for Anime Studio (the animation, which was taken from the Anime Studio website, cost about $12), ran the animation against a blank background in that package, and created a QuickTime movie (`.mov`) of the result. I then used AnimGet to capture frames as I played the movie. The animation has a lot more detail than we need for the small sprites in V3, but that's okay. I edited each frame in GIMP, scaled the images down to 60 × 60 pixels, and created a transparent background for each one. I also flipped the images, because the animation happened to be walking left to right, and all of our sprites will be moving the other way.

Using the images and GIMP to combine them, I created the sprite sheet shown in Figure 6.6.

Figure 6.6 Scrum tiled texture

The modified version of `Level1Activity.java` is shown in Listing 6.2.

Listing 6.2 **Level1Activity with Animated Vampires**

```
package com.pearson.lagp.v3;

import java.util.Arrays;
import java.util.Random;

import org.anddev.andengine.engine.Engine;
import org.anddev.andengine.engine.camera.Camera;
import org.anddev.andengine.engine.options.EngineOptions;
import org.anddev.andengine.engine.options.EngineOptions.ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy
        .RatioResolutionPolicy;
import org.anddev.andengine.entity.modifier.AlphaModifier;
import org.anddev.andengine.entity.modifier.DelayModifier;
import org.anddev.andengine.entity.modifier.FadeInModifier;
import org.anddev.andengine.entity.modifier.MoveModifier;
import org.anddev.andengine.entity.modifier.MoveYModifier;
import org.anddev.andengine.entity.modifier.ParallelEntityModifier;
import org.anddev.andengine.entity.modifier.RotationModifier;
import org.anddev.andengine.entity.modifier.ScaleModifier;
import org.anddev.andengine.entity.modifier.SequenceEntityModifier;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.sprite.AnimatedSprite;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.entity.util.FPSLogger;
import org.anddev.andengine.opengl.texture.BuildableTexture;
import org.anddev.andengine.opengl.texture.Texture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.builder.BlackPawnTextureBuilder;
import org.anddev.andengine.opengl.texture.builder.ITextureBuilder-
        .TextureSourcePackingException;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.opengl.texture.region.TiledTextureRegion;
import org.anddev.andengine.ui.activity.BaseGameActivity;
import org.anddev.andengine.util.modifier.ease.EaseQuadOut;

import android.content.Intent;
import android.os.Handler;
import android.util.Log;
public class Level1Activity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================
```

```
private static final int CAMERA_WIDTH = 480;
private static final int CAMERA_HEIGHT = 320;
private String tag = "Level1Activity";

// ===========================================================
// Fields
// ===========================================================

private Handler mHandler;

protected Camera mCamera;

protected Scene mMainScene;

private Texture mLevel1BackTexture;
private Texture mScrumTexture;
private BuildableTexture mObstacleBoxTexture;
private TextureRegion mBoxTextureRegion;
private TextureRegion mLevel1BackTextureRegion;
private TextureRegion mBulletTextureRegion;
private TextureRegion mCrossTextureRegion;
private TextureRegion mHatchetTextureRegion;
private TiledTextureRegion mScrumTextureRegion;

private AnimatedSprite[] asprVamp = new AnimatedSprite[10];
private int nVamp;
Random gen;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   mHandler = new Handler();
   gen = new Random();
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
```

```
      new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT),
         this.mCamera));
}

@Override
public void onLoadResources() {
   /* Load Textures. */
   TextureRegionFactory.setAssetBasePath("gfx/Level1/");
   mLevel1BackTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mLevel1BackTextureRegion =
      TextureRegionFactory.createFromAsset(
      this.mLevel1BackTexture, this, "level1bk.png", 0,
         0);
   mEngine.getTextureManager().loadTexture(
      this.mLevel1BackTexture);

   mObstacleBoxTexture = new BuildableTexture(512, 256,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mBoxTextureRegion =
      TextureRegionFactory.createFromAsset(
         mObstacleBoxTexture,
         this, "obstaclebox.png");
   mBulletTextureRegion =
      TextureRegionFactory.createFromAsset(
         mObstacleBoxTexture,
         this, "bullet.png");
   mCrossTextureRegion =
      TextureRegionFactory.createFromAsset(
         mObstacleBoxTexture,
         this, "cross.png");
   mHatchetTextureRegion =
      TextureRegionFactory.createFromAsset(
         mObstacleBoxTexture,
         this, "hatchet.png");
      try    {
        mObstacleBoxTexture.build(
        new BlackPawnTextureBuilder(2));
      } catch (final TextureSourcePackingException e) {
        Log.d(tag,
        "Sprites won't fit in mObstacleBoxTexture");
     }
   this.mEngine.getTextureManager().loadTexture(
      this.mObstacleBoxTexture);

   mScrumTexture = new Texture(512, 256,
```

```
            TextureOptions.DEFAULT);
        mScrumTextureRegion =
            TextureRegionFactory.createTiledFromAsset(
                mScrumTexture,
                this, "scrum_tiled.png", 0, 0, 8, 4);
        mEngine.getTextureManager().loadTexture(
            this.mScrumTexture);

    }

    @Override
    public Scene onLoadScene() {
        this.mEngine.registerUpdateHandler(new FPSLogger());

        final Scene scene = new Scene(1);

        /* Center the camera. */
        final int centerX = (CAMERA_WIDTH -
            mLevel1BackTextureRegion.getWidth()) / 2;
        final int centerY = (CAMERA_HEIGHT -
            mLevel1BackTextureRegion.getHeight()) / 2;

        /* Create the sprites and add them to the scene. */
        final Sprite background = new Sprite(centerX, centerY,
            mLevel1BackTextureRegion);
        scene.getLastChild().attachChild(background);
        final Sprite obstacleBox = new Sprite(0.0f, CAMERA_HEIGHT -
            mBoxTextureRegion.getHeight(), mBoxTextureRegion);
        scene.getLastChild().attachChild(obstacleBox);
        final Sprite bullet = new Sprite(20.0f, CAMERA_HEIGHT -
            40.0f,
            mBulletTextureRegion);
        bullet.registerEntityModifier(
            new SequenceEntityModifier(
                new ParallelEntityModifier(
                  new MoveYModifier(3, 0.0f, CAMERA_HEIGHT -
                   40.0f, EaseQuadOut.getInstance() ),
                  new AlphaModifier(3, 0.0f, 1.0f),
                  new ScaleModifier(3, 0.5f, 1.0f)
                ),
                new RotationModifier(3, 0, 360)
            )
        );
        scene.getLastChild().attachChild(bullet);
        final Sprite cross = new Sprite(bullet.getInitialX() +
            40.0f,
```

```
        CAMERA_HEIGHT - 40.0f, mCrossTextureRegion);
    cross.registerEntityModifier(
        new SequenceEntityModifier(
            new ParallelEntityModifier(
              new MoveYModifier(4, 0.0f, CAMERA_HEIGHT -
               40.0f, EaseQuadOut.getInstance() ),
              new AlphaModifier(4, 0.0f, 1.0f),
              new ScaleModifier(4, 0.5f, 1.0f)
            ),
            new RotationModifier(2, 0, 360)
        )
    );
    cross.registerEntityModifier(new AlphaModifier(10.0f, 0.0f,
          1.0f));
    scene.getLastChild().attachChild(cross);
    final Sprite hatchet = new Sprite(cross.getInitialX() +
        40.0f,
        CAMERA_HEIGHT - 40.0f, mHatchetTextureRegion);
    hatchet.registerEntityModifier(
        new SequenceEntityModifier(
            new ParallelEntityModifier(
              new MoveYModifier(5, 0.0f, CAMERA_HEIGHT -
               40.0f, EaseQuadOut.getInstance() ),
              new AlphaModifier(5, 0.0f, 1.0f),
              new ScaleModifier(5, 0.5f, 1.0f)
            ),
            new RotationModifier(2, 0, 360)
        )
    );
    hatchet.registerEntityModifier(new AlphaModifier(15.0f,
          0.0f,
          1.0f));
    scene.getLastChild().attachChild(hatchet);
    scene.registerEntityModifier(new AlphaModifier(10, 0.0f,
        1.0f));

           // Add first vampire (which will add the others)
           nVamp = 0;
    mHandler.postDelayed(mStartVamp,5000);
    return scene;
}

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

```
    private Runnable mStartVamp = new Runnable() {
        public void run() {
           int i = nVamp++;
           Scene scene = Level1Activity.this.mEngine.getScene();
              float startY = gen.nextFloat()*(CAMERA_HEIGHT - 50.0f);
              asprVamp[i] = new AnimatedSprite(CAMERA_WIDTH - 30.0f,
         startY,
         mScrumTextureRegion.clone());
              final long[] frameDurations = new long[26];
              Arrays.fill(frameDurations, 500);
              asprVamp[i].animate(frameDurations, 0, 25, true);
              asprVamp[i].registerEntityModifier(
                 new SequenceEntityModifier (
                    new AlphaModifier(5.0f, 0.0f, 1.0f),
                      new MoveModifier(60.0f,
asprVamp[i].getX(), 30.0f,
                         asprVamp[i].getY(),
                    (float)CAMERA_HEIGHT/2)));
              scene.getLastChild().attachChild(asprVamp[i]);
              if (nVamp < 10){
                 mHandler.postDelayed(mStartVamp,5000);
              }
   }
     };
}
```

When you run the game, you'll see the vampires come to life, one by one, moving across the screen to the left side, where they all pile up like the marching band in *Animal House*. Notice that the sprites are visible wherever they walk. They were the last items added to the Scene, so they appear as the uppermost objects in the layers that make up the scene.

Let's look at code in more detail. First we define some new variables that we're going to need:

- `Handler mHandler`: an Android Handler we can use to post runnable routines. If you aren't familiar with Android Handlers and Runnables, see the note "Android Handlers and Runnables."
- `Texture mScrumTexture`: a new Texture to hold the animation frames.
- `TiledTextureRegion mScrumTextureRegion`: the region within `mScrumTexture`.
- `AnimatedSprite[] asprVamp[10]`: an array of AnimatedSprites we'll use for the vampire sprites.
- `int nVamp`: a counter for vampires.
- `Random gen`: a random number generator.

> **Note: Android Handlers and Runnables**
>
> Android applications are thread based and centered on a message loop executed by the "main" or "UI" thread. Handlers are used to interact with the message queue. Handlers can post either a Message or a Runnable for the thread to execute, with the Message or Runnable being set to run immediately or after a delay.
>
> This mechanism is used in a number of ways in Android applications, and we use it for two tasks in V3:
>
> 1. We use it in StartActivity and MainMenuActivity to start other Activities running. We use a Handler to post a Runnable that executes the `startActivity()` method with an Intent that describes the Activity we wish to run.
> 2. We use it here in Level1Activity to delay the running of a method. If we want something to happen at some later time, we can't stop the Android message loop and force it to wait until it's the desired time; instead, we post a delayed Runnable whose `run()` method is the one we want to execute. The Handler queues the message for us and Android executes it when the delay ends.

onLoadEngine()

Only a few additions are made in this method:

- `Handler mHandler`: We create the Handler that we can use to delay running of the `mStartVampire` Runnable until later.
- `Random gen`: We initialize the random number generator we can use to stagger the vampires—er, send the vampires staggering across the graveyard.

onLoadResources()

The following additions are made to load the textures needed for the vampire animation:

- `Texture mScrumTexture`: We create a new Texture just for our new animation.
- `TiledTextureRegion mScrumTextureRegion`: We create this region to hold the animation tiles and load them from the tiled bitmap image `scrum_tiled.png`, which we've placed in `assets/gfx/Level1`. Because we're creating a TiledTextureRegion, we use the `createTiledFromAsset()` method and pass the number of columns and rows in the tiled image.
- As usual, we ask the TextureManager to load the Texture once we're done.

onLoadScene()

Nothing changes here until we reach the "//Add first vampire…" comment.

- `int nVamp`: We'll use this counter to track the number of vampires we've started, so we initialize it to 0.

- We set up our Handler to post a request to run the `mStartVampire` Runnable in 5 seconds. We'll restart `mStartVampire` every 5 seconds until 10 vampires appear on the screen.

mStartVampire

`mStartVampire` is a new Runnable whose `run()` method will create a vampire, send it walking onto the screen, and post a request for the next vampire:

- `int i`: We copy the `nVamp` counter and increment it—mostly because I'm lazy and I know I'm going to have to type the index a lot of times in this method.
- `Scene scene`: We recover the current Scene from the Engine, as it is otherwise out of scope in this method.
- `float startY`: We compute a starting position for this vampire using the random number generator. The vampire will appear at some random *Y* location on the right screen margin.
- `AnimatedSprite asprVamp[i]`: We create the sprite for this vampire at the computed location and pass it a clone of the `mScrumTextureRegion`. The cloning step is important. Without cloning, all the vampires would animate in lockstep, and that's not what we want. By cloning the TiledTextureRegion used, we ensure that each vampire will be animated independently of the others.
- `asprVamp[i].animate`: Recall that the Scrum tiles don't completely fill the tile array in `scrum_tiled.png` (see Figure 6.6). We create the animation using a version of `.animate()` that allows us to pass in a `pFirstTileIndex` and a `pLastTileIndex` (the second and third parameters, respectively). This method requires the use of a `pframeDurations` array, so we create that array first and fill in each element with the same duration, given in milliseconds. The final parameter (`true`) says to loop the animation when it's done.
- We register some Modifiers to the vampire that will make it fade in and walk to the center left of the screen (in front of Miss B's door).
- We attach the vampire to the current Scene.
- If 10 vampires are not yet walking around, we ask Android to schedule the Runnable in another 5 seconds to start another vampire animation.

Animation Problems

Animation is famously difficult to get right. It's an art, and even with all the new tools that make it easier and faster to generate animations, it still takes a lot of time and effort to achieve smooth, believable action.

A lot of information is available for free on the Internet about creating good animations and about debugging bad animations. If your characters seem to jump and jerk

around in odd ways (and you didn't intend for them to do that), and you can't figure out how to make them behave properly, step back for a few minutes and browse the tips and hints, and then go back and look carefully at your frames.

If you can afford to use an animation package, its capabilities will greatly facilitate your own work. The process of getting from artwork to game can involve many steps (render, frame grab, insert alpha channel, resize, load into Eclipse project, …), and the earlier in that process that you can fix problems, the better.

Advanced Topic: 2D Animations from 3D Models

With the popularity of 3D games and 3D illustration in general, it has been said that 3D computer artists might outnumber 2D artists. I'm not so sure that's true, but using 3D models to generate 2D animation sequences certainly has other advantages. You often need animation sequences for your characters in which characters must move in different directions. At one point, they might be running from left to right; at another point, they might need to run away from you; and at yet another point, they might need to run toward you. If you're generating animations from 2D art, you would have to come up with at least three animation sequences to support moving in these different directions (you can usually just flip the left-to-right sequence to get the right-to-left version).

If you start with a 3D model, however, it's a simple task to reposition the model or the camera and generate 2D animation sequences from any number of angles. You just need to render each version as a 2D film, and then use AnimGet to collect the frames for your game.

The learning curve for 3D animation packages can be steep. Nevertheless, if you already know how to use one, or if you have a friend who knows how to use one, you might think of tapping into this option as a way to generate the animation sequences for your characters.

Summary

This chapter provided our first look at animation and covered just about everything we will need to know for all the animations we'll need in our V3 game. The book isn't devoted to animation art, but we talked about different ways of creating animation sequences, ways of loading those sequences into the TiledTextureRegion, and ways of using the `animate()` method to create the actual animations.

Recapping the process of preparing and animating a sprite:

- Load the animation frames from a tiled image into a TiledTextureRegion, and load that region into a Texture. Ask TextureManager to load the Texture into its cache.
- Create the AnimatedSprite object, being careful to clone the TiledTextureRegion if you are using the region with multiple sprites and you want them to be animated independently.

- Animate the sprite using whichever version of the `animate()` method is most applicable.
- Add the sprite to the Scene where it will be shown.

Now we know how to create sprites for characters and entities in our game, and we know how to make them move and animate them as they move. Next we want to take a closer look at using text in our game.

Exercises

1. On the V3 splash screen, make the animated bat fly back and forth, instead of just hovering by the tombstone. Make it fly in front of the tombstone. What would you need to do to make it fly behind the tombstone?
2. Draw a series of animation frames for a character walking. If you're artistically challenged (like) me, create a stick figure—it really doesn't matter for our purposes here. The point is to get a feel for what it takes to generate animation frames and get them into a suitable form for use in a game.
3. Repeat Exercise 2 using an animation software package (most companies offering such packages have trial editions that they make available for a limited time for free) to see how much effort it saves in producing animation frames. Where is the tradeoff point for you, between creating frames by hand and using an animation package to do the tweening?
4. Change Level1Activity so the vampires disappear when they get to Miss B's door.

7

Text

We used text when we were creating the text menus in Chapter 3. In this chapter, however, we look at the ways text is used in AndEngine in more detail. We examine fonts, we look at the game elements that use text, and we learn how to create and use our own unique fonts with AndEngine.

Fonts and Typefaces

The terms "font" and "typeface" are sometimes used interchangeably, but they have distinct meanings. A typeface is something like Arial or Droid Sans, each of which defines a set of characters that map to the character codes for ASCII and Unicode. A font combines a typeface with a size (e.g., 16 points, where a point is 1/72nd of 1 inch) and a style (e.g., bold or italic).

Three basic methods can be used to generate the physical representation of characters:

- Bitmap: In bitmap fonts, a bitmap image is created for each character. Because bitmaps do not scale well, a separate bitmap is usually available for each font size. As a result, bitmap fonts can be large and can consume of precious memory.
- Outline or vector: In a vector font, a vector representation is provided for each character, which can be scaled to different font sizes. Just storing the vectors typically means vector fonts can be stored in less space.
- Stroke: Stroke fonts as meant by most artists (AndEngine is different) are similar to vector fonts, defining each character as a set of strokes. They are particularly well suited to East Asian languages, which are stroke based. These fonts can reduce the size of a large font image (such as the 5,000 or so commonly used Chinese characters) to a significant extent.

Just as we saw with graphics in general, vector typefaces tend to scale well to different sizes (particularly larger sizes), and bitmap fonts have the advantage of being precisely tailored at each font size. AndEngine supports vector fonts. As we'll see,

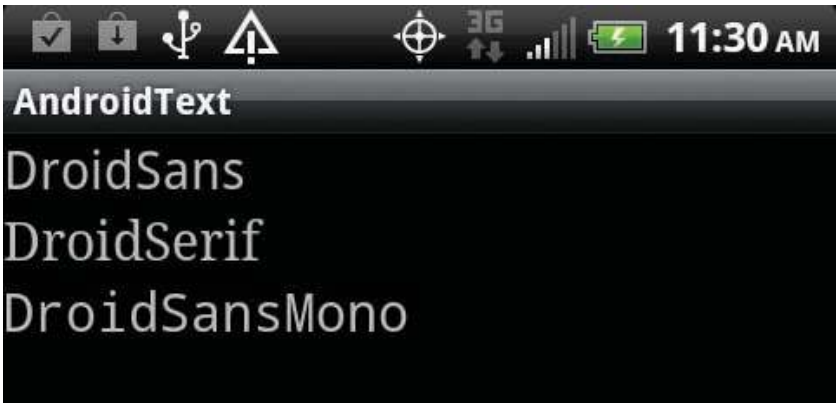


Figure 7.1 Standard Android typefaces

AndEngine fonts called StrokeFont are available, but they are really the outlines of vector fonts, so they are not the same as the stroke fonts mentioned earlier. You could add your own classes to handle bitmap and real stroke fonts, but that is beyond the scope of this book.

Android uses vector typefaces for all of its text components. Every Android device (so far) ships with three typefaces installed: Droid Sans, Droid Sans Mono, and Droid Serif. "Sans" means "sans serif"; the serif is that little line at the base of characters that is drawn with a serif font. "Mono" means `monospaced`; each character in such a typeface has the same width as every other character, much like the type you get from a typewriter. Figure 7.1 shows the three standard Android typefaces.

Android also allows you to load your own custom typefaces and fonts. It knows how to load TrueType fonts and OpenType fonts, although some developers have reported having issues with particular OpenType fonts. AndEngine works closely with the Android Typeface class. We'll look at an example of using a custom TrueType font later in this chapter.

Loading Fonts

Before you can create any text-based entities in AndEngine, you need to load a font. AndEngine entities are all displayed using OpenGL; thus, even if you're using a standard Android font, you will need to load it into a Texture before AndEngine can use it. AndEngine provides a Font class (distinct from the Android Font class) and a FontManager to load and manage fonts.

Font

In AndEngine, the Font class is part of the OpenGL code. From our game developer point of view, the only thing we need to know about is the constructor:

**Font(final Texture pTexture, final Typeface pTypeface,
final float pSize, final boolean pAntiAlias, final int pColor)**

The parameters to this call are described here:

- `Texture pTexture`: This parameter points to the Texture where the font image will be stored (discussed in the next section).
- `Typeface pTypeface`: This parameter specifies the Android Typeface that is the source of the font images.
- `float pSize`: This parameter gives the size (height) of the font in pixels (not points).
- `boolean pAntiAlias`: If this parameter is true, the font will be anti-aliased when displayed. If you're not familiar with anti-aliasing, it can improve the way a font looks when displayed.
- `int pColor`: The color to use when displaying the font. Note there are not separate fill and stroke colors, but rather just one color for both.

We'll see an example of using a Font with standard Android fonts in Listing 7.1 later in this chapter. Fonts are displayed as filled characters with no outline.

StrokeFont

AndEngine provides a StrokeFont class that extends Font. These fonts draw each character in outline, but they can also draw the characters with a different fill color, or with the outline alone. The constructors are similar to those for Font. The most complete constructor is

**StrokeFont(final Texture pTexture, final Typeface pTypeface, final float pSize,
final boolean pAntiAlias, final int pColor, final float pStrokeWidth,
final int pStrokeColor, final boolean pStrokeOnly)**

The parameters that do not appear in the Font constructor are:

- `float pStrokeWidth`: This parameter specifies the width of the stroke to be painted
- `int pStrokeColor`: The strokes will be painted with this color.
- `boolean pStrokeOnly`: If this parameter is true, only the outline (strokes) will be painted, and not the fill for each character. This parameter is optional and defaults to false if not given.

FontFactory

Rather than using the constructors for Font and StrokeFont, there is another way to create new instances of the classes. The FontFactory class has a number of create methods, the most versatile of which are these two:

Font createFromAsset(final Texture pTexture, final Context pContext, final String pAssetPath, final float pSize, final boolean pAntiAlias, final int pColor)
StrokeFont createStrokeFromAsset(final Texture pTexture, final Context pContext, final String pAssetPath, final float pSize, final boolean pAntiAlias, final int pColor, final int pStrokeWidth, final int pStrokeColor, final boolean pStrokeOnly)

The parameters are the same as for the constructors, except that now we can access Typefaces in the `assets` folder. We'll make use of these methods to create fonts from custom Typefaces.

FontManager

AndEngine provides a FontManager analogous to the TextureManager we've seen earlier. AndEngine creates this singleton FontManager for us, and we then use it to manage our library of fonts. The pattern for loading a Font is

```
this.mEngine.getFontManager().loadFont(this.mFont);
```

where `mFont` is the Font we are loading. We'll see the FontManager code in Listing 7.1.

Typeface

In our game, we will use the Android Typeface class to access the standard Android fonts. Typeface is very well documented in the Android developer documents, but here is a quick refresher on the methods and constants we'll be using.

Typeface.create(Typeface family, int style)

We'll use this method when we want to use one of the default Android fonts.

- The `family` in that case can just be one of the following:
  - `Typeface.DEFAULT`
  - `Typeface.DEFAULT_BOLD`
  - `Typeface.MONOSPACE`
  - `Typeface.SANS_SERIF`
  - `Typeface.SERIF`
- The `style` is one of the following:
  - `Typeface.NORMAL`
  - `Typeface.BOLD`

- `Typeface.ITALIC`
- `Typeface.BOLD_ITALIC`

Text in AndEngine

Other than in MenuItems (which we discussed in Chapter 3), AndEngine uses text in three ways:

- `Text`: Creates a label with a fixed message.
- `ChangeableText`: Creates a label whose message can change.
- `TickerText`: Creates a fixed label whose message appears letter by letter. It does not scroll, the way a tickertape does, but it's an interesting effect.

The Android Views are also available, of course, and AndEngine games often make use of `Toast`, in particular. We'll cover that View briefly, as a refresher.

Text APIs in AndEngine

We use the Text class to create labels that we don't expect to change. There are three constructors for Text, each a bit more specific, as explained next:

Text(final float pX, final float pY, final Font pFont, final String pText)
Text(final float pX, final float pY, final Font pFont, final String pText,
final HorizontalAlign pHorizontalAlign)
Text(final float pX, final float pY, final Font pFont, final String pText,
final HorizontalAlign pHorizontalAlign, final int pCharactersMaximum)

The common parameters are as follows:

- `float pX, pY`: The position of the label on the screen.
- `String pText`: The text string itself. The string can be multiline, with embedded `\n`'s.
- `HorizontalAlign pHorizontalAlign`: There are three choices, with the default being LEFT:
  - `HorizontalAlign.LEFT`
  - `HorizontalAlign.CENTER`
  - `HorizontalAlign.RIGHT`
- `int pCharactersMaximum`: The number of characters in `pText`, not counting newlines. When you use one of the first two constructors, this value is computed for you.

A short example of creating and displaying a Text is shown in Listing 7.1.

Listing 7.1 **Text Example**

```
. . .

public class TextExample extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 720;
   private static final int CAMERA_HEIGHT = 480;

   // ===========================================================
   // Fields
   // ===========================================================

   private Camera mCamera;
   private Texture mFontTexture, mStrokeFontTexture;
   private Font mFont;
   private StrokeFont mStrokeFont;


   @Override
   public Engine onLoadEngine() {
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), this.mCamera));
   }

   @Override
   public void onLoadResources() {
      this.mFontTexture = new Texture(256, 256,
            TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      this.mStrokeFontTexture = new Texture(256, 256,
            TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      this.mFont = new Font(this.mFontTexture,
            Typeface.create(Typeface.DEFAULT,
            Typeface.BOLD), 32, true, Color.BLACK);
      this.mStrokeFont = new StrokeFont(this.mStrokeFontTexture,
            Typeface.create(Typeface.DEFAULT,
            Typeface.BOLD), 32, true, Color.RED, 2.0f,
            Color.WHITE, true);
      this.mEngine.getTextureManager().loadTexture(
             this.mFontTexture);
```

```
        this.mEngine.getTextureManager().loadTexture(
                this.mStrokeFontTexture);
        this.mEngine.getFontManager().loadFont(this.mFont);
        this.mEngine.getFontManager().loadFont(this.mStrokeFont);
    }

    @Override
    public Scene onLoadScene() {
        this.mEngine.registerUpdateHandler(new FPSLogger());
        final Scene scene = new Scene(1);
        scene.setBackground(new ColorBackground(0.1f, 0.6f, 0.9f));
        final Text textCenter = new Text(100, 60, this.mFont,
                "Show this centered \n on two lines.",
                HorizontalAlign.CENTER);
        final Text textStroke = new Text(100, 160,
            this.mStrokeFont,
                "Stroke font example \n also on two lines.",
                HorizontalAlign.CENTER);
        scene.getLastChild().attachChild(textCenter);
        scene.getLastChild().attachChild(textStroke);

        return scene;
    }

    @Override
    public void onLoadComplete() {
    }

}
```

Listing 7.1 produces the screen shown in Figure 7.2. Most of the code is self-explanatory, but there are a few points to note:

- The font Texture, `mFontTexture` in this case, has to be big enough to hold the entire font we are loading. If the Texture is not big enough, you won't get an error, but some characters will be missing when you display some text. The size needed depends on the typeface and on the size of the characters. The size used here (256 × 256 pixels) works fine for the DEFAULT, SANS, and MONOSPACE typefaces at 32 pixels. For the SERIF font at 64 pixels, you'll need a Texture size of 256 × 512 pixels (recall that Texture dimensions always have to be a power of 2). For larger fonts, you'll have to make the Texture dimensions correspondingly larger (although you have to be careful, because some Android devices are limited to Textures as small as 512 × 512 pixels).
- We need separate Textures for the Font and StrokeFont, even though they are the same typeface and size.

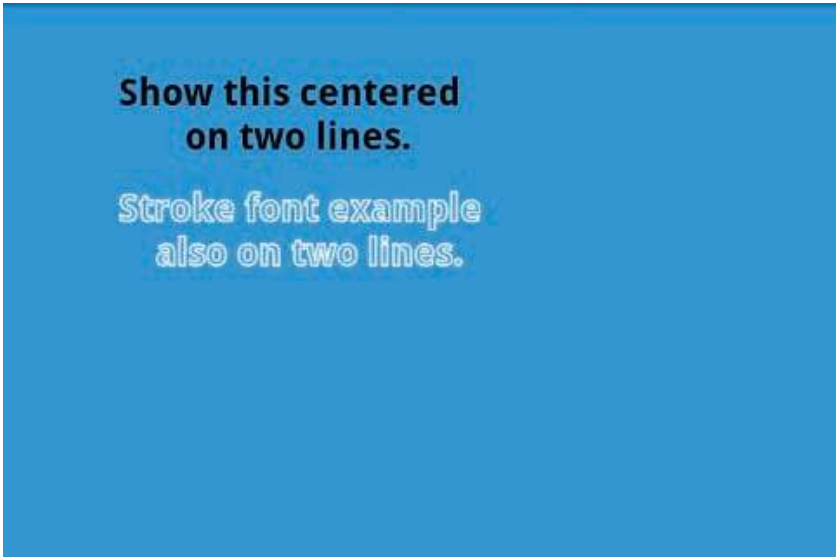


Figure 7.2 Text example

- When we create `mFont`, we ask for the default font (usually Droid Sans), at 32 pixels, with anti-aliasing, and in the color black. `mStrokeFont` is the same, with colors for fill and stroke, and a Boolean value that says "just show the stroke."
- We load the Textures using the TextureManager, and then load the Fonts using the FontManager.
- In `onLoadScene()`, we asked that the text be displayed at position (100, 60) with an alignment of `HorizontalAlign.CENTER`. That meant we wanted all the lines aligned at the center of the text block, not at the center of the screen, as shown in Figure 7.2.

Toast

Toast is a standard Android widget. We mention it here because it is so often used as a way to display quick messages for the game player. The pattern for creating a Toast message is

```
Toast.makeText(context, text, duration).show();
```

The parameters are as follows:

- `Context context`: The current application context (e.g., `StartActivity.this`).
- `String text`: The text to be displayed.
- `Int duration`: There are two duration options for a Toast:
  - `Toast.LENGTH_SHORT` (the default)
  - `Toast.LENGTH_LONG`

If we add a Toast to our Text Example (in the overridden `onLoadComplete()` method), we get the screen shown in Figure 7.3, on which the Toast appears for a few seconds.

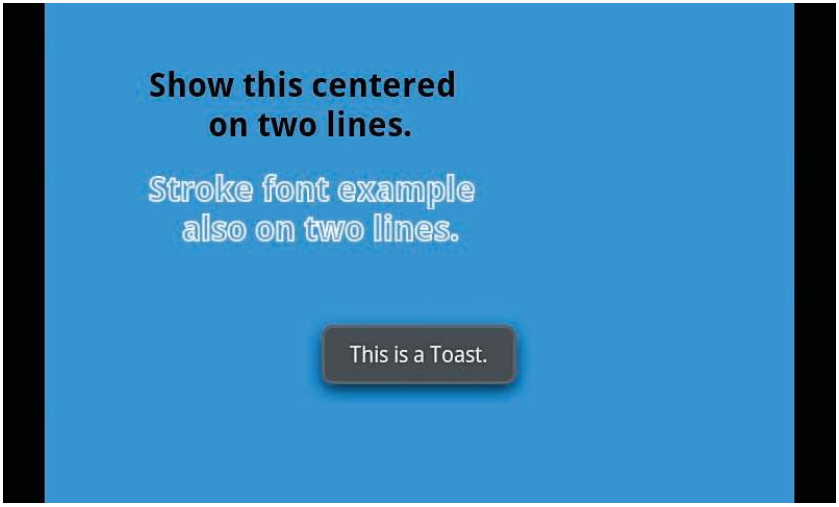


Figure 7.3 Text with Toast

Custom Fonts

As mentioned earlier, Android comes with a few TrueType fonts. AndEngine examples include a few more, but you may want to use a font that you find on the Internet or even create your own font. As with any other intellectual property, if you plan to use a downloaded font for your game, make sure it comes with a license that allows you to use it. If you plan to sell your game, be particularly cautious, as many "free" fonts don't come with a commercial license.

Creating Your Own TrueType Fonts

In V3, I've made use of a TrueType font called Flubber, which I downloaded from `www.1001fonts.com`. The Flubber download comes with a very lenient license that says, "You use this font in any way that you see fit." All that the font artist, Ben McGehee, asks in turn is that we include the license file that gives him credit for creating the font.

I don't have a need to edit Flubber, but if I did, many TrueType font editors are available. Some are expensive, professional font creation tools; others are free or inexpensive basic font editing tools.

If you want to create your own font from scratch (as opposed to editing someone else's font), the tool for doing so—as mentioned in Chapter 2—is FontStruct (`www.fontstruct.com`). FontStruct is a free service offered by `fontshop.com`. To use it, you establish a free account at the website and use the web application to build your font from a set of predefined building blocks. You can clone one of the existing fonts to start with, as long as the license allows it, and license the result back to the community with any of several Creative Commons options, which are explained on the

website. In Figure 7.4, I've cloned a font called SwiftedStrokes, which was created by Mike Lee.

I used FontStruct to make the clone, and, as you can see in Figure 7.5, I'm now free to edit the font. Here I'm looking at the character for the letter 'A'; I can change any of the blocks that make up that character—or any other character, for that matter. I can then save the font, make it available to others if I wish, or just keep it for myself.

Figure 7.4 Clone of the SwiftedStrokes TrueType font

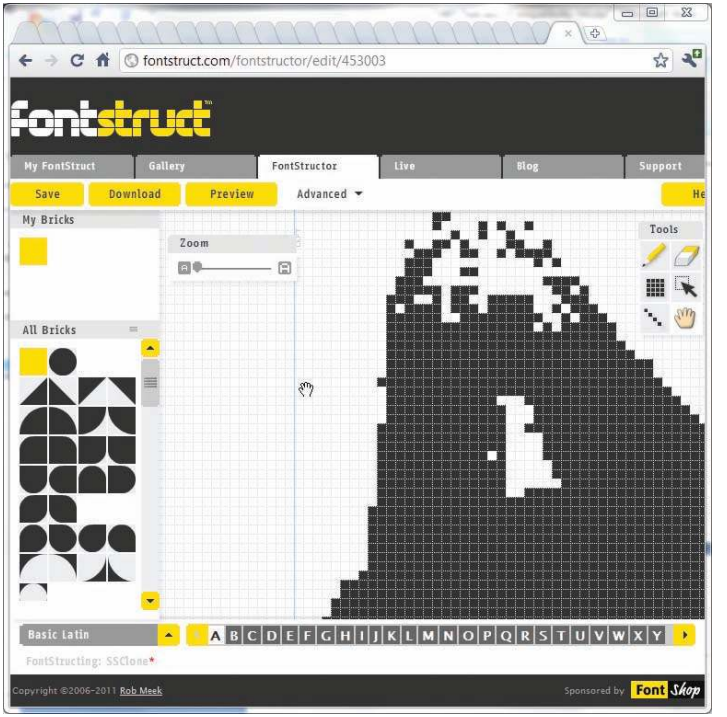


Figure 7.5 FontStruct used to edit a clone of the SwiftedStrokes TrueType font

Adding Custom Fonts to V3

So far in V3, we've embedded labels into the graphics that we've used, either for backgrounds or for items such as the obstacle box. One screen we haven't implemented yet is the Options screen, which will contain just a few menu items to perform the following tasks:

- Turn music on and off
- Turn sound effects on and off

For consistency, we'd like the Options screen text to be displayed in the Flubber font, and we'll display the on/off status by changing the text in the menu items. Because we don't actually have any sound in the game yet (we'll add audio features in Chapter 11), we'll just flip a Boolean toggle switch and set the menu item text for now. We want the Options screen to appear as shown in Figure 7.6.

We implement this screen by making a few changes to our game program. First, in `MainMenuLayer.java`, we need to add the code that will start the OptionsMenuActivity class when that item is chosen (before we just had a Toast for a stub). Listing 7.2 shows the changed code.

Listing 7.2 **Changes to MainMenuLayer.java to Add OptionsActivity**

```
. . .
       @Override
   public boolean onMenuItemClicked(final MenuScene pMenuScene,
      final IMenuItem pMenuItem, final float pMenuItemLocalX,
```

Figure 7.6 Options screen

```
      final float pMenuItemLocalY) {
      switch(pMenuItem.getID()) {
. . .
         case MENU_OPTIONS:
            mMainScene.registerEntityModifier(
              new ScaleModifier(1.0f, 1.0f, 0.0f));
            mStaticMenuScene.registerEntityModifier(
              new ScaleModifier(1.0f, 1.0f, 0.0f));
            mHandler.postDelayed(mLaunchOptionsTask,
             1000);
            return true;
. . .
}
. . .
     private Runnable mLaunchOptionsTask = new Runnable() {
         public void run() {
           Intent myIntent = new Intent(MainMenuActivity.this,
         OptionsActivity.class);
           MainMenuActivity.this.startActivity(myIntent);
         }
. . .
```

We also need to add the OptionsMenuActivity to our manifest file, as shown in Listing 7.3.

Listing 7.3 **New Version of AndroidManifext.xml with the Added OptionsActivity**

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.pearson.lagp.v3"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
   android:label="@string/app_name">
        <activity android:name=".StartActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"
         />
            </intent-filter>
        </activity>
      <activity android:name="MainMenuActivity"></activity>
      <activity android:name="Level1Activity"></activity>
      <activity android:name="OptionsActivity"></activity>
    </application>
    <uses-sdk android:minSdkVersion="4" />
```

```
  <uses-permission android:name="android.permission.WAKE_LOCK">
  </uses-permission>
</manifest>
```

We implement OptionsActivity as usual, by adding a class to our project that extends BaseGameActivity. The new class is shown in Listing 7.4.

Listing 7.4 **OptionsActivity.java**

```
package com.pearson.lagp.v3;

. . .
// imports here
. . .

public class OptionsActivity extends BaseGameActivity implements
   IOnMenuItemClickListener {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;

   protected static final int MENU_MUSIC = 0;
   protected static final int MENU_EFFECTS = MENU_MUSIC + 1;

   // ===========================================================
   // Fields
   // ===========================================================

   protected Camera mCamera;

   protected Scene mMainScene;
   protected Handler mHandler;

   private Texture mMenuBackTexture;
   private TextureRegion mMenuBackTextureRegion;

   protected MenuScene mOptionsMenuScene;
   private TextMenuItem mTurnMusicOff, mTurnMusicOn;
   private TextMenuItem mTurnEffectsOff, mTurnEffectsOn;
   private IMenuItem musicMenuItem;
   private IMenuItem effectsMenuItem;

   private Texture mFontTexture;
   private Font mFont;
```

```
public boolean isMusicOn = true;
public boolean isEffectsOn = true;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   mHandler = new Handler();
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
   /* Load Font/Textures. */
   this.mFontTexture = new Texture(256, 256,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);

   FontFactory.setAssetBasePath("font/");
   this.mFont = FontFactory.createFromAsset(this.mFontTexture,
      this, "Flubber.ttf", 32, true, Color.WHITE);
   this.mEngine.getTextureManager().loadTexture(
      this.mFontTexture);
   this.mEngine.getFontManager().loadFont(this.mFont);

   this.mMenuBackTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   this.mMenuBackTextureRegion =
      TextureRegionFactory.createFromAsset(
         this.mMenuBackTexture, this,
         "gfx/OptionsMenu/OptionsMenuBk.png", 0, 0);
   this.mEngine.getTextureManager().loadTexture(
      this.mMenuBackTexture);
```

```
    mTurnMusicOn = new TextMenuItem(MENU_MUSIC, mFont,
        "Turn Music On");
    mTurnMusicOff = new TextMenuItem(MENU_MUSIC, mFont,
        "Turn Music Off");
    mTurnEffectsOn = new TextMenuItem(MENU_EFFECTS, mFont,
        "Turn Effects On");
    mTurnEffectsOff = new TextMenuItem(MENU_EFFECTS, mFont,
        "Turn Effects Off");
}

@Override
public Scene onLoadScene() {
    this.mEngine.registerUpdateHandler(new FPSLogger());

    this.createOptionsMenuScene(true, true);

    /* Center the background on the camera. */
    final int centerX = (CAMERA_WIDTH -
        this.mMenuBackTextureRegion.getWidth()) / 2;
    final int centerY = (CAMERA_HEIGHT -
        this.mMenuBackTextureRegion.getHeight()) / 2;

    this.mMainScene = new Scene(1);
    /* Add the background and static menu */
    final Sprite menuBack = new Sprite(centerX, centerY,
        this.mMenuBackTextureRegion);
    mMainScene.getLastChild().attachChild(menuBack);
    mMainScene.setChildScene(mOptionsMenuScene);

    return this.mMainScene;
}

@Override
public void onLoadComplete() {
}

@Override
public boolean onMenuItemClicked(final MenuScene pMenuScene,
    final IMenuItem pMenuItem, final float pMenuItemLocalX,
    final float pMenuItemLocalY) {
    switch(pMenuItem.getID()) {
        case MENU_MUSIC:
            if (isMusicOn) {
                isMusicOn = false;
            } else {
                isMusicOn = true;
            }
```

```
            createOptionsMenuScene();
            mMainScene.clearChildScene();
            mMainScene.setChildScene(mOptionsMenuScene);
            return true;
         case MENU_EFFECTS:
            if (isEffectsOn) {
                  false);
               isEffectsOn = false;
            } else {
                  true);
               isEffectsOn = true;
            }
            createOptionsMenuScene()
            mMainScene.clearChildScene();
            mMainScene.setChildScene(mOptionsMenuScene);
            return true;
         default:
            return false;
      }
   }

   // ===========================================================
   // Methods
   // ===========================================================

   protected void createOptionsMenuScene() {
      this.mOptionsMenuScene = new MenuScene(this.mCamera);

      if (isMusicOn) {
         musicMenuItem = new ColorMenuItemDecorator(
            mTurnMusicOff, 0.5f, 0.5f, 0.5f, 1.0f,
            0.0f, 0.0f);
      } else {
         musicMenuItem = new ColorMenuItemDecorator(
            mTurnMusicOn, 0.5f, 0.5f, 0.5f, 1.0f,
            0.0f, 0.0f);
      }
      musicMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mOptionsMenuScene.addMenuItem(musicMenuItem);

      if (isEffectsOn) {
         effectsMenuItem = new ColorMenuItemDecorator(
            mTurnEffectsOff, 0.5f, 0.5f, 0.5f,
               1.0f, 0.0f, 0.0f);
      } else {
```

```
            effectsMenuItem = new ColorMenuItemDecorator(
               mTurnEffectsOn, 0.5f, 0.5f, 0.5f,
                  1.0f, 0.0f, 0.0f);
         }
         effectsMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
            GL10.GL_ONE_MINUS_SRC_ALPHA);
         this.mOptionsMenuScene.addMenuItem(effectsMenuItem);
         this.mOptionsMenuScene.buildAnimations();
         this.mOptionsMenuScene.setBackgroundEnabled(false);
         this.mOptionsMenuScene.setOnMenuItemClickListener(this);
      }
   }
   . . .
```

This code looks a lot like `MainMenuLayer.java`, which we saw in Chapter 3. The difference is that now we know why it works:

- `onLoadEngine()` is the same as always.
- In `onLoadResources()`, we load the font resources for `Flubber.ttf`.
  - The Texture size is 256 × 256 pixels: We've tried the 32-pixel font in TextExample, and it fits in that size Texture.
  - The font asset base path is set as "`/font`", because we want to keep font assets separate from the rest of our assets, and because we've imported a copy of `Flubber.ttf` into `asset/font`.
  - We create the Font and use TextureManager and FontManager to load the Texture and Font, respectively.
  - The Texture for the Scene background is loaded in the way we've seen before.
  - We define four TextMenuItems for the Options menu. We'll use two of these at a time, depending on the current status of two methods that return Boolean values, `isMusicOn()` and `isEffectsOn()`. (I apologize for the verb/subject disagreement, but Boolean-returning methods just have to start with "is" somehow.)
- In `onLoadScene()`, we invoke the `createOptionsMenuScene()` method to create the menu. We'll also use this method to re-create the menu with appropriate TextMenuItems as they change. Recall that the text in a TextMenuItem cannot be changed once it is created.
- In `onMenuItemClicked()`, we catch touches on the menu items, set the Boolean values appropriately, and call `createOptionsMenuScene()` again to re-create the menu. Then we clear away the old menu and add the new version to the Scene.

- The `createOptionsMenuScene()` method uses the Boolean values to put the right text in each menu item and uses ColorMenuItemDecorators to set the text to red when not selected and gray when selected.

If you run the changed code, selecting Options from the Main Menu now takes you to the Options Menu, where you can select Music or Effects to turn those sounds on or off. Of course, we don't have any sound yet, but the Boolean values are set for our sound methods—to be created in Chapter 11—to access.

Summary

In this chapter, we learned the most important stuff about Text in AndEngine games. We can create Text labels in our choice of styles, use custom vector fonts, make Toasts, and use them all in our games.

A pattern should be emerging by now in the way AndEngine treats objects that it will display with OpenGL:

- Create a Texture big enough to hold all the image options for the displayable Entity.
- Load the Texture with the images for the Entity. The images might come from an Android object (Typeface), an asset (custom TrueType font), or a resource (as we saw with Sprites).
- Use the singleton TextureManager to cache the Texture.
- Sometimes use an Entity manager (such as FontManager) to cache the Entity.
- Create the displayable object (e.g., Sprite, Text), and attach it to a Scene for display.

We've seen that pattern over and over, and we'll see a lot more of it as we continue to develop our example game.

Exercises

1. Change one Boolean value in `TextExample.java` to get the screen shown in Figure 7.7.
2. Make some modifications to the original `TextExample.java`:
   - Change the size of `mStrokeFont` to 64 pixels.
   - Change the position of `textStroke` to (60, 160).
   - Change both fonts to use `Typeface.SERIF`.
   - When you run the program, you should see a screen like that shown in Figure 7.8. What happened to the "i" in lines?
   - Fix the program so the "i" displays properly with the larger font.

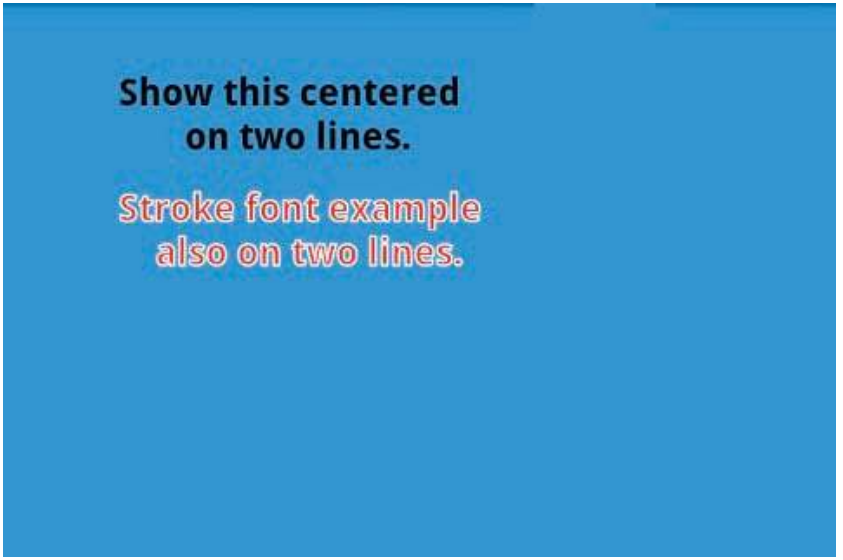


Figure 7.7 TextExample with filled stroke

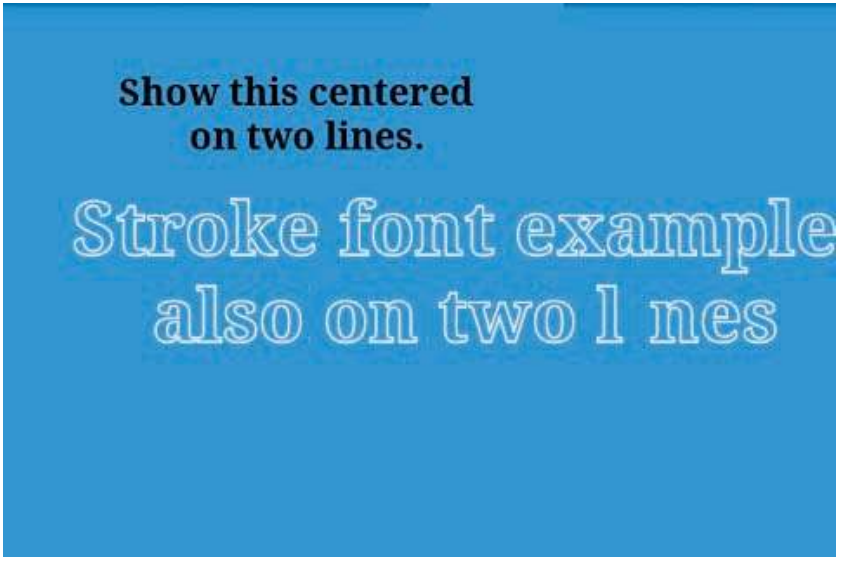


Figure 7.8 TextExample with changes

3. Add an option to the Options Menu to access the Help screen (which is only a Toast right now) directly, without returning to the Main Menu.
4. Use Toasts as debug messages to follow the change in value of the music and effects Boolean values in OptionsMenuActivity.

This page intentionally left blank

8

User Input

There wouldn't be much point to a computer game if there weren't user input to the game. Android and AndEngine together have a particularly rich set of input capabilities that developers can take advantage of to make their games more interesting. Let's first take a look at each of the user input features and see how AndEngine makes use of the underlying Android features. You might also want to use Android features that are not directly supported by AndEngine (EditText Views, for example), so we'll take a look at those capabilities as well.

Android and AndEngine Input Methods

Android has a very flexible user interface, anticipating a wide variety of Android devices with different styles of user input. It is likely that the users who play our games will do so on many different devices, so we need to support as many of them as possible. So far, Android devices have shipped with the following input methods (and undoubtedly more):

- Hard QWERTY keyboards (for various languages)
- Soft QWERTY keyboards (for various languages)
- D-Pad (up/down/left/right/select)
- Keypad (dedicated buttons or touch points):
  - Send
  - Home
  - Back
  - End
  - Vol+/−
  - Search
  - Menu

- Trackball
- Single-touch screen
- Single-touch pad
- Multi-touch screen
- Speech recognition
- Accelerometer
- Location (GPS or AGPS)
- Orientation (compass heading)

You might not normally think of the last three as user input methods, but they are used that way, particularly for games.

Keyboard and Keypad

Android simplifies programming for the plethora of keyboard types and layouts to produce standard key events that applications can use. In addition to the key event type (up, down, or multiple keys), Android passes a keycode that indicates which key was pressed. The key events are completely accessible to AndEngine games.

There are currently more than 200 keycodes, including the ones you would expect to see on a keyboard, as well as media control keys, camera keys, TV keys, and more. The keycodes are listed in the Android developer documentation at the following site:

`http://developer.android.com/reference/android/view/KeyEvent.html`

Android activities, including AndEngine games, can register themselves as listeners for key events, and notification of key events will then be passed to them. Flags may be set for each event that tell whether the event was generated by a hard key, by a soft key, or by a program (not by a key at all).

Android will also handle the details of key events in relation to standard Android Views. If an application creates a Text View that is editable, for example, Android will take care of collecting the user's keystrokes and make a string available to the application once the user is done. If an Android View is collecting the keystrokes, those data are not passed on as key events.

A typical snippet of code for capturing a key stroke is shown in Listing 8.1.

Listing 8.1 **Capturing Keystrokes**

```
@Override
public boolean onKeyDown(final int pKeyCode,
      final KeyEvent pEvent) {
   if(pKeyCode == KeyEvent.<insert keycode> &&
      pEvent.getAction() == KeyEvent.ACTION_DOWN) {
      // Do something because key was pressed
      return true;
```

```
        } else {
            return super.onKeyDown(pKeyCode, pEvent);
        }
    }
```

The `onKeyDown()` method overrides the default method in the Activity class; it is called when a key is pressed, and no Android View handles the key press. Returning a value of `true` indicates that your method has handled the key press, and it shouldn't be propagated to other key handlers that might be part of the chain. AndEngine games use this same API, just as with any Android application.

Touch

Android originally had only a single-touch interface, which is what we've used so far in the example game. Android quickly evolved to include multi-touch capabilities, however—first for gestures such as zooming, and then to include a more general gesture interface. AndEngine wraps Android's basic touch capabilities with code that manages touch events and makes it easier to use touch in games.

Single-Touch Mode

In the single-touch mode, when the user touches a point on the touchscreen, a `MotionEvent` is generated for listening applications. The standard Android API lets you register `OnTouchListener()` methods to Views. When the View is touched, the appropriate MotionEvents (down, drag, and up) are propagated to that method. Listing 8.2 is a snippet of code showing the capture of touch events in a standard Android application.

Listing 8.2 **Android Touch Capture**

```
package com.pearson.lagp.example;
. . .
public class TouchExample extends Activity implements OnTouchListener {
    private LinearLayout linear;
    private ImageView image;

. . .
    @Override
    public boolean onTouch(View v, MotionEvent e) {
        if (e.getAction() == MotionEvent.ACTION_DOWN) {
            Toast.makeText(this, "Down",
                Toast.LENGTH_SHORT).show();
        }
        return false;
    }
}
```

This approach is great for working with standard Android Views, but our AndEngine games may have only one View (the GLSurfaceView that serves as the canvas for all of our OpenGL drawing). To make effective use of touch in such a case, we would have to keep track of where everything is on the screen and map the touch event coordinates to different Entities.

Fortunately, AndEngine does all of this work for us. The Android touch paradigm is extended so we can declare a touch listener for any object (including a Sprite) that inherits from the Shape class. We can also declare a touch listener for any Scene. The relevant methods are shown here:

void Scene.setOnSceneTouchListener(final IOnSceneTouchListener pOnSceneTouchListener)
boolean Shape.onAreaTouched(final TouchEvent pSceneTouchEvent, final float pTouchAreaLocalX, final float pTouchAreaLocalY)

The parameters are self-explanatory. The TouchEvent passed to `onAreaTouched()` is an AndEngine TouchEvent. It's very much like an Android MotionEvent, but to minimize garbage collection AndEngine manages a pool of TouchEvents for us.

Listing 8.3 shows an example of capturing both Scene and Sprite touch events.

Listing 8.3 **AndEngine Touch Capture**

```
. . .
public class AndEngineTouchExample extends BaseGameActivity {
. . .

   // ===========================================================
   // Methods for/from SuperClass/Interfaces
   // ===========================================================

. . .

   @Override
   public void onLoadResources() {
      mIconTexture = new BuildableTexture(128, 128,
            TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      mIconTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mIconTexture, this, "icon.png");
         try {
         mIconTexture.build(
            new BlackPawnTextureBuilder(2));
         } catch (final TextureSourcePackingException e) {
            Log.d(tag, "Sprites won't fit in mIconTexture");
         }
```

```
    this.mEngine.getTextureManager().loadTexture(
                    this.mIconTexture);
}

@Override
public Scene onLoadScene() {
    final Scene scene = new Scene(1);
    scene.setBackground(new ColorBackground(0.1f, 0.6f, 0.9f));
    scene.setOnSceneTouchListener(new IOnSceneTouchListener() {
        @Override
        public boolean onSceneTouchEvent(
            final Scene pScene,
            final TouchEvent pSceneTouchEvent) {
            switch(pSceneTouchEvent.getAction()) {
                case TouchEvent.ACTION_DOWN:
                    Toast.makeText(
                    AndEngineTouchExample.this,
                    "Scene touch DOWN",
                    Toast.LENGTH_SHORT).show();
                    break;
                case TouchEvent.ACTION_UP:
                    Toast.makeText(
                    AndEngineTouchExample.this,
                    "Scene touch UP",
                    Toast.LENGTH_SHORT).show();
                    break;
            }
            return true;
        }
    });

    mIcon = new Sprite(100, 100, this.mIconTextureRegion) {
        @Override
        public boolean onAreaTouched(
            final TouchEvent pAreaTouchEvent,
            final float pTouchAreaLocalX,
            final float pTouchAreaLocalY) {
            switch(pAreaTouchEvent.getAction()) {
            case TouchEvent.ACTION_DOWN:
                Toast.makeText(
                AndEngineTouchExample.this,
                "Sprite touch DOWN",
                Toast.LENGTH_SHORT).show();
                break;
            case TouchEvent.ACTION_UP:
                Toast.makeText(
                AndEngineTouchExample.this,
                "Sprite touch UP",
```

```
                Toast.LENGTH_SHORT).show();
                break;
            }
            return true;
        }
    };

    scene.getLastChild().attachChild(mIcon);
    scene.registerTouchArea(mIcon);
    scene.setTouchAreaBindingEnabled(true);

    return scene;
}

@Override
public void onLoadComplete() {
}

}
```

When this application runs on an Android device, it displays a light blue background with an Android icon. If you touch the icon, Toasts describing the Sprite touch events appear. If you touch anywhere else, Toasts describing Screen touch events appear.

Notice the method `setTouchAreaBindingEnabled()` found almost at the end of the program. It notifies AndEngine that you want any additional area touch events (e.g., Move, Up) to be associated with this same Sprite. This action can be very useful if you are using a Move event to reposition the Sprite, for example. It would be easy for the update lag to cause the user's finger to stray outside the Sprite area. By using this method, however, you ensure that AndEngine associates events with the Sprite until the next Down event. To demonstrate this feature, touch the icon (you'll get a "Sprite touch DOWN" Toast), drag your finger, and then let up: You'll get a "Sprite touch UP" Toast, instead of the "Scene touch UP" that you might expect, because you're now off the Sprite.

Multi-Touch Mode

To handle multiple touches, Android uses the single-touch actions for the first touch event, and `MotionEvent.ACTION_POINTER` events for subsequent touches and moves. AndEngine does not add anything to the standard Android multi-touch APIs. These APIs are well documented on the Android developer site, in case you want to use them in your game.

Gestures

In addition to simple touches, we'd like to be able to generate gesture events for touch motion sequences. Single touch gestures have been part of Android since API level 4 was introduced (Android 1.6), and limited support for multi-touch gestures was

introduced with API level 8 (Android 2.2). The multi-touch support is limited to the two-finger zoom gesture.

As of this writing, the AndEngineExamples do not include any examples for gestures. Listing 8.4 shows a short example of the basic Android API that generates a Toast for each of the available gestures.

Listing 8.4 **Android Gestures Example**

```
. . .
public class GestureExample extends BaseGameActivity {
. . .
   @Override
   public Engine onLoadEngine() {
      mGestureDetector = new GestureDetector(this,
         new ExampleGestureListener());
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), this.mCamera));
   }

. . .

   @Override
   public boolean onTouchEvent(MotionEvent event) {
     if (mGestureDetector.onTouchEvent(event))
       return true;
     else
       return false;
   }

   class ExampleGestureListener extends
      GestureDetector.SimpleOnGestureListener{
        @Override
        public boolean onSingleTapUp(MotionEvent ev) {
         Toast.makeText(GestureExample.this,
         "Single tap up.", Toast.LENGTH_SHORT).show();
             return true;
        }

        @Override
        public void onShowPress(MotionEvent ev) {
         Toast.makeText(GestureExample.this,
         "Show press.", Toast.LENGTH_SHORT).show();
         return true;
```

```
        }
        @Override
        public void onLongPress(MotionEvent ev) {
         Toast.makeText(GestureExample.this,
         "Long press.", Toast.LENGTH_SHORT).show();
         return true;
        }

        @Override
        public boolean onScroll(MotionEvent e1, MotionEvent e2,
         float distanceX, float distanceY) {
         Toast.makeText(GestureExample.this, "Scroll.",
            Toast.LENGTH_SHORT).show();
             return true;
        }

        @Override
        public boolean onDown(MotionEvent ev) {
         Toast.makeText(GestureExample.this, "Down.",
            Toast.LENGTH_SHORT).show();
             return true;
        }

        @Override
        public boolean onFling(MotionEvent e1, MotionEvent e2,
         float velocityX, float velocityY) {
         Toast.makeText(GestureExample.this,
            "Fling.", Toast.LENGTH_SHORT).show();
          return true;
        }
    }
}
```

If you run this app (preferably on a real Android device—not on the emulator), you'll see that multiple events are generated for each gesture. The Android gesture interface is very flexible. As this book was being written, work was ongoing to simplify some of the gestures for use in games. You can follow this work on the AndEngine forum:

`http://www.andengine.org/forums`

Custom Gestures

The Android SDK comes with Gestures Builder, an application that makes it easy to create and record single-touch gestures. Although you can run Gestures Builder on the emulator, the easiest way to create a gesture is to run it on an Android device. The

resulting gestures library is stored on the device's sdcard using the filename `gestures`. If the phone is connected via USB you can retrieve it with the `adb` command

```
adb -d pull /sdcard/gestures
```

The `gestures` file will be pulled into your current directory, and will need to be loaded into `assets` for use in your game. We won't be using any custom gestures in V3, but you can think of many ways it might be used to make a game more fun to play. If you want to use custom gestures in your game, take a look at this Android Developer site article:

```
http://developer.android.com/resources/articles/gestures.html
```

On-Screen Controllers

One very neat feature of AndEngine is the ability to show and use simulated game controllers on the game screen. These controllers are usually shown transparently floating above the game, and they respond to touch inputs to realistically simulate a toggle-type game controller. Two flavors of on-screen controllers are available:

- Analog controllers, where the distance of the toggle from center is an analog value that can be used to control the game
- Digital controllers, which are more like a D-Pad, where the toggle value is just up, down, left, or right

Figure 8.1 shows the analog controllers, as demonstrated in the AndEngineExamples program `AnalogOnScreenControlsExample.java`.

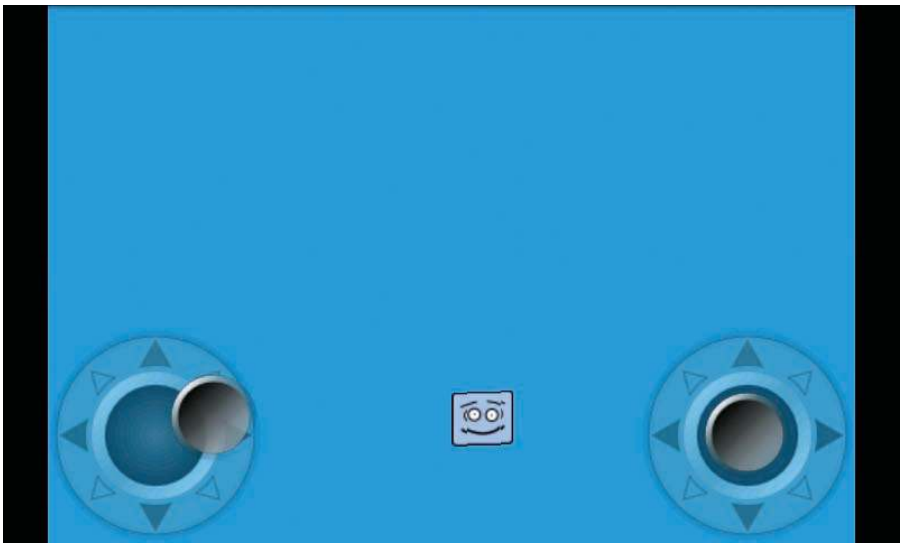


Figure 8.1 AndEngineExamples on-screen controller

The controllers are very realistic, but even though they are semi-transparent, they take up quite a bit of screen space. We won't include controllers in V3, but if you're interested in using them in your own game, take a look at the examples in AndEngine-Examples or at the answer to Exercise 1 in the Appendix.

Accelerometer

Most Android devices include an accelerometer, and AndEngine makes it readily available for use in our games. As a user input method, the accelerometer is often used to measure the tilt of the Android device. A popular mode of operation is to have something apparently rolling around the screen and to have the user tilt the phone to make it roll to a target while avoiding obstacles.

The accelerometer features in AndEngine are most often used in combination with the Physics modules, which we will discuss in detail in Chapter 12. For now, Exercise 3 at the end of this chapter (and its solution in the Appendix) shows briefly how to use the accelerometer.

Location and Orientation

Most Android devices also include sensors for location (usually GPS or AGPS) and orientation (a magnetometer to produce a compass heading). These items aren't often thought of as user input methods, but they can be used as inputs to a game. AndEngine wraps the basic Android APIs with classes that simplify their use in a game.

To use the orientation APIs, you need to say that your Activity implements IOrientationListener, and override the following method:

```
void onOrientationChanged(final OrientationData pOrientationData)
```

The OrientationData returned includes a lot of information, with the most frequently used pieces of data being roll, pitch, and yaw—the three axes of rotation. Their values range from 0.0 to 360.0, and there are getters for each item.

The AndEngine ILocationListener interface requires us to implement more methods:

```
void onLocationChanged(final Location pLocation)
void onLocationLost()
void onLocationProviderDisabled()
onLocationProviderStatusChanged(final LocationProviderStatus
        pLocationProviderStatus, final Bundle pBundle)
```

Of these methods, `onLocationChanged()` is the most often used, as it is where we can find out the new location of the device. Just as with any Android program that uses location information, we need to request the appropriate permissions and obtain access to a location provider by creating a new Location object (see the notes following Listing 8.5). Listing 8.5 shows a simple example of using location and orientation within AndEngine.

Listing 8.5 **Location and Orientation Example**

```
package com.pearson.lagp.example;

import org.anddev.andengine.engine.Engine;
import org.anddev.andengine.engine.camera.Camera;
import org.anddev.andengine.engine.options.EngineOptions;
import org.anddev.andengine.engine.options.EngineOptions
   .ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy
   .RatioResolutionPolicy;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.scene.background.ColorBackground;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.opengl.texture.BuildableTexture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.builder
   .BlackPawnTextureBuilder;
import org.anddev.andengine.opengl.texture.builder.ITextureBuilder
   .TextureSourcePackingException;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.sensor.location.ILocationListener;
import org.anddev.andengine.sensor.location.LocationProviderStatus;
import org.anddev.andengine.sensor.location.LocationSensorOptions;
import org.anddev.andengine.sensor.orientation.IOrientationListener;
import org.anddev.andengine.sensor.orientation.OrientationData;
import org.anddev.andengine.ui.activity.BaseGameActivity;
import org.anddev.andengine.util.Debug;
import android.location.Criteria;
import android.location.Location;
import android.location.LocationManager;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;

public class AndEngineSensorExample extends BaseGameActivity implements
IOrientationListener, ILocationListener {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;
   private String tag = "AndEngineSensorExample";

   // ===========================================================
   // Fields
   // ===========================================================
```

```
protected Camera mCamera;
private Location mUserLocation;

protected Scene mMainScene;
protected Sprite mIcon;

private BuildableTexture mIconTexture;
private TextureRegion mIconTextureRegion;
private Location mLocation ;

// ===========================================================
// Constructors
// ===========================================================
// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
   this.mLocation = new Location(
      LocationManager.GPS_PROVIDER);
}

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);

   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
      CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
   mIconTexture = new BuildableTexture(128, 128,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mIconTextureRegion =
      TextureRegionFactory.createFromAsset(
         this.mIconTexture, this, "icon.png");
      try {
      mIconTexture.build(new BlackPawnTextureBuilder(2));
```

```
      } catch (final TextureSourcePackingException e) {
      Log.d(tag, "Sprites won't fit in mIconTexture");
      }
   this.mEngine.getTextureManager().loadTexture(
      this.mIconTexture);
}

@Override
public Scene onLoadScene() {
   final Scene scene = new Scene(1);
   scene.setBackground(new ColorBackground(0.1f, 0.6f, 0.9f));

   mIcon = new Sprite(100, 100, this.mIconTextureRegion);
   scene.getLastChild().attachChild(mIcon);

   return scene;
}

@Override
public void onLoadComplete() {
}

@Override
protected void onResume() {
   super.onResume();

   this.enableOrientationSensor(AndEngineSensorExample.this);

   final LocationSensorOptions locationSensorOptions =
      new LocationSensorOptions();
   locationSensorOptions.setAccuracy(
      Criteria.ACCURACY_COARSE);
   locationSensorOptions.setMinimumTriggerTime(0);
   locationSensorOptions.setMinimumTriggerDistance(0);
   this.enableLocationSensor(AndEngineSensorExample.this,
      locationSensorOptions);
}

@Override
protected void onPause() {
   super.onPause();
   this.mEngine.disableOrientationSensor(this);
   this.mEngine.disableLocationSensor(this);
}

@Override
public void onOrientationChanged(
         final OrientationData pOrientationData) {
```

```
        float yaw = pOrientationData.getYaw() / 360.0f;
        mIcon.setPosition( CAMERA_WIDTH/2, yaw * CAMERA_HEIGHT);
    }

    @Override
    public void onLocationChanged(final Location pLocation) {
        String tst = "Lat: " + pLocation.getLatitude()+
                "  Lng: " + pLocation.getLongitude();
        Toast.makeText(AndEngineSensorExample.this, tst,
            Toast.LENGTH_LONG).show();
    }

    @Override
    public void onLocationLost() {
    }

    @Override
    public void onLocationProviderDisabled() {
    }

    @Override
    public void onLocationProviderEnabled() {
    }

    @Override
    public void onLocationProviderStatusChanged(
        final LocationProviderStatus pLocationProviderStatus,
        final Bundle pBundle) {
    }
}
```

The important points about this example include the following:

- To access location information, your application must have the proper permissions. For this example (asking for the GPS location provider), the manifest includes

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION">
```

- We need to override `onCreate()`, which we normally leave to BaseGameActivity, but we need to enable the location provider before `onResume()` runs.
- We usually leave the calls to `onPause()` and `onResume()` to BaseGameActivity, but we need to override these methods here so we can turn orientation and location listening off and on as necessary. This approach allows Android to optimize power consumption by disabling the sensors if no application is listening.
- The ILocationListener interface requires that we implement all of the location listener methods, most of which we leave as stubs.

Speech

Android provides a generalized voice recognition API that makes use of programs that run on Google's servers. Because the raw speech must go back to the server and the recognized text must return to the Android device, you need to factor a bit of latency into any use of speech in your game. Another point to consider is that the default speech recognition user interface obscures most of the screen while it is running. Listing 8.6 provides some example code so you can give voice recognition a try.

Listing 8.6 **Android Speech Recognition**

```
package com.pearson.lagp.example;

import java.util.ArrayList;

import org.anddev.andengine.engine.Engine;
import org.anddev.andengine.engine.camera.Camera;
import org.anddev.andengine.engine.options.EngineOptions;
import org.anddev.andengine.engine.options.EngineOptions
   .ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy
   .RatioResolutionPolicy;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.scene.background.ColorBackground;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.input.touch.TouchEvent;
import org.anddev.andengine.opengl.texture.BuildableTexture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.builder
   .BlackPawnTextureBuilder;
import org.anddev.andengine.opengl.texture.builder.ITextureBuilder
   .TextureSourcePackingException;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.ui.activity.BaseGameActivity;

import android.content.Intent;
import android.speech.RecognizerIntent;
import android.util.Log;
import android.widget.Toast;

public class VoiceRecExample extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;
```

```
private String tag = "VoiceRecExample";
    private static final int VOICE_RECOGNITION_REQUEST_CODE = 1234;

// ===========================================================
// Fields
// ===========================================================

protected Camera mCamera;

protected Scene mMainScene;
protected Sprite mIcon;

private BuildableTexture mIconTexture;
private TextureRegion mIconTextureRegion;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
      CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
   mIconTexture = new BuildableTexture(128, 128,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mIconTextureRegion =
      TextureRegionFactory.createFromAsset(
         this.mIconTexture, this, "icon.png");
      try {
        mIconTexture.build(
         new BlackPawnTextureBuilder(2));
```

```
      } catch (final TextureSourcePackingException e) {
        Log.d(tag, "Sprites won't fit in mIconTexture");
      }
   this.mEngine.getTextureManager().loadTexture(this.mIconTexture);
}

@Override
public Scene onLoadScene() {
   final Scene scene = new Scene(1);
   scene.setBackground(new ColorBackground(0.1f, 0.6f, 0.9f));

   mIcon = new Sprite(CAMERA_WIDTH/2, CAMERA_HEIGHT/2,
      this.mIconTextureRegion) {
      @Override
      public boolean onAreaTouched(
         final TouchEvent pAreaTouchEvent,
         final float pTouchAreaLocalX,
         final float pTouchAreaLocalY) {
           switch(pAreaTouchEvent.getAction()) {
           case TouchEvent.ACTION_DOWN:
            startVoiceRecognitionActivity();
            break;
           case TouchEvent.ACTION_UP:
            break;
         }
         return true;
      }
   };

   scene.getLastChild().attachChild(mIcon);
   scene.registerTouchArea(mIcon);
   scene.setTouchAreaBindingEnabled(true);
   Toast.makeText(VoiceRecExample.this,
      "Touch icon and say move left/right/up/down",
      Toast.LENGTH_SHORT).show();
   return scene;
}

@Override
public void onLoadComplete() {
}

// ===========================================================
// Methods
// ===========================================================
```

```
    private void startVoiceRecognitionActivity() {
        Intent intent = new Intent(RecognizerIntent
                .ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
                RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
         intent.putExtra(RecognizerIntent.EXTRA_PROMPT,
       "Speech recognition demo");
         startActivityForResult(intent, VOICE_RECOGNITION_REQUEST_CODE);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode,
       Intent data) {
        if (requestCode == VOICE_RECOGNITION_REQUEST_CODE &&
            resultCode == RESULT_OK) {
            ArrayList<String> matches = data.getStringArrayListExtra(
                    RecognizerIntent.EXTRA_RESULTS);
            for (String match : matches){
               if (match.equalsIgnoreCase("move left")){
                  mIcon.setPosition(mIcon.getX()-10.0f,
          mIcon.getY());
               }
               if (match.equalsIgnoreCase("move right")){
                  mIcon.setPosition(mIcon.getX()+10.0f,
          mIcon.getY());
               }
               if (match.equalsIgnoreCase("move up")){
                  mIcon.setPosition(mIcon.getX(),
          mIcon.getY()-10.0f);
               }
               if (match.equalsIgnoreCase("move down")){
                  mIcon.setPosition(mIcon.getX(),
          mIcon.getY()+10.0f);
               }
            }
        }

        super.onActivityResult(requestCode, resultCode, data);

    }

   // ============================================================
   // Inner and Anonymous Classes
   // ============================================================
}
```

If the Voice Recognizer application is installed on your Android device (as it is on all recent devices, for Voice Search), when you touch the icon you will see the speech

recognition UI. If you say, "Move left," "Move up," or something similar, you will see the icon move once the speech is recognized and the speech UI goes away.

Adding User Input to V3

We'd like to add some user input capabilities to the example game we're developing. Recall the scene we've developed for Level 1 of the game, with vampires walking toward Miss B's and a box of obstacles (see Figure 8.2). We want to make it possible for the player to touch a weapon Sprite and drag it into the graveyard to hold back the vampire attack. We won't implement the weapons' effects just yet; we'll just allow the player to move them around.

The changes to implement this ability are all made in `Level1Activity.java`, as shown in Listing 8.7.

Listing 8.7 **Changes to Level1Activity.java**

```
package com.pearson.lagp.v3;
. . .
   @Override
   public Scene onLoadScene() {
      this.mEngine.registerUpdateHandler(new FPSLogger());
```

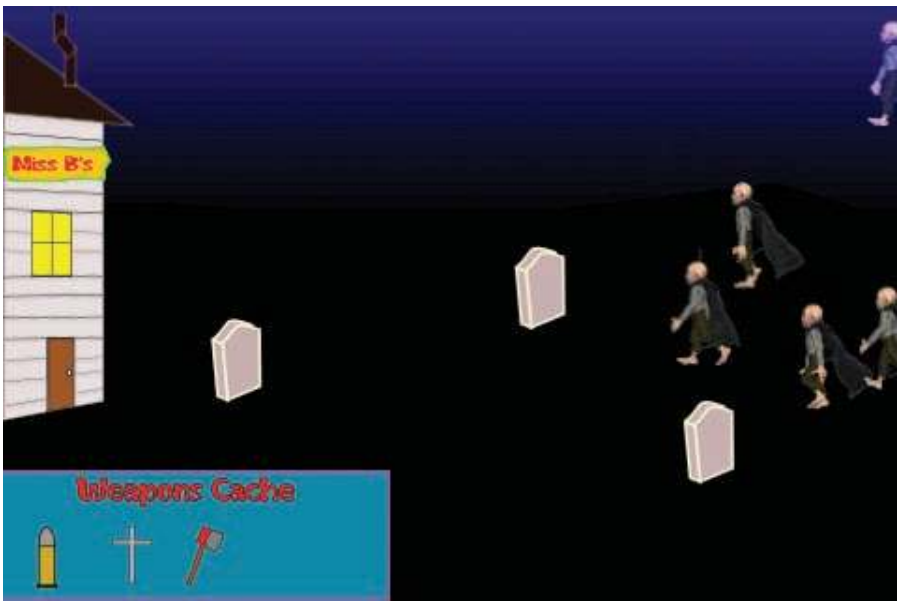


Figure 8.2 V3 Level 1 scene

```
final Scene scene = new Scene(1);

/* Center the camera. */
final int centerX = (CAMERA_WIDTH -
   mLevel1BackTextureRegion.getWidth()) / 2;
final int centerY = (CAMERA_HEIGHT -
   mLevel1BackTextureRegion.getHeight()) / 2;

/* Create the sprites and add them to the scene. */
final Sprite background = new Sprite(centerX, centerY,
   mLevel1BackTextureRegion);
scene.getLastChild().attachChild(background);
final Sprite obstacleBox = new Sprite(0.0f, CAMERA_HEIGHT -
   mBoxTextureRegion.getHeight(), mBoxTextureRegion);
scene.getLastChild().attachChild(obstacleBox);
final Sprite bullet = new Sprite(20.0f, CAMERA_HEIGHT -
   40.0f, mBulletTextureRegion){
   @Override
   public boolean onAreaTouched(
     final TouchEvent pAreaTouchEvent,
     final float pTouchAreaLocalX,
     final float pTouchAreaLocalY) {
      switch(pAreaTouchEvent.getAction()) {
      case TouchEvent.ACTION_DOWN:
        Toast.makeText(Level1Activity.this,
          "Sprite touch DOWN",
          Toast.LENGTH_SHORT).show();
        break;
      case TouchEvent.ACTION_UP:
        Toast.makeText(Level1Activity.this,
          "Sprite touch UP",
          Toast.LENGTH_SHORT).show();
        break;
      case TouchEvent.ACTION_MOVE:
        this.setPosition(pAreaTouchEvent.getX() -
          this.getWidth() / 2,
          pAreaTouchEvent.getY() -
          this.getHeight() / 2);
        break;
      }
      return true;
   }
};
bullet.registerEntityModifier(
   new SequenceEntityModifier(
      new ParallelEntityModifier(
         new MoveYModifier(3, 0.0f,
```

```
            CAMERA_HEIGHT - 40.0f,
            EaseQuadOut.getInstance() ),
          new AlphaModifier(3, 0.0f, 1.0f),
          new ScaleModifier(3, 0.5f, 1.0f)
            ),
       new RotationModifier(3, 0, 360)
       )
);
scene.registerTouchArea(bullet);
scene.setTouchAreaBindingEnabled(true);
scene.getLastChild().attachChild(bullet);

final Sprite cross = new Sprite(bullet.getInitialX() +
   40.0f, CAMERA_HEIGHT - 40.0f, mCrossTextureRegion){
   @Override
   public boolean onAreaTouched(
     final TouchEvent pAreaTouchEvent,
     final float pTouchAreaLocalX,
     final float pTouchAreaLocalY) {
      switch(pAreaTouchEvent.getAction()) {
      case TouchEvent.ACTION_DOWN:
        Toast.makeText(Level1Activity.this,
          "Sprite touch DOWN",
          Toast.LENGTH_SHORT).show();
        break;
      case TouchEvent.ACTION_UP:
        Toast.makeText(Level1Activity.this,
          "Sprite touch UP",
          Toast.LENGTH_SHORT).show();
        break;
      case TouchEvent.ACTION_MOVE:
        this.setPosition(pAreaTouchEvent.getX() -
          this.getWidth() / 2,
          pAreaTouchEvent.getY() -
          this.getHeight() / 2);
        break;
      }
      return true;
   }
};
cross.registerEntityModifier(
   new SequenceEntityModifier(
      new ParallelEntityModifier(
         new MoveYModifier(4, 0.0f,
           CAMERA_HEIGHT - 40.0f,
           EaseQuadOut.getInstance() ),
         new AlphaModifier(4, 0.0f, 1.0f),
```

```
            new ScaleModifier(4, 0.5f, 1.0f)
               ),
         new RotationModifier(2, 0, 360)
         )
);
cross.registerEntityModifier(
   new AlphaModifier(10.0f, 0.0f, 1.0f));
scene.registerTouchArea(cross);
scene.getLastChild().attachChild(cross);

final Sprite hatchet = new Sprite(cross.getInitialX() +
  40.0f, CAMERA_HEIGHT - 40.0f, mHatchetTextureRegion){
   @Override
   public boolean onAreaTouched(
     final TouchEvent pAreaTouchEvent,
     final float pTouchAreaLocalX,
     final float pTouchAreaLocalY) {
      switch(pAreaTouchEvent.getAction()) {
      case TouchEvent.ACTION_DOWN:
        Toast.makeText(Level1Activity.this,
          "Sprite touch DOWN",
          Toast.LENGTH_SHORT).show();
        break;
      case TouchEvent.ACTION_UP:
        Toast.makeText(Level1Activity.this,
          "Sprite touch UP",
          Toast.LENGTH_SHORT).show();
        break;
      case TouchEvent.ACTION_MOVE:
        this.setPosition(pAreaTouchEvent.getX() -
          this.getWidth() / 2,
          pAreaTouchEvent.getY() -
          this.getHeight() / 2);
        break;
      }
      return true;
   }
};
hatchet.registerEntityModifier(
   new SequenceEntityModifier(
      new ParallelEntityModifier(
         new MoveYModifier(5, 0.0f,
           CAMERA_HEIGHT - 40.0f,
           EaseQuadOut.getInstance() ),
         new AlphaModifier(5, 0.0f, 1.0f),
         new ScaleModifier(5, 0.5f, 1.0f)
            ),
```

```
            new RotationModifier(2, 0, 360)
            )
      );
      hatchet.registerEntityModifier(
         new AlphaModifier(15.0f, 0.0f, 1.0f));
      scene.getLastChild().attachChild(hatchet);
      scene.registerTouchArea(hatchet);
      scene.registerEntityModifier(
         new AlphaModifier(10, 0.0f, 1.0f));

          // Add first vampire (which will add the others)
          nVamp = 0;
      mHandler.postDelayed(mStartVamp,5000);
      return scene;
   }
. . .
}
```

There are basically two changes to the `onLoadScene()` method:

1. As each weapon Sprite is created, we add an `onAreaTouched()` listener method that will capture touches. For the moment, we're displaying the Toasts for the up and down events, just as we did in the touch example in Listing 8.3, but we'll want to take these Toasts out eventually. The `onAreaTouched()` methods are exactly alike at this point, and we could factor them out, but we want to keep them separate to do different things later. In each one we've added a case for `ACTION_MOVE` and reset the position each time to drag the Sprite along.
2. For each weapon Sprite, we registered the touch area with the Scene, so the touches would be propagated to us. Once we enable touch area binding (with the bullet Sprite registration), the move events will follow the Sprite.

Summary

We can see from this chapter that Android and AndEngine offer a wide variety of user input capabilities. The variety is almost too complex, and given that most inputs are touch based at their core, it can be a challenge to keep track of how touches are propagated and where they should be captured.

As game designers, our first task is to map the available input methods with the player's inputs to our game, thereby providing the most natural user interface. Let's recap the available Android and AndEngine input methods:

- Touch
  - Scene touches
  - Area (Sprite) touches

- Multi-touch
- Gesture
- On-screen controller
  - Analog controller
  - Digital controller
- Speech recognition
- Accelerometer
- Location
- Orientation

For most of these methods AndEngine provides wrappers that not only make the input easier to gather, but also manage pools of object resources, so inputs don't generate a bunch of objects that need to be garbage collected. Having those wrappers at hand puts us ahead of the curve in creating killer games.

Exercises

1. Write a small example game that moves a Sprite around the screen based on inputs from a single on-screen controller.
2. Write an example that uses an on-screen analog controller to move a Sprite around the screen. Have the Sprite wrap around when it hits the edge of the screen in any direction.
3. Write a simple game that changes the screen (background) color depending on the whether the Android device is held flat or is held up (i.e., the way a phone is normally used). Have it be green if flat and red if upright.

9

Tile Maps

Whether you know it or not, you've seen tile maps in other games that you've played. Many of the original arcade games were tile based, as well as many games for game consoles. Tiles are a good fit for a variety of game situations.

In this chapter, we look at the way AndEngine manages tiles. We also examine a tile editor for AndEngine, which enables you to make your own tiles or to adapt tile sets that you get from others. We'll implement tiles for the V3 game by incorporating a small game within the game.

Why Tile Maps?

Tile maps have two principal advantages that were critical to early computer games and remain valuable in today's games:

- For many games, the designer wants the playing field to be large and possibly extensible. Rather than having to create huge bitmaps for those large fields, tiles make it possible to construct seemingly endless fields on the fly, using just a few tiles that are repeated to build the larger image. Tile maps save both memory and time, as the same tile images are rendered repeatedly. In a related (non-game) area, that is exactly the reason why tiles are used in geographic mapping applications—they make it unnecessary to have the whole map in memory at once.
- Tiles can facilitate collision detection. Several types of collision detection are used in modern games, but you can detect collisions with tiles using very few computing resources.

Types of Tile Maps

A tile map is just an image formed from an array of repeating polygons. For simple tile maps, the polygons are usually squares. Thus the tile map is a grid, with each cell showing one of a small set of tiles, and collectively all of the cells creating a whole image. Figure 9.1 shows an example tile map that comes with the tile map editor we'll discuss later in this chapter; this figure shows a desert scene from above. The entire scene was created using just the tiles shown in its tile set, as depicted in Figure 9.2.

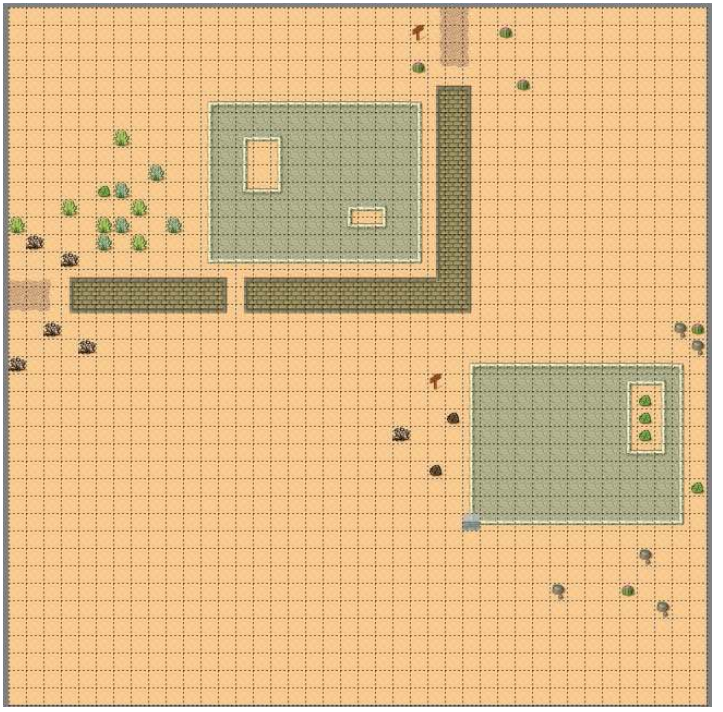


Figure 9.1 Desert example from Tiled

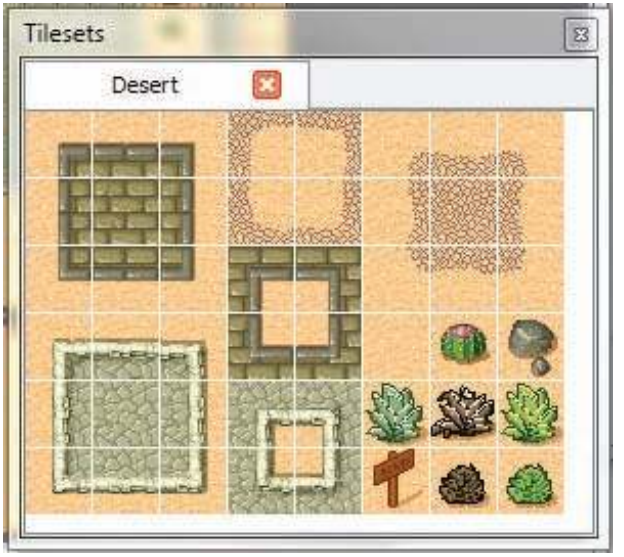


Figure 9.2 Tile set for desert example

Using just the 48 tiles in this tile set, a game designer can create as large a desert scene as needed, complete with varied features and obstacles. The game simply needs to store the images for the tiles, along with a map that tells it which tile to render in each location of the grid.

Orthogonal Tile Maps

The tile map we looked at in the last section is an orthogonal tile map. The view of the territory is from directly overhead, and the tiles are most commonly square, although some orthogonal tile maps use rectangular tiles. Orthogonal tile maps were the first type created and perhaps the most popular type of tile map used in games. They are the simplest type of tile map, as tile images never overlap. We'll use orthogonal tiling in our "game within a game."

Isometric Tile Maps

Another popular style of tile map uses isometric tiles. The idea behind isometric maps is to create a pseudo-3D effect by effectively moving the location of the viewing camera. Instead of viewing the territory from directly overhead, we view it from about 45 degrees off vertical. Figure 9.3 shows an example forest scene that comes with the Tiled editor.

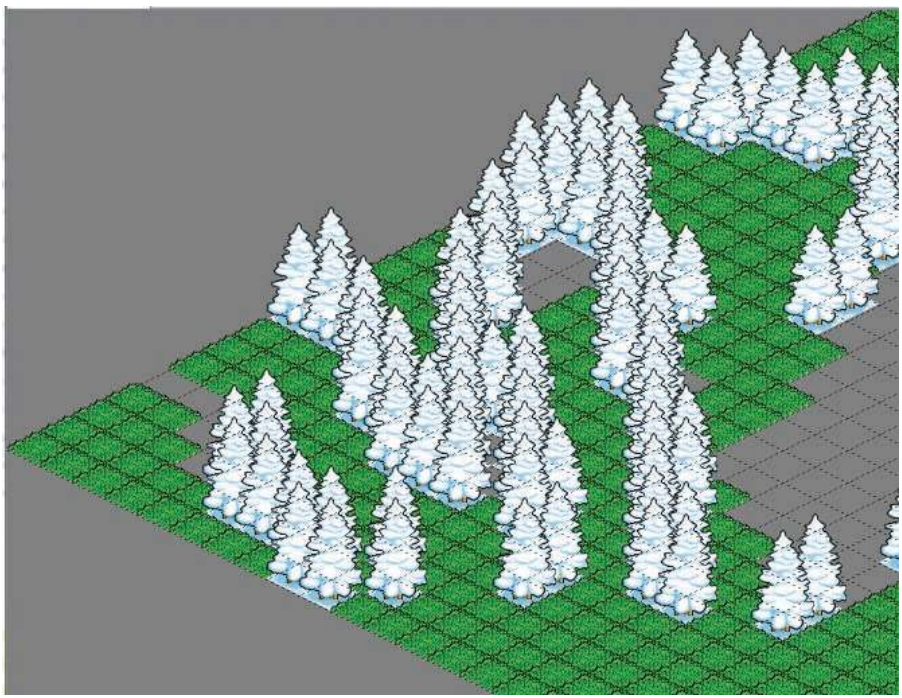


Figure 9.3 Isometric tile set

With the change in viewpoint, two things happen:

- The square tiles of the orthogonal view are transformed into a rhombus (diamond) shape.
- Features of nearer tiles will overlap the tiles that are farther away. In the example in Figure 9.3, the closer trees overlap tiles that are farther away.

The 3D effect of isometric tiling doesn't match what can be achieved with OpenGL and a real 3D game, but for the cost it's very effective. We won't develop a detailed isometric tile example for V3, but you should recognize that the basic techniques for orthogonal tiling apply to isometric tile maps as well.

Structure of Tile Maps

A tile map consists of three parts: the map, the tile set, and the image that provides the textures for the tiles. The map itself is structured hierarchically:

- Each tile has its characteristics, including its ID (its position in the map) and a global ID (which identifies the tile texture from the tile set that should be displayed there).
- Tiles are mapped into layers, each of which is a complete map of the 2D area covered by the tiles.
- The layers combine to make the map.

Tile maps also allow for construction of object groups, which are contiguous groups of tiles that the user wants to treat as a single entity for some reason. Object groups are implemented as rectangular areas on the map. Maps, layers, tiles, and object groups can all have their own properties assigned. Properties are just name–value pairs that can be set in the Tiled Qt editor (as we'll see later) and retrieved at runtime.

Tile Maps in AndEngine

Tile maps have been used for a long time in games, and standard formats have been developed for storing the tile sets and the resulting tile maps. Tile map utilities are freely available to build and edit tile maps based on these standards.

TMX and TSX Files

Two very popular XML-based formats for tile maps are TMX, for tile maps, and TSX, for tile sets. The two go together, as most TMX files reference a TSX file to obtain the tile set information. We'll look at these file formats in more detail later in this chapter, when we look at the Tiled tile map editor.

TMX tile maps integrate very easily into AndEngine. They can be loaded, manipulated, and rendered using a rather straightforward API that is detailed in the next few sections.

TMXLoader

AndEngine includes a TMXLoader class that knows how to parse and load TMX and TSX files into TMXTiledMap objects. You use TMXLoader by first creating a new object using one of its constructors. The most complete constructor is

```
TMXLoader(final Context pContext, final TextureManager pTextureManager,
        final TextureOptions pTextureOptions,
        final ITMXTilePropertiesListener pTMXTilePropertyListener)
```

The `Context` and `TextureManager` parameters are required, whereas `TextureOptions` and `ITMXTilePropertiesListener` are both optional. `TextureOptions` tells OpenGL how to render the tile textures, just as it did for Sprites. If not otherwise specified, `TextureOptions` defaults to `NEAREST_PREMULTIPLYALPHA`.

`ITMXTilePropertiesListener` gives us a chance to do something when a tile with properties is loaded. In the description of the Tiled Qt editor in a later section, we'll see that these properties are just name–value pairs. An example using the listener method also appears later in this chapter.

Once we have a TMXLoader object, we can use it to load the tile map with one of the load methods:

```
TMXTiledMap loadFromAsset(final Context pContext, final String pAssetPath)
TMXTiledMap load(final InputStream pInputStream)
```

The first method, `loadFromAsset()`, is more convenient, unless you happen to have the `.tmx` file open for some other reason. Both loaders will throw a TMXLoadException if anything goes wrong with parsing and loading the tile map. The returned TMXTileMap is the root node that we will use to access all of the tile map information.

TMXTiledMap

TMXTiledMap is the AndEngine class corresponding to a whole tile map, with all its layers, tiles, and properties. You normally won't have to use the constructor to create the entire map, but rather will get the object by loading a tile map as mentioned previously. Instead, we typically use TMXTiledMap getter methods to access the contents of the tile map:

```
int getTileColumns()
int getTileRows()
int getTileWidth()
int getTileHeight()
```

These methods retrieve dimensional information that might be of use. The width and height are the pixel width and height, respectively, of a single tile.

```
ArrayList<TMXTileSet> getTMXTileSets()
ArrayList<TMXLayer> getTMXLayers()
ArrayList<TMXObjectGroup> getTMXObjectGroups()
```

These methods retrieve lists of the layers, tile sets, and object groups in the tile map. Each of these objects has methods that provide more detailed information about that part of the map.

```
TMXProperties<TMXTileProperty> getTMXTilePropertiesByGlobalTileID(
        final int pGlobalTileID)
TMXProperties<TMXTiledMapProperty> getTMXTiledMapProperties()
TMXProperties<TMXTileProperty> getTMXTileProperties(
        final int pGlobalTileID)
```

These methods retrieve lists of properties that can be associated with the map, with a layer, or with a tile.

```
TextureRegion getTextureRegionFromGlobalTileID(final int pGlobalTileID)
```

This method allows us to retrieve the resulting TextureRegion that was created when the tile map was loaded.

TMXLayer

Tile maps are composed of layers, with each layer having its own map of tiles from the tile set. We saw earlier that we can retrieve the list of layers in a map from the TMXTiledMap object. We can select a particular layer and use its methods to retrieve or set information about that layer of the map:

```
String getName()
int getTileColumns()
int getTileRows()
```

These methods retrieve basic information about the layer. The name can be set in the Tiled Qt editor.

```
TMXTile[][] getTMXTiles()
TMXTile getTMXTile(final int pTileColumn, final int pTileRow)
TMXTile getTMXTileAt(final float pX, final float pY)
```

These three methods retrieve tiles. The first places all of the tiles in an array of TMXTiles, the second retrieves a tile by its row/column position in the grid, and the last one gets the tile that intersects with the given scene coordinates.

```
TMXProperties<TMXLayerProperty> getTMXLayerProperties()
```

This method provides a list of properties associated with the layer.

TMXTile

Once we have a reference to a TMXTile object, we can use its getter/setter methods to access all the interesting things about it:

```
int getTileRow()
int getTileColumn()
```

```
int getTileX()
int getTileY()
int getTileWidth()
int getTileHeight()
TextureRegion getTextureRegion()
```

These methods are straightforward. You might think you would use `getTextureRegion()` to figure out which tile image was mapped to this tile, but there's actually a better way. Remember the global ID?

```
int getGlobalTileID()
void setGlobalTileID(final TMXTiledMap pTMXTiledMap, final int pGlobalTileID)
```

Global IDs are assigned by Tiled Qt when you build the tile map (discussed in the next section). Although you can tell Tiled Qt where to start the numbering, the default is to start at 0, and each tile image in the tile set has its own unique global ID. When you set the global ID with `setGlobalTileID()`, doing so also sets the tile's texture to match. Thus you can use this approach to manipulate the images shown at runtime.

The Tile Editor: Tiled

AndEngine knows how to parse tile map files that are created by an editor called Tiled Qt. We encountered that editor briefly in Chapter 2, but now we'd like to look at it in detail and use it to build some tile maps for our game. Tiled knows how to edit both orthogonal and isometric tile maps. We'll focus on an orthogonal tile map example for V3.

If you haven't already done so, download Tiled (from `www.mapeditor.org`) and install it on your computer. Be sure you're installing "Tiled Qt," which is based on the Qt library from Nokia and written in C++. An earlier version of Tiled, which you may also run across, was written in Java and is no longer supported. For the examples in this book, we're using Tiled Qt 0.6.0.

When you open Tiled and open the `desert.tmx` example (in the `examples` subdirectory), the workspace shown in Figure 9.4 will appear.

The default editing tool is the "stamp." You can edit the tile map by picking tiles from the tile set in the lower-right corner of the screen and then clicking a grid position in the main work area to stamp that location with the selected tile. When you're ready to save your work, use either the File > Save or Save As… menu command to create a `.tmx` file that AndEngine can use to render your tile map. Right-clicking on the Tile Sets pane brings up a menu that lets you edit or save the tile set you're using into a `.tsx` file.

The Layers pane in the upper-right corner of the Tiled workspace shows the layers in the map. This view says there is only one layer in this tile map, it is named Tile Layer 1, and it is currently being displayed.

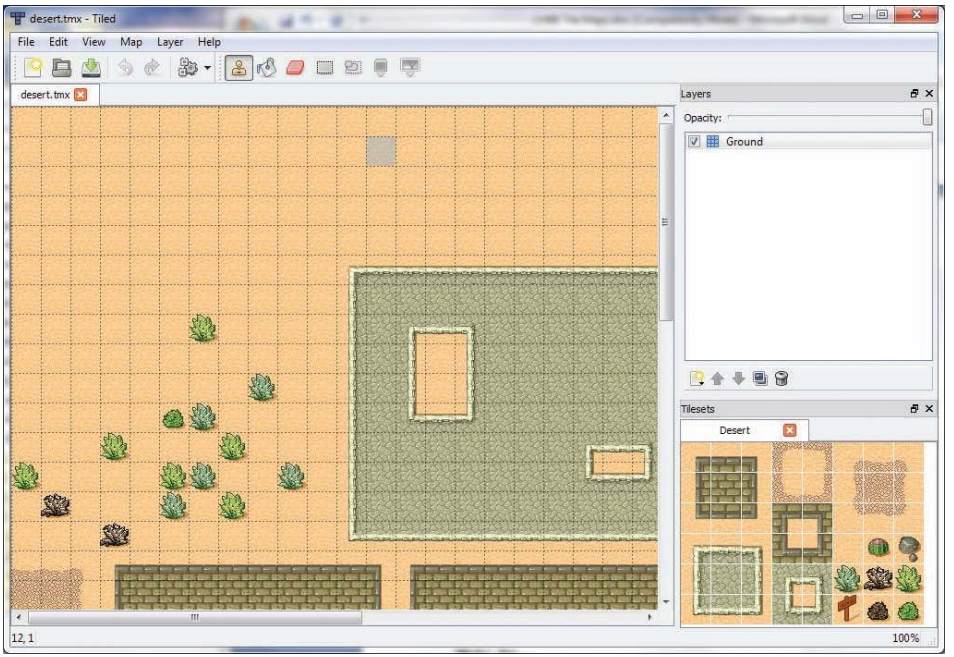


Figure 9.4 Tiled workspace

TMX Files

The file format used by Tiled and AndEngine is TMX, and the tile map files all have a file extension of `.tmx`. The set of tile images is found in an accompanying `.png` file. TMX files are XML based. They include the map indicating which tile appears in which location, for which compression is used to reduce the tile map data to a very small footprint. The TMX file for the desert landscape shown in Figure 9.4 is just 845 bytes long. Listing 9.1 provides the complete file.

Listing 9.1 **desert.tmx**

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.0" orientation="orthogonal" width="40" height="40"
tilewidth="32" tileheight="32">
<tileset firstgid="1" name="Desert" tilewidth="32" tileheight="32"
spacing="1" margin="1">
  <image source="tmw_desert_spacing.png" width="265" height="199"/>
 </tileset>
 <layer name="Ground" width="40" height="40">
  <data encoding="base64" compression="zlib">

eJztmNkKwjAQRaN9cAPrAq5Yq3Xf6v9/nSM2VIbQJjEZR+nDwQZScrwztoORECLySBcIgZ7nc2
y4KfyWDLx+Jb9nViNgDEwY+KioAXUgQN4+zpoCMwPmQAtoAx2CLFbA2oDEo9+hwG8DnIDtF/
```

```
2K8ks086Tw2zH0uyMv7HcRr/6/EvvhnsPrsrxwX7rwU/0ODig/eV3mh3N1ld8eraWPaX6+
64s9McesfrqcHfg1MpoifxcVEWjukyw+9AtFPl/I71pER3Of6j4bv7HI54s+MChhqLlPdZ/
P3qMmFuo5h5NnTOhjM5tReN2yT51n5/v7J3F0vi46fk+ne7aX0i9l6If7mpufTX3f5wsqv9
TAD2fJLT9VrTn7UeZnM5tR+v0LMQOHXwFnxe2/warGFRWf8QDjOLfP
  </data>
 </layer>
</map>
```

A few interesting bits about the contents of the file are worth highlighting:

- `width` and `height` are both measured in tiles, so Desert is 40 tiles wide and 40 tiles high.
- The `<tileset>` entity describes a tile set, and the `<image source>` is the filename for the tile set—`tmw_desert_spacing.png`, in this case. Tile sets can also be described by a separate XML file, called a TSX file. We'll use a TSX file when we build onto our V3 game later in this chapter.
- `tilewidth` and `tileheight` are both measured in pixels, so each tile in Desert is 32 × 32 pixels.
- There's only one `<layer>` entity, so there's only one layer in Desert; its name is Ground.
- The tile data for layer Ground has been compressed using zlib compression (`compression="zlib"`).

Orthogonal Game: Whack-A-Vampire

To demonstrate the use of tile maps in AndEngine, we will introduce a small, tiled subgame into V3. Between game layers (anywhere we like, really), we will ask the user to play a separate game that uses orthogonal tiling.

The game within a game will be called Whack-A-Vampire (WAV); it will be played on a 15 × 10 grid. When the game begins, the player will see a map of tiles, some of which contain closed caskets. At random times a closed casket will pop open to reveal a vampire. The player should whack (touch) each vampire as the casket opens.

WAV Tile Map

Fire up Tiled and use the menus on the opening screen (File > New …) to create a new tile map. Figure 9.5 shows the resulting New Map dialog.

We need to change a few of the fields to match the map we want to create. Make the following changes, and then click OK:

- Orientation: Orthogonal
- Map size: width: 15; height: 10
- Tile size: width: 32; height: 32

The Tiled workspace now appears as shown in Figure 9.6.

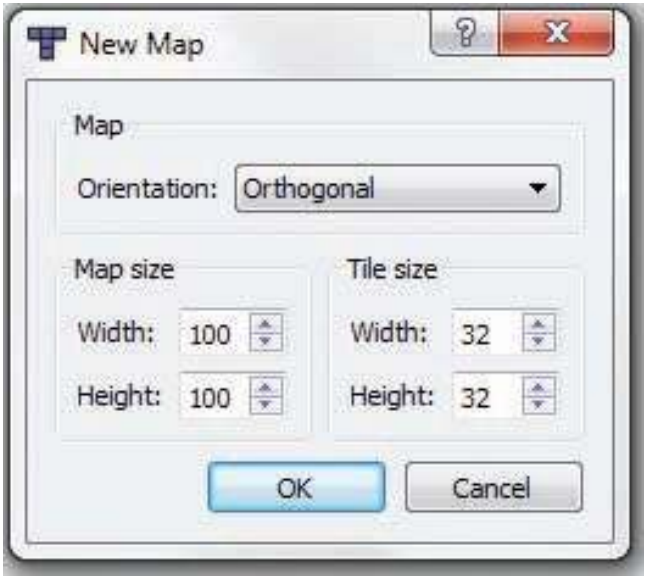


Figure 9.5 Tiled New Map dialog

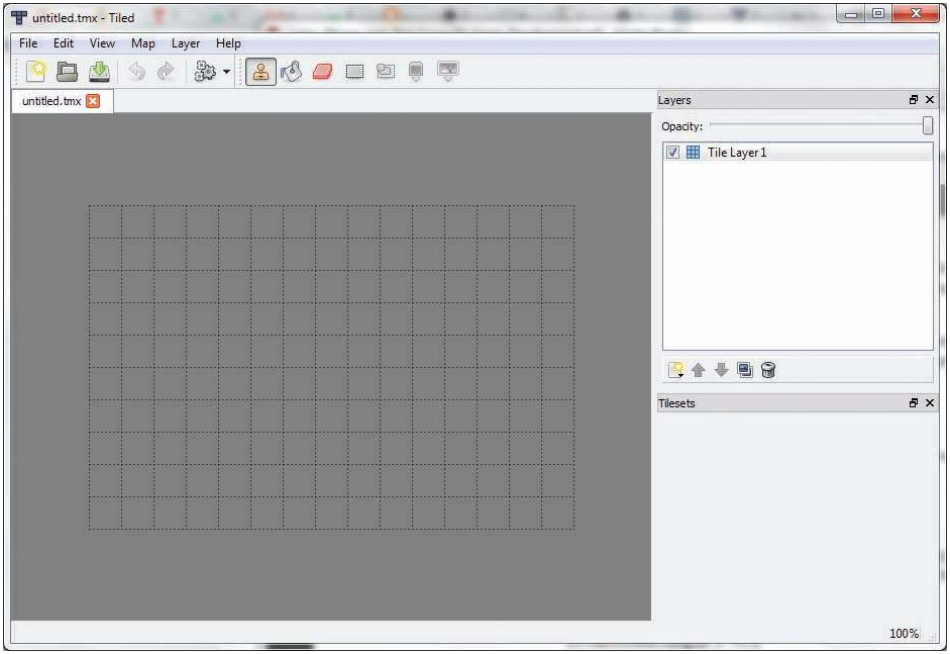


Figure 9.6 Tiled empty workspace

Figure 9.7 WAV tile set image

Creating the WAV Tile Set

You can use your favorite graphics editor to create the tiles you need for your game. For WAV, I used Inkscape to create six 32 × 32 pixel tiles. Tiled Qt expects tile sets to be in an image (`.png`) file, all piled together like a sprite sheet. Zwoptex (`www.zwoptex.com`) is a great tool for creating these mini-sprite sheets, or you can just use your favorite graphics editor to put all the tiles together in one `.png` file. The tile set texture (`.png`) file I created for WAV is shown in Figure 9.7.

Creating the WAV Tile Map

We import the tile set into Tiled by selecting Map > New Tileset … from the menu. The dialog shown in Figure 9.8 then appears.

Using this dialog, you can browse for the image file that contains your tile set image, and Tiled will fill in the Name to match that filename. Make sure the tile

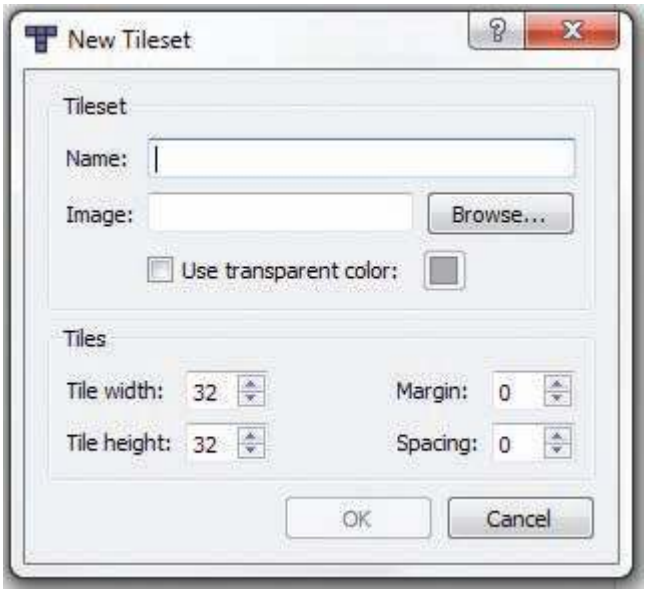


Figure 9.8 Tiled New Tileset dialog

sizes, margin, and spacing match the way you created the tiles. You can import as many tile sets as you'd like to build your map. We'll use just one tile set for WAV.

If you right-click on the closed coffin tile in the Tile Sets pane (second tile from the right), you should see a pop-up menu that includes Tile Properties…. Click on this menu item to bring up the dialog shown in Figure 9.9.

We need to insert a new property for this tile, so click on "<new property>" and fill in the Name `coffin` and the Value `true`. We will use this property in the game as we load the tiles to identify which tiles are coffin images. Click OK.

If you now right-click again anywhere in the Tile Sets pane, you'll see a pop-up menu including Export Tileset As…. Selecting this menu item creates the TSX file that describes the tile set, and which you are asked to name. For WAV, we've cleverly titled it `WAVTileset.tsx`.

With the tile set in place, you can use the editing tools in Tiled to build your map. The final result for WAV is shown in Figure 9.10.

To create the map, I first used the fill tool (paint can) to paint the entire map with the plain black tile. I then selected the stamp tool and used it to create the grass paths (or maybe they're hedges—it's hard to tell from this angle). You can use the click-and-drag technique to fill in a whole string of tiles with the selected tile image. Next, I used the stamp tool to insert coffins, graves, and headstones at random points

Figure 9.9 Tiled Tile Properties dialog

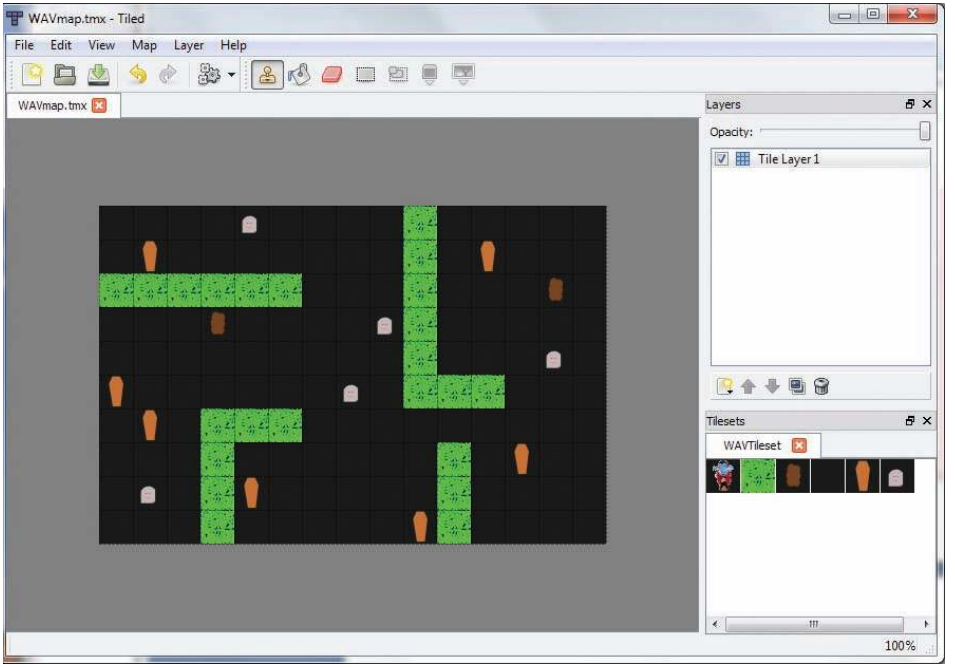


Figure 9.10 Tiled WAV tile map

on the map. I did not add any open coffin tiles, as we'll add those dynamically during the game.

We won't need object groups for WAV, but we could also identify object groups on the map. Object groups in Tiled are invisible rectangular regions that you plan to identify later in your application. Objects exist in their own layers, which are created with the Layer > Add Object Layer… command.

Properties are not just limited to tiles, of course. Any of the components in a TMX tile map (map, layer, tile, object group) can have associated properties, which are just name–value pairs of strings. For example, if you click the Layer > Layer Properties sequence, you will see the dialog shown in Figure 9.11.

At some point, we will want to save our tile map, but first we have to make sure Tiled uses gzip compression for the tile map. Click on Edit > Preferences… and you will see the dialog shown in Figure 9.12.

You just need to ensure that the item "Store the layer data as:" is set to Base64 (gzip compressed). That's not the default, at least on the version I'm using. Now we can save the map with File > Save. This action creates the `WhackAVampire.tmx` file, which we will import into the `assets` folder for our game.

Figure 9.11 Tiled set Layer Properties dialog

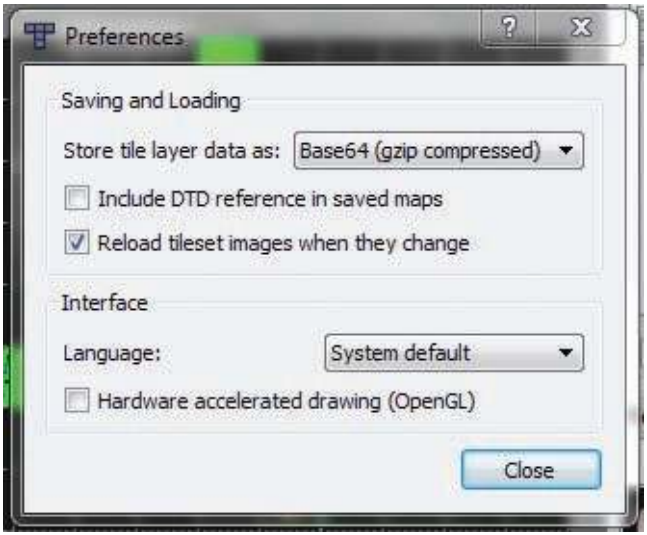


Figure 9.12 Tiled Preferences dialog

Whack-A-Vampire: The Code

So far, we don't have a way to actually complete levels in V3, so we'll need a way to get to the WAV game. We'll add an access point for this game to the Options Menu for the moment, and wire it in between layers when we can actually complete them.

Adding WAV to OptionsActivity

The new version of OptionsActivity is shown in Listing 9.2, where the changes appear in bold. The modifications should be largely self-explanatory, given our experience with menus from Chapter 3.

Listing 9.2 **OptionsActivity with WAV Added**

```
package com.pearson.lagp.v3;


import javax.microedition.khronos.opengles.GL10;

import org.anddev.andengine.engine.Engine;
import org.anddev.andengine.engine.camera.Camera;
import org.anddev.andengine.engine.options.EngineOptions;
import org.anddev.andengine.engine.options.EngineOptions
   .ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy
   .RatioResolutionPolicy;
import org.anddev.andengine.entity.modifier.ScaleModifier;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.scene.menu.MenuScene;
import org.anddev.andengine.entity.scene.menu.MenuScene
   .IOnMenuItemClickListener;
import org.anddev.andengine.entity.scene.menu.item.IMenuItem;
import org.anddev.andengine.entity.scene.menu.item.TextMenuItem;
import org.anddev.andengine.entity.scene.menu.item.decorator
   .ColorMenuItemDecorator;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.entity.util.FPSLogger;
import org.anddev.andengine.opengl.font.Font;
import org.anddev.andengine.opengl.font.FontFactory;
import org.anddev.andengine.opengl.texture.Texture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.ui.activity.BaseGameActivity;

import android.content.Intent;
import android.graphics.Color;
import android.os.Handler;

public class OptionsActivity extends BaseGameActivity implements
IOnMenuItemClickListener {
   // ===========================================================
   // Constants
   // ===========================================================
```

```
private static final int CAMERA_WIDTH = 480;
private static final int CAMERA_HEIGHT = 320;

protected static final int MENU_MUSIC = 0;
protected static final int MENU_EFFECTS = MENU_MUSIC + 1;
protected static final int MENU_WAV = MENU_EFFECTS + 1;

// ===========================================================
// Fields
// ===========================================================

protected Camera mCamera;

protected Scene mMainScene;
protected Handler mHandler;

private Texture mMenuBackTexture;
private TextureRegion mMenuBackTextureRegion;

protected MenuScene mOptionsMenuScene;
private TextMenuItem mTurnMusicOff, mTurnMusicOn;
private TextMenuItem mTurnEffectsOff, mTurnEffectsOn;
private TextMenuItem mWAV;
private IMenuItem musicMenuItem;
private IMenuItem effectsMenuItem;
private IMenuItem WAVMenuItem;

private Texture mFontTexture;
private Font mFont;

public boolean isMusicOn = true;
public boolean isEffectsOn = true;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
```

```
    mHandler = new Handler();
    this.mCamera = new Camera(0, 0, CAMERA_WIDTH, CAMERA_HEIGHT);
    return new Engine(new EngineOptions(true,
        ScreenOrientation.LANDSCAPE,
        new RatioResolutionPolicy(CAMERA_WIDTH,
        CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
    /* Load Font/Textures. */
    this.mFontTexture = new Texture(256, 256,
        TextureOptions.BILINEAR_PREMULTIPLYALPHA);

    FontFactory.setAssetBasePath("font/");
    this.mFont = FontFactory.createFromAsset(
        this.mFontTexture, this, "Flubber.ttf", 32,
        true, Color.WHITE);
    this.mEngine.getTextureManager().loadTexture(this.mFontTexture);
    this.mEngine.getFontManager().loadFont(this.mFont);

    this.mMenuBackTexture = new Texture(512, 512,
        TextureOptions.BILINEAR_PREMULTIPLYALPHA);
    this.mMenuBackTextureRegion =
        TextureRegionFactory.createFromAsset(
            this.mMenuBackTexture, this,
            "gfx/OptionsMenu/OptionsMenuBk.png", 0, 0);
    this.mEngine.getTextureManager().loadTexture(this.mMenuBackTexture);

    mTurnMusicOn = new TextMenuItem(MENU_MUSIC, mFont,
        "Turn Music On");
    mTurnMusicOff = new TextMenuItem(MENU_MUSIC, mFont,
        "Turn Music Off");
    mTurnEffectsOn = new TextMenuItem(MENU_EFFECTS, mFont,
        "Turn Effects On");
    mTurnEffectsOff = new TextMenuItem(MENU_EFFECTS, mFont,
        "Turn Effects Off");

    mWAV = new TextMenuItem(MENU_WAV, mFont,
        "Whack A Vampire");
}

@Override
public Scene onLoadScene() {
    this.mEngine.registerUpdateHandler(new FPSLogger());

    this.createOptionsMenuScene();
```

```
    /* Center the background on the camera. */
    final int centerX = (CAMERA_WIDTH -
       this.mMenuBackTextureRegion.getWidth()) / 2;
    final int centerY = (CAMERA_HEIGHT -
       this.mMenuBackTextureRegion.getHeight()) / 2;

    this.mMainScene = new Scene(1);
    /* Add the background and static menu */
    final Sprite menuBack = new Sprite(centerX, centerY,
       this.mMenuBackTextureRegion);
    mMainScene.getLastChild().attachChild(menuBack);
    mMainScene.setChildScene(mOptionsMenuScene);

    return this.mMainScene;
}

@Override
public void onLoadComplete() {
}

@Override
public boolean onMenuItemClicked(final MenuScene pMenuScene,
    final IMenuItem pMenuItem, final float pMenuItemLocalX,
    final float pMenuItemLocalY) {
    switch(pMenuItem.getID()) {
       case MENU_MUSIC:
          if (isMusicOn) {
             isMusicOn = false;
          } else {
             isMusicOn = true;
          }
          createOptionsMenuScene();
          mMainScene.clearChildScene();
          mMainScene.setChildScene(mOptionsMenuScene);
          return true;
       case MENU_EFFECTS:
          if (isEffectsOn) {
             isEffectsOn = false;
          } else {
             isEffectsOn = true;
          }
          createOptionsMenuScene();
          mMainScene.clearChildScene();
          mMainScene.setChildScene(mOptionsMenuScene);
          return true;
        case MENU_WAV:
           mMainScene.registerEntityModifier(
             new ScaleModifier(1.0f, 1.0f, 0.0f));
```

```
            mOptionsMenuScene.registerEntityModifier(
              new ScaleModifier(1.0f, 1.0f, 0.0f));
            mHandler.postDelayed(mLaunchWAVTask,1000);
            return true;

         default:
            return false;
      }
   }

   // ===========================================================
   // Methods
   // ===========================================================

   protected void createOptionsMenuScene() {
      this.mOptionsMenuScene = new MenuScene(this.mCamera);

      if (isMusicOn) {
         musicMenuItem = new ColorMenuItemDecorator(
           mTurnMusicOff, 0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
      } else {
         musicMenuItem = new ColorMenuItemDecorator(
           mTurnMusicOn, 0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
      }
      musicMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mOptionsMenuScene.addMenuItem(musicMenuItem);

      if (isEffectsOn) {
         effectsMenuItem = new ColorMenuItemDecorator(
           mTurnEffectsOff, 0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
      } else {
         effectsMenuItem = new ColorMenuItemDecorator(
           mTurnEffectsOn, 0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
      }
      effectsMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mOptionsMenuScene.addMenuItem(effectsMenuItem);

      WAVMenuItem = new ColorMenuItemDecorator(mWAV, 0.5f, 0.5f,
         0.5f, 1.0f, 0.0f, 0.0f);
      WAVMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mOptionsMenuScene.addMenuItem(WAVMenuItem);

      this.mOptionsMenuScene.buildAnimations();
      this.mOptionsMenuScene.setBackgroundEnabled(false);
      this.mOptionsMenuScene.setOnMenuItemClickListener(this);
```

```
    }

     private Runnable mLaunchWAVTask = new Runnable() {
         public void run() {
           Intent myIntent = new Intent(OptionsActivity.this,
          WAVActivity.class);
           OptionsActivity.this.startActivity(myIntent);
         }
      };

    // ===========================================================
    // Inner and Anonymous Classes
    // ===========================================================

}
```

Creating the Activity for Whack-A-Vampire

We create `WAVActivity.java` in the usual manner, remembering to add it to the Manifest, so Android will know there is a new Activity. The code is shown in Listing 9.3, with a detailed explanation to follow.

Listing 9.3 **WAVActivity.java**

```
package com.pearson.lagp.v3;

+imports

public class WAVActivity extends BaseGameActivity {
    // ===========================================================
    // Constants
    // ===========================================================

    private static final int CAMERA_WIDTH = 480;
    private static final int CAMERA_HEIGHT = 320;
    private String tag = "WAVActivity";

    // ===========================================================
    // Fields
    // ===========================================================

    private Handler mHandler;

    protected Camera mCamera;

    protected Scene mMainScene;
```

```
private TMXTiledMap mWAVTMXMap;
private TMXLayer tmxLayer;
private TMXTile tmxTile;

private int[] coffins = new int[50];
private int coffinPtr = 0;
private int mCoffinGID = -1;
private int mOpenCoffinGID = 1;

private Random gen;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   mHandler = new Handler();
   gen = new Random();
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
}

@Override
public Scene onLoadScene() {
   this.mEngine.registerUpdateHandler(new FPSLogger());

   final Scene scene = new Scene(1);
   try {
      final TMXLoader tmxLoader = new TMXLoader(
      this, this.mEngine.getTextureManager(),
```

```
      TextureOptions.BILINEAR_PREMULTIPLYALPHA,
      new ITMXTilePropertiesListener() {
      @Override
      public void onTMXTileWithPropertiesCreated(
         final TMXTiledMap pTMXTiledMap,
         final TMXLayer pTMXLayer,
         final TMXTile pTMXTile,
         final TMXProperties<TMXTileProperty>
           pTMXTileProperties) {
           if(pTMXTileProper-
      ties.containsTMXProperty("coffin", "true")) {
            coffins[coffinPtr++] =
               pTMXTile.getTileRow() * 15 +
               pTMXTile.getTileColumn();
            if (mCoffinGID<0){
              mCoffinGID =
               pTMXTile.getGlobalTileID();
               }
            }
         }
      });
      this.mWAVTMXMap = tmxLoader.loadFromAsset(this,
         "gfx/WAV/WAVmap.tmx");
   } catch (final TMXLoadException tmxle) {
      Debug.e(tmxle);
   }

   tmxLayer = this.mWAVTMXMap.getTMXLayers().get(0);
   scene.getFirstChild().attachChild(tmxLayer);
   scene.setOnSceneTouchListener(new IOnSceneTouchListener() {
      @Override
      public boolean onSceneTouchEvent(
         final Scene pScene,
         final TouchEvent pSceneTouchEvent) {
           switch(pSceneTouchEvent.getAction()) {
              case TouchEvent.ACTION_DOWN:
                 /* Get the touched tile */
                 tmxTile = tmxLayer.getTMXTileAt(
                   pSceneTouchEvent.getX(),
                   pSceneTouchEvent.getY());
                 if((tmxTile != null) &&
      (tmxTile.getGlobalTileID() == mOpenCoffinGID)) {
                   tmxTile.setGlobalTileID(mWAVTMXMap, mCoffinGID);
                 }
            break;
             }
   return true;
```

```
        }
      });

      mHandler.postDelayed(openCoffin,gen.nextInt(2000));
      return scene;
    }

    @Override
    public void onLoadComplete() {
    }

     private Runnable openCoffin = new Runnable() {
         public void run() {
            int openThis = gen.nextInt(coffinPtr);
            int tileRow = coffins[openThis]/15;
            int tileCol = coffins[openThis] % 15;
            tmxTile = tmxLayer.getTMXTileAt(tileCol*32 + 16,
          tileRow*32 + 16);
            tmxTile.setGlobalTileID(mWAVTMXMap, mOpenCoffinGID);
           mHandler.postDelayed(openCoffin,gen.nextInt(4000));
         }
      };
}
```

After the usual imports, variable declarations, and Engine setup, the serious differences in this module start in `onLoadScene()`. We try to load the tile map into AndEngine by creating a new TMXLoader object and asking it to create the map from an asset. As we create the TMXLoader, we override the `onTMXTileWithPropertiesCreated()` method and use it to track whenever a tile is created whose `coffin` property is set to `true`. When we load a coffin tile, we keep track of its position in an array, `int coffins[]`.

AndEngine will expect to find the named `.tmx` file in the `assets` folder, as modified by the `setAssetBasePath()` call in `onLoadResources()`. If you are using subfolders, as we are here, you will also need to manually edit the`.tmx` and `.tsx` files produced by Tiled to add the complete path from assets. See, for example, the `source` attributes in the edited versions of `WAVmap.tmx` and `WAVTileset.tsx` in Listing 9.4 and Listing 9.5, respectively, where we've added the complete paths. This filename juggling can get a little tricky, especially if you are creating the TMX and TSX files in another directory.

Listing 9.4 **WAVmap.tmx with Edits for Subfolders**

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.0" orientation="orthogonal" width="15" height="10"
   tilewidth="32" tileheight="32">
```

```
<tileset firstgid="1" source="gfx/WAV/WAVTileset.tsx"/>
 <layer name="Tile Layer 1" width="15" height="10">
  <data encoding="base64" compression="gzip">

H4sIAAAAAAAAAC5VQQQ4AMAQTG/9/8rKDpBFlO/RAldYWkQ3wVF9o0QtYw6E2zylBd3cV+6ueD5
4nZG38pMvqoK2yoL6b+fX28mv0xjJMXhhvhDtug+XEWAIAAA==
  </data>
 </layer>
</map>
```

Listing 9.5 **WAVTileset.tsx with Edits for Subfolders**

```
<?xml version="1.0" encoding="UTF-8"?>
<tileset name="WAVTileset" tilewidth="32" tileheight="32" spacing="2"
   margin="2">
<image source="gfx/WAV/WAVTileSet.png" width="256" height="64"/>
<tile id="4">
  <properties>
   <property name="coffin" value="true"/>
  </properties>
</tile>
</tileset>
```

With the tile map loaded, we retrieve Layer 0 (the only layer we created in Tiled) and attach it as a child of the current Scene. We can now create an `onSceneTouchListener()` method to capture user touch events. When an `ACTION_DOWN` touch event occurs, we retrieve the touched tile from the layer, using the `getTileAt()` method. We check the global ID of the tile to see if it's showing an open coffin image. If it is, we change it to show a closed coffin.

After that setup, we can post a delayed runnable, `openCoffin`, to actually start opening coffins. `openCoffin` picks a coffin at random from the `coffins` array, and changes its image to that of an open coffin. `openCoffin`'s final act is to post itself to run again at some random time between now and 4 seconds from now.

When you run the Activity, coffins start popping open at random intervals. If you touch an open coffin, it closes immediately. Right now, WAV is not a terribly challenging gamelet, but we will improve it later by adding sound and some distractions to make it more difficult to keep track of the coffins.

Isometric Tile Maps

Isometric tile maps can make a game more interesting, providing a sense of "3Dness" without going to the expense of creating 3D models and animations. They can be quite tricky to create and debug, however. The first challenge relates to the issue of drawing the tiles from the appropriate perspective. Then there are the *z*-ordering

issues that can either make or break the illusion of 3D. Some graphic artists like to create isometric tiles using 3D models, then position the virtual camera in the appropriate position and render a 2D representation. That's not unlike the technique we discussed earlier for creating animation sequences from 3D models.

We don't use any isometric tile maps in V3, but they are fully supported by AndEngine. You can create your own isometric tiles with your favorite graphics editor, use Zwoptex to create a tile set, and use Tiled Qt to create tile maps. AndEngine knows how to parse the resulting `.tmx` and `.tsx` files to create the map, and you use the same TMXLayer methods to access individual tiles.

Summary

Tile maps are an important part of game programming. As discussed in this chapter, AndEngine provides support that makes it easy to integrate into a game both tile sets and tile maps created with a set of standard, easy-to-use tools (and did I mention the tools are all available at no charge?).

The steps in building a tile map are as follows:

1. Create the tiles needed using your favorite graphics editor. For an orthogonal tile map, the end result should be a series of uniform tile images in `.png` files.
2. Using the tile images, create a tile set using a specialized utility such as Zwoptex or your favorite graphics program. Make the minimum-size sheet (whose dimensions should always be a power of 2), set the spacing as you like (I use 2 pixels), and save the texture as your tile set image.
3. Using Tiled Qt, create a tile map that is the size you need, with tile sizes that match those of the tiles in the tile set. Import the tile set you created in step 2 and build your tile map using the tiles. Use Tiled Qt to create an external `.tsx` file describing the tile set.
4. Use Tiled Qt to save the tile map in a TMX file, with gzip compression of the tile data. Import the `.tmx` file and the tile set `.tsx` and `.png` files into your Android `assets` folder.
5. Now you're ready to rock and roll. Load the tile map and use the TMX… methods to access individual tiles and get or set their characteristics.

Exercises

1. Using Tiled Qt, add a property to the tombstone tile in `WAVTileset.tmx`, with name = "tombstone" and value = "true".
2. Using the property added in Exercise 1, change WAVActivity so that when the user touches a tombstone tile, it is briefly replaced by a tile that says "Boo!" To make this modification, you'll have to create the tile, create a new tile set, and use Tiled Qt to add the new tile to the map.

This page intentionally left blank

10

Particle Systems

Particle systems are another aspect of computer games that you have seen many times before, even if you didn't explicitly recognize them. Particle systems solve a fundamental problem with the representation of effects that involve a whole bunch of small particles generating patterns. Think of fire, or smoke, or rain, where many particles are needed to create a convincing effect. The particles can behave according to rules, but usually a random element is present as well to make them look realistic.

The particle system provides the environment to create particle emitters, which are the individual effects. Particle systems are most common in 3D game environments, but AndEngine provides some example particle emitters as part of its examples package, and you can also create your own. We'll look at example and custom emitters in this chapter, and show how particle emitters are integrated into the V3 game program.

An example of a particle system effect is shown in Figure 10.1. This screenshot, which is taken from V3, illustrates the particle effect enhancements we'll add later in this chapter.

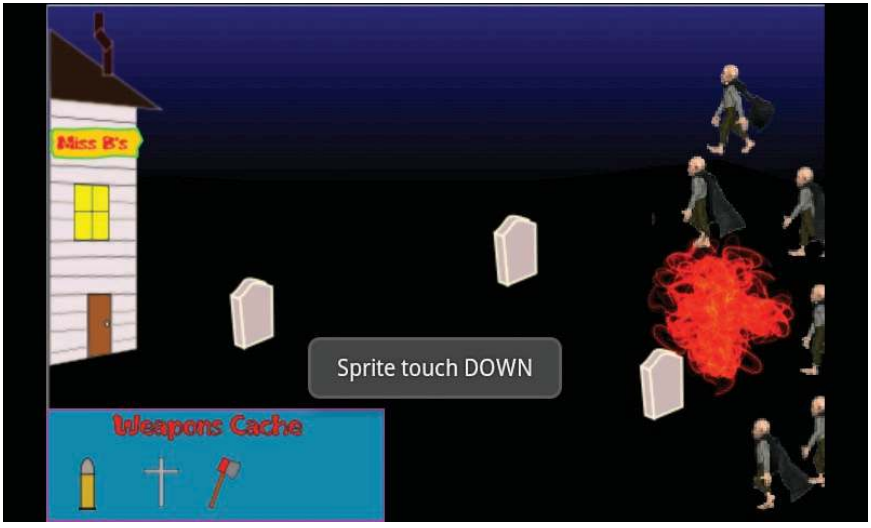


Figure 10.1 Explosion particle effect

What Is a Particle Emitter?

The AndEngine particle subsystem provides the environment to create and run particle emitters. A particle emitter creates a pattern of particles and issues them from a position on the screen.

Trying to represent those particles as sprites would create far too much overhead for a typical game engine, including AndEngine. What is needed instead is a special subsystem that can start with a representative texture and modify it over time, mimicking the physics of particles to produce a convincing animation.

Creating a convincing particle emitter is as much an art as a science. The coding for a particle effect is not difficult, but creating a truly convincing effect typically takes a lot of trial and error, and even the subtlest of changes can make a huge difference in the outcome. To complicate matters, the animation for particle emitters typically doesn't run especially well on the Android emulator. The emulator, after all, doesn't take advantage of any hardware graphics on the host platform, so animation is performed via software, in an emulated instruction set. It is very difficult to debug a particle emitter using just an emulator.

How Do Particle Systems Work?

The art in particle systems comes in part from the nature of the phenomena they model. We all know what a fire looks like, but any real fire also features a lot of randomness. Think of a candle flame. It usually has a dark inner flame, a brighter outer flame, and a gradation of color from bluish to bright yellow. The heat of the flame causes the wax molecules to evaporate from the wick and eventually ignite, creating more or less random currents that affect other luminous wax molecules and contributing to what we recognize as a candle flame.

When modeling that behavior in a particle emitter, there are some things we know concretely and some things we would like to randomize. For example, we know all the wax molecules will start at the wick and travel away from it, generally upward. The exact angle at which they leave the wick, however, needs to be randomized. In addition, we know the color of a wax molecule over time, as it first leaves the wick (dark—really transparent), then bluish as it ignites, bright yellow at peak illumination, and fading to red and finally black smoke. We know that all the particles don't change color in exactly the same way, so there needs to be some randomization in this behavior as well. Finally, we know something about the path of the molecule—rising faster as it gets hotter—but once again that factor needs randomization.

Particle systems allow us to generate a mass (thousands) of particles, establish some basic parameters, and apply variances to those parameters to influence their randomization. Particles can respond to gravity, positive or negative or absent. They can have an associated life span, speed, direction, and acceleration. Their size, color, and spin may all change. We can also select the way the particles blend into the background or replace it.

The AndEngine Particle System

A particle system in AndEngine is a hierarchy of components:

- ParticleSystem: This component comprises the whole effect, including a ParticleEmitter, a small OpenGL texture, and some parameters that control the rate and quantity of particle generation.
- ParticleEmitter: This component generates and displays the particles, modifying them as required for their lifetimes. The type of ParticleEmitter (Circle, CircleOutline, Point, Rectangle, RectangleOutline) defines the shape of the area where particles will be generated. For each particle, the ParticleEmitter randomizes its initial position, velocity, acceleration, rotation, gravity, color, and transparency within the limits that we define in ParticleInitializers.
- Particle: Particles are the individual elements generated and displayed by the ParticleEmitter. All Particles for an effect share the same OpenGL texture, which can be any (small) texture. At the same time, each Particle has its own lifetime and its own set of parameters, such as color, transparency, rotation and scale, which can be modified over its lifetime using ParticleModifiers.
- ParticleInitializers: These components set the initial conditions for all Particles in the effect. They provide a range of values (minimum and maximum) for the velocity, acceleration, rotation, gravity, color, and transparency of the Particles. The ParticleEmitter then picks random values within the given range for each Particle as it is generated.
- ParticleModifiers: These components tell the ParticleSystem how to modify each particle over its lifetime. Modifiers exist for the Particle's lifetime, color, transparency, rotation, and scale, and they define a start time, an end time, and minimum and maximum values. During each update, the ParticleEmitter adjusts the values for each Particle so they fit the directions in the ParticleModifiers. The transitions are gradual. Thus, if a particle's ColorModifier asks to change from reddish to bluish starting at 0 seconds and ending at 1 second, the color of each Particle changes gradually (and individually) over that time.

ParticleSystem

ParticleSystem has one constructor, which simply passes in the needed parameters:

```
ParticleSystem(final IParticleEmitter pParticleEmitter, final float pMinRate,
        final float pMaxRate, final int pMaxParticles,
        final TextureRegion pTextureRegion)
```

The parameters are quite straightforward. The rates are given in particles per second, and the ParticleSystem randomly varies the generation rate through that range. The TextureRegion will be used for every Particle in this system. Obviously, you have to

load the TextureRegion before calling this constructor, using the methods we've seen used with Sprites and other AndEngine elements.

AndEngine doesn't come with particle textures, but several are included with AndEngineExamples, and we've added a few more to the `assets/gfx/particles` file. For most effects, specialized particle shapes are used that have a color and transparency that have been modified to create the desired effect, but you can use any small texture for particles. You can easily create a fountain of Androids, for example, as we'll see in the Exercises at the end of this chapter.

ParticleEmitters

If we want to create a ParticleSystem, we need a ParticleEmitter. Separate classes have been developed for each type of ParticleEmitter based on the desired shape of the effect. Each emitter generates particles at random positions in the shape. AndEngine uses the Java Random utilities to randomize particles and provide a uniform distribution of particles within the shape.

CircleParticleEmitter

This set of methods would have been better named EllipseParticleEmitter—but no matter. It's the emitter you use when you want the particles to emanate from an elliptical or circular region on the screen.

```
CircleParticleEmitter(final float pCenterX, final float pCenterY,
        final float pRadius)
CircleParticleEmitter(final float pCenterX, final float pCenterY,
        final float pRadiusX, final float pRadiusY)
```

The first constructor uses a circular area with the center at (`pCenterX`, `pCenterY`), the second uses an elliptical one. Particles are generated from random positions anywhere inside the circle or ellipse. The ellipse can be rotated to any angle with a RotationInitializer.

CircleOutlineParticleEmitter

This emitter also uses a circular or elliptical shape but generates particles only on the outer rim of the shape.

```
CircleOutlineParticleEmitter(final float pCenterX, final float pCenterY,
        final float pRadius)
CircleOutlineParticleEmitter(final float pCenterX, final float pCenterY,
        final float pRadius)
```

The parameters are the same as they are for CircleParticleEmitter.

PointParticleEmitter

All the particles emerge from a single point with this emitter. The format is as follows:

```
PointParticleEmitter(final float pCenterX, final float pCenterY)
```

RectangleParticleEmitter

This time the emitting shape is a rectangle, which, obviously, can also be a line if one of the dimensions is 0.

```
RectangleParticleEmitter(final float pCenterX, final float pCenterY,
        final float pWidth, final float pHeight)
```

As with the elliptical emitters, the initial angle of the rectangle or line can be adjusted with a RotationInitializer.

RectangleOutlineEmitter

As with the circles, this emitter generates particles on the rim of the rectangle. If the shape is really a line (`pWidth` or `pHeight` equal 0), this method is equivalent to RectangleParticleEmitter. It follows this format:

```
RectangleOutlineParticleEmitter(final float pCenterX, final float pCenterY,
        final float pWidth, final float pHeight)
```

ParticleInitializers

When creating a ParticleSystem, ParticleInitializers can be created and added to the ParticleSystem to control the initial setup of the Particle Emitter. Initializers are available for the acceleration, alpha (transparency), color, gravity, rotation, and velocity characteristics. All of the initializers allow you to either set a fixed value or define a range for the ParticleEmitter to randomize.

AccelerationInitializer

This group of methods sets the starting acceleration for the particles generated by the ParticleSystem:

```
AccelerationInitializer(final float pAcceleration)
AccelerationInitializer(final float pAccelerationX, final float pAccelerationY)
AccelerationInitializer(final float pMinAccelerationX,
        final float pMaxAccelerationX, final float pMinAccelerationY,
        final float pMaxAccelerationY)
```

The first two constructors create initializers with fixed (not random) values. The third constructor is more flexible, allowing us to specify ranges for the acceleration values. The *X* and *Y* directions are relative to the ParticleEmitter.

AlphaInitializer

These two methods specify the beginning transparency of particles:

```
AlphaInitializer(final float pAlpha)
AlphaInitializer(final float pMinAlpha, final float pMaxAlpha)
```

Again we have the option of setting a fixed value or specifying a range for randomization.

ColorInitializer

These methods set the beginning color for all the particles:

```
ColorInitializer(final float pRed, final float pGreen, final float pBlue)
ColorInitializer(final float pMinRed, final float pMaxRed, final float pMinGreen,
        final float pMaxGreen, final float pMinBlue, final float pMaxBlue)
```

The first constructor initializes the particle color, which is a tint, multiplied onto the texture colors for each particle. The second constructor uses a different random tint for each particle.

GravityInitializer

This convenience method can be used only to turn gravity on (which is the same as setting acceleration to the fixed value of earth's gravity).

RotationInitializer

These methods set the initial rotation of the ParticleEmitter, which can range from 0 .0 to 360.0 degrees:

```
RotationInitializer(final float pRotation)
RotationInitializer(final float pMinRotation, final float pMaxRotation)
```

VelocityInitializer

This set of methods specifies the initial velocity of the particles being generated:

```
VelocityInitializer(final float pVelocity)
VelocityInitializer(final float pVelocityX, final float pVelocityY)
VelocityInitializer(final float pMinVelocityX, final float pMaxVelocityX, final float
        pMinVelocityY, final float pMaxVelocityY)
```

VelocityInitializer is very similar to AccelerationInitializer, except that now we're setting the initial velocities of the particles. Again the *X* and *Y* directions are relative to the ParticleEmitter.

ParticleModifiers

Now that we've initialized the ParticleSystem, we need to tell it how the particles change over their lifetimes. We do so using ParticleModifiers, which unfortunately have the same names as their Entity counterparts. It's not a big deal and makes them easy to remember, but it does mean that we need to use fully qualified class names if we plan to sue both Entity and Particle modifiers in the same class. We'll see that arrangement later in the V3 example for this chapter.

AndEngine provides ParticleModifiers for the particle's alpha, color, lifetime, rotation, and scale values. Most of the modifiers include a start time and an end time for the modification to occur, both of which are defined relative to the time the particle was created.

AlphaModifier

AlphaModifier gradually modifies the transparency of particles from `pFromAlpha` to `pToAlpha` over the indicated portion of each particle's lifetime:

```
AlphaModifier(final float pFromAlpha, final float pToAlpha,
        final float pFromTime, final float pToTime)
```

ColorModifier

ColorModifier gradually changes the color tint applied to each particle's texture over the indicated time:

```
ColorModifier(final float pFromRed, final float pToRed, final float pFromGreen,
        final float pToGreen, final float pFromBlue, final float pToBlue,
        final float pFromTime, final float pToTime)
```

ExpireModifier

This modifier defines the lifetime of a particle. It is the only modifier that does not include start or end times.

```
ExpireModifier(final float pLifeTime)
ExpireModifier(final float pMinLifeTime, final float pMaxLifeTime)
```

The first constructor gives all particles the same lifetime, whereas the second provides a range of lifetimes, with each particle randomly assigned a lifetime in that range.

RotationModifier

This method rotates each particle's texture over the indicated time:

```
RotationModifier(final float pFromRotation, final float pToRotation,
        final float pFromTime, final float pToTime)
```

ScaleModifier

The ScaleModifier methods gradually change each particle's scale over the indicated time.

```
ScaleModifier(final float pFromScale, final float pToScale,
        final float pFromTime, final float pToTime)
ScaleModifier(final float pFromScaleX, final float pToScaleX,
        final float pFromScaleY, final float pToScaleY, final float pFromTime,
        final float pToTime)
```

The first constructor modifies the scale uniformly in both the *X* and *Y* directions (i.e., it doesn't change the aspect ratio), whereas the second allows you to change the particle size asymmetrically.

Useful ParticleSystem Methods

ParticleSystem includes some built-in methods that are useful for dealing with particle effects in our code. We can start and stop a ParticleSystem, and set the way it blends with other textures.

Starting and Stopping the Effect

ParticleSystem includes a method that starts or stops the spawning of particles:

```
void setParticlesSpawnEnabled(final boolean pParticlesSpawnEnabled)
```

Just set the parameter as `true` to start spawning particles, or as `false` to shut it off. You can also test whether spawning is currently on or off:

```
boolean isParticlesSpawnEnabled()
```

Setting the OpenGL Blend Function

We've seen this approach before when looking at Sprites. Again, OpenGL is beyond the scope of this book, but basically this method sets the way the particle textures will be blended with the other textures on the screen:

```
void setBlendFunction(final int pSourceBlendFunction,
        final int pDestinationBlendFunction)
```

The blend function values are defined in the following file:

```
javax.microedition.khronos.opengles.GL10
```

Creating Particle Systems

When this book was written, AndEngine required game developers to create each ParticleSystem from scratch, using the initializers and modifiers described earlier. This approach has some problems, however:

- The particle effect created from a combination of texture, initializers, and modifiers is not at all obvious.
- It can be difficult to reuse particle systems across projects, as doing so involves cutting and pasting code that is not packaged separately.
- Particle effect development is tedious: You have to recompile and download the project every time you make the smallest change to the particle effect parameters.

My solution to some of those problems was to introduce XML-based particle effects to AndEngine. By the time you read these words, this functionality may be incorporated into AndEngine as an extension. I've given the XML files the extension `.px`, and included all of the needed code with the example code for this book. As we add particle effects to V3, we'll show both methods so you can get an idea of which you'd prefer.

ParticleSystems the Traditional Way

The original way of creating an AndEngine ParticleSystem is to follow these steps:

1. Create a TextureRegion that contains the Texture you want to use for the particles in your system, and use TextureManager to load it. This process is the same as that for creating any other TextureRegion.

2. Create a new ParticleEmitter using the constructor for the shape you need for your emitter.
3. Create a ParticleSystem, passing it the ParticleEmitter, Texture, minimum rate, maximum rate, and maximum number of particles.
4. Optionally, set the ParticleSystem's OpenGL blend function.
5. Add initializers to the ParticleSystem.
6. Add modifiers to the ParticleSystem.
7. Add the ParticleSystem as a child of the current Scene.

ParticleSystems with XML

The steps when using an XML file are the same in principle, but the PX classes carry out many of these tasks for you. The process is similar to the steps we saw when loading a TMX tile map:

1. Create a PXLoader object, as described in the next section.
2. Use the PXLoader to create the ParticleSystem from the PX file that describes the ParticleSystem.
3. Optionally, set conditions such as the OpenGL blend function.
4. Attach the ParticleSystem as a child of the Scene.

We won't look at all the code for PXLoader and its associated classes. The complete source is included with downloadable code for this chapter (and may be on AndEngineExtensions by the time you read this book). We do need to look at the interfaces we will use, however.

PX Files

The XML files that describe a ParticleSystem are fairly straightforward. XML elements are used for each component, initializer, and modifier. A complete list of XML elements can be found in `PXConstants.java`, along with the example code. An example of a typical PX file (and the one we'll use with V3) is shown in Listing 10.1.

Listing 10.1 **PX File explo.px**

```
<ParticleConfig>
   <emitter
      shape="circle"
      center_x="0.0"
      center_y="0.0"
      radius_x="40.0"
      radius_y="40.0">
   </emitter>
   <system
      texture="particle_fire.png"
      min_rate="100"
```

```
max_rate="100"
max_particles="500">
<init_color
   min_red="1"
   max_red="1"
   min_green="0"
   max_green="0"
   min_blue="0"
   max_blue="0">
</init_color>
<init_alpha
   min_alpha="0"
   max_alpha="0">
</init_alpha>
<init_velocity
   min_velocity_x="-2"
   max_velocity_x="2"
   min_velocity_y="-2"
   max_velocity_y="-2">
</init_velocity>
<init_rotation
   min_rotation="0.0"
   max_rotation="360.0">
</init_rotation>
<mod_scale
   from_scale_x="1.0"
   to_scale_x="2.0"
   from_scale_y="1.0"
   to_scale_y="2.0"
   from_time="0"
   to_time="5">
</mod_scale>
<mod_color
   from_red="1"
   to_red="1"
   from_green="0"
   to_green="0.5"
   from_blue="0"
   to_blue="0"
   from_time="0"
   to_time="2">
</mod_color>
<mod_color
   from_red="1"
   to_red="1"
   from_green="0.5"
```

```
            to_green="1"
            from_blue="0"
            to_blue="1"
            from_time="2"
            to_time="4">
         </mod_color>
         <mod_alpha
            from_alpha="0"
            to_alpha="1"
            from_time="0"
            to_time="1">
         </mod_alpha>
         <mod_alpha
            from_alpha="1.0"
            to_alpha="0"
            from_time="3"
            to_time="4">
         </mod_alpha>
         <mod_expire
            min_lifetime="2"
            max_lifetime="4">
         </mod_expire>
      </system>
</ParticleConfig>
```

The file is easy enough to read without a reference, and you can readily guess the names of the other elements if you don't want to browse through `PXConstants.java`.

Editing PX Files: PXEditor

Also included with the downloadable code are the binaries for an editor you can use to easily edit PX files. At some point (maybe by the time you read this book), we'll add a particle viewer, so you can edit and view particles in real time. For now, you can still create and edit ParticleSystems without having to hand-code all the initializers and modifiers you need in the traditional manner.

Figure 10.2 shows a screenshot of PXEditor, whose use is fairly straightforward. The File menu lets you create a new PX file or load and edit an existing one. The spinners allow you to adjust the parameters, and the image shows you the texture for the ParticleSystem.

ParticlePlayer

In the downloadable code for this chapter, there is a particle viewer application that knows how to load and display PX files. One line of `ParticlePlayer.java` has to be edited for each new PX file you want to view. This line is in `onLoadScene()`:

```
String pxFileName = "gfx/particles/explo.px";
```

Figure 10.2 PXEditor

You just put the PX file you want to view under `assets` and edit the string to tell ParticlePlayer where to find it. The particle texture images in your PX file should be imported as referenced to `assets/gfx`.

When you run ParticlePlayer, touch the screen anywhere and the particle effect should appear there. If any errors occur while loading or displaying the particle effect, you will see them in LogCat. If the PX file contains an error, the LogCat error will tell you the line in which it appears and the type of parsing error.

Figure 10.3 shows a snapshot of the `explo.px` particle system in various stages.

PXLoader

PXLoader is the class that knows how to load `.px` files that describe a ParticleSystem. Creating a PXLoader is just like creating a TMXLoader for a tile map:

```
PXLoader(final Context pContext, final TextureManager pTextureManager)
PXLoader(final Context pContext, final TextureManager pTextureManager,
        final TextureOptions pTextureOptions)
```

If you don't include any TextureOptions, the PXLoader uses `TextureOptions .DEFAULT`.

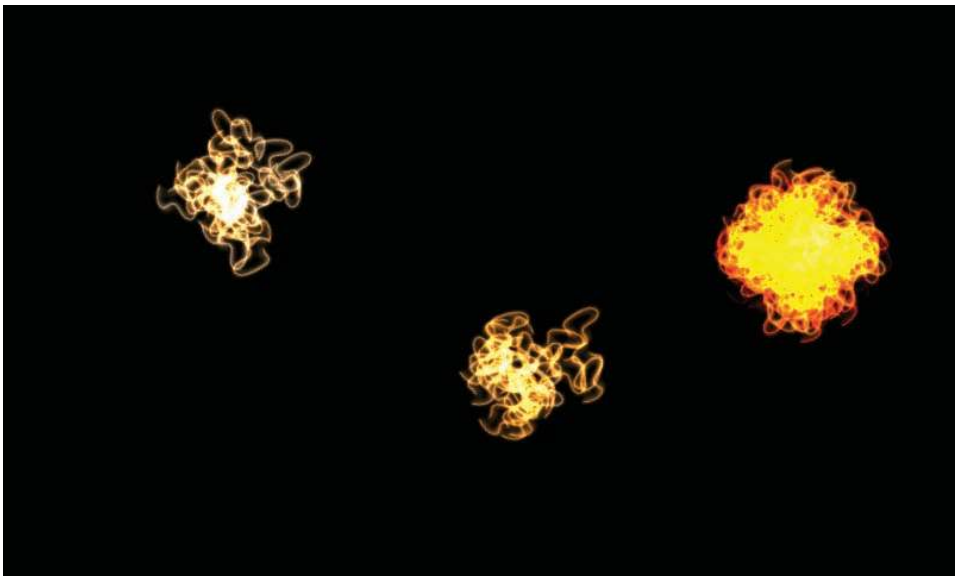


Figure 10.3 ParticlePlayer showing explo.px

PXLoader has two methods that are of use to us in loading ParticleSystem definitions:

```
ParticleSystem createFromAsset(final Context pContext,
        final String pAssetPath)
ParticleSystem load(final InputStream pInputStream)
```

The first method is the one we would normally use to load a particle definition from a subfolder of `assets`. The second method can be used to load a particle definition from some other `InputStream`. In both cases, a complete ParticleSystem is returned. If a problem occurs while loading the definition, a PXLoadException or a PXParseException will be thrown.

Particle Emitters in V3

Let's incorporate some particle emitters into V3 to help liven up the game. We don't have real collision detection enabled yet, but we do effectively have the vampires walking into Miss Bliss's school on the left side of the screen. We will add an effect to the main graveyard scene: When the player touches a vampire, that vampire will burst into flame and disappear from the screen (nice job, Buffy!).

V3 Explosion the Traditional Way

To create this particle effect, we'll use both the traditional method of creating ParticleSystems and the XML method. First let's look at the traditional approach. Listing 10.2 shows the changes to `Level1Activity.java` from the version we saw last in Chapter 9.

Listing 10.2 **Level1Activity.java with Particle Effects: Traditional Way**

```
package com.pearson.lagp.v3;
. . .
public class Level1Activity extends BaseGameActivity {
. . .
   private TextureRegion mParticleTextureRegion;
   private ParticleSystem particleSystem;
   private CircleParticleEmitter particleEmitter;
. . .
   @Override
   public void onLoadResources() {
      /* Load Textures. */
. . .
      TextureRegionFactory.setAssetBasePath("gfx/particles/");
      mParticleTexture = new Texture(32, 32,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      mParticleTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mParticleTexture, this,
            "particle_fire.png",
         0, 0);
      mEngine.getTextureManager().loadTexture(
         this.mParticleTexture);
   }

   @Override
   public Scene onLoadScene() {
. . .
      particleEmitter = new CircleParticleEmitter(
        CAMERA_WIDTH * 0.5f, CAMERA_HEIGHT * 0.5f + 20, 40);
      particleSystem = new ParticleSystem(particleEmitter,
         100, 100, 500, this.mParticleTextureRegion);

      particleSystem.addParticleInitializer(
         new ColorInitializer(1, 0, 0));
      particleSystem.addParticleInitializer(
         new AlphaInitializer(0));
      particleSystem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE);
      particleSystem.addParticleInitializer(
         new VelocityInitializer(-2, 2, -2, -2));
      particleSystem.addParticleInitializer(
         new RotationInitializer(0.0f, 360.0f));

      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-            .
ScaleModifier(1.0f, 2.0f, 0, 5));
```

```
      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-
            .ColorModifier(1, 1, 0, 0.5f, 0, 0, 0, 3));
      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-
            .ColorModifier(1, 1, 0.5f, 1, 0, 1, 2, 4));
      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-            .
AlphaModifier(0, 1, 0, 1));
      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-
            .AlphaModifier(1, 0, 3, 4));
      particleSystem.addParticleModifier(
         new ExpireModifier(2, 4));

      particleSystem.setParticlesSpawnEnabled(false);
      scene.getLastChild().attachChild(particleSystem);

      return scene;
   }

   @Override
   public void onLoadComplete() {
   }

    private Runnable mStartVamp = new Runnable() {
        public void run() {
           int i = nVamp++;
           Scene scene = Level1Activity.this.mEngine.getScene();
              float startY = gen.nextFloat()*(CAMERA_HEIGHT - 50.0f);
              asprVamp[i] = new AnimatedSprite(CAMERA_WIDTH - 30.0f,
        startY, mScrumTextureRegion.clone()) {
                 @Override
                 public boolean onAreaTouched(
            final TouchEvent pAreaTouchEvent,
            final float pTouchAreaLocalX,
            final float pTouchAreaLocalY) {
                    switch(pAreaTouchEvent.getAction()) {
                    case TouchEvent.ACTION_DOWN:
                       /* Is there a vampire close by? */
                       for (int j=0; j<nVamp; j++){
                          if (
(Math.abs(asprVamp[j].getX() + (asprVamp[j].getWidth()/2) -
pAreaTouchEvent.getX()) < 10.0f) &&
                                (Math.abs(asprVamp[j].getY() +
(asprVamp[j].getHeight()/2) - pAreaTouchEvent.getY()) < 10.0f)) {
                    particleEmitter.setCenter(
               pAreaTouchEvent.getX(),
```

```
                pAreaTouchEvent.getY());
                     particleSystem.setParticlesSpawnEnabled(
                true);
                         mHandler.postDelayed(mEndPESpawn,3000);
                         asprVamp[j].clearEntityModifiers();
asprVamp[j].registerEntityModifier(
                           new AlphaModifier(1.0f, 1.0f, 0.0f));
                                asprVamp[j].setPosition(
                   CAMERA_WIDTH,
                   gen.nextFloat()*
                   CAMERA_HEIGHT);
                            }
                        }
                        break;
                   }
                     return true;
          }
              };
          scene.registerTouchArea(asprVamp[i]);
. . .

     private Runnable mEndPESpawn = new Runnable() {
         public void run() {
           particleSystem.setParticlesSpawnEnabled(false);
         }
     };
}
```

Let's look at each of the methods in turn to understand how particle effects are displayed in the traditional way:

- In `onLoadResources()`, we load the Texture we need for the ParticleSystem. We've previously placed the image in `assets/gfx/particles`. The rationale for using a separate folder is that ParticleSystems can apply to multiple places in our game.
- In `onLoadScene()`, we create a CircleParticleEmitter with which we will create our ParticleSystem, and assign an appropriate number of particles and rates ("appropriate" is determined by trial and error for the most part).
- We add "appropriate" initializers and modifiers to our ParticleSystem to create the desired effect, and for the moment `setSpawnEnabled(false)`. We don't need to generate any particles until a vampire is touched, but we go ahead and attach the ParticleSystem to the Scene.
- In the Runnable `mStartVamp`, we add an `onAreaTouched()` listener, so we can capture touches. This method tells us when a vampire is touched, but it doesn't tell us which vampire was touched, so we iterate through the vampires that have been launched so far. We can get away with that because we know there are no more

than 10 vampires on the screen at any given time. In a game with potentially hundreds of active sprites, of course, we would want to use a more sophisticated collision detection strategy (which we discuss in Chapter 12). The formatting is a little messy in Listing 10.2, but is much easier to read in the real source.

- If the touch point is within 10 pixels of the vampire's position, we'll call it a hit.
- When we find a hit, we position the particle emitter at that vampire and turn on spawning to generate the explosion effect. We position the effect at the hit point. We could have chosen the vampire's position, but somehow having the flame where you touch is more appropriate.
- We ask Android to run a Runnable that will stop the explosion effect in 3 seconds.
- We remove the modifiers from the vampire sprite, so we're not wasting cycles in trying to move it to a new starting position, and we set an alpha modifier to make it disappear from the screen in 1 second.
- We tell Android that we've handled the touch by returning `true` at this point and don't waste time looking at the other vampires.
- The Runnable `mEndPESpawn()` shuts off particle generation to end the particle effect.

V3 Explosion the XML Way

A modified version of the same particle effect is included with the downloadable source code as V310PX. In this version, we replace the ParticleSystem creation code in Listing 10.2 with the equivalent code that creates the ParticleSystem from the `explo.px` file we showed in Listing 10.1. These changes are shown in Listing 10.3.

Listing 10.3 **Level1Activity.java with Particle Effects: XML Way**

```
package com.pearson.lagp.v3;
. . .
public class Level1Activity extends BaseGameActivity {
. . .
   private ParticleSystem particleSystem;
   private BaseParticleEmitter particleEmitter;
. . .
   @Override
   public Scene onLoadScene() {
. . .
      try {
         final PXLoader pxLoader = new PXLoader(this,
            this.mEngine.getTextureManager(),
            TextureOptions.BILINEAR_PREMULTIPLYALPHA);
         particleSystem = pxLoader.createFromAsset(this,
            "gfx/particles/explo.px");
      } catch (final PXLoadException pxle) {
```

```
            Debug.e(pxle);
        }
        particleSystem.setBlendFunction(GL10.GL_SRC_ALPHA,
            GL10.GL_ONE);
        particleSystem.setParticlesSpawnEnabled(false);
        particleEmitter =
         (BaseParticleEmitter) particleSystem.getParticleEmitter();

        scene.getLastChild().attachChild(particleSystem);
. . .
```

Looking at the changes from the last (traditional) approach:

- We don't need to declare a Texture or TextureRegion for the particle in this version, as that task is handled by PXLoader. We do need to import the texture image into wherever the PX file says PXLoader should expect it. The file `explo.px` says that a texture `particle_fire.png` should appear in the current Texture asset path (which we set to `gfx/Level1` for all the Textures in this activity). We declare our ParticleEmitter as class BaseParticleEmitter because we don't know which type of emitter it is yet.
- We no longer need to load the TextureRegion or Texture in `onLoadResources()`.
- In `onLoadScene()`, we create a PXLoader and use its `createFromAsset()` method to create the ParticleSystem described in the file `explo.px`. We previously imported this file into `assets/gfx/particles`. We retrieve the ParticleEmitter from our ParticleSystem using the `getParticleEmitter()` method so that we can use it later to reset its position.
- Everything else is the same. The sprite touch detection logic is unchanged from the traditional version. The ParticleSystem created by PXLoader is like any other ParticleSystem. Thus you can add modifiers, reposition the emitter, and do anything you would do with a ParticleSystem created in the traditional way.

Summary

Particle effects can make our games more interesting, and just plain more fun, by implementing complex, lifelike effects. The AndEngine particle system makes this happen without consuming a huge amount of computing resources, as would be required if we tried to duplicate the same effects with animations.

Particle effects are rather easy to include in your game, and not that difficult to invent. You can add particle systems to your games in at least three ways:

1. Code the particle system into your game using the initializer and modifier classes provided by AndEngine.

2. Create a PX file that describes a particle system and use the PXLoader classes to load it into your application at runtime.
3. Use a particle system that's been created and debugged by someone else. It can consist of code that you cut and paste or a PX file that you build into your project.

However you choose to create the particle system, it can be positioned and triggered based on any event in your game that you choose. In Chapter 11, we'll see how to make the special effects even more compelling by introducing sounds that go with them.

Exercises

1. Working with the traditional way of creating particle effects (as shown in Listing 10.2), change `particleSystem` so the effect looks more like a puff of smoke. Use the `particle_smoke.png` image file (found in the `assets/px` folder in the downloadable code) and appropriate colors. Adjust the particle system until you think the effect is convincing.
2. Repeat Exercise 1, using the PXLoader approach to building particle effects (as in Listing 10.3). You can either create your `smoke.px` file by hand or use PXEditor to create it. Either way, it's easiest to start from `explo.px`.
3. Create a brand-new effect of your own, illustrating falling rain. Create the particle system using PXLoader, and show your effect using the ParticlePlay application.

This page intentionally left blank

11

Sound

Sound enhances the mood of a game and can help bring the action to life. It provides an emotional connection that can't be achieved any other way. Filmmakers have recognized this fact ever since talkies started with *The Jazz Singer,* and they invest huge sums of money in creating just the right background music, and just the right sound effects, for their films. You probably don't have their budget, and you may not be as gifted as John Williams, but sound will be important for your game. The AndEngine library for Android provides APIs that build on Android's native multimedia capabilities and make it easy to incorporate these features.

How Sound Is Used in Games

An entire profession is built around the design of sounds for video games. There are even trade conferences devoted to the subject (e.g., `http://www.gamesoundcon.com/`). Although we certainly won't become game sound experts in the next few paragraphs, we will learn some basic principles that will guide our work.

There are two basic types of sound in a video game:

- Music, which is usually background music
- Sound effects that accompany events in real time within the game play

Music

Music is important for setting the mood of the game. Somber music creates a subdued game-playing mood, whereas music that is reminiscent of happy times (carousels, hurdy-gurdys) sets a happy mood. The tempo of the background music often reflects the current tempo of the game. In a game that features every increasing layers of difficulty, the tempo might increase as the challenges get harder and villains are attacking faster and faster.

There don't seem to be any hard-and-fast rules about what makes a piece of music reflect a particular emotion. Major keys are usually associated with "happy" music, and minor keys with "sad" music, but there are exceptions to those rules. The best

advice seems to be "You'll know it when you hear it." When picking background music for your game, it is important to listen to a lot of other games, and to music in general. Try to pick background music that won't become too boring or annoying when it's played over and over.

Increasingly, especially on mobile devices, players don't listen to the music you pick while they're playing your game. Instead, they prefer to listen to their own music files that they've loaded onto the device. Android devices do a great job of allowing the device music player to keep playing while the user activates other applications, including games, so there's nothing you have to do to enable that use case.

Sound Effects

Whereas music generally plays continuously in the background, sound effects are short sounds that play coincident with events in the game. If a gun is fired, the player should hear a bang. If a villain bumps into an object, there should be a bump noise, and maybe an "Ugh!" from the villain. If a vampire goes up in flames, the player should hear the sound of a fiery explosion.

The tools used to create and edit sound effects overlap those used for music, but they are different, in that sound effects are short and the focus is on trimming and filtering to enhance the effect. Music, by comparison, is, well, music. The emphasis is more on the succession of tones and silences that make up a musical composition, along with the timbres of the musical instruments that are used to play it.

Sources of Music and Effects

Companies building professional games for game consoles or the PC spend hundreds of thousands of dollars creating unique sounds and getting these sounds just right. They often have an in-house composer who is charged with creating new music for their games. You probably don't have pockets quite that deep, but you can still create great sound for your game.

A wide variety of music and sound files are available on the Internet, but be very careful about license restrictions. The same kinds of considerations exist for music and sound effects as apply to images and animations. The last thing you want to do is publish a game and have somebody's lawyer come after you because you don't have a license to use the client's intellectual property.

When looking for sources of music for your game, you have the following options:

- Make your own music and sound effects. If you have the talent and the equipment and can record your own original composition for your game, that option is by far the best choice. With this approach, you own all the rights to your own creation and performance.
- Use a friend's music. If you have a friend with musical talent and the equipment to create an original musical background, that's almost as good. Just make sure

you have a clear agreement with your friend about the use of the music—and it's best to put that agreement in writing.

- License someone else's music. This can be expensive if "someone else" is the Rolling Stones, but there are also websites where you can purchase a license to use professionally created music for a reasonable fee. The following sites offer such services, for example:
  - `http://www.partnersinrhyme.com`: includes both music and sound effects
  - `http://www.5alarmmusic.com/`
  - `http://www.mymusicsource.com`
- Use music and sound effects that are in the public domain. We used this option for the background music in V3, using a J. S. Bach composition to set the mood for the main game screen. The MIDI file for this piece of music was found at `http://www.midiworld.com`.

Tools for Music and Effects

Music and sound effects files are both audio files, so the toolsets used to manipulate them overlap to some extent. Sound development can happen in several different ways, and the most appropriate tools depend on exactly what you need to do. Examples of different approaches include the following:

- If you are capturing music being played live, many professional music editing programs are available, such as Sony Sound Forge, M-Audio Pro Tools, Cubase, and Adobe Soundbooth. None of these programs are free, but most offer a "lite" version for a token amount of money.
- If you are adapting an existing composition, you can use tools such as MuseScore and MakeMusic Finale to make minor edits, change voices, alter tempo, and even transpose notes. We'll see an example of the use of MuseScore later in this chapter.
- If you are creating sound effects, an audio editor, such as Audacity, will likely be sufficient. These kinds of programs don't have the musical flexibility of the tools mentioned previously, but they're great for capturing and editing raw audio. We'll also see a detailed example of using Audacity to capture sound effects later in this chapter.

Sound Codec Considerations

The Android media player knows how to interpret a wide variety of media types. Some of these (as of Android 3.0) are listed here, along with pros and cons for using that codec for game sound files:

- AAC: Including this option in the list is a bit misleading. As of Android 3.0, the media player does know about AAC-encoded audio, but only when it is embedded in 3GP and MP4 video. AAC isn't an option for Android games (yet).

- MP3: MP3 is the old standby for psycho-acoustically encoded audio, and it works quite well for game music and sound effects. It is not supported by all tools, due in part to the need for encode licensing. Many Android devices include hardware MP3 decoders that stream MP3 directly from a file, saving precious memory and processor cycles.
- MIDI: Rather than compressing an audio waveform, MIDI lists the notes to be played for a piece of music, along with metadata such as which instrument should be playing which notes. MIDI files are very compact compared with any other alternative, but the sound can be rather mechanical. MIDI does have limited provision for support of sound effects—by naming instruments for "gunshot," "applause," and so on. On most Android devices, MIDI is computation and memory intensive compared with MP3, Ogg, and WAV, as the processor streams the file and calculates the waveform from wavetables that are kept in memory.
- Ogg Vorbis: This codec is the open licensed equivalent of MP3 and is quite widely supported. It offers the same advantages as MP3, with the additional advantage of being freely available. As is true for MP3, most Android devices contain dedicated hardware to stream and decode Ogg-encoded sound files.
- WAV: The WAV format is really digitized audio, most often digitized using linear pulse code modulation (LPCM), and most often uncompressed. The digitization can take on different parameters, but WAV files can quite literally offer "CD quality": WAV is the format used on commercial music CDs. Because it is uncompressed, this codec also results in the largest files (by far), which is an important consideration in a mobile game, particularly for the background music.

Sound in AndEngine

The AndEngine interfaces refer to music as "music," and to sound effects as "sounds." As a rule of thumb, think of music as a piece of audio lasting more than 5 seconds, and sound effects as lasting less than 5 seconds, and preferably less than 3 seconds.

Most game developers settle on one musical theme per activity, and AndEngine uses objects of the Music class and keeps them around. Sound effects, in contrast, come and go, so AndEngine uses Android SoundPool objects that facilitate management of the short clips. All of that work occurs behind the scenes, so you don't have to worry about it as a game developer—but it's nice to know that Nicolas and the other AndEngine developers have thought through the differences for us.

From our point of view, there are four classes we need to know about:

1. Music: This class represents a stream of music.
2. Sound: This class represents a sound effect.

3. MusicFactory: This singleton class knows how to extract a musical stream from a file.
4. SoundFactory: This singleton class loads sound effects.

The music and sound APIs in AndEngine are very similar to the Android MediaPlayer APIs, if you are familiar with them. AndEngine uses the MediaPlayer beneath the covers, so it makes sense that they are consistent. There are also MusicManager, SoundManager, and SoundLibrary classes, but we don't need to deal with them directly just to play and control music and sounds.

Music Class

We don't normally use the constructor provided for Music; instead, we let the MusicFactory create Music objects for us (see the next section). Once we have a Music object, the class provides a set of methods that let us control the playing of the piece:

```
void play()
void stop()
void pause()
void resume()
void release()
```

These methods do just what they say. If you `release()` a Music object, it is no longer available for playing.

```
void setLooping(final boolean pLooping)
void setVolume(final float pLeftVolume, final float pRightVolume)
void seekTo(final int pMilliseconds)
boolean isPlaying()
```

These methods control aspects of playing the music. Again they're pretty obvious. The `isPlaying()` method returns `false` if the music is paused or stopped.

```
void setOnCompletionListener(final OnCompletionListener pOnCompletionListener)
```

Using this method, you can register a listener method to run when the musical piece is finished playing.

Sound Class

The methods provided for the Sound class are very similar to those for the Music class, as you'd expect. There are a few important differences, all due to the fact that sound effects are short bursts of sound:

- There is no `seekTo()` method.
- There is no `isPlaying()` method.
- There is no way to set a completion listener.

Some additional methods come easily because Sound uses the Android SoundPool class to manage the sound effects,

```
void setLoopCount(final int pLoopCount)
```

This method works together with `setLooping()`. If `setLooping()` is set to `true`, the effect loops until you `stop()` it. If `setLooping()` is `false`, and `setLoopCount` is non-zero, it loops that number of times. Oddly enough, `pLoopCount` is zero based, so if you want an effect to play 5 times, you should set `pLoopCount` to 4.

```
void setRate(final float pRate)
```

You can use this method to vary the playback rate. A `pRate` of 1.0f is normal playback speed. The allowable range is 0.5f (playback at half the speed) to 2.0f.

MusicFactory

As with some of the other AndEngine media factory classes, MusicFactory provides a group of `createfrom...` methods to load music from various sources:

```
Music createMusicFromFile(final MusicManager pMusicManager,
        final Context pContext, final File pFile)
Music createMusicFromAsset(final MusicManager pMusicManager,
        final Context pContext, final String pAssetPath)
Music createMusicFromResource(final MusicManager pMusicManager, final
        Context pContext, final int pMusicResID)
```

These methods work just like the factory classes we've seen for Textures, TMX files, and PX files. Some developers have reported issues with the `createMusicFrom-Resource()` method. For creation from assets, MusicFactory provides a method to set a default `assets` subfolder path, just like TextureFactory and the other classes:

```
void setAssetBasePath(final String pAssetBasePath)
```

SoundFactory

An analogous group of methods come with SoundFactory, this time taking advantage of some SoundPool capabilities:

```
Sound createSoundFromPath(final SoundManager pSoundManager,
        final Context pContext, final String pPath)
Sound createSoundFromAsset(final SoundManager pSoundManager,
        final Context pContext, final String pAssetPath)
Sound createSoundFromResource(final SoundManager pSoundManager,
        final Context pContext, final int pSoundResID)
Sound createSoundFromFileDescriptor(final SoundManager pSoundManager,
        final FileDescriptor pFileDescriptor, final long pOffset,
        final long pLength)
```

In this book we will always load media assets from the `assets` folder, but it's good to know the other methods are there, should you need to load items from a file or other resource. SoundFactory has the same `setAssetBasePath()` method provided by the other factory classes in AndEngine.

Adding Sound to V3

We want to create background music and sound effects for V3. In this section we will look at examples of both:

- **Creating and implementing the background music for the game.** We want the music to start when we start the game and to play in the background until we enter a game-playing level (e.g., Level1Activity). If the game pauses, the music should stop, and then resume when the game resumes. If the player turns the music off on the Options page, it should stop and not play again until the player turns it back on. The music and effects on/off settings should be persistent, even if our Android device is turned off.
- **Creating and implementing a sound effect for the particle effect we added in Chapter 10.** When the user touches the screen to blow away a vampire, a blast sound should accompany the player's triumph. If the player has turned sound effects off on the Options page, then no sound effects should play. The music and the sound effects can be turned on and off independently.
- **Adding sound effects to the game weapons.** In Chapter 10, we made it possible for the player to reposition weapons from the Weapons box to the playing field. Now we'd like to have the weapon do something when the player releases the touch. We'll need sound effects to accompany those actions (i.e., firing the bullet and throwing the hatchet).

Creating the Sound Effects

We need a sound effect for the blasts that immolate the vampires, and I selected a public domain WAV file from the PartnersInRhyme site mentioned earlier in this chapter. This site includes both public domain sound effects and royalty-free licensed sound effects. The difference between the two is that the public domain effects have no restrictions on their use. The royalty-free sound effects are sold for a small fee and come with a license that defines how they may be used, including no need to pay royalties for their use.

I selected a public domain effect: `Fireball3.wav`. This file is not very big (18KB) and has a good whoomph to it, but I'd like to tune it a bit. I'll first import it into Audacity, using File > Open. The screen shown in Figure 11.1 appears.

The length of the clip is short (less than 1 second), which is appropriate for a sound effect, and it fades in and out nicely. Audacity gives you tools to do an amazing amount of waveform analysis. As an example, let's look at the audio spectrum of the file by choosing Analyze > Plot Spectrum.... The resulting screen (Figure 11.2) shows us the relative amplitudes of the waveform across the spectrum but does not factor in the nonlinearity of human hearing. Even so, this spectrum plot is still useful to help us adjust the sound effect.

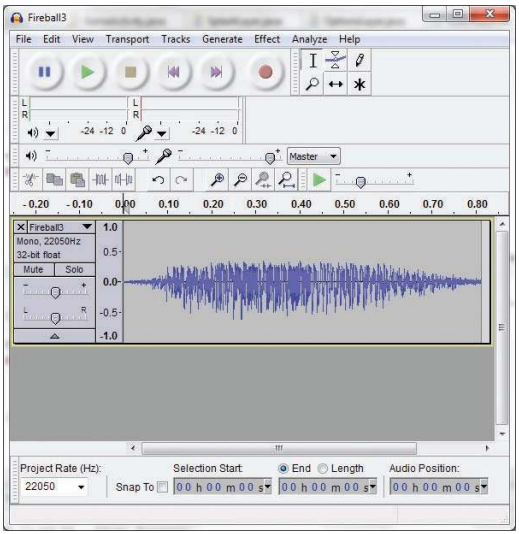


Figure 11.1 Audacity, with Fireball3.wav loaded

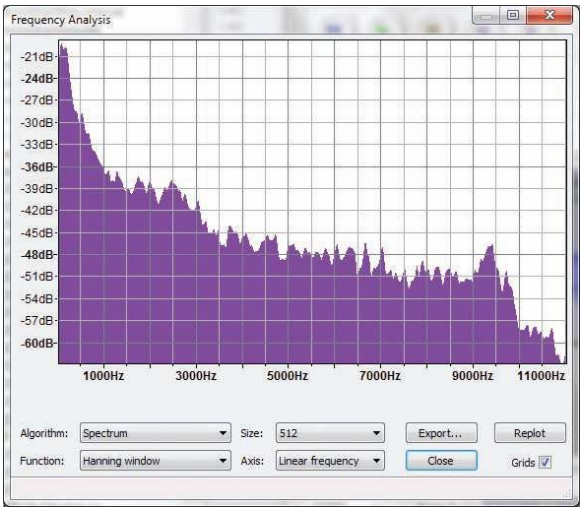


Figure 11.2 Audacity, showing Fireball3.wav spectrum

There's some good rumble at the low end of the spectrum, but we'd like to add some more. We want these fireballs to be worthy of a truly vile vampire going up in flames. One way Audacity lets us make these adjustments is with the Effects menu (not to be confused with game sound effects—these effects are various changes Audacity can help you make to the waveform you are editing). Just pulling down the Effects menu, you can see the long list of effects types that Audacity provides. A few of the more interesting ones for effects are listed here:

- Amplify: makes the sound louder
- Bass boost: boosts the low end sounds—just what we want in the example case
- Change pitch
- Compressor: compresses the amplitude of the waveform in a nonlinear way
- Echo
- Equalization
- Fade In
- Fade Out
- Inverter: inverts the waveform (its amplitude, not its duration)
- Leveller: levels out the peaks in a waveform
- Noise Removal
- Normalize: adjusts the amplitude to remove DC offset and normalize to a given volume
- Reverse: reverses the waveform in time
- Truncate Silence

For the fireball sound effect, I will boost the bass using the Bass Boost effect, as shown in Figure 11.3.

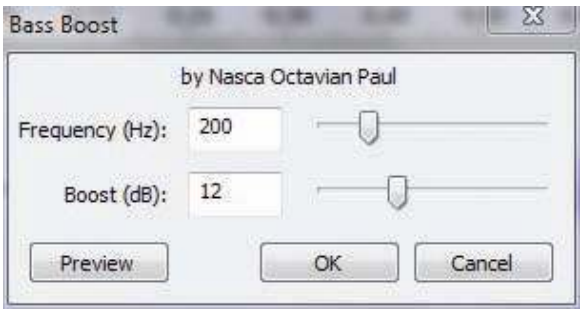


Figure 11.3 Audacity: bass boost of Fireball3.wav

I can try different boost points by adjusting the frequency and boost sliders and listening to Preview until the sound is just the way I'd like it to be. We could apply any of the other effects, and when we're done tweaking, we can save it, using File > Save Project. This step creates an Audacity project file (`.aup`) and saves all our changes, but it does *not* create a new WAV file. To do that, we use File > Export…, which allows us to save the waveform in its original WAV format, or to change codecs and save the file in MP3, Ogg/Vorbis, or many other formats.

Saving the file in the `.wav` format results in a file that is 18.01KB—not too bad, but we're going to import a bunch of these files for different sound effects, so collectively they could add up. To save a sound file as an MP3, at least on Windows, Audacity will ask you to download an encoder library, `lame_enc.dll`. The legal implications of using the LAME libraries are unclear, at least in the United States, where I live. As a practical matter, it's unlikely that the MP3 license holders would come after me for creating a game (or a book) using MP3, but there is an alternative that involves no risk at all—namely, Ogg/Vorbis.

Ogg/Vorbis does psycho-acoustic compression of audio files, just like MP3, but it is open-source format, with no licenses required for its use. Audacity comes with an Ogg/Vorbis encoder standard. Using File > Export… and selecting the Ogg/Vorbis format from the Save As Type drop-down menu produces a file, `fireball.ogg`, that is 9.73KB—a bit more than half the size of the uncompressed WAV file. Listening to the `.ogg` file, I can't hear any difference from the original, so I'll elected to use that format in V3.

In a similar way, I created sound effects for firing the bullet (`gunshot.ogg`) and throwing the hatchet (`whiffle.ogg`). All of these are loaded into the `assets/mfx` folder for use in the game.

Creating the Background Music

As mentioned earlier, we picked a J. S. Bach composition, Fugue in G Minor, as the background music for the main screen of the game. The rights to this composition have long been in the public domain, so there's no problem there. The rights to specific performances of the composition are a different matter, however: The rights to a performance would be licensed separately.

Fortunately, this composition is available in a form that is also in the public domain—namely, as a MIDI file. Not all MIDI files are in the public domain, but it happens that a public domain version is available at MIDIWorld (its URL was given earlier in the section "Sources of Music and Effects").

To use this file with cocos2d for Android, I downloaded it and used MuseScore to open it up and edit it, as shown in Figure 11.4. The author was apparently Jason Fortunato, but there is no contact information for him on the website. Thanks, Jason, for taking the time to enter Bach's score.

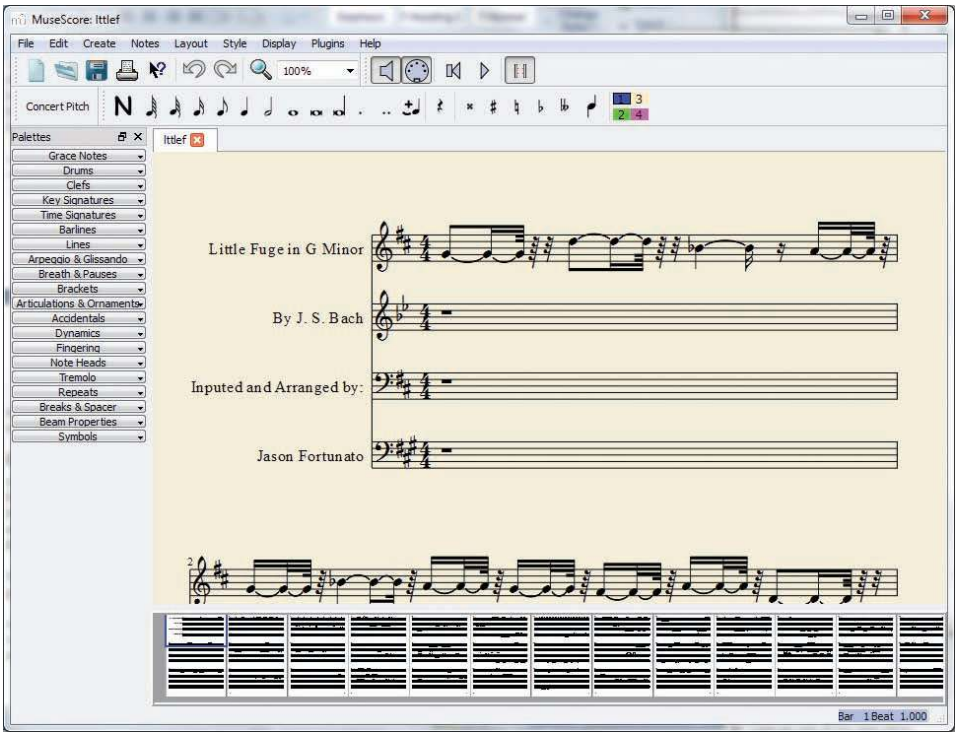


Figure 11.4 MuseScore with Bach Fugue opened

As you can see, MuseScore (and similar software, such as MakeMusic's Finale products) provides a user interface that is much more "music friendly." Instead of manipulating waveforms, as in Audacity, here we are manipulating notes and rests on staves. MuseScore is very music oriented and knows about musical interfaces, such as MIDI.

After listening to the piece, there were two things I wanted to change—the tempo and the voicing, or which instrument was assigned to each of the staves in the piece. MuseScore will easily let me change both voicing and tempo. I could have used Audacity to change just the tempo, but it makes sense to make all of the changes in one place.

In MuseScore, the voicing for a MIDI file is available from the menu under Display > Mixer. The panel that comes up is titled "MuseScore:Part List," as shown in Figure 11.5, and you can see the "Sound:" pull-down options for each staff. For this file the voices all default to "Strings CLP."

Using the mixer panel, I changed the voicing of each staff. There are many voices to choose from (128 or so), and I chose a set that sounded good to me. I don't claim to have a musical ear.

To change the tempo using MuseScore, you select the menu commands Display > Play Panel. This brings up the panel shown in Figure 11.6.

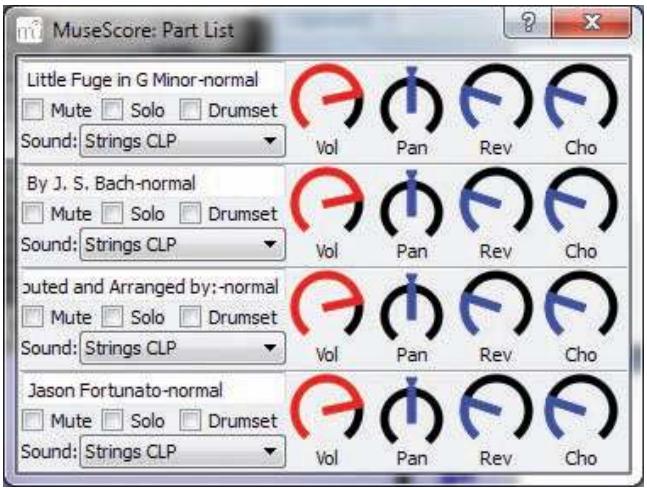


Figure 11.5 MuseScore mixer panel

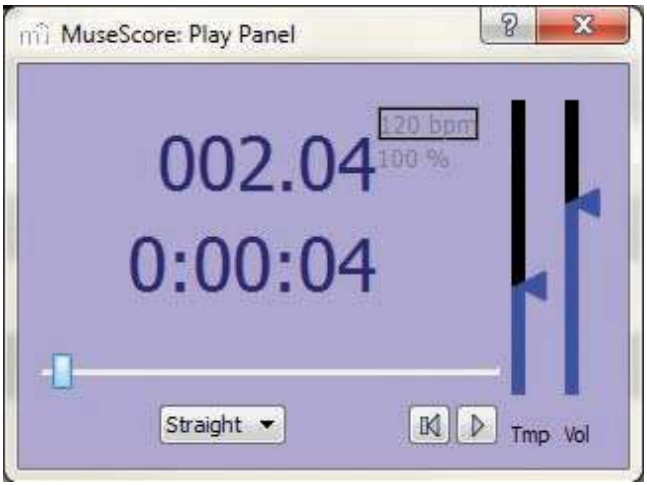


Figure 11.6 MuseScore Play Panel

Table 11.1 **Bach Fugue File Sizes**

Format	Size (KB)
MIDI	12
Ogg/Vorbis	1979
WAV	23,537

The large numbers are the current playback position in measures (on top) and the current playback position in time (on the bottom). To the upper right of the total time, you can see the tempo, in beats per minute (bpm). The horizontal slider near the bottom of the dialog tracks the current play point in the piece, while the vertical sliders adjust the tempo and volume. The MuseScore Help doesn't explain the pull-down options (Straight/Swing/Shuffle), but we don't need them. I adjusted the tempo down to 70 bpm to slow the piece down and make it a bit more dirge-like. With the slowdown, the total time of the piece is slightly less than 4 minutes.

Choosing File > Save from the menu allows us to save a MuseScore project file. Choosing File > Save As… gives us the same dialog, but now the "Save As Type" pull-down menu allows us to save the file in a variety of audio formats. For comparison purposes, I saved the file as MIDI (`.mid`), Ogg/Vorbis (`.ogg`), and WAV (`.wav`) files. The results are shown in Table 11.1.

The WAV file is huge—and out of the question for our application. We really don't want to increase the size of our game file by 20MB for each piece of music. We could go with the tiny MIDI file, but Android doesn't have all the MIDI voices that are in MuseScore, so the result sounds a little too mechanical. With the Ogg/Vorbis version, we can use all the power of MuseScore to adjust the music, and the file size is still quite small. We'll go with that option. As with the sound effects, the result is imported into `assets/mfx`.

Making the Coding Changes to V3

We need to make a number of changes to V3 to incorporate the sounds into the game:

- We need global Boolean functions to signify music on/off and effects on/off. In earlier chapters, we just toggled local Boolean values in `OptionsActivity.java` to achieve this kind of outcome, but now we need the results available on a game-wide basis. Fortunately, Android provides an easy way to do this with SystemPreferences.
- The Music and Sound objects need to be managed in the activities' `onPause()` and `onResume()` methods so that the sound works properly as we start and stop the game, or move between activities.
- The background music and sound effects must be loaded before they are played.

- If music is turned on, the background music begins to play as StartActivity starts, and then stops when we enter Level1Activity.
- The weapons animations need to be enhanced to actually shoot the bullet and throw the hatchet (crosses don't do much except wait for a vampire to bump into them). If effects are turned on, appropriate sound effects should sync with the animations.
- If effects are turned on, the fireball sound effect must be triggered whenever we register a touch hit on a vampire in Level1Activity.

Listing 11.1 (`StartActivity.java`), Listing 11.2 (`OptionsActivity.java`), and Listing 11.3 (`Level1Activity.java`) show the primary changes that are required.

Listing 11.1 **Changes to StartActivity.java**

```
package com.pearson.lagp.v3;

. . .

public class StartActivity extends BaseGameActivity {
. . .
   static protected Music mMusic;
   private SharedPreferences audioOptions;
   private SharedPreferences.Editor audioEditor;

. . .
   @Override
   public Engine onLoadEngine() {
      mHandler = new Handler();
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      audioOptions = getSharedPreferences("audio", MODE_PRIVATE);
      audioEditor = audioOptions.edit();
      if (!audioOptions.contains("musicOn")){
         audioEditor.putBoolean("musicOn", true);
         audioEditor.putBoolean("effectsOn", true);
         audioEditor.commit();
      }
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), this.mCamera).setNeedsMusic(true));
   }

   @Override
   public void onLoadResources() {
. . .
```

```
      MusicFactory.setAssetBasePath("mfx/");
      try {
        StartActivity.mMusic = MusicFactory.createMusicFromAsset(
         this.mEngine.getMusicManager(),
         getApplicationContext(),
         "bach_fugue.ogg");
         StartActivity.mMusic.setLooping(true);
      } catch (final IOException e) {
         Debug.e(e);
      }
   }

   @Override
   public Scene onLoadScene() {
. . .

      //Start the music!
      mMusic.play();
         if (!audioOptions.getBoolean("musicOn", false)) {
             mMusic.pause();
         }
      return scene;

   }

   @Override
   public void onGamePaused() {
      super.onGamePaused();
      StartActivity.mMusic.pause();
   }

   @Override
   public void onGameResumed() {
      super.onGameResumed();
         if (audioOptions.getBoolean("musicOn", false))
         StartActivity.mMusic.resume();
      mHandler.postDelayed(mLaunchTask,3000);
   }

. . .
}
```

The changes are shown in bold, and here is a description of each:

- We added a static variable for the Music object. This is a bit of a kludge, but we need to have access to the object across our game, and SharedPreferences doesn't provide an interface for storing and retrieving Objects. In a very un-object-oriented

fashion, we will access `mMusic` directly from other activities. We can get away with this strategy because there is only one Music object, so management is easy.

- We also added private variables for SharedPreferences and its editor. As mentioned at the beginning of this section, we will use SharedPreferences for the Boolean values that track whether music and sound effects are turned on.
- In `onLoadEngine()`, we initialize the SharedPreferences, allowing for the fact that they may already exist from a previous run of the game.
- Also in `onLoadEngine()`, we add a method call, `setNeedsMusic(true)` to the EngineOptions constructor, which informs AndEngine that this module will be referencing the Music methods.
- In `onLoadResources()`, we use the MusicFactory to create and load the Music object with the music file we've previously installed in `assets/mfx`.
- We `setLooping(true)` so the music will keep playing until the player loses his or her mind and throws the Android device against the nearest wall.

Listing 11.2 **Changes to OptionsActivity.java**

```
package com.pearson.lagp.v3;
. . .
public class OptionsActivity extends BaseGameActivity implements
IOnMenuItemClickListener {
. . .
   private SharedPreferences audioOptions;
   private SharedPreferences.Editor audioEditor;

. . .
   @Override
   public Engine onLoadEngine() {
      mHandler = new Handler();
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      audioOptions = getSharedPreferences("audio", MODE_PRIVATE);
      audioEditor = audioOptions.edit();
      return new Engine (new EngineOptions (true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
            CAMERA_HEIGHT, this.mCamera));
   }
. . .
   @Override
   public void onGamePaused() {
      super.onGamePaused();
      StartActivity.mMusic.pause();
   }
```

```
   @Override
   public void onGameResumed() {
      super.onGameResumed();
      if (audioOptions.getBoolean("musicOn", false))
         StartActivity.mMusic.resume();
      mMainScene.registerEntityModifier(new ScaleAtModifier(0.5f,
         0.0f, 1.0f, CAMERA_WIDTH/2, CAMERA_HEIGHT/2));
      mOptionsMenuScene.registerEntityModifier(
         new ScaleAtModifier(0.5f, 0.0f, 1.0f,
            CAMERA_WIDTH/2, CAMERA_HEIGHT/2));
   }

   @Override
   public boolean onMenuItemClicked (final MenuScene pMenuScene,
      final IMenuItem pMenuItem, final float pMenuItemLocalX,
      final float pMenuItemLocalY) {
      switch(pMenuItem.getID()) {
         case MENU_MUSIC:
         if (audioOptions.getBoolean("musicOn",
            true)) {
            audioEditor.putBoolean("musicOn", false);
            if (StartActivity.mMusic.isPlaying()) {
               StartActivity.mMusic.pause();
         } else {
            audioEditor.putBoolean("musicOn", true);
            StartActivity.mMusic.resume();
         }
         audioEditor.commit();
         createOptionsMenuScene();
         mMainScene.clearChildScene();
         mMainScene.setChildScene(mOptionsMenuScene);
         return true;
         case MENU_EFFECTS:
         if (audioOptions.getBoolean("effectsOn", true)) {
            audioEditor.putBoolean("effectsOn", false);
         } else {
            audioEditor.putBoolean("effectsOn", true);
         }
         audioEditor.commit();
         createOptionsMenuScene();
         mMainScene.clearChildScene();
         mMainScene.setChildScene(mOptionsMenuScene);        return true;
. . .
      }
   }

. . .
}
```

- We again establish a local variable for SharedPreferences and its editor, so we can get and put the contents. We initialize those variables in `onLoadEngine()`.
- We override `onGamePaused()` and `onGameResumed()` so we can shut the music on and off should the game be paused. These methods are also called every time the activity is run. When we resume playing the game, we don't want to turn the music on unless it was playing when the game was paused, so we check the preference values.
- The switch in `onMenuItemClicked()` takes care of changing the options on the menu and taking the appropriate action, such as turning off any music that is playing when the player selects "Music Off." The case is simpler for sound effects, because we assume they don't continue to play. (We don't use `setLoop(true)` anywhere in this game for sound effects.) If you're not familiar with SharedPreferences, note that we have to call the `commit()` method before any changes are propagated to the shared storage.

Listing 11.3 **Changes to Level1Activity.java**

```
package com.pearson.lagp.v3;
. . .
public class Level1Activity extends BaseGameActivity {
. . .

   private Sound mExploSound, mGunshotSound, mWhiffleSound;
   private SharedPreferences audioOptions;

. . .
   @Override
   public Engine onLoadEngine() {
      mHandler = new Handler();
      gen = new Random();
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      audioOptions = getSharedPreferences("audio", MODE_PRIVATE);
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
            CAMERA_HEIGHT),
            this.mCamera).setNeedsSound(true));
   }
   @Override
   public void onLoadResources() {
. . .

      SoundFactory.setAssetBasePath("mfx/");
      try {
```

```
            this.mExploSound = SoundFactory.createSoundFromAsset(
              this.mEngine.getSoundManager(),
            getApplicationContext(), "fireball.ogg");
            this.mGunshotSound = SoundFactory.createSoundFromAsset(
              this.mEngine.getSoundManager(),
            getApplicationContext(), "gunshot.ogg");
            this.mWhiffleSound = SoundFactory.createSoundFromAsset(
              this.mEngine.getSoundManager(),
            getApplicationContext(), "whiffle.ogg");
         } catch (final IOException e) {
            Debug.e(e);
         }

      }

      @Override
      public Scene onLoadScene() {
. . .
         bullet = new Sprite(20.0f, CAMERA_HEIGHT - 40.0f,
          mBulletTextureRegion){
           @Override
            public boolean onAreaTouched(
             final TouchEvent pAreaTouchEvent,
             final float pTouchAreaLocalX,
             final float pTouchAreaLocalY) {
             switch(pAreaTouchEvent.getAction()) {
             case TouchEvent.ACTION_DOWN:
                break;
             case TouchEvent.ACTION_UP:
                fireBullet(pAreaTouchEvent.getX(),
                   pAreaTouchEvent.getY());
                break;
             case TouchEvent.ACTION_MOVE:
                this.setPosition pAreaTouchEvent.getX() -
                   this.getWidth() / 2,
                   pAreaTouchEvent.getY() -
                      this.getHeight() / 2);
                break;
             }
             return true;
            }
         };
. . .
         hatchet = new Sprite(cross.getInitialX() + 40.0f,
            CAMERA_HEIGHT - 40.0f, mHatchetTextureRegion){
            @Override
            public boolean onAreaTouched(
```

```
            final TouchEvent pAreaTouchEvent,
            final float pTouchAreaLocalX,
            final float pTouchAreaLocalY) {
            switch(pAreaTouchEvent.getAction()) {
            case TouchEvent.ACTION_DOWN:
               break;
            case TouchEvent.ACTION_UP:
               throwHatchet(pAreaTouchEvent.getX(),
                  pAreaTouchEvent.getY());
               break;
            case TouchEvent.ACTION_MOVE:
               this.setPosition(pAreaTouchEvent.getX() -
                  this.getWidth() / 2,
                  pAreaTouchEvent.getY() -
                     this.getHeight() / 2);
               break;
            }
            return true;
            }
         };
. . .
      }

      @Override
      public void onGamePaused() {
         super.onGamePaused();
         mGunshotSound.stop();
         mExploSound.stop();
      }

      private void fireBullet(float pX, float pY){
         // rotate bullet sprite 90 degrees cw,
         // move rapidly to right, and play gunshot effect
         bullet.registerEntityModifier(new SequenceEntityModifier (
            new IEntityModifierListener() {
               @Override
               public void onModifierFinished(
                  final IModifier<IEntity>
                  pEntityModifier,
                  final IEntity pEntity) {
                  Level1Activity.this.runOnUiThread(
                    new Runnable() {
                    @Override
                    public void run() {
                     bullet.setVisible(false);
                     bullet.setPosition(0,0);
                        }
```

```
                });
            }
        },
        new RotationModifier(0.5f, 0.0f, 90.0f),
        new MoveXModifier(0.5f, pX, CAMERA_WIDTH),
        new AlphaModifier(0.1f, 1.0f, 0.0f)));

    mHandler.postDelayed(mPlayGunshot, 500);
}

private void throwHatchet(float pX, float pY){
    // hatchet flies to right, rotating about eccentric point
    hatchet.registerEntityModifier(new ParallelEntityModifier (
        new IEntityModifierListener() {
            @Override
            public void onModifierFinished(
                final IModifier<IEntity>
                pEntityModifier,
                final IEntity pEntity) {
                Level1Activity.this.runOnUiThread(
                  new Runnable() {
                  @Override
                  public void run() {
                   hatchet.setVisible(false);
                   hatchet.setPosition(0,0);
                  }
                });
            }
        },
        new RotationAtModifier(5.0f, 0.0f, 5.0f*360.0f,
            20.0f, 20.0f),
        new MoveXModifier(5.0f, pX, CAMERA_WIDTH)));
    playSound(mWhiffleSound);
}

 private Runnable mPlayGunshot = new Runnable() {
     public void run() {
        playSound(mGunshotSound);
     }
 };

 private Runnable mStartVamp = new Runnable() {
     public void run() {
. . .
             asprVamp[i] = new AnimatedSprite(CAMERA_WIDTH - 30.0f,
      startY, mScrumTextureRegion.clone()) {
             @Override
```

```
                public boolean onAreaTouched(
            final TouchEvent pAreaTouchEvent,
            final float pTouchAreaLocalX,
            final float pTouchAreaLocalY) {
                    switch(pAreaTouchEvent.getAction()) {
                    case TouchEvent.ACTION_DOWN:
                       /* Is there a vampire close by? */
               if (. . .
                           playSound(mExploSound);
. . .

     };
. . .
     private void playSound (Sound mSound){
        if (audioOptions.getBoolean("effectsOn", false)) {
           mSound.play();
        }
     }
}
```

- We create variables for the Sound objects, and for SharedPreferences. We don't need the `SharedPreferences.Editor` in this activity, as we're only getting values, not putting any.
- In `onLoadEngine()`, we again run an EngineOptions method, but this time it is `setNeedSound(true)`, so we can access the sound effects methods.
- In `onLoadResources()`, we use the `SoundFactory.createFromAsset()` method to load all three sound effects.
- In `onLoadScene()`, we want to modify the TouchEvent switch statements for both the bullet and the hatchet. The modifications are slightly different:
  - For the bullet, when the user lets it go (after dragging it from the weapons cache), we want it to rotate 90 degrees clockwise and then "fire," by shooting across the screen and playing the gunshot sound effect. We create the method `fireBullet()` to make all action this happen.
  - For the hatchet, when the player lets it go, we want it to immediately start rotating eccentrically and moving to the right of the screen, while playing the whiffle sound effect. We've artfully coordinated the whiffle sound and the animation of the hatchet so there are seven swoops as it crosses the screen. We create the method `throwHatchet()` for this action.
- We've overridden `onGamePause()` to stop any sound effects (although they would stop themselves in a few seconds anyway). We don't really need `onGameResumed()` in this activity, as we never shrink the activity, and there are no "sub" activities.

- The `fireBullet()` method uses a collection of modifiers to carry out the movement and rotation we want. We don't want the sound effect to start until after the rotation, so we post a delayed runnable, `mPlayGunshot()`, timed to match the time taken by the rotation. We could have also used a listener on the rotation modifier, but this approach seemed simpler. After the routine, we set the bullet to invisible and park it at the coordinates (0, 0). The idea is that in the finished game, we'll cause the bullet to come to life at some random later time, so the player can use it again.
- The `throwHatchet()` method is similar to `fireBullet()`, but here the sound effect can take place immediately.
- Finally, we created a `playSound()` method so we don't have to check the sound effects preference throughout the rest of the code.

Summary

This chapter really just scratched the surface of sound in games development. Creating effective background music and convincing sound effects that add to the game play are arts in themselves. If you are building a team of people to put together a game, you should definitely try to attract an expert to the team who can concentrate on just the aural aspects of the game.

The discussion here barely touched on the subject of moods in music. It's a particularly hot topic of research lately, as ventures would like to be able to characterize the mood of recorded music without having people listen to it. For now, you need a person to select or create music that fits your game.

Sound effects are similarly subjective, and difficult to get right. A strong case can be made for paying the modest licensing fees necessary to obtain sound effects that professionals have already refined for you.

The good news is that a wealth of tools to support music and sound effect creation and editing are available, and the capabilities of even the open-source tools are probably beyond what you have time to learn to use. The commercial tools are more sophisticated, (arguably) better supported, and available at a very reasonable cost.

An AndEngine interface is not available that would allow game developers to take advantage of Android features such as the JetPlayer audio engine, which knows about segments of a musical composition and can combine those segments to create dynamic compositions. Making full use of the Android capabilities such as JetPlayer represents a great opportunity to expand upon the work that's been done for AndEngine so far.

Exercises

1. I don't know about you, but I'm feeling a little guilty about enhancing the bullet and the hatchet in this chapter, but leaving the cross with nothing to do except sit there. Why don't you fix that shortcoming by having the cross play a short

tune after it is dragged from the Weapons box? A short chorus of "Onward Christian Soldiers" will do.[1] You don't need to get fancy with the audio—just sing it into your development laptop and record it with Audacity (you can sing it with audacity, too, if you like). You will then need to add music-playing capability to Level1Activity; right now, it is set up to play only sound effects.

2. A MIDI version of the Bach Fugue music is included in `assets/mfx`. Change StartActivity to use this MIDI file. Notice that the tempo is quite a bit faster, but unfortunately the MediaPlayer doesn't give us control of the playback tempo.
3. As the Android developers guide points out, SharedPreferences are useful for things other than user preferences. When StartActivity starts up, create a SharedPreference called "scores" that maintains key–value pairs for each game screen ("WAV" and "Level1" for now). Change the ScoresActivity to just display those values for now.

1. "Onward Christian Soldiers" has an interesting history, by the way. Originally a processional written by Sabine Baring-Gould, the music was later rewritten by none other than Sir Arthur Sullivan, of Gilbert and Sullivan fame. G&S were early proponents of international copyright laws, as many of their operettas were used by others who did not bother to obtain or pay for the rights. It's worth reminding ourselves of the need to secure proper rights to any intellectual property we're using in our games. (For the record, "Onward Christian Soldiers" has long been in the public domain, and the version used in the Exercise Solutions was performed [badly] by yours truly.)

12

Physics

In game development, "physics" is the collection of effects that mimic the physics of the real world. Physics don't enter into every type of game, of course. For example, you generally won't need physics if you are developing a board game. Other games are almost entirely physics based, with missiles of various kinds following natural arcs, crashing into piles of objects that then break or fall down under the influence of gravity.

As we'll see, a very complete physics engine for use in our games is available as an extension to AndEngine. We'll investigate that engine and related tools in this chapter, and then build a little physics-based gamelet to augment our V3 example game. We won't come anywhere close to investigating all that the physics engine can do for us, but we'll make a start at understanding it.

The gamelet we'll end up with is shown in Figure 12.1. The player can launch wooden stakes at the objects in the gamelet by touching, dragging, and letting go of the stakes. The idea is to get the vampire heads to touch the ground.

Figure 12.1 Physics gamelet in V3

Box2D Physics Engine

The physics engine used by AndEngine is called Box2D. The Box2D engine was originally written in C++ by Erin Catto as part of a tutorial on physics engines. It has since been expanded considerably, ported to a number of other languages, and incorporated into many 2D game engines.

We have space in this book to explore only a small part of Box2D, but the information and examples in this chapter should give you a head start in exploring the rest. We'll describe what you need to know to create Box2D-based worlds in AndEngine, but for a more complete discussion, I highly recommend the Box2D Manual, which you should be able to find at the following website:

`http://www.box2d.org/manual.html`

Box2D Concepts

To talk about a physics engine such as Box2D, we first need to define some terms. These terms are more fully defined in the Box2D Manual, in case you want to explore them more deeply.

Units

The physics simulation that Box2D performs is sensitive to the units we use for mass, velocity, and other quantities. Box2D is tuned for MKS (meter–kilogram–second) units, and is designed to simulate "normal-sized" objects as they interact. In this case "normal-sized" means moving objects between 0.1 and 10 meters long, and static objects as much as 50 meters long. We'll see later that AndEngine provides a useful constant, `PIXEL_TO_METER_RATIO_DEFAULT`, that we can use to translate between pixel and physics world coordinates. If you use pixel units directly with Box2D, the resulting simulations won't look realistic.

World

Box2D enables us to build virtual worlds where bodies behave much like the physical objects we encounter in real life. A Box2D world consists of bodies, fixtures, joints, and constraints that add up to a physical simulation of that world.

Rigid Body

Bodies are the basic simulation objects in Box2D. They have only a few characteristics of their own, but can take on more complex physical attributes through association with shapes, fixtures, and constraints, as described later in this section. The primary restriction that Box2D places on the bodies is that they are all rigid bodies, meaning their shapes never become distorted. To quote from the Box2D Manual, a rigid body is

> A chunk of matter that is so strong that the distance between any two bits of matter on the chunk is completely constant. They are hard like a diamond.

Bodies have a type, which can be any of three values:

1. Static: Bodies that are normally fixed in place. The user can move these bodies, but they are not moved as part of the physics simulation. Static bodies act as though they have infinite mass (represented by a mass of zero when we create them), and they collide only with dynamic bodies.
2. Kinematic: Bodies that move only by virtue of their velocity. Kinematic bodies are not part of the physics simulation, and don't respond to forces. They also behave as though they have infinite mass, and collide only with dynamic bodies.
3. Dynamic: The moving, fully simulated bodies in a physics world. Dynamic bodies always have a finite, non-zero mass, and can collide with static, kinematic, and dynamic bodies. If you try to set the mass of a dynamic body to zero, it is automatically reset to 1 kilogram.

Shape

Box2D supports two basic shapes that approximate the shape of two real objects—namely, a circle and a polygon. When we create a Box2D body, we normally associate a shape with it through a fixture (discussed next). When Box2D is simulating the physics of bodies interacting, it uses the shape of the body to determine when collisions occur.

Fixture

Bodies and shapes become associated through a fixture. In addition to giving a body a shape, a fixture provides values for the body's density, elasticity, and friction. The same fixture values are typically used by multiple bodies, so it's easiest to define them once and reuse them as we create bodies.

Constraint

A constraint prevents a body from moving in some dimension. An unconstrained body in 2D really has three degrees of freedom: *x, y,* and rotation. Constraints are used to remove one or more of those degrees of freedom.

Joint

In Box2D, a joint is a constraint that connects two bodies together. Joints can have limits (such as an elbow, which cannot bend backward) and motors, which can drive movement of the joint. Box2D supports revolute, prismatic, distance, pulley, mouse, line, and weld joints, all of which are further explained in the Box2D Manual.

Sensor

Perhaps your game needs a body that detects collisions, but doesn't respond to them. Sensors fill that need, and we'll see that you can declare any body to be a sensor.

Bullet

Box2D normally looks for collisions between dynamic and static bodies by sweeping through the motion of the dynamic bodies between simulation frames (referred to as continuous collision detection). This practice ensures that a dynamic body cannot

tunnel through a static body between frames. For performance reasons, Box2D does not normally do this when detecting collisions between dynamic bodies. If you label a dynamic body as a bullet, however, Box2D will perform continuous collision detection for that body with other dynamic bodies. If your game includes a fast-moving body that will collide with other dynamic bodies, you should label it a bullet.

Running the physics engine on the Android emulator is problematic, because the frame rate is so slow in emulation. If tunneling is a problem in your testing, you can turn on continuous collision detection for everything. If `mPhysicsWorld` is the world you've created, then you would use this line:

```
mPhysicsWorld.setContinuousPhysics(true);
```

Of course, now everything will run even more slowly, as more calculation is being done. It's best to test this capability on a real device.

Setting Up Box2D

The pattern for building a physics world and starting the simulation includes the following steps:

1. Create a PhysicsWorld, using that class's constructor. In the constructor, you can optionally create a gravitational acceleration vector. You can also tell the newly created world to save cycles by not simulating inactive objects ("allow sleeping").
2. Create the static objects in the simulation, which might include a floor, some walls, and a ceiling, to keep objects from flying off out of screen range. We'll do this in two steps:
   - Create the shapes for the objects (usually Sprites).
   - Create the bodies, attaching the bodies to our world, and attaching shapes to the bodies through an appropriate fixture.
3. Attach the shapes to the AndEngine Scene so they will be displayed.
4. Connect the Sprites to the Physics with PhysicsConnectors.
5. Register the Box2D PhysicsWorld as an UpdateHandler for our Scene, so it can update Sprite positions.

We'll put this pattern into practice in the example code toward the end of this chapter.

Building Levels for Physics Games

Physics games often contain many levels. In each level, the player is presented with an arrangement of physical bodies that represents a puzzle. The player resolves that puzzle and gets a score. As game developers, we have several approaches we can use to put together the needed levels:

- Create levels in code: We can always write Dalvik code that will create bodies and position them appropriately in the physical world. You, as the developer,

must mentally do the translation from code to physical space in such a case, and this translation is vulnerable to errors. This approach quickly becomes a lot of work, and I don't recommend it.

- Use an available Box2D level editor: Generous developers have created level editing programs and made them available on the Internet. The editors are typically visual in nature—you create and size bodies by dragging them onto the canvas and assigning them the desired properties. Once the level is arranged the way you want it, the editor can produce an XML file that describes all of the bodies in that physical world. Your game can then read the XML file to dynamically create the level.
- Create your own custom level editor: Many developers prefer to have an editor customized for their particular game. Creating one is not particularly hard to do, but it is a task beyond the scope of this book.

In this book, we will use one of the available Box2D level editors, Bison Kick, which was created by Jacob Schatz. This Flash application runs in your favorite web browser. You can find the beta version of Bison Kick at the following website:

`http://www.jacobschatz.com/bisonkick-beta/`

A typical screen from Bison Kick is shown in Figure 12.2. In the examples provided later in this chapter, we show how to use Bison Kick, and how to create the code needed to load Bison Kick levels into AndEngine.

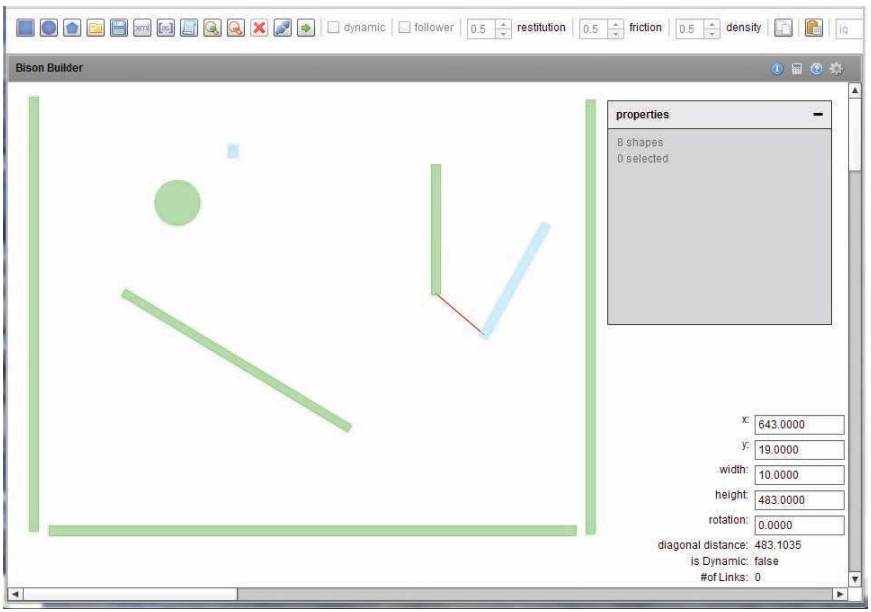


Figure 12.2 Bison Kick level editor

AndEngine and Box2D

Physics is an extension to the basic AndEngine game engine. It must be downloaded and installed separately. AndEngine uses the `libgdx` JNI wrapper library, which provides a very complete interface to the underlying C++ Box2D libraries; Mario Zechner is the lead developer for `libgdx`. AndEngine provides an API layer that makes it easy to use the `libgdx` interfaces in an AndEngine game.

Download and Add the AndEnginePhysicsBox2DExtension

You need to add two libraries to your Eclipse project to use Box2D physics. The first library is the AndEngine extension that contains the `libgdx` wrappers and the AndEngine API extensions. The second is an ARM library that contains the Box2D C++ code. The easiest way to get the libraries is to copy them from the AndEngineExamples project. (If you want to build the AndEngine extension library from scratch, see the sidebar titled "Building AndEngine Extensions from Source.")

andenginephysicsbox2dextension.jar

The AndEngineExamples website (`http://code.google.com/p/andengine-examples/`) includes several physics-based activities. If you look in the `lib` folder of the project, you will see the `.jar` file for each of the AndEngine extensions, including `andenginephysicsbox2dextension.jar`. The AndEngine team (particularly Nicolas) is very good about keeping these `.jar` files up-to-date.

1. Download the physics extension `.jar` file to your development system.
2. Import the `.jar` file into your project's `lib` folder.
3. In the Eclipse Project Explorer, right-click on the `.jar` file, and select Build Path > Add to Build Path.

armeabi

In the same AndEngineExamples project, there is another folder called `libs`. In this folder you will find a subfolder, `armeabi`.

1. Download the shared library `libandenginephysicsbox2dextension.so` from the `armeabi` subfolder to your development system.
2. Create a new first-level folder under your Eclipse project named `libs` (right-click on project in Project Explorer, select New > Folder). Create a subfolder under `libs` called `armeabi`.
3. Import the shared library into the `armeabi` subfolder.
4. You do not have to add the shared library to the build path—it'll get sucked in automatically.

Building AndEngine Extensions from Source

The `.jar` files in AndEngineExamples are kept very up-to-date by the AndEngine maintainers, but sometimes you may want to build your own from scratch. After all, one of

the advantages of open-source programs is that you can change the sources if necessary and then rebuild the `.jar` file for use in your project.

The steps are the same, no matter which extension you are building (or for that matter, if you are building AndEngine itself):

1. Use your favorite form of Mercurial (usually TortoiseHg for Windows, or the command-line interface from Linux or Macintosh OS X) to download the sources from the appropriate `code.google.com` site. Separate sites are maintained for AndEngine and for each of the extensions. Mercurial will create a clone repository on your development machine.
2. In Eclipse, create a new Android project. On the New Android Project dialog, select Create Project from Existing Source, and then navigate the "Location:" box to the repository created by Mercurial. When you are done, click Finish. Eclipse will create the project with a long name that reflects the extension's package name.
3. Right-click on the project name in the Project Explorer, and select Export... from the pop-up menu. In the resulting Export dialog, select Java > JAR file, and click the Next button. The resulting JAR Export dialog is shown in Figure 12.3.

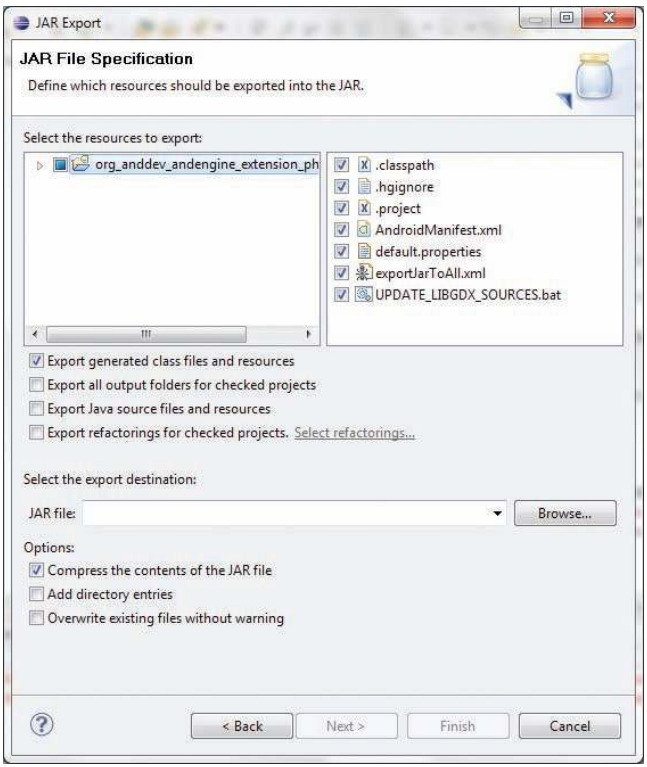


Figure 12.3 JAR Export dialog box

4. You *must* deselect (uncheck) all of the boxes in the right-hand pane under "Select the resources to export." In addition, you must indicate a destination (complete filename, not just the directory) for the box under "Select the export destination."
5. Click Finish, and Eclipse will create the `.jar` file for you.

Box2D APIs

The Box2D physics extension adds 16 classes to AndEngine. It also adds the Box2D classes themselves, which are part of the `libgdx` library. For most game development, we need to know about only three classes:

- PhysicsWorld: which represents Box2D worlds
- PhysicsFactory: which helps us generate Box2D bodies and fixtures
- PhysicsConnector: which connects Box2D bodies with AndEngine entities

Creating Worlds: PhysicsWorld

We use the PhysicsWorld class to create the physics world where our game action takes place. After we create it, we add it to our AndEngine Scene so it will be displayed. There are two constructors:

PhysicsWorld(final Vector2 pGravity, final boolean pAllowSleep)
PhysicsWorld(final Vector2 pGravity, final boolean pAllowSleep, final int pVelocityIterations, final int pPositionIterations)

The parameters are not difficult to figure out:

`Vector2 pGravity`: This parameter sets the gravity acceleration vector for the world. This is the first time we've seen the Vector2 type, which is used by Box2D to represent two-dimensional vectors. If you want your world to include gravity, you'll normally set this vector as follows:

```
new Vector2(0, SensorManager.GRAVITY_EARTH)
```

Here we've taken advantage of the gravity constant that Android makes available to us, which is conveniently in MKS units. If you don't want gravity, you just pass the following parameters:

```
new Vector2(0, 0)
```

Android provides gravity constants for the moon, the sun, all of the planets (even Pluto), and `GRAVITY_DEATH_STAR_I`. I'm not sure how they figured that last one out, but it's there.

- `boolean pAllowSleep`: If this value is `true`, Box2D will skip the simulation calculations for any body that is at rest. That's generally a good thing to do, as it saves simulation cycles and improves performance.

- `int pVelocityIterations`: This parameter sets the number of iterations Box2D will go through when it does its velocity calculations. If you use the first constructor, `pVelocityInterations` defaults to 8. You can improve performance (at the cost of accuracy) by setting this value lower than 8.
- `int pPositionIterations`: Similarly, this parameter sets the number of iterations when calculating position. It also defaults to 8.

We'll also use PhysicsWorld methods to create and manage joints between bodies. Those methods are covered in the Joints section later in this chapter. Many other methods in the class can also be used to set and read other aspects of our model world. If you're interested in them, take a look at the `PhysicsWorld.java` sources at this site:

`http://code.google.com/p/andenginephysicsbox2dextension/source/browse/src/org/anddev/andengine/extension/physics/box2d/PhysicsWorld.java`

Creating Bodies: PhysicsFactory

Now that we have a world, we can start creating bodies, shapes, fixtures, and joints to create the objects in our model. AndEngine provides the PhysicsFactory singleton class to facilitate these tasks. We don't have to instantiate PhysicsFactory; instead, AndEngine creates one for us. We use it much in the same way that we have been using TextureRegionFactory, SoundFactory, and other classes. PhysicsFactory defines the following methods to create physics bodies:

Body createBoxBody(final PhysicsWorld pPhysicsWorld, final IShape pIShape, final BodyType pBodyType, final FixtureDef pFixtureDef)

Body createCircleBody(final PhysicsWorld pPhysicsWorld, final IShape pIShape, final BodyType pBodyType, final FixtureDef pFixtureDef)

Body createLineBody(final PhysicsWorld pPhysicsWorld, final Line pLine, final FixtureDef pFixtureDef)

Body createPolygonBody(final PhysicsWorld pPhysicsWorld, final IShape pIShape, final Vector2[] pVertices, final BodyType pBodyType, final FixtureDef pFixtureDef)

Body createTrianglulatedBody(final PhysicsWorld pPhysicsWorld, final IShape pIShape, final List<Vector2> pTriangleVertices, final BodyType pBodyType, final FixtureDef pFixtureDef)

Each of these methods also has a variation with an additional parameter (the last parameter), `float pPixelToMeterRatio`, in case you need to set your own pixel scaling, different from the default value (32.0f). The other parameters are as follows:

- `PhysicsWorld pPhysicsWorld`: The world that the body will be in. Usually it will be the world we just created with PhysicsWorld.

- `IShape pIShape` or `Line pLine`: The shape that we want to attach to the body. Shapes are just geometric shapes—either Sprites or geometric shapes created with something like the AndEngine Rectangle method.
- `BodyType pBodyType`: The parameter in which we tell Box2D which type the body is:
  - `BodyType.StaticBody`
  - `BodyType.KinematicBody`
  - `BodyType.DynamicBody`
- `FixtureDef pFixtureDef`: The fixture we want to use with the body. Fixture creation is described in the next section.
- `TrianglulatedBody`: Let's hope the spelling of this method name is fixed by the time you use AndEngine. Here we pass a list of triangle vertices as Vector2 variables: `List<Vector2> pTriangleVertices`.

Creating Fixtures: PhysicsFactory

The PhysicsFactory class also includes some methods that create fixtures for us:

FixtureDef createFixtureDef(final float pDensity, final float pElasticity, final float pFriction, final boolean pSensor)
FixtureDef createFixtureDef(final float pDensity, final float pElasticity, final float pFriction, final boolean pSensor, final short pCategoryBits, final short pMaskBits, final short pGroupIndex)

The parameters to the first method are obvious except perhaps for `boolean pSensor`, which is optional. If this parameter is included and `true`, the fixture is a sensor as defined earlier in the "Box2D Concepts" section.

PhysicsConnector

The PhysicsConnector class allows us to bridge the divide between AndEngine entities (such as Sprites) and Box2D bodies. That way we can use all of the AndEngine goodies, such as Modifiers, on the combined object, and Box2D will do the physics simulation for us.

There are four constructors for PhysicsConnector:

PhysicsConnector(final IShape pShape, final Body pBody)
PhysicsConnector(final IShape pShape, final Body pBody, final float pPixelToMeterRatio)
PhysicsConnector(final IShape pShape, final Body pBody, final boolean pUdatePosition, final boolean pUpdateRotation)
PhysicsConnector(final IShape pShape, final Body pBody, final boolean pUdatePosition, final boolean pUpdateRotation, final float pPixelToMeterRatio)

The first two parameters in each case are just the shape (Sprite) and the body that we want to connect. Note that creating and registering this connection is required, even though you might have used the Sprite as the shape when you created the body.

The `pPixelToMeterRatio` is optional; it has the same meaning as for PhysicsFactory methods, and the same default (32.0f). The two `boolean` parameters specify whether Box2D will update the Sprite's position and rotation as it simulates the physical world. Usually we want both of those characteristics updated, so the defaults are `true`.

Simple Physics Example

If this is the first time you've encountered Box2D, the material presented so far probably seems overwhelming. Matters will become clearer if we look at a simple physics game that demonstrates how to put everything together. Figure 12.4 shows an activity we'll call Demolition; it is a simplified version of one of the AndEngineExamples activities. As you touch the screen, objects are added. As you tilt the device, the objects fall to the lowest corner of the screen. They bounce off one another as they collide. Listing 12.1 provides the code for the activity.

Figure 12.4 Demolition screenshot

Listing 12.1 **Demolition: A Simple Physics Application**

```
package com.pearson.lagp.demolition;

+imports . . .

public class Demolition extends BaseGameActivity implements
   IAccelerometerListener, IOnSceneTouchListener {

   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;

   private static final FixtureDef FIXTURE_DEF =
      PhysicsFactory.createFixtureDef(1, 0.5f, 0.5f);
   private static final int MAX_BODIES = 50;

   // ===========================================================
   // Fields
   // ===========================================================

   private Texture mTexture;
   private TextureRegion mTStoneTextureRegion;
   private TextureRegion mMatHeadTextureRegion;
   private PhysicsWorld mPhysicsWorld;
   private int mBodyCount = 0;

   @Override
   public Engine onLoadEngine() {
      Toast.makeText(this, "Touch the screen to add objects.",
         Toast.LENGTH_LONG).show();
      final Camera camera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      final EngineOptions engineOptions = new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), camera);
      engineOptions.getTouchOptions().setRunOnUpdateThread(true);
      return new Engine(engineOptions);
   }

   @Override
   public void onLoadResources() {
      /* Textures. */
```

```
    mTexture = new Texture(64, 128,
        TextureOptions.BILINEAR_PREMULTIPLYALPHA);

    /* TextureRegions. */
    mTStoneTextureRegion =
        TextureRegionFactory.createFromAsset(mTexture,
            getApplicationContext(), "tombstone.png",
            0, 0); // 50x50
    mMatHeadTextureRegion =
        TextureRegionFactory.createFromAsset(mTexture,
            getApplicationContext(), "mathead.png", 0, 50); // 32x32
    this.mEngine.getTextureManager().loadTexture(mTexture);
    this.enableAccelerometerSensor(this);
}

@Override
public Scene onLoadScene() {
    this.mEngine.registerUpdateHandler(new FPSLogger());

    final Scene scene = new Scene(2);
    scene.setBackground(new ColorBackground(0, 0, 0));
    scene.setOnSceneTouchListener(this);

    this.mPhysicsWorld = new PhysicsWorld(new Vector2(0, 0),
            true);
    final Shape ground = new Rectangle(0, CAMERA_HEIGHT - 2,
        CAMERA_WIDTH, 2);
    final Shape roof = new Rectangle(0, 0, CAMERA_WIDTH, 2);
    final Shape left = new Rectangle(0, 0, 2, CAMERA_HEIGHT);
    final Shape right = new Rectangle(CAMERA_WIDTH - 2, 0, 2,
        CAMERA_HEIGHT);

    final FixtureDef wallFixtureDef =
        PhysicsFactory.createFixtureDef(0, 0.5f, 0.5f);
    PhysicsFactory.createBoxBody(mPhysicsWorld, ground,
        BodyType.StaticBody, wallFixtureDef);
    PhysicsFactory.createBoxBody(mPhysicsWorld, roof,
        BodyType.StaticBody, wallFixtureDef);
    PhysicsFactory.createBoxBody(mPhysicsWorld, left,
        BodyType.StaticBody, wallFixtureDef);
    PhysicsFactory.createBoxBody(mPhysicsWorld, right,
        BodyType.StaticBody, wallFixtureDef);

    scene.getFirstChild().attachChild(ground);
    scene.getFirstChild().attachChild(roof);
    scene.getFirstChild().attachChild(left);
    scene.getFirstChild().attachChild(right);
```

```
        scene.registerUpdateHandler(mPhysicsWorld);

        return scene;
    }

    @Override
    public void onLoadComplete() {
    }

    @Override
    public boolean onSceneTouchEvent(final Scene pScene,
            final TouchEvent pSceneTouchEvent) {
        if(mPhysicsWorld != null) {
            if(pSceneTouchEvent.isActionDown()) {
                addBody(pSceneTouchEvent.getX(),
                    pSceneTouchEvent.getY());
                return true;
            }
        }
        return false;
    }

    @Override
    public void onAccelerometerChanged(
        final AccelerometerData pAccelerometerData) {
        final Vector2 gravity =
            Vector2Pool.obtain(pAccelerometerData.getY(),
                pAccelerometerData.getX());
        mPhysicsWorld.setGravity(gravity);
        Vector2Pool.recycle(gravity);
    }

    // ===========================================================
    // Methods
    // ===========================================================

    private void addBody(final float pX, final float pY) {
        final Scene scene = this.mEngine.getScene();
        if (mBodyCount >= MAX_BODIES) return;
        mBodyCount++;

        final Sprite matSprite;
        final Body body;

        if(mBodyCount % 2 == 0) {
            matSprite = new Sprite(pX, pY,
                mTStoneTextureRegion);
            body = PhysicsFactory.createBoxBody(mPhysicsWorld,
```

```
                matSprite, BodyType.DynamicBody,
                FIXTURE_DEF);
        } else {
            matSprite = new Sprite(pX, pY,
                mMatHeadTextureRegion);
            body = PhysicsFactory.createCircleBody(
                mPhysicsWorld, matSprite,
                BodyType.DynamicBody, FIXTURE_DEF);
        }

        scene.getLastChild().attachChild(matSprite);
        mPhysicsWorld.registerPhysicsConnector(
            new PhysicsConnector(matSprite, body, true, true));
    }

    // ===========================================================
    // Inner and Anonymous Classes
    // ===========================================================
}
```

Here are some key things to note in the code:

- In the Constants section, we define the fixture that we will use for the dynamic bodies in the application. We'll define a different fixture for the static walls, floor, and ceiling.
- The `onLoadEngine()` and `onLoadResources()` methods should look very familiar by now. At the end of `onLoadResources()`, we enable the Accelerometer and tell it that the listener method `onAccelerometerChanged()` is in this class.
- The `onLoadScene()` method includes a few twists we haven't seen yet:
  - We create the PhysicsWorld and pass it a gravity vector of [0, 0] for now. We'll update it as the Accelerometer tells us how our device is tilted. The application can skip simulation calculations for bodies at rest.
  - We define the walls, floor. and ceiling shapes as slim rectangles. We create a fixture to use for these static bodies; the difference between this fixture and the one we created earlier is that we set density = 0 here, which is what Box2D would do anyway (static bodies have infinite mass, so they are not accelerated by forces on them). The walls, floor, and ceiling don't have to be on screen, but we will keep them where we can see them for our examples.
  - We create box bodies for the walls, floor, and ceiling and attach them to the Scene, so they'll be displayed.
  - We register our PhysicsWorld as an update handler for our Scene, so it can update the bodies with new positions each time the Scene is rendered.

- We override `onSceneTouchEvent()` so we can add a body every time the screen is touched.
- We override `onAccelerometerChanged()` to reset the gravity vector each time we get a new reading on the tilt of our Android device. We recycle the Vector2 variables that we use, so we don't end up doing a lot of garbage collection.
- The `addBody()` method handles that task, alternating between Mat Heads and Tombstones. In addition to adding the new Sprite to our scene, we create a PhysicsConnector that connects the Sprite to the new physics body we created.

Level Loading

In our earlier discussion of levels, we noted that you would probably want to load physics game levels from an XML file created by a level editor. AndEngine provides some classes that help you build the level loading code that you need to parse the XML file and use it to build Box2D objects.

SAX Parsing

AndEngine uses a SAX (Simple API for XML) parser to read XML files. The parser comes as part of Android, so it only makes sense to use it. If you're not familiar with SAX, you can get a quick overview with the following document:

`http://www.saxproject.org/quickstart.html`

If you want to delve more deeply into SAX, there is a lot of documentation available on that same site and around the Internet. For our purposes, we need just a few basic concepts and methods:

- SAX parsers are event-driven sequential access parsers that take a stream of XML data and generate callbacks to your code when they find XML elements. The callbacks are made in order as the parser reads the stream.
- For each element, the parser reports which XML element it found, the attributes of that element, and the values assigned to those attributes.
- The parser also indicates when it reaches the end of an element.

We will access the SAX library through two sets of APIs provided by AndEngine:

- LevelLoader, a class that handles the details of setting up the parser and forwarding the callback data
- SAX Utilities, which simplify the retrieval of attributes from the SAX callbacks

LevelLoader

LevelLoader is a class that's analogous to the TMXLoader class we used to load tile maps in Chapter 9. Both classes set up the SAX parser to read in an XML data file,

but there is one big difference between them. TMXLoader does all of the parsing for us, because it knows exactly what is in a TMX file. LevelLoader is designed to work with a variety of level editors, which makes it more flexible, but it also means that some of the parsing is left up to the user. In the Irate Villagers example that follows this section, we will show how to write the parsing code for the XML schema used by Bison Brick.

Creating an instance of LevelLoader couldn't be simpler. There are no parameters to the constructor:

```
LevelLoader()
```

Another constructor accepts an asset path as its only parameter, or you can set the `assetPath` (as we did with Textures, Sounds, and TMX files) by using a LevelLoader method:

```
void setAssetBasePath(final String pAssetBasePath)
```

We can register XML entities with a LevelLoader instance by using one of the `registerEntityLoader()` methods:

```
void registerEntityLoader(final String pEntityName, final IEntityLoader
        pEntityLoader)
void registerEntityLoader(final String pEntityNames, final IEntityLoader
        pEntityLoader)
```

The only difference between the two methods is whether we pass in a single entity name or an array of entity names. For the second parameter, in either case we will pass an inner class that overrides the `onLoadEntity()` method.

The other LevelLoader method we will use is the one that actually connects the SAX parser to the XML stream:

```
void loadLevelFromAsset(final Context pContext, final String pAssetPath)
```

The parameters for this method are just the Application Context and the path to the XML file from the asset base path.

SAXUtils

The SAXUtils class provides methods that make it easier to extract attribute values from the SAX callbacks. The methods we use to extract attribute values are all of the following form:

```
xxx SAXUtils.getXXXAttribute(final Attributes pAttributes, final String pAttribute-
        Name, final xxx pDefaultValue)
```

The actual method names substitute a class type (Double, Float, Long, Int, Short, Byte, String, Boolean) for the `XXX`, and the related primitive type for the `xxx` return and `pDefaultValue`. If that sounds confusing, take a look at the calls in Listing 12.2, and the usage should be clearer. The first parameter, `pAttributes`, is the list of attributes passed to the `onLoadEntity()` method by SAX.

In addition, another set of `getXXXAttribute` methods will throw an exception if the attribute is not found:

xxx SAXUtils.getXXXAttributeOrThrow(final Attributes pAttributes, final String pAttributeName, final xxx pDefaultValue)

In all other respects, these methods are identical to the methods that don't throw an exception.

Still other SAXUtils methods append values to the list of attributes, but we don't need them for what we want to do.

Using LevelLoader

To use LevelLoader, we first define and register all the XML entities that we want SAX to find, and then connect SAX to the XML input stream (usually a file). As we register each XML entity, we override the `onLoadEntity()` method of the unnamed IEntity class that is a parameter. In `onLoadEntity()`, we use SAXUtils methods to examine the attributes for the entity and build appropriate Box2D objects. LevelLoader takes care of all the underlying SAX work for us. Listing 12.2 is a short example of using LevelLoader to extract the entities in a simple XML file. The XML file is shown first, followed by the AndEngine code.

Listing 12.2 **Using LevelLoader to Parse a Simple XML file**

```
. . .XML file. . .
<level>
   <entity x="40.0" y="100.0" width="20.0" height="10.0"
      isDynamic="true">
    </entity>
</level>

. . .AndEngine game code. . . .
final LevelLoader levelLoader = new LevelLoader();
levelLoader.registerEntityLoader("level", new IEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes) {
      mLevels++;
   }
});

levelLoader.registerEntityLoader("entity", new IEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes) {
      final float x = SAXUtils.getFloatAttributeOrThrow(
         pAttributes, "x");
      final float y = SAXUtils.getFloatAttributeOrThrow(
         pAttributes, "y");
```

```
final float width = SAXUtils.getFloatAttributeOrThrow(
   pAttributes, "width");
final float height = SAXUtils.getFloatAttributeOrThrow(
   pAttributes, "height");
final boolean isDyn = SAXUtils.getBooleanAttributeOrThrow(
   pAttributes, "isDynamic");
if (isDynamic) bodyType = BodyType.DynamicBody;
   . . .

try {
   levelLoader.loadLevelFromAsset(this, "ex");
} catch (final IOException e) {
   Debug.e(e);
}
```

Irate Villagers: A Physics Gamelet for V3

To demonstrate game physics, collisions, and level loading, let's add another gamelet to V3. In Chapter 9, we used the Whack-A-Vampire gamelet to show tile maps, and now we'll add a typical physics gamelet called Irate Villagers (IV—somehow appropriate for a vampire game).

The idea is that the villagers of the town where Miss B's is located have surrounded a vampire, Mad Mat (remember Mat from the discussion of Sprites in Chapter 5?), and his clones, who have gathered in a heap of rubble—boards, glass, and rocks. The villagers are using a slingshot to hurl wooden stakes at the vampires. In this case, the wooden stakes don't actually have to pierce the vampires to kill them; instead, the vampires expire when they touch the ground. We'll give the villagers all the stakes they want to hurl. The first illustration in this chapter, Figure 12.1, is a screenshot of the IV gamelet.

Implementing IV

We need to add a number of things to V3 to implement IV:

- We need artwork for the vampires, the wooden stakes, the wooden barriers, and the pieces of glass and stone that will be part of the vampires' pile of rubble.
- `OptionsActivity.java` must change so we can select IV as an option from the menu. Eventually we'll chain all these gamelets together, but for now we just want to run them.
- A new Activity, `IVActivity.java`, will create the IV world and facilitate the simulation. IV must perform the following tasks:
  - Create our physics world and populate it with a level that we create using the Bison Kick level editor

- Handle updates from Box2D
- Implement the slingshot that is used to launch wooden stakes at the vampires

Creating a Level

We'll use the Bison Kick level editor to create a level for our gamelet. Figure 12.5 shows the Bison Kick screen as it opens.

The toolbar just above the drawing area gives us the tools with which to draw our level. Figure 12.6 is a close-up of the left end of that toolbar as shown in Figure 12.5.

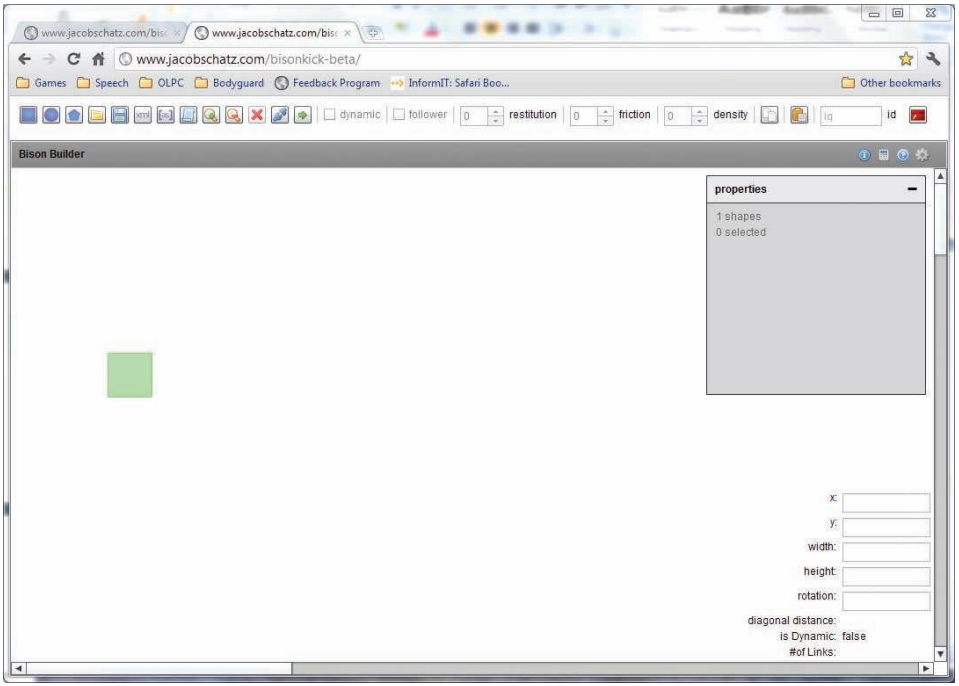


Figure 12.5 Bison Kick opening screen

Figure 12.6 Bison Kick toolbar, left end

Here's a quick rundown on the icons shown in Figure 12.6:

- Clicking on one of the first three icons creates a new rectangle, circle, or polygon, respectively. For our gamelet, we will use only circles and rectangles. You can also create objects by typing on the keyboard—that is, by entering r for a rectangle, and c for a circle.
- You can save your level at any time. The fourth icon lets you open a saved level, and the fifth lets you save the current level.
- The sixth icon (containing the text "xml") shows the XML version of what is currently in the drawing area. We'll use this icon to extract the XML file that we will include in our gamelet.
- We can ignore the next two icons, which are used when creating levels for Box2D and Flash (ActionScript).
- The magnifying glass icons allow us to zoom in and out.
- The red X icon deletes the currently selected object from the drawing area.
- The connector icon lets us create joints between bodies. Unfortunately, the joints are not reflected in the XML descriptions, but having them here makes it easier to visualize what happens when we run the level.
- The right arrow icon starts the simulation for the drawing area. This simulation continues to run until you click on the X button in the upper-right corner of the simulation screen.

Continuing toward the right in that same toolbar (refer back to Figure 12.5):

- If you check the "dynamic" check box, the current object will be marked as a dynamic body. Otherwise, it is assumed to be static (there is no provision in this version of Bison Kick for kinematic bodies).
- A similar check box is used to mark the body as a "follower." The idea behind a follower body is that the camera will attempt to follow this body as the simulation plays out. We haven't implemented followers for AndEngine.
- The next three spinners, labeled "restitution," "friction," and "density," allow you to set the fixture parameters for the current object. Restitution is equivalent to elasticity.
- The next two icons allow you to copy and paste objects.
- The "id" edit box allows us to enter an identifier for the current object. We will use this feature to specify the material the object is made from—either "wood," "stone," or "glass." Stone is unbreakable, wood will break if hit hard enough, and glass will shatter easily. We'll use a circle with the id "vamp" for a vampire head.
- The rightmost button is used by Flash developers.

We've used the tools in Bison Kick to put together the level shown in Figure 12.7.

We click on the XML icon to bring up a listing of the XML output for the level, as shown in Figure 12.8. You must select the text in the window and then press the key combination Ctrl-C to actually copy it. You can then paste the copy into your favorite editor to actually create the XML file.

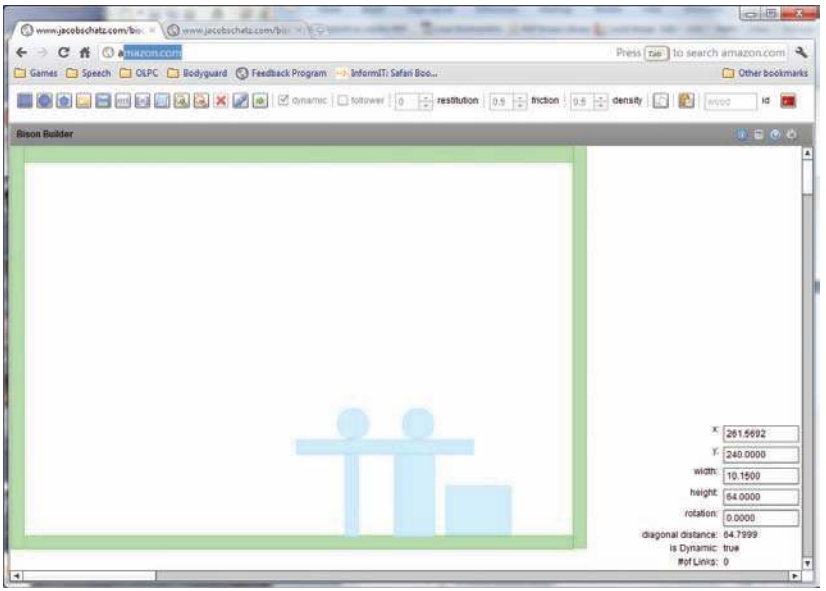


Figure 12.7 Bison Kick showing Level 1

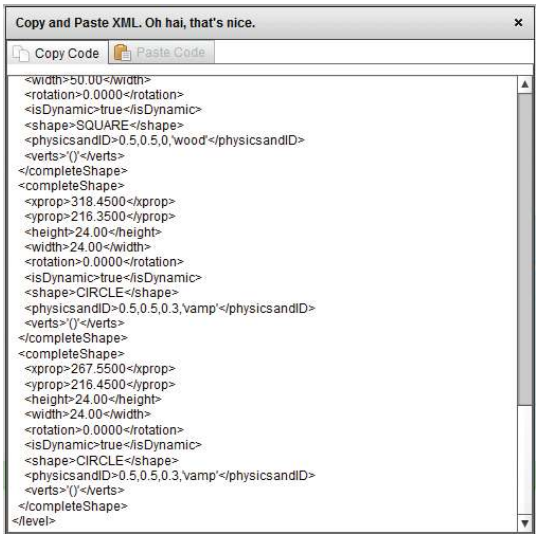


Figure 12.8 Bison Kick XML copy window

Listing 12.3 shows the beginning of a listing of the resulting file.

Listing 12.3 **XML Version of IV Level 1**

```
<level>
  <completeShape>
    <xprop>15.0000</xprop>
    <yprop>272.5000</yprop>
    <height>525.00</height>
    <width>10.00</width>
    <rotation>0.0000</rotation>
    <isDynamic>false</isDynamic>
    <shape>SQUARE</shape>
    <physicsandID>0.5,0.5,0.5,'stone'</physicsandID>
    <verts>'()'</verts>
  </completeShape>
  <completeShape>
    <xprop>423.0000</xprop>
    <yprop>533.0000</yprop>
    <height>10.00</height>
    <width>802.00</width>
    <rotation>0.0000</rotation>
    <isDynamic>false</isDynamic>
    <shape>SQUARE</shape>
    <physicsandID>0.5,0.5,0.5,'stone'</physicsandID>
    <verts>'()'</verts>
  </completeShape>
. . .
</level>
```

Notice that the XML file consists solely of entities. The LevelLoader class cannot parse this file properly, as it is set up to read entities with attributes, like those found in the equivalent XML file in Listing 12.4.

Listing 12.4 **XML Version of IV Level 1 with Attributes**

```
<level>
  <completeShape>
    xprop=15.0000
    yprop=272.5000
    height=525.00
    width=10.00
    rotation=0.0000
    isDynamic='false'
    shape=SQUARE
    physicsandID='0.5,0.5,0.5,stone'
    verts='()'
  </completeShape>
```

```
  <completeShape>
    xprop=423.0000
    yprop=533.0000
    height=10.00
    width=802.00
    rotation=0.0000
    isDynamic='false'
    shape=SQUARE
    physicsandID='0.5,0.5,0.5,stone'
    verts='()'
  </completeShape>
. . .
</level>
```

Unfortunately, LevelLoader doesn't give us a way to retrieve the non-attribute contents of an XML entity. No problem: We've created a special loader for our Bison Kick files, called BKLoader. We use it much the same way we used LevelLoader, but in our `onLoadEntity()` method, we'll also return the text contents of the entity loaded. We also need one more change: We don't want `onLoadEntity()` to be called until we reach the end of the entity. That way, for example, we'll know that all the parameters are loaded when `onLoadEntity()` is called for the `<completeShape>` tag.

BKLoader extends LevelLoader and creates a new Interface, IBKEntityLoader, which in turn extends IEntityLoader. We've added and overridden the necessary methods so that BKLoader can give us the entity text. We've also created a BKParser class that extends LevelParser, again with the needed methods and overrides. If you are curious, the complete source is included with the download for this chapter.

Creating IVActivity.java

We'll skip looking at the changes to `OptionsActivity.java`. They are much the same as the additions we made previously to add the Whack-A-Vampire gamelet in Chapter 9. The sources are there in the download if you want to take a look at them.

`IVActivity.java` is more interesting. The code is found in Listing 12.5, parts 1 to 7, along with explanations of the key changes.

Listing 12.5 **IVActivity.java: Part 1**

```
package com.pearson.lagp.v3;

+imports. . .

public class IVActivity extends BaseGameActivity implements
IOnSceneTouchListener, BKConstants {
   // ===========================================================
   // Constants
   // ===========================================================
```

```
private static final int CAMERA_WIDTH = 480;
private static final int CAMERA_HEIGHT = 320;

private static final FixtureDef FIXTURE_DEF =
   PhysicsFactory.createFixtureDef(1, 0.5f, 0.5f);

// ===========================================================
// Fields
// ===========================================================

private BuildableTexture mTexture;

private TextureRegion mStakeTextureRegion;
private TextureRegion mGlassTextureRegion;
private TextureRegion mWoodTextureRegion;
private TextureRegion mStoneTextureRegion;
private TextureRegion mMatHeadTextureRegion;
private Sprite stakesprite;
private Scene mScene;

private PhysicsWorld mPhysicsWorld;

private boolean isStakeSpawning = false;
private float stakeX, stakeY;
private float velX, velY;
private Line stakeLine;

private Vector2 gravity;
private Body stake;

private float mX, mY;
private float mWidth, mHeight;
private float mRotation;
private boolean mIsDynamic;
private BodyType mBodyType;
private String mShape;
private String mPhysicsAndID;
private float mDensity;
private float mFriction;
private float mElasticity;
private String mID;
private String mVerts;

private float PtoM =
   PhysicsConstants.PIXEL_TO_METER_RATIO_DEFAULT;
```

Not much new in Part 1—just initializing the needed variables and one constant.

Listing 12.5 **IVActivity.java: Part 2**

```
@Override
public Engine onLoadEngine() {
   final Camera camera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   final EngineOptions engineOptions = new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
      CAMERA_HEIGHT), camera).setNeedsSound(true);
   engineOptions.getTouchOptions().setRunOnUpdateThread(true);
   return new Engine(engineOptions);
}

@Override
public void onLoadResources() {
   /* Textures. */
   this.mTexture = new BuildableTexture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   TextureRegionFactory.setAssetBasePath("gfx/IV/");

   /* TextureRegions. */
   this.mStakeTextureRegion =
      TextureRegionFactory.createFromAsset(this.mTexture,
         getApplicationContext(), "stake40.png");
   this.mGlassTextureRegion =
      TextureRegionFactory.createFromAsset(this.mTexture,
         getApplicationContext(), "glass.png");
   this.mWoodTextureRegion =
      TextureRegionFactory.createFromAsset(this.mTexture,
         getApplicationContext(), "wood.png");
   this.mStoneTextureRegion =
      TextureRegionFactory.createFromAsset(this.mTexture,
         getApplicationContext(), "stone.png");
   this.mMatHeadTextureRegion =
      TextureRegionFactory.createFromAsset(this.mTexture,
         getApplicationContext(), "mathead.png");
      try   {
         mTexture.build(
         new BlackPawnTextureBuilder(2));
         } catch (final TextureSourcePackingException e){
         Log.d("V3",
         "Sprites won't fit in mTexture");
         }
   this.mEngine.getTextureManager().loadTexture(this.mTexture);

}
```

The code in Part 2 should look fairly familiar by now. We've created a BuildableTexture and loaded in the textures we'll use for the different bodies.

Listing 12.5 **IVActivity.java: Part 3**

```
@Override
public Scene onLoadScene() {
   this.mEngine.registerUpdateHandler(new FPSLogger());

   mScene = new Scene(2);
   mScene.setOnSceneTouchListener(this);

   /* Center the camera. */
   final int centerX = CAMERA_WIDTH / 2;
   final int centerY = CAMERA_HEIGHT / 2;
   mScene.setBackground(new ColorBackground(0.0f, 0.0f,
      0.0f));
   mPhysicsWorld = new PhysicsWorld(new Vector2(0,
      SensorManager.GRAVITY_EARTH), false);

   final BKLoader bkLoader = new BKLoader();
   bkLoader.setAssetBasePath("level/iv/");
```

In Part 3, things start to get more interesting. We've created a BKLoader and we'll start registering entities with it. Every time BKLoader finds one of the registered entities, it will call the `onLoadEntity()` method of the IBKEntityLoader we registered with it. The tag definitions (`TAG_LEVEL`, `TAG_BODY`, and so on) are found in the file `BKConstants.java`.

Listing 12.5 **IVActivity.java: Part 4**

```
   bkLoader.registerEntityLoader(TAG_LEVEL,
      new IBKEntityLoader() {
      @Override
      public void onLoadEntity(final String pEntityName,
         final Attributes pAttributes,
         final String pValue) {
      }
      public void onLoadEntity(final String pEntityName,
         final Attributes pAttributes) {
      }
   });

   bkLoader.registerEntityLoader(TAG_BODY,
      new IBKEntityLoader() {
      @Override
      public void onLoadEntity(final String pEntityName,
         final Attributes pAttributes,
```

```
            final String pValue) {
            if(mShape.equals(TAG_SHAPE_VALUE_SQUARE)) {
               final TextureRegion mTR = selectTexture(mID);
               final Sprite bodyShape = new Sprite(mX - mTR.getWidth()/2,
                  mY - mTR.getHeight()/2, mTR);
               bodyShape.setScaleX(mWidth/mTR.getWidth());
               bodyShape.setScaleY(mHeight/mTR.getHeight());
               if (mRotation != 0.0f) {
                  bodyShape.setRotation(mRotation);
               }
               final Body mBody =
               PhysicsFactory.createBoxBody(mPhysicsWorld,
         bodyShape, mBodyType,
                     PhysicsFactory.createFixtureDef(mDensity,
            mElasticity, mFriction));
               mScene.getLastChild().attachChild(bodyShape);
               mPhysicsWorld.registerPhysicsConnector(
         new PhysicsConnector(bodyShape, mBody, true,
            true));
            } else if(mShape.equals(TAG_SHAPE_VALUE_CIRCLE)) {
               final TextureRegion mTR = mMatHeadTextureRegion;
               Sprite bodyShape = new Sprite(mX - mTR.getWidth()/2,
         mY - mTR.getHeight()/2, mTR);
               bodyShape.setScaleX(mWidth/mTR.getWidth());
               bodyShape.setScaleY(mHeight/mTR.getHeight());
               if (mRotation != 0.0f) {
                  bodyShape.setRotation(mRotation);
               }
               final Body mBody =
         PhysicsFactory.createCircleBody(mPhysicsWorld,
         bodyShape, mBodyType,
                     PhysicsFactory.createFixtureDef(mDensity,
           mElasticity, mFriction));
               mScene.getFirstChild().attachChild(bodyShape);
               mPhysicsWorld.registerPhysicsConnector(
         new PhysicsConnector(bodyShape, mBody, true,
            true));
            } else if(mShape.equals(TAG_SHAPE_VALUE_POLYGON)) {
               // Unimplemented
            } else {
               throw new IllegalArgumentException();
            }
         }
         public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes) {
         }
      });
```

The `onLoadEntity()` method for `TAG_BODY` is the most interesting element in Part 4. It will be called after BKLoader has loaded a completeShape definition, including the final tag. At this point all the parameters for the body have been encountered and set, so we can create the body and fixture we need to make it part of our scene. Here's a breakdown of what's going on:

- We figure out whether the body is a rectangle (`SQUARE`) or a circle. Polygons are not implemented because, well, polygons are hard. There's no Shape class to create polygons. Thus, if you plan to use them, you need a texture that is custom made for each polygon type.
- If the body is rectangular, we select the appropriate texture, based on the ID of the body, which tells us whether it's made of wood, stone, or glass. The method `selectTexture()` is defined later in Listing 12.5, for convenience. If the body is circular, we use the texture that is Mad Mat's head.
- We scale and rotate the Sprite according to the width, height, and rotation parameters retrieved from the XML file.
- We create an appropriate body and, as we're doing so, fill in the fixture parameters based on the information retrieved from the XML file.
- We add our Sprite as a child of the current Scene and create a PhysicsConnector that connects the Sprite to the body we just created.
- Finally, we include a stub `onLoadEntity()` method without the `pValue` parameter to satisfy the IBKEntityLoader interface.

Listing 12.5 **IVActivity.java: Part 5**

```
bkLoader.registerEntityLoader(TAG_X, new IBKEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes, final String pValue) {
      mX = new Float(pValue);
   }
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes) {
   }
});

bkLoader.registerEntityLoader(TAG_Y, new IBKEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes, final String pValue) {
      mY = new Float(pValue);
   }
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes) {
   }
});
```

```
bkLoader.registerEntityLoader(TAG_WIDTH, new IBKEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes, final String pValue) {
      mWidth = new Float(pValue);
   }
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes) {
   }
});
bkLoader.registerEntityLoader(TAG_HEIGHT, new IBKEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes, final String pValue) {
      mHeight = new Float(pValue);
   }
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes) {
   }
});
bkLoader.registerEntityLoader(TAG_ROTATION, new IBKEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes, final String pValue) {
      mRotation = new Float(pValue);
   }
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes) {
   }
});
bkLoader.registerEntityLoader(TAG_ISDYNAMIC, new IBKEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes, final String pValue) {
      mIsDynamic = true;
      if (pValue.equals("false")) mIsDynamic = false;
      mBodyType = BodyType.StaticBody;
      if (mIsDynamic) mBodyType = BodyType.DynamicBody;
   }
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes) {
   }
});
bkLoader.registerEntityLoader(TAG_SHAPE, new IBKEntityLoader() {
   @Override
   public void onLoadEntity(final String pEntityName,
final Attributes pAttributes, final String pValue) {
```

```
            mShape = pValue;
         }
         public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes) {
         }
      });
      bkLoader.registerEntityLoader(TAG_PHYSICSANDID,
      new IBKEntityLoader() {
         @Override
         public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes, final String pValue) {
            mPhysicsAndID = pValue;
            final String[] physTokens = mPhysicsAndID.split(",");
            mDensity = Float.valueOf(physTokens[1]).floatValue();
            mFriction = Float.valueOf(physTokens[0]).floatValue();
            mElasticity = Float.valueOf(physTokens[2]).floatValue();
            mID = trimQuotes(physTokens[3]);
         }
         public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes) {
         }
      });
      bkLoader.registerEntityLoader(TAG_VERTS, new IBKEntityLoader() {
         @Override
         public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes, final String pValue) {
         }
         public void onLoadEntity(final String pEntityName,
      final Attributes pAttributes) {
         }
      });

      try {
         bkLoader.loadLevelFromAsset(getApplicationContext(), "iv1.lvl");
      } catch (final IOException e) {
         Debug.e(e);
      }
      mScene.registerUpdateHandler(this.mPhysicsWorld);
      return mScene;
   }

   @Override
   public void onLoadComplete() {
      this.mPhysicsWorld.setContactListener(new ContactListener() {
            @Override
            public void beginContact(Contact contact) {
            }
```

```
            public void endContact(Contact contact) {
            }
        });
    }
```

The rest of the `onLoadEntity()` methods shown in Part 5 just extract each parameter from the XML information. Some of them, such as PhysicsAndID, have to be parsed out of the string.

Listing 12.5 **IVActivity.java: Part 6**

```
@Override
public boolean onSceneTouchEvent(final Scene pScene,
   final TouchEvent pSceneTouchEvent) {
   if(this.mPhysicsWorld != null) {
      final Scene scene = this.mEngine.getScene();
      if(pSceneTouchEvent.isActionDown()) {
         isStakeSpawning = true;
         stakeX = pSceneTouchEvent.getX();
         stakeY = pSceneTouchEvent.getY();
         return true;
      }
      if (pSceneTouchEvent.isActionMove()){
         if (isStakeSpawning){
            // Draw line, angle stake
            if (stakeLine == null){
               stakeLine = new Line(pSceneTouchEvent.getX(),
      pSceneTouchEvent.getY(), stakeX, stakeY);
            } else{
               stakeLine.setPosition(pSceneTouchEvent.getX(),
      pSceneTouchEvent.getY(), stakeX, stakeY);
            }
            scene.getLastChild().attachChild(stakeLine);
            return true;
         }
      }
      if (pSceneTouchEvent.isActionUp()){
         // Launch stake
         velX = (stakeX - pSceneTouchEvent.getX())/6.0f;
         velY = (stakeY - pSceneTouchEvent.getY())/6.0f;
         this.addStake(stakeX, stakeY, velX, velY);
         if (stakeLine != null) stakeLine.setPosition(0.0f, 0.0f,
      0.0f, 0.0f);
         isStakeSpawning = false;
         return true;
      }
```

```
    }
    return false;
}
```

The section of code found in Part 6 handles touch events so we can create flying stakes for the player. It does not contain anything particularly unusual: We note where the player touches first and where the touch ends, and we create a velocity vector in that direction, with magnitude relative to the distance from `ACTION_DOWN` to `ACTION_UP`.

Listing 12.5 **IVActivity.java: Part 7**

```
// ============================================================
// Methods
// ============================================================

private void addStake(final float pX, final float pY, float velX,
   float velY) {
   final Scene scene = this.mEngine.getScene();
   stakesprite = new Sprite(pX, pY, this.mStakeTextureRegion);
   stakesprite.registerEntityModifier(new RotationModifier(0.1f, 0.0f,
   (float) ((360.0f/Math.PI)*Math.atan(velY/velX))));
   stake = PhysicsFactory.createBoxBody(this.mPhysicsWorld,
   stakesprite, BodyType.DynamicBody, FIXTURE_DEF);
   stake.setBullet(true);
   stake.setLinearVelocity(new Vector2(velX, velY));
   stake.setSleepingAllowed(true);

   scene.getLastChild().attachChild(stakesprite);
   this.mPhysicsWorld.registerPhysicsConnector(
   new PhysicsConnector(stakesprite, stake, true, true));
}

private TextureRegion selectTexture(String id){
   if (id.equals("wood")){
      return mWoodTextureRegion;
   } else if (id.equals("stone")){
      return mStoneTextureRegion;
   } else {
      return mGlassTextureRegion;
   }
}

  public static String trimQuotes(String value)
  {
    if (value == null)
```

```
        return value;

      value = value.trim();
      if (value.startsWith("\'") && value.endsWith("\'"))
        return value.substring(1, value.length() - 1);
      return value;
    }
}
```

The final bits in Part 7 include some methods that make the previous calculations a little easier. Again, there is nothing unusual here.

Summary

We covered a lot of ground in this chapter, introducing both game physics and level loading. Even so, we merely scratched the surface of game physics. There is much more to investigate, including the following topics:

- Joints:
  - Distance joint
  - Revolute joint
  - Prismatic joint
  - Pulley joint
  - Gear joint
  - Mouse joint
  - Line joint
  - Weld joint
- Collision filtering
- Fixed-step physics
- Damping
- Fixed rotation
- Activation
- User data

The best place to start exploring these other features is the Box2D manual.

Exercises

1. Use Bison Kick or your favorite level editor to create a new level (Level 2) for Irate Villagers. Substitute it for `lvl1`, and run the gamelet. Adjust the density, friction, and restitution parameters until the level performs in the desired way.

2. As the gamelet now stands, the user can launch a stake from anywhere on the screen. Create a slingshot Sprite toward the left of the screen that is static. Change the stake creation code in `IVActivity.java` so you can create stakes only near the slingshot.

This page intentionally left blank

13

Artificial Intelligence

Games are more fun when the player has to use his or her intelligence to outwit an opponent. The opposing intelligence might be another player, as in a multiplayer game, or it might be built into the game by the developer and displayed through the actions of the other characters in the game. In games, the latter approach is often called artificial intelligence (AI), and it can make the games a lot more fun to play.

We can draw a distinction between AI in games and the research field of AI, though the distinction gets fuzzier as games progress. AI in research and AI in games have different goals. AI research has the aim of reproducing aspects of human intelligence using computers. AI in games aims to make a game more fun to play, and is usually limited to relatively simple strategies that the computer can devise for characters. But even at the frontiers of AI research games and research can come together. Consider IBM's Watson, whose deductive powers were recently showcased on the television game *Jeopardy*.

This chapter discusses some of the more popular AI strategies and algorithms used in games. AndEngine provides no special support for AI, but we will implement one of those strategies for the V3 example game using Java.

Game AI Topics

There will never be a definitive list of game AI algorithms and strategies, because new ones are devised and included in games every day. In this section, we consider some of the more classic strategies, including how they have been used in different types of games.

Simple Scripts

The simplest and most pervasive game artificial intelligence isn't very intelligent at all; it is just the defining of a character's behavior based on one or more scripts. The script can tell the character where to move, which noises to make, and so on, but the point is that the behaviors are fixed and usually triggered by a small group of events.

Everything we've done so far in V3 falls into this category, although instead of some scripting language, we've coded the behaviors into Java.

Decision Trees, Minimax Trees, and State Machines

The next step up the algorithmic ladder is to use more complicated software algorithms that can take advantage of a character's current and previous states to make more complicated (and therefore more interesting) decisions. If you aren't already familiar with these type of algorithms, here's a quick overview.

Decision Trees

In a decision tree, we progress through a series of choices until we reach a conclusion. The choices are characterized as nodes on a graph, and the conclusion is a leaf node on that graph. The thought process is similar to what you commonly see in troubleshooting charts or biology identification keys. A simple example for a hypothetical game is shown in Figure 13.1.

The implementation for a decision tree can be as simple as a chain of if-else statements (for binary decisions) or as complex as a chain of switch statements for (multivalued decisions).

Minimax Trees

Minimax trees are graphs that are often used to calculate the computer player's best move in a zero-sum game such as a board game. In other words, they help the computer pick the best action from a graph of possible actions. Part of a small minimax tree is illustrated in Figure 13.2; it shows the first few moves in a Tic-Tac-Toe game (Naughts and Crosses, if you're British). Each level in the tree corresponds to a move by one player. The levels are commonly called "plies" for minimax trees.

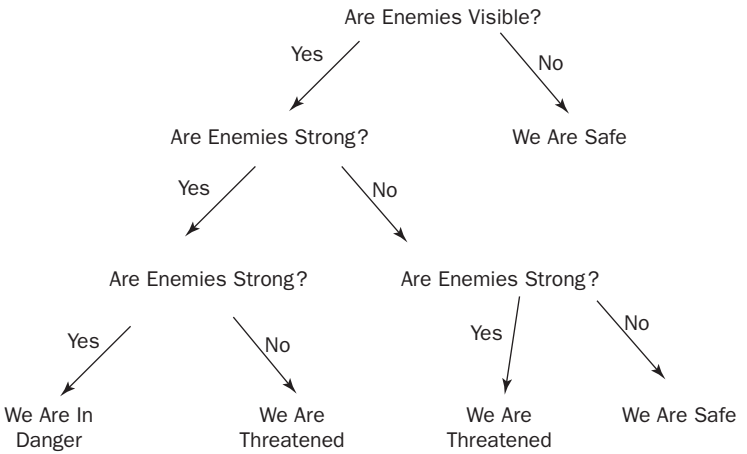


Figure 13.1 Simple decision tree

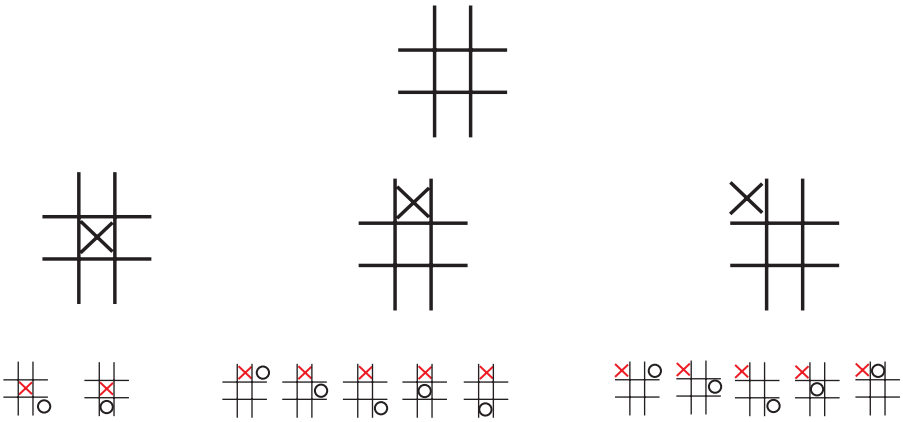


Figure 13.2 Partial minimax tree

Typically, values are assigned to each outcome (let's say, +10 if Player X wins and −10 if Player O wins). A value is then assigned to each intermediate node using some kind of heuristic (or guess) that assigns a level of "goodness" to that path. The algorithm's goal is reduced to either maximize (for Player X) or minimize (for Player O) the node value as the game progresses (hence the name "minimax"). The graphs and resulting search spaces can become very large very quickly, so schemes have been developed to prune the search.

Minimax trees are most useful when the computer has perfect and complete information about the game state. They are less useful in a card game, for example, where you can't see your opponent's hand. Minimax is a well-studied algorithm, with many examples of its application available on the Internet.

State Machines

Scripts and trees are pretty much unidirectional. The decision or play proceeds down the tree, resulting in a decision to take an action (although computation of values for the tree nodes often proceeds in a bottom-up direction). Finite state machines (FSMs), in contrast, model situations where there are known states and known events that cause transitions between states. The current state of the machine is a result of the history of transition events that have occurred up until now. The events might be known with certainty, or they might be triggered based on some probability distribution. In parsers, state machines are commonly used for lexical and syntactic analysis of source code.

We didn't implement a state machine for the vampires in V3. If we had, and if we'd included the behavior described in Exercise 2 at the end of this chapter (where the vampires try to evade weapons), it might look something like the diagram shown in Figure 13.3.

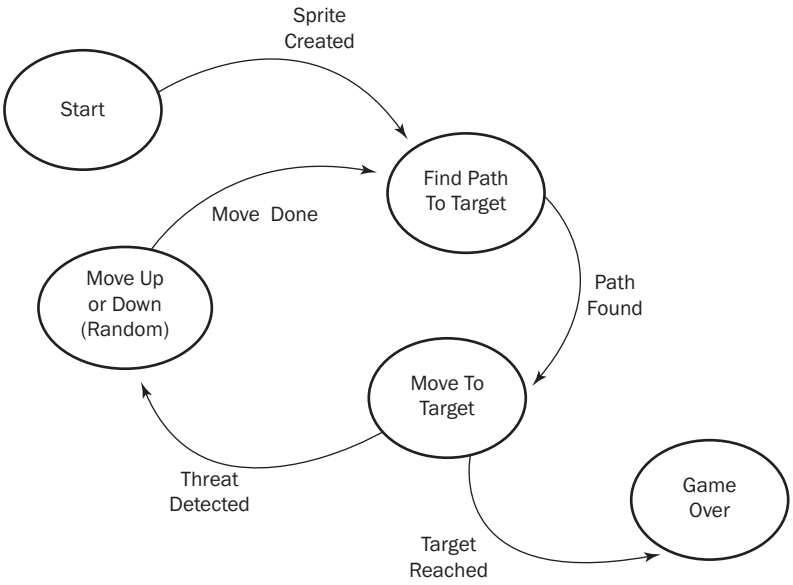


Figure 13.3 Vampire state machine

Expert or Rule-Based Systems

Moving closer to the academic field of artificial intelligence, expert systems attempt to distill the expertise of an expert player into a set of if-then rules that the computer can follow to derive an optimal strategy. Such a rule-based system typically consists of the following elements:

- The set of rules coded in a rule-based language, each of which has a condition (or predicate) and an action.
- A working memory, which provides state for the rules and can be modified by the actions.
- An inference engine, which iterates over the set of rules and determines which rule should "fire" or execute. Typically only one rule is allowed to fire on each pass, so if there are multiple rules whose conditions are satisfied, a conflict resolution policy is established to decide which will fire.
- An interface for getting events and data into the rule-based system and getting decisions out.

You could achieve the same ends by applying brute force, with if-then-else statements, but the result would be inefficient, messy, and difficult to maintain. Rule-based systems provide a clear structure and an efficient way of resolving the rule set. Rule-based languages are said to be "declarative" instead of "imperative."

In other words, you tell the rule engine what you know to be true and which rules you know to be true, and it reaches conclusions by combining those facts.

Some rule-based systems (e.g., OPS5 and JESS) are forward-chaining: "If these things are true, what is the resulting action?" Others (e.g., Prolog) are backward-chaining: "Is this thing true?" Most newer systems use both forward- and backward-chaining. Some rule-based systems can "learn" by creating and including rules dynamically.

Rule-based systems oriented toward computer games were popular some time ago, but lately most of the work in this area has been devoted to business rules. A Java standard, JSR-94 has been created for such systems, and several Java implementations are available, such as JESS and Drools (also known as JBoss Rules).

Listing 13.1 gives some sense of the way a rule-based language is used. To really make use of such a system, you'd need to learn the syntax of a complete language and connect the system to Android. That effort might be worth it if you need to implement some complex reasoning.

Listing 13.1 **Prolog Example**

```
. . . Prolog . . .
Vampire(Igor).
Ghoul(X) :- Vampire(X).
?- Ghoul(Igor).
Yes
?- Ghoul(X).
X = Igor
```

The Prolog lines translate as follows:

- Igor is a Vampire.
- Anything that is a Vampire is a Ghoul.
- Is Igor a Ghoul? The answer (yes) is printed by the Prolog engine.
- What Ghouls are there? The answer (Igor) is printed by the Prolog engine.

Just glancing at the rule syntax doesn't do justice to the major change in thinking required to adopt a rule-based approach. If you're interested, take a look at some of the many examples and tutorials on the Internet that show how rules are configured and used to solve complex problems.

Neural Networks

Neural networks are yet another approach to tackling problems that are difficult to solve with procedural languages such as Java. These networks are particularly good at pattern recognition. The idea is to establish a network of nodes that simulate simple

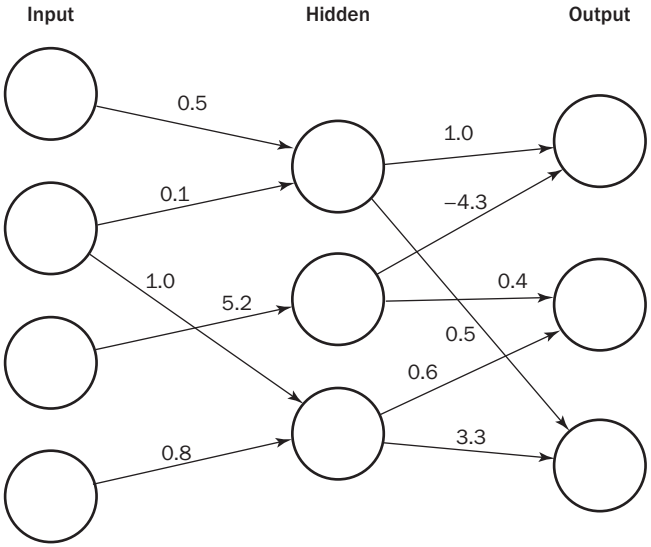


Figure 13.4 Simple neural network

neuron behavior (hence the name). The nodes are organized in layers and interconnected by weights. As each node fires, its firing propagates to the connected nodes, modified by the associated weight. Each node has a threshold, and if the sum of the incoming stimuli exceeds the threshold, that node in turn fires.

Figure 13.4 shows a simple neural network. There must always be an input layer and an output layer, but there can be any number of hidden layers.

The interesting thing about neural networks is that you don't program the weights or thresholds directly yourself, but rather train the network by introducing training inputs and comparing the output of the neural net with the correct output patterns. The differences are then "back-propagated" to the hidden and input layer weights. As the network is trained, it becomes increasingly better at recognizing similar input patterns. The mathematics needed to perform back-propagation is all well worked out and is part of what you get with neural network software. The ideal sizing of the network (how many nodes reside at each layer) is an active topic of research, but a vague rule of thumb is that most pattern recognition can be done with a single hidden layer (some sources argue for two layers), and the number of nodes at each layer should be one less than the number of patterns to be matched.

I'm not aware of a neural net package that's been ported to Android, but there are many good references and open-source implementations available on the Internet. Here are a couple of sites to point you in the right direction:

- Neuroph: `http://neuroph.sourceforge.net/download.html`
- The Encog Framework: `http://www.heatonresearch.com/encog`

Genetic Algorithms

Genetic algorithms are another approach to training software to solve problems that are too difficult to solve procedurally. More accurately, if there is a space containing potential solutions to a given problem, we can use a genetic algorithm to search that space for a solution that meets some criterion of "good enough." The idea is to mimic biological evolution, including the concepts of mutation and selection, to arrive at a solution. Genetic software typically includes the following elements:

- Encoding: The algorithm requires some way of representing components of the solution as a "chromosome" or coded string. Typically this is a string of bits, but it could be more complicated (e.g., strings of integers, letters).
- Selection: The algorithm needs some way to select the "best" parent solutions from each generation. We can create a new population of possible solutions by crossbreeding the most successful solutions from each generation; that is, each pair of selected parents combines to form two offspring solutions for the next generation.
- Crossover: We need some technique to combine chromosomes from successful parents to create the new generation. A simple way of combining chromosomes selects a "crossover point" randomly somewhere in the middle of the chromosome. One offspring inherits the first *n* bits from Parent A, and the last bits from Parent B. The other offspring inherits the first *n* bits from Parent B, and the last bits from Parent A.
- Mutation: Just as in biological evolution, it is desirable to have some random changes introduced to the chromosomes between generations in the algorithm. This technique can help our search avoid getting stuck at a local optimum.
- Termination: We don't want to continue searching forever, so there must be some definition of when we've found a solution that is "good enough." The termination condition could be manual inspection, for example.

We aren't using any genetic algorithms in V3, and it's difficult to give a concise example here. If you're interested, some good examples can be found on the Internet. One that includes an animation showing the evolution of generations of solutions can be found at the following address:

```
http://www.obitko.com/tutorials/genetic-algorithms/example-function-minimum.php
```

Path Finding

Path finding is a common problem in computer games that AI algorithms can solve. Typically a character is at a known position and would like to get to another known position, but it faces obstacles that block the way. Path finding can find the optimal route for the character to take.

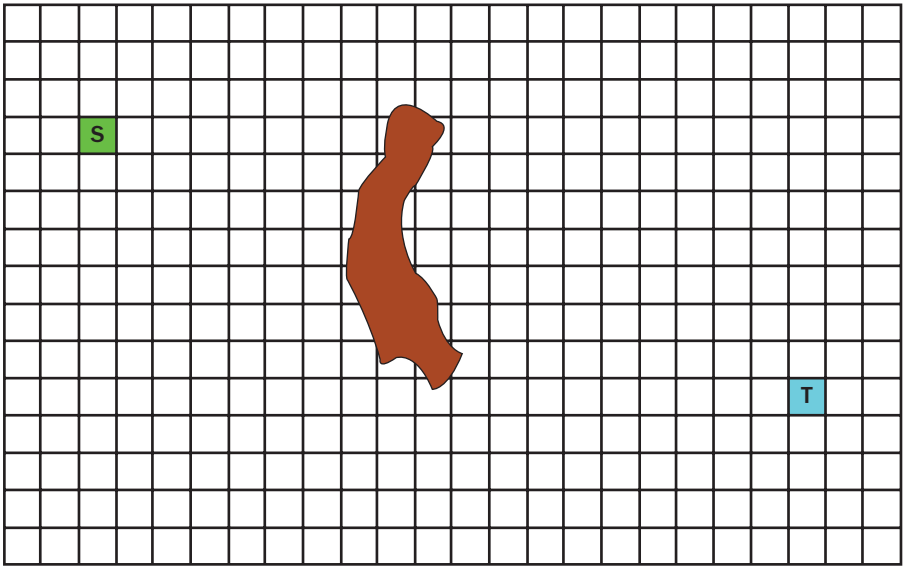


Figure 13.5 A* example grid

The A* ("A Star") path finding algorithm divides the playing space into a grid, something like that shown in Figure 13.5. The idea is to find the (near) optimal path for a character to get from S to T, avoiding any obstacles in between. The algorithm is fairly simple:

- Mark the grid locations where it is "legal" for the character to go (where there is no obstacle).
- Beginning at the S location:
  - Consider the adjacent legal grid locations (up to 8).
  - Compute values for each adjacent location:

    ```
    F = G + H
    ```

    F: is the location's value.

    G: is the location's distance from S, where vertical and horizontal moves count 1.0, and diagonal moves count 1.4.

    H: is a heuristic that approximates the distance to T.

    A commonly used heuristic is "Manhattan," which is the distance from the location to T when you are making only vertical or horizontal moves, ignoring obstacles. It is just the number of rows to T, plus the number of columns to T.
  - The algorithm then calls for moving to the location with the lowest value of F, and iterating this cycle until you reach T. The path of visited locations is the near optimal path for getting from S to T. We say "near" because the choice of heuristic can cause the path to be a bit suboptimal in certain situations.

If you'd like to see an animation of the A* algorithm in action, a good one can be found on the Wikipedia page for A* (at least there was as this book was written—Wikipedia changes):

```
http://en.wikipedia.org/wiki/A*_search_algorithm
```

Dynamic Difficulty Balancing

Another possible use of AI techniques in games is in dynamically modifying the difficulty of a game to match the ability of the current player. Ideally a game should be rewarding for the novice as well as the expert player, providing just enough challenge for both so that rewards are achievable, but not too easy. If we look at a player's scores on a given level over a period of time (it can be a relatively short period), we can view those data as a pattern that represents the player's ability at playing the game. Any of the AI techniques described previously could be used to assess that pattern and dynamically adjust variables, such as the number of antagonists, the time between threats, and the number of distractions, to make the game either easier or harder to play.

Procedural Music Generation

At the leading edge of current AI research are topics such as the automatic generation of appropriate music, which could also be introduced into games. These algorithms are very large and burn a lot of cycles, so we probably wouldn't want to have them running on a mobile Android device. Instead, we'd call on server-based AI processing to generate appropriate music given the parameters we pass to the program, which might include key, mood, tempo, and other musical attributes.

Once you have the power of the cloud at your disposal, many things become possible. We've already seen how Google's speech recognition can be used as an input method, and Microsoft Research has an application called Songsmith that will generate backup music for any tune you care to sing. There just *has* to be a way to work that into your game.

Implementing AI in V3

Let's add a bit of AI to the V3 game. We won't be so ambitious as to try adding a rule-based system or implementing procedural music generation, but we can make the game more interesting with some moderate AI.

Consider the part of the game we've been calling Layer1, which is shown in Figure 13.6. Right now, the vampires start at random positions on the right-hand side of the screen, and they walk straight toward Miss B's, trampling on the tombstones that have been placed in their way. Furthermore, when the player positions a bullet, hatchet, or cross in front of a vampire, the villain keeps walking straight for Miss B's door, making no attempt to dodge the weapon.

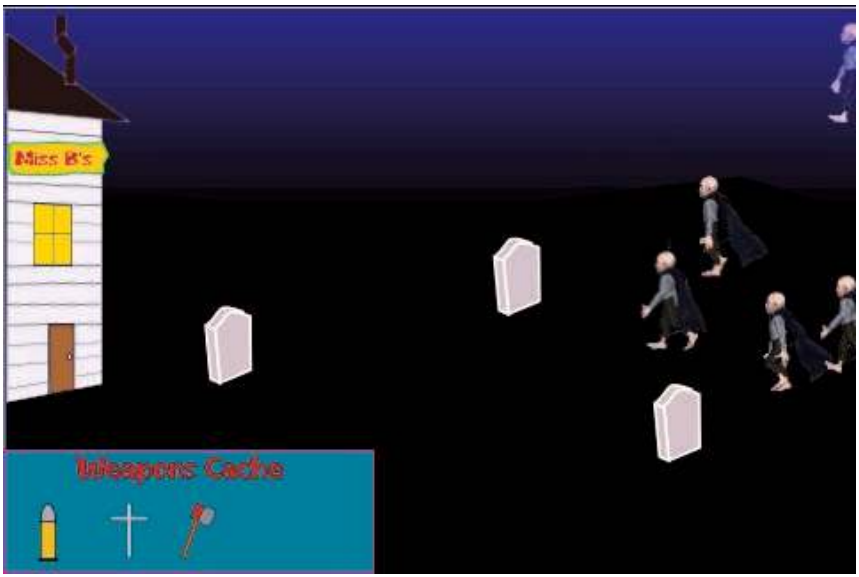


Figure 13.6 Level 1 in midplay

Let's make some changes so that the vampires are a little smarter:

- Overlay an invisible grid that we can use to compute each vampire's A* path to Miss B's.
- If there's a tombstone in the way, make the vampire walk around it. To be really effective, we'd change the vampire animation so it looks like the vampires are walking either toward us or away us from when they're going around a tombstone, but we won't try to be that fancy here.

Implementing A*

The Internet offers many implementations of A*, written in many languages. We won't try for the most efficient implementation here, but rather seek out the most understandable, so you can clearly see what's happening as the paths are computed.

A* is relatively expensive in terms of computing resources, so we will limit our path computations to events that require them—either a vampire Sprite is created on the right of the screen, or a weapon causes a path recalculation.

We'll divide the playing area by setting up a 24 × 16 grid to match the aspect ratio we've been using for the game example. We will manually map the tombstone obstacles onto that grid. Note that if we were using a tiled map scene, the gridding and mapping would have already been done for us. Figure 13.7 shows the scene with the background omitted for clarity, and the grid superimposed on the contents. When we create the A* grid in our program, we will note each location containing an obstacle (the shaded areas in Figure 13.7).

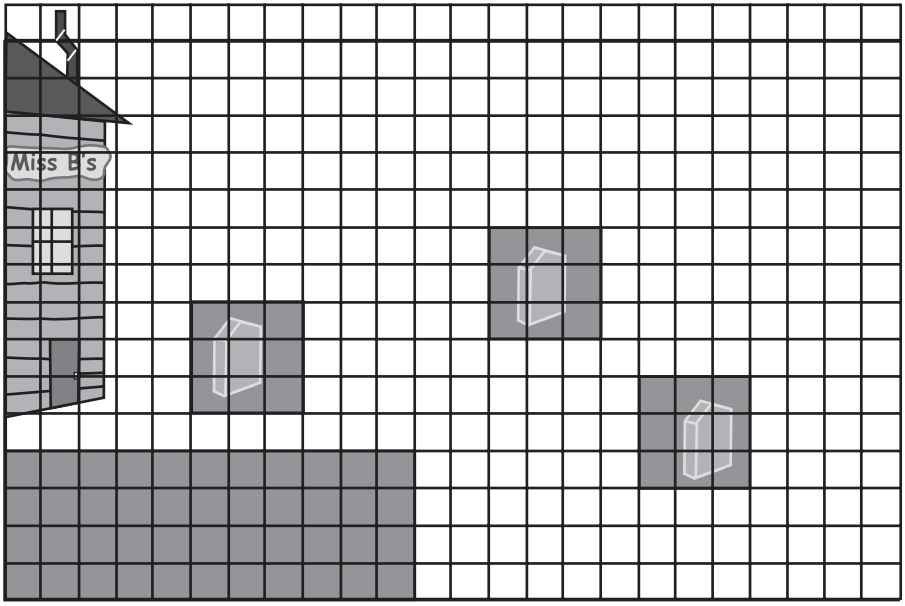


Figure 13.7 Level 1 with A* grid

GridLoc Class

We'll create a class, GridLoc, that represents each location on the grid. Listing 13.2 shows the essentials of GridLoc. It's really more of a structure than a class; in other words, it's a place to hold the pertinent parameters of a grid location.

Listing 13.2 **GridLoc.java**

```
package com.pearson.lagp.v3;

public class GridLoc {
   public float g = 0.0f;   // dx from start (measured)
   public float h = 0.0f; // dx to target (heuristic)
   public boolean obstacle = false;
   public boolean footprint = false;
}
```

The variables are pretty obvious, but to avoid confusion we'll summarize them here:

- `float g`: The measured distance from the starting location. As noted earlier, horizontal and vertical moves are valued at 1.0f and diagonal moves at 1.4f (roughly the square root of 2).
- `float h`: The heuristic value for the Manhattan distance to the target, ignoring obstacles.

- `boolean obstacle`: A value that is `true` only if this grid location is part of an obstacle.
- `boolean footprint`: A value that is `true` if we've already visited this grid location in our path finding. It helps prevent the program from getting stuck in loops or "dancing."

AStar Class

We also need a class that implements the A* algorithm, finding a path from any starting location on the grid to any target location. We'll call that class AStar, and the code for it is shown in Listing 13.3.

Listing 13.3 **AStar.java**

```
package com.pearson.lagp.v3;

import java.util.ArrayList;

import org.anddev.andengine.entity.modifier.PathModifier.Path;

import android.util.Log;

public class AStar {
   // ===========================================================
   // Constants
   // ===========================================================

   // ===========================================================
   // Fields
   // ===========================================================
   private GridLoc[][] grid;
   private int rowMax, colMax;
   private int cellWidth, cellHeight;
   private String tag = "AStar:";

   // ===========================================================
   // Constructors
   // ===========================================================
   public AStar(int pRows, int pCols, int pWidth, int pHeight) {
      // pWidth = total width in pixels
      // pHeight = total height in pixels
      grid = new GridLoc[pRows][pCols];
      rowMax = pRows-1;
      colMax = pCols-1;
      cellWidth = pWidth/pCols;
      cellHeight = pHeight/pRows;
       for (int i=0; i<pRows; i++) {
          for (int j=0; j<pCols; j++) {
             grid[i][j] = new GridLoc();
```

```
        }
     }
}
// ============================================================
// Getter and Setter
// ============================================================
public void setObstacle(int pObstacleRow, int pObstacleCol){
   if (grid != null){
      grid[pObstacleRow][pObstacleCol].obstacle = true;
   }
}

public Path getPath(float pStartX, int pTargetCol, float pStartY,
   int pTargetRow, float pSpriteWidth, float pSpriteHeight){
   // Use A* pathfinding to find the near optimal path
   int nextCol, nextRow;
   int startCol, startRow;
   ArrayList<Integer> pathCols = new ArrayList<Integer>();
   ArrayList<Integer> pathRows = new ArrayList<Integer>();
   startCol = (int)pStartX/cellWidth;
   startRow = (int)pStartY/cellHeight;
   int currCol = startCol;
   int currRow = startRow;
   float[] f = new float[8];
   grid[currRow][currCol].g = 0.0f;
   grid[currRow][currCol].h =
      pTargetCol - currCol + pTargetRow - currRow;
   grid[currRow][currCol].footprint = true;

   while ((currCol != pTargetCol) || (currRow != pTargetRow)){
      //Consider the eight surrounding locations
      for (int i=0; i<8; i++) f[i] = 0;

      f[0] = fComp(currRow, currCol, -1, -1, 1.4f,
         pTargetRow, pTargetCol);
      f[1] = fComp(currRow, currCol, 0, -1, 1.0f,
         pTargetRow, pTargetCol);
      f[2] = fComp(currRow, currCol, +1, -1, 1.4f,
         pTargetRow, pTargetCol);
      f[3] = fComp(currRow, currCol, -1, 0, 1.0f,
         pTargetRow, pTargetCol);
      f[4] = fComp(currRow, currCol, +1, 0, 1.0f,
         pTargetRow, pTargetCol);
      f[5] = fComp(currRow, currCol, -1, +1, 1.4f,
         pTargetRow, pTargetCol);
      f[6] = fComp(currRow, currCol, 0, +1, 1.0f,
         pTargetRow, pTargetCol);
```

```
f[7] = fComp(currRow, currCol, +1, +1, 1.4f,
   pTargetRow, pTargetCol);

int lowidx = 0;
float pos = 10000.0f;
for (int j=0; j<8; j++){
   if (f[j]<pos){
      pos = f[j];
      lowidx = j;
   }
}
nextCol = currCol;
nextRow = currRow;
switch (lowidx){
   case (0):
      nextRow = currRow - 1;
      nextCol = currCol - 1;
      break;
   case (1):
      nextRow = currRow;
      nextCol = currCol - 1;
      break;
   case (2):
      nextRow = currRow + 1;
      nextCol = currCol - 1;
      break;
   case (3):
      nextRow = currRow - 1;
      nextCol = currCol;
      break;
   case (4):
      nextRow = currRow + 1;
      nextCol = currCol;
      break;
   case (5):
      nextRow = currRow - 1;
      nextCol = currCol + 1;
      break;
   case (6):
      nextRow = currRow;
      nextCol = currCol + 1;
      break;
   case (7):
      nextRow = currRow + 1;
      nextCol = currCol + 1;
      break;
}
```

```
            //Add next location to Path,
            //set footprint and update currCol, currRow
            pathCols.add(nextCol);
            pathRows.add(nextRow);
            grid[currRow][currCol].footprint = true;
            currCol = nextCol;
            currRow = nextRow;
        }
        float[] xArray = new float[pathCols.size()+1];
        float[] yArray = new float[pathRows.size()+1];
        xArray[0] = pStartX;
        yArray[0] = pStartY;


        for (int i = 1; i < xArray.length; i++) {
            Float tmpX = (float)pathCols.get(i-1) * cellWidth
              - pSpriteWidth/2;
            xArray[i] = (tmpX != null ? tmpX : 0.0f);
            Float tmpY = (float)pathRows.get(i-1) * cellHeight
              - pSpriteHeight/2;
            yArray[i] = (tmpY != null ? tmpY : 0.0f);
        }
        return (new Path(xArray, yArray));
    }
    // ============================================================
    // Methods for/from SuperClass/Interfaces
    // ============================================================

    // ============================================================
    // Methods
    // ============================================================

    private float fComp(int pCurrRow, int pCurrCol, int pRowDiff,
        int pColDiff, float pDx, int pTargetRow, int pTargetCol){
        // Computes the A* values for a grid location:
        //    g: distance from start
        //  h: distance to target
        // returns f = g + h
        // If the grid location is marked as an obstacle,
        //  footprint is true; if it is outside the grid,
        //  returns a high number (5,000)
        if (((pCurrRow + pRowDiff) > rowMax) ||
              ((pCurrCol + pColDiff) > colMax) ||
              ((pCurrRow + pRowDiff) < 0) ||
              ((pCurrCol + pColDiff) < 0)) {
            return 5000.0f;
        }
```

```
        if((grid[pCurrRow + pRowDiff]
           [pCurrCol + pColDiff].obstacle) ||
           (grid[pCurrRow + pRowDiff]
              [pCurrCol + pColDiff].footprint)){
           return 5000.0f;
        }

        grid[pCurrRow + pRowDiff][pCurrCol + pColDiff].g =
           grid[pCurrRow][pCurrCol].g + pDx;
        grid[pCurrRow + pRowDiff][pCurrCol + pColDiff].h =
           Math.abs(pTargetRow - (pCurrRow + pRowDiff)) +
           Math.abs(pTargetCol - (pCurrCol + pColDiff));
        return (grid[pCurrRow + pRowDiff][pCurrCol + pColDiff].g +
           grid[pCurrRow + pRowDiff][pCurrCol + pColDiff].h);
    }
    // ============================================================
    // Inner and Anonymous Classes
    // ============================================================
}
```

This code is not difficult to follow. The significant details are highlighted here:

- We set up for the algorithm by declaring a variable grid, which is a two-dimensional array of GridLoc locations. This is an exact analogy of the grid we laid over our scene in Figure 13.7.
- The constructor for AStar initializes that grid, along with some other useful variables that let us test for things like array bounds easily. We will create an AStar object for each Path that we want to find.
- We'll use the `setObstacle()` to tell the A* algorithm where the obstacles are located in the grid. We'll call `setObstacle()` for each location in the grid that is part of an obstacle we want to avoid.
- The `getPath()` method is where we actually implement the A* algorithm. It's worth looking at the return value and the parameters, so we're clear about what we're dealing with:
  - Path: The return from the method is a PathModifier Path. Recall that we can attach a PathModifier to a Sprite, and the Sprite will then follow that path to its conclusion. That's exactly what we want here. We don't know how long the Path will be, but we do know that it will have two arrays, one for the x coordinates and one for the y coordinates.
  - `float pStartX` and `float pStartY`: These parameters set the coordinates of the starting location in pixels. We need the pixel location so we can create a path where the Sprite doesn't jump at the beginning. This information gets prepended to the Path that we find from the starting grid location.

  - `int pTargetCol` and `int pTargetRow`: The column and row for the target location. The target is expressed in column by row dimensions, because the algorithm can only find paths to a cell.
  - `float pSpriteWidth` and `float pSpriteHeight`: We will find a path consisting of a series of grid locations, and we need to convert those data into a pixel path for a Sprite. To do so, we need the dimensions of the Sprite.
- The `getPath()` method starts by initializing some variables. We use `ArrayList<Integer>` variables to develop the Path, as we don't know how long it will be in advance.
- The while loop continues looping until we reach the target location. On each pass of the loop, it adds one location to the Path. As we add each location, we take a number of steps:
  - We initialize an array to hold the A* values (F in the equation F = G + H) and call a method, `fcomp()`, that will compute those values for each of the eight locations that surround the current location. If the test location is an obstacle, or if we've already been there, or if the location is outside the grid, the method will return a large number (5000.0f). Otherwise, it returns the A* value calculated as the sum of the distance from the starting point and the Manhattan distance to the target.
  - Once we have all eight values, we use a for loop to find the smallest value. That value indicates the direction in which we'd like to move—and now it's clear why we were returning 5000 as the value for locations where we don't want to go in the last step.
  - A switch statement updates the next row and column, matching the row and column difference parameters that we fed to `fcomp()` for the index that turned out to have the lowest value.
  - We add the next row and column to the path ArrayLists, and mark the current location as one that we've visited (`footprint = true`).
  - We advance the current row and column to the next row and column.
- When we return from the while loop, we've found our complete path; thus we can create the arrays of pixel locations needed to construct a Path for PathModifier. We initialize the arrays, fill them from the grid location path ArrayLists, and return the completed Path to the caller.
- The `fcomp()` method is also straightforward. We check the test location to see whether it is outside of the grid, part of an obstacle, or a place we've already visited. If any of those conditions is true, we return a large value, so the location will be out of the running for the chosen direction in which to move. If it's otherwise legal, we just compute the values for `g` (distance from the start) and `h` (Manhattan heuristic for distance to the target), and we return their sum, `f`.

Adding Path Finding to Level1Activity

Just a few changes to Level1Activity are needed to take advantage of the path finding algorithm for the vampire Sprites. The changes are shown in Listing 13.4.

Listing 13.4 **Changes to Level1Activity.java**

```
. . .
public class Level1Activity extends BaseGameActivity {
. . .
   // ===================================================
   // Fields
   // ===================================================
   private AStar[] aStar = new AStar[10];
   private Path[] pathVamp = new Path[10];
. . .
   @Override
   public void onLoadResources() {
. . .
      for (int i=0; i<10; i++) {
         aStar[i] = new AStar(16, 24,
            CAMERA_WIDTH, CAMERA_HEIGHT);
         aStar[i].setObstacle(12,0);
         aStar[i].setObstacle(13,0);
         aStar[i].setObstacle(14,0);
         aStar[i].setObstacle(15,0);
         aStar[i].setObstacle(8,5);
         aStar[i].setObstacle(8,6);
         aStar[i].setObstacle(8,7);
         aStar[i].setObstacle(9,5);
         aStar[i].setObstacle(9,6);
         aStar[i].setObstacle(9,7);
         aStar[i].setObstacle(10,5);
         aStar[i].setObstacle(10,6);
         aStar[i].setObstacle(10,7);
         aStar[i].setObstacle(6,13);
         aStar[i].setObstacle(7,13);
         aStar[i].setObstacle(8,13);
         aStar[i].setObstacle(6,14);
         aStar[i].setObstacle(7,14);
         aStar[i].setObstacle(8,14);
         aStar[i].setObstacle(6,15);
         aStar[i].setObstacle(7,15);
         aStar[i].setObstacle(8,15);
         aStar[i].setObstacle(10,17);
         aStar[i].setObstacle(11,17);
         aStar[i].setObstacle(12,17);
         aStar[i].setObstacle(10,18);
```

```
            aStar[i].setObstacle(11,18);
            aStar[i].setObstacle(12,18);
            aStar[i].setObstacle(10,19);
            aStar[i].setObstacle(11,19);
            aStar[i].setObstacle(12,19);
        }
. . .
    private Runnable mStartVamp = new Runnable() {
        public void run() {
. . .
               pathVamp[i] = aStar[i].getPath(asprVamp[i].getX(), 1,
            asprVamp[i].getY(), 10, asprVamp[i].getWidth(),
            asprVamp[i].getHeight());
                 asprVamp[i].registerEntityModifier(
                    new SequenceEntityModifier (
                       new AlphaModifier(5.0f, 0.0f, 1.0f),
                      new PathModifier(60.0f, pathVamp[i])
                    ));
. . .
}
```

The steps again are easy to follow:

- We declare grids and paths for each of the 10 possible vampire Sprites.
- For each of the grids, we tell AStar where the obstacles are. The grids must remain separate so the Sprites don't interfere with one another as they are calculating their individual paths.
- In the `mStartVamp()` Runnable, as each vampire Sprite is created it asks its grid for the best path from the starting point to the target location. The resulting Path is attached to the Sprite with a PathModifier, replacing the previous MoveModifier, which simply moved the Sprite in a straight line.

Summary

This chapter took a very high-level view of some of the AI techniques that can be applied to mobile games and a specific look at one of those techniques, A* path finding. Volumes have been written about AI techniques, and volumes more about the way those techniques can be used in games. This chapter should give you a starting point for exploring the algorithms that might be of use in your own games.

Exercises

1. If you've run the example code for this chapter, you'll see an issue with it. Yes, all of the vampires find their way to Miss B's door, no matter where they start from, and no matter which obstacles are in the way. But because of the way the tombstones

and weapons box are positioned, the vampires tend to funnel themselves into a single line leading across the screen.

Change `Level1Activity.java` so that each vampire moves straight to the left for some random period of time before creating a path to Miss B's door. Does that strategy help spread the vampires out on the screen?

2. The vampires are still pretty dumb. When faced with a weapon, they just continue on their pre-computed path, even if it leads straight into the teeth of the weapon. Change `Level1Activity.java` so vampires react to a player's placement of a bullet or hatchet by moving laterally a few steps and then calculating a new path to Miss B's.

14

Scoring and Collisions

Scores are key to most games, giving the players feedback on how well they are playing the game and a measurement they can strive to improve.

Scoring systems can be very complicated or very simple. A simple scoring system should at least meet these needs:

- Keep track of the current score as a game is being played
- Record some number of highest scores
- Display the current score to the player
- Display the list of highest scores

More complicated scoring systems could include these functions:

- Keep track of scores for players on other (connected) clients, perhaps with ranking systems and tournament play.
- Keep track of scores for multiple players on a single device.
- Record the time and date of high scores.
- Provide scoring by time (e.g., time to solve a puzzle), rather than points.
- Give out in-game rewards based on reaching a given score. You might award the player a new weapon, for example, after the player has achieved a given score, to make the game more interesting.
- Reward achievements that are not directly tied to game play (awards could include anything from a trophy to special found objects). Achievements are being used increasingly in mobile games to extend the playability of games beyond the basic play of the game.

We will tend toward simpler score keeping and leave the more complicated scoring and achievements to your inventiveness.

Collisions of one kind or another are the most frequently scored events in games. We've used a bit of collision detection in V3 examples so far, but in this chapter

we explore the two major methods of collision detection for AndEngine as part of implementing our scoring enhancements:

- Collisions using AndEngine's `collidesWith()` methods for Sprites and Shapes
- Collisions using the Box2D Physics Engine

Scoring Design

AndEngine doesn't provide any special APIs or structures for scoring, so we'll have to create some. Of course, we will take advantage of AndEngine's display capabilities and use Android's APIs to build a simple scoring system that works well with AndEngine. We did a little of this work in Exercise 3 of Chapter 11 (you can see the details in the Exercise Solutions Appendix), but we will expand that effort greatly here.

Here is the list of requirements for our scoring system:

- Update scores from any gamelet.
- Track the five highest scores.
- Display the five highest scores for each gamelet on the Scores page, ranked from highest to lowest.
- Display a gamelet's current score as part of the gamelet's Scene.
- Have scores persist across invocations of the game.

Update the Scores from Any Gamelet

We will use Android SharedPreferences as the storage location for our scores. Recall that we used SharedPreferences for the on/off Boolean values associated with music and sound effects in Chapter 11. Android creates one SharedPreferences object for our entire application, and we access that object using code like that shown in Listing 14.1.

Listing 14.1 **SharedPreferences Reading**

```
   scores = getSharedPreferences("scores", MODE_PRIVATE);
   highScores[4] = scores.getInt("Level1-4", 0);
   highScores[3] = scores.getInt("Level1-3", 0);
   highScores[2] = scores.getInt("Level1-2", 0);
   highScores[1] = scores.getInt("Level1-1", 0);
   highScores[0] = scores.getInt("Level1-0", 0);
. . .
```

We've labeled the SharedPreference for scores as "scores"; it contains key–value pairs, where the key is a String and the value is an integer (the score). We'll keep five such values for each gamelet to record the highest five scores. The `getInt()` method parameters are just the key String and a default value to return in case the key String

is not found. Thus, in the calls shown in Listing 14.1, `scores.getInt()` will return a value of 0 if the key is not found.

Putting the score values into the SharedPreference is just as easy, as shown in Listing 14.2.

Listing 14.2 **SharedPreferences Loading**

```
   scores = getSharedPreferences("scores", MODE_PRIVATE);
   scoresEditor = scores.edit();
   scoresEditor.putInt("Level1-4", highScores[4]);
   scoresEditor.commit();
. . .
```

There is one caveat to using SharedPreferences to store scores: If you are attaching any real values to scores, you need to be careful. Players who have rooted their phones (gained Linux root user access by loading alternative system firmware) can load your game, and then access and edit the information in SharedPreferences in any way they like. In the V3 case, the scores are just scores, so it's okay to use SharedPreferences. The same concern applies to SQLite files: If you need to hide some persistent piece of information from all players, you'll need to encrypt it.

Track the Five Highest Scores

Inside each gamelet, we'll retrieve the five highest scores at the beginning of the game and update them at the end if they change. The list of scores will change only if the player wins the gamelet and his or her score is higher than one of the scores already in the list. Listing 14.3 shows typical code for the end of a gamelet, for the case when the player has won and might have a score higher than one already on the list.

Listing 14.3 **End-of-Gamelet Coding for Scores**

```
       int[] newHighScores = {0,0,0,0,0};
       for (int i=4; i>-1; i--){
           if (thisScore > highScores[i]){
              newHighScores[i] = thisScore;
              for (int j=i-1; j>-1; j--){
                 newHighScores[j] = highScores[j+1];
              }
              if (i==4) newHigh = true;
              break;
           } else {
              newHighScores[i] = highScores[i];
           }
       }
       for (int i=0; i<5; i++) highScores[i] = newHighScores[i];
   scoresEditor.putInt("Level1-4", highScores[4]);
```

```
   scoresEditor.putInt("Level1-3", highScores[3]);
   scoresEditor.putInt("Level1-2", highScores[2]);
   scoresEditor.putInt("Level1-1", highScores[1]);
   scoresEditor.putInt("Level1-0", highScores[0]);
   scoresEditor.commit();
. . .
```

Coming into this code, the previous high scores are held in the integer array `highScores[]`, and the score for this gamelet is found in `thisScore`. We create a new array, `newHighScores[]`, where we'll create the potentially modified list. We copy values from `highScores[]` into `newHighScores[]` as long as they are larger than or equal to `thisScore`. If we come to a `highScore[]` value that is smaller than `thisScore`, we insert `thisScore` and continue copying the remaining scores into `newHighScores[]`.

Once we have the new list of high scores, we copy it back into `highScores[]` and update the SharedPreferences. Note that we must commit the changes once we're done, or the `putInt()` methods will not have any effect.

Display the Score on the Gamelet's Scene

We examined Text entities in Chapter 7. At this point, we need to use a ChangeableText entity to display the current score while a gamelet is running. We'll always display the score in the upper-right corner of the screen, and we'll keep it up-to-date by using a private method that updates both the `thisScore` integer and the display. A short example and the private method are shown in Listing 14.4.

Listing 14.4 **Score ChangeableText**

```
   // Score display
   mCurrScore = new ChangeableText(0.75f*CAMERA_WIDTH, 10.0f,
      mFont32, "Score: 0", "Score: XXXXXX".length());
   scene.getLastChild().attachChild(mCurrScore);
. . .
   mAddScore(BULLET_VAMP_SCORE);
. . .
     private void mAddScore(int pAdder){
        thisScore += pAdder;
        mCurrScore.setText("Score: " + thisScore);
     }
. . .
```

We've allowed for a six-digit score in our ChangeableText, and we've positioned it in the upper-right corner of the screen. We loaded `mFont32` from the Flubber TrueType font earlier in the activity. Whenever a game event occurs that results in updating the score, we call the method `mAddScore()`, which updates both the internal `thisScore` variable and the `mCurrScore` display.

Figure 14.1 Scores page

Scores Page Display

Nothing fancy is needed here—we just want a simple list of scores for each gamelet, displayed in our Flubber typeface so that it matches the rest of our game. We continue to navigate to the Scores page from the Options Menu, just as we have been doing ever since we added preliminary scoring in Chapter 11, Exercise 3. Figure 14.1 shows a screenshot of the Scores page filled with some high scores.

The updated version of `ScoresActivity.java` is shown in Listing 14.5.

Listing 14.5 **ScoresActivity.java with Highest Score Lists**

```
package com.pearson.lagp.v3;

+imports

public class ScoresActivity extends BaseGameActivity {
   // ============================================================
   // Constants
   // ============================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;

   // ============================================================
   // Fields
   // ============================================================
```

```
    protected Camera mCamera;

    protected Scene mScoresScene;
    private Text mTitle, mHeaders;
    private Text[] mScoreL = new Text[5];
    private Text[] mScoreW = new Text[5];
    private Text[] mScoreI = new Text[5];
    private Texture mFontTexture;
    private Font mFont;
    private SharedPreferences scores;
    private SharedPreferences.Editor scoresEditor;

    // ===========================================================
    // Constructors
    // ===========================================================

    // ===========================================================
    // Getter and Setter
    // ===========================================================

    // ===========================================================
    // Methods for/from SuperClass/Interfaces
    // ===========================================================

    @Override
    public Engine onLoadEngine() {
        this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
            CAMERA_HEIGHT);
        scores = getSharedPreferences("scores", MODE_PRIVATE);
        scoresEditor = scores.edit();
        return new Engine(new EngineOptions(true,
            ScreenOrientation.LANDSCAPE,
            new RatioResolutionPolicy(CAMERA_WIDTH,
            CAMERA_HEIGHT), this.mCamera));
    }

    @Override
    public void onLoadResources() {
        /* Load Font/Textures. */
        this.mFontTexture = new Texture(256, 256,
            TextureOptions.BILINEAR_PREMULTIPLYALPHA);

        FontFactory.setAssetBasePath("font/");
        this.mFont = FontFactory.createFromAsset(this.mFontTexture,
            this, "Flubber.ttf", 32, true, Color.RED);
        this.mEngine.getTextureManager().loadTexture(
this.mFontTexture);
```

```
        this.mEngine.getFontManager().loadFont(this.mFont);
    }

    @Override
    public Scene onLoadScene() {
        /* Center the background on the camera. */
        final int centerX = (CAMERA_WIDTH) / 2;
        final int centerY = (CAMERA_HEIGHT) / 2;

        this.mScoresScene = new Scene(1);
        /* Add the background and scores */
        mScoresScene.setBackground(new ColorBackground(
            0.0f, 0.0f, 0.0f));
        mTitle = new Text( centerX - 200, centerY - 120, mFont,
            "Scores");
        mScoresScene.getLastChild().attachChild(mTitle);
        mHeaders = new Text( centerX - 150, centerY - 80, mFont,
            "Level1     WhAV     IV");
        mScoresScene.getLastChild().attachChild(mHeaders);
        for (int i=0; i<5; i++){
            mScoreL[i] = new Text( centerX - 150,
                (centerY - 40) + (4-i)*40, mFont, "" +
                scores.getInt("Level1-"+i, -1));
            mScoreW[i] = new Text( centerX, (centerY - 40) +
                (4-i)*40, mFont, "" +
                scores.getInt("WhAV-"+i, -1));
            mScoreI[i] = new Text( centerX + 150,
                (centerY - 40) + (4-i)*40, mFont, "" +
                scores.getInt("IV-"+i, -1));
            mScoresScene.getLastChild().attachChild(mScoreL[i]);
            mScoresScene.getLastChild().attachChild(mScoreW[i]);
            mScoresScene.getLastChild().attachChild(mScoreI[i]);
        }
        return this.mScoresScene;
    }

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

The code is very straightforward, so we won't belabor it. It's worth pointing out that the positions are fairly well hard-coded for the 480 × 320 display size. If your game is going to be more flexible with respect to screen size than the example V3 game, you might want to improve that part of the code.

Collisions in AndEngine

We've largely ignored collision detection so far, but now we need to get a better handle on this issue so that we can score our gamelets properly. As mentioned briefly at the beginning of this chapter, there are two ways of detecting collisions in an AndEngine game, depending on whether you are using physics in your game:

- If you're not using the Box2D physics engine, AndEngine provides a collision detection method for Sprites and Shapes in your game. This method allows you to test whether two objects are currently colliding, and the recommendation is that this test take place in an UpdateHandler routine that you register with the current Scene.
- If you are using Box2D, you can take advantage of its embedded collision detection system, which is a bit more flexible than the AndEngine capability and can be more efficient. You can register a ContactListener method with the PhysicsWorld you have defined, and Box2D will then call `beginContact()` and `endContact()` methods in that ContactListener every time any two objects collide in that world. You can even use the Box2D collision detector if you've excluded the objects from the physics simulation, although we won't be doing that for V3 (see the Box2D manual [referenced in Chapter 12] if you want to find out more).

AndEngine Shape Collisions

Detecting collisions between AndEngine Shapes (including Sprites) is really easy. The detection operation just has to be performed at the right time, after the Scene has done all its position updates and you can compare a snapshot of where all the Shapes are located. Listing 14.6 shows the pattern for testing for collisions.

Listing 14.6 **AndEngine Shape Collision Testing**

```
scene.registerUpdateHandler(new IUpdateHandler() {
   @Override
   public void reset() { }

   @Override
   public void onUpdate(final float pSecondsElapsed) {
      if (spriteA.collidesWith(spriteB)){
         //Do something as a result of collision
      }
   });
```

That's all there is to it! You just have to test for every collision of interest to you.

Box2D Collisions

Box2D collision detection can get much more complicated, however. By default, Box2D will call your ContactListener methods for every collision that occurs in each frame of the physics simulation. A collision filtering system is also included, through which you can specify bitmasks that filter out unimportant collisions. For our scoring purposes, we won't need these kinds of filters.

Listing 14.7 shows the setup for the ContactListener methods. Both methods [`beginContact()` and `endContact()`] are passed a single parameter, a Contact. That parameter contains all the details of the collision:

- Which two Fixtures collided, and their characteristics:
  - Density
  - Friction
  - Elasticity (restitution)
- Which Bodies are attached to those Fixtures, and their characteristics:
  - Mass
  - Density
  - Linear velocity
  - Angle
  - Angular velocity

In short, a Contact contains just about any information you'd need to know to characterize the collision. You can also get the UserData for each Body in the collision, which is any object you care to assign to the Body. We will use that as a way of identifying the type of Body.

Listing 14.7 **Box2D Collision Listening**

```
this.mPhysicsWorld.setContactListener( new ContactListener() {
    @Override
    public void beginContact(Contact contact) {
        Body bodyA = contact.getFixtureA().getBody().;
        Body bodyB = contact.getFixtureB().getBody();
    }
         public void endContact(Contact contact) {
    }
});
```

Notice one thing about the code in Listing 14.7: Given the method that Box2D uses to detect collisions, you can never be sure which Body will be `bodyA` and which will be `bodyB`. If you're looking for a collision between two Body types, you will have to test both ways (the example later in this chapter for IVActivity does just that).

Letting the Player Score

Now that we have a mechanism for viewing and recording scores, we need to give players a way to actually score some points. The scoring will be different for each gamelet, and we'll assume that a Level 1 point is somehow equivalent to a Whack-A-Vampire point. We'll go through each gamelet and point out the places where we've added code that implements player scoring.

Graveyard (Level 1)

The Level 1 gamelet is the most complex of the ones in V3, and it should include many ways to score. Here's a list of the things we need to do to add scoring to this gamelet:

- Ways to score: We need to implement the scoring mechanism described in the previous section so the player receives a score for the number of vampires killed.
- Vampire deaths: The player can currently kill a vampire by touching it. We also need the weapons to kill vampires as they collide with them. When a vampire dies, that event should prompt an update of the score (we'll give the player 50 points for killing a vampire by touching it, 100 points for killing it with a bullet, 200 points for using a hatchet, and 500 points for killing the vampire with a cross). To add some interest, and some justification for the different scores, we'll say that a bullet kills everything in its path, the hatchet kills only the first vampire in its path, and the cross just sits there and waits for the first vampire to run into it.
- Gamelet completion: To have a final score, the gamelet must end. The Level 1 gamelet will end either when a vampire reaches Miss B's door (the vampires win) or when all the vampires are dead (the player wins). Upon completion of the gamelet, the player should be presented with a results screen, along with the choice to play again or go back to the main menu. The completion scene in Figure 14.2 depicts the case in which the player wins. The scene in Figure 14.3 appears when the vampires win.

This is a rather large activity (approximately 700 lines), so let's take it section by section.

Constants and Fields

The changes to the Constants and Fields sections of the activity are shown in Listing 14.8.

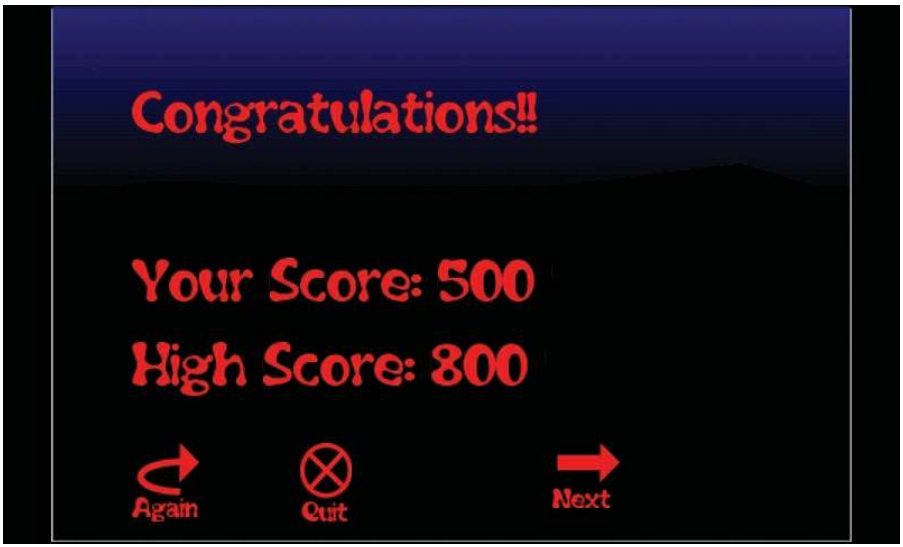


Figure 14.2 Level 1: “the player wins” scene

Figure 14.3 Level 1: “the vampires win” scene

Listing 14.8 **Level1Activity.java Scoring Additions: Constants and Fields**

```
public class Level1Activity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================
```

```
. . .
   private static final int TOUCH_VAMP_SCORE = 50;
   private static final int BULLET_VAMP_SCORE = 100;
   private static final int HATCHET_VAMP_SCORE = 200;
   private static final int CROSS_VAMP_SCORE = 500;

   private static final int NUKE_BULLET = 1;
   private static final int NUKE_HATCHET = NUKE_BULLET+1;
   private static final int NUKE_CROSS = NUKE_HATCHET+1;
   private static final int NUKE_TOUCH = NUKE_CROSS+1;

   private static final int MAX_VAMPS = 10;
   private static final int VAMP_RATE = 2000;

   private static final boolean PLAYER_WINS = true;
   private static final boolean VAMPIRES_WIN = false;

   // Location of MissB's front door
   private static final Rectangle MissBs = new Rectangle(
               35.0f, 195.0f, 15.0f, 35.0f);

   // ===========================================================
   // Fields
   // ===========================================================
. . .
   private Texture mPopUpTexture;
. . .
   private TextureRegion mEndBackTextureRegion;
   private TextureRegion mAgainButtonTextureRegion;
   private TextureRegion mQuitButtonTextureRegion;
   private TextureRegion mNextButtonTextureRegion;
   private TextureRegion mNewHighTextureRegion;
   private Texture mFontTexture;
   private Font mFont32;
   private ChangeableText mCurrScore;

   private Sprite endBack, newHigh, againButton, quitButton,
      nextButton;
   private int nVamp, nVampsKilled;
. . .
   private SharedPreferences scores;
   private SharedPreferences.Editor scoresEditor;
   private int[] highScores = new int[5];
   private int thisScore = 0;
. . .
```

First, we create some constants for the score values, along with a set of flags we'll use later when killing vampires. We want to put the vampire killing code in a separate method, as it is much the same no matter how the vampire got killed. We'll use the flags `NUKE_BULLET`, `NUKE_HATCHET`, and so on to tell the method how the vampire was killed, so it can apply the correct score.

Next, we create constants for the number of vampires on the screen at any one time and the rate at which they appear. In a previous version of the gamelet, we hard-coded these values as numbers before; the changes here represent the first step toward making them become variables so we can modify the difficulty of the gamelet. We won't implement difficulty balancing until Chapter 16.

Finally in the Constants section, we define some Boolean values for indicating whether the player or the vampires won. We also define a Rectangle that overlays the front door of Miss B's. We'll use this Rectangle to identify when a vampire reaches its target.

The Fields additions are mostly intended to support the pop-up completion scenes (shown in Figures 14.2 and 14.3, respectively) and the score display. We've also added an integer to count the number of vampires killed (so we'll know when they're all dead and the player has won), and the variables needed for the `scores` SharedPreferences.

onLoadEngine and onLoadResources

The only change to `onLoadEngine()` is to add the code for the `scores` SharedPreferences, as shown in Listing 14.1. The current values of the high scores are just read into the integer array `highScores[]`.

The changed section of code for `onLoadResources()` is shown in Listing 14.9.

Listing 14.9 **Level1Activity.java Scoring Additions: onLoadResources()**

```
   @Override
   public void onLoadResources() {
. . .
      mPopUpTexture = new Texture(512, 512,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      mEndBackTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mPopUpTexture, getApplicationContext(),
            "endback.png", 0, 0);
      mAgainButtonTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mPopUpTexture, getApplicationContext(),
            "againbutton.png", 0, 330);
      mQuitButtonTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mPopUpTexture, getApplicationContext(),
            "quitbutton.png", 50, 330);
```

```
      mNextButtonTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mPopUpTexture, getApplicationContext(),
            "nextbutton.png", 100, 330);
      mNewHighTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mPopUpTexture, getApplicationContext(),
            "newhigh.png", 100, 400);
      mEngine.getTextureManager().loadTexture(
         this.mPopUpTexture);

      this.mFontTexture = new Texture(256, 512,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      FontFactory.setAssetBasePath("font/");
      mFont32 = FontFactory.createFromAsset(this.mFontTexture,
         this, "Flubber.ttf", 32, true, Color.RED);
      mEngine.getTextureManager().loadTexture(this.mFontTexture);
      mEngine.getFontManager().loadFont(this.mFont32);
. . .
```

No real mysteries here. We've added the bitmaps that will go into the completion screens and the Font that we need to write the scores.

onLoadScene

As usual, life gets more interesting in the `onLoadScene()` method. We'll use this method to add the collisions with weapons that kill the vampires, create and add the score display, and create the Sprites for the completion scenes. Listing 14.10 shows the additions.

Listing 14.10 **Level1Activity.java Scoring Additions: onLoadScene()**

```
   @Override
   public Scene onLoadScene() {
. . .
      scene.registerUpdateHandler(new IUpdateHandler() {
         @Override
         public void reset() { }

         @Override
         public void onUpdate(final float pSecondsElapsed) {
           for (int i=0; i<nVamp; i++){
            if (asprVamp[i].collidesWith(bullet)){
              mNukeVamp(i, NUKE_BULLET);
            }
            if (asprVamp[i].collidesWith(hatchet)){
              mNukeVamp(i, NUKE_HATCHET);
            }
```

```
      if (asprVamp[i].collidesWith(cross)){
        mNukeVamp(i, NUKE_CROSS);
      }
      if (asprVamp[i].collidesWith(MissBs)){
        //gamelet over, vampires win
        mGameOver(VAMPIRES_WIN);
      }
      if ((touchActive) &&
          (asprVamp[i].collidesWith(touchRect))){
        mNukeVamp(i, NUKE_TOUCH);
      }
     }
   }
});

// Score display
mCurrScore = new ChangeableText(0.75f*CAMERA_WIDTH,
   10.0f, mFont32, "Score: 0",
   "Score: XXXXXX".length());
scene.getLastChild().attachChild(mCurrScore);

// Create Sprites for result screens – don't attach yet
endBack = new Sprite(
  (CAMERA_WIDTH – mEndBackTextureRegion.getWidth()) / 2,
  (CAMERA_HEIGHT – mEndBackTextureRegion.getHeight()) / 2,
  mEndBackTextureRegion);
newHigh = new Sprite(0.0f, 0.0f, mNewHighTextureRegion);
againButton = new Sprite(0.0f, 0.0f,
   mAgainButtonTextureRegion){
         @Override
         public boolean onAreaTouched(final TouchEvent
     pAreaTouchEvent, final float pTouchAreaLocalX,
     final float pTouchAreaLocalY) {
          switch(pAreaTouchEvent.getAction()) {
          case TouchEvent.ACTION_DOWN:
             //TBD
          }
          return true;
         }
};
nextButton = new Sprite(0.0f, 0.0f,
   mNextButtonTextureRegion){
         @Override
         public boolean onAreaTouched(final TouchEvent
     pAreaTouchEvent, final float pTouchAreaLocalX,
     final float pTouchAreaLocalY) {
          switch(pAreaTouchEvent.getAction()) {
          case TouchEvent.ACTION_DOWN:
```

```
                    //TBD
                }
                return true;
            }
        };
        quitButton = new Sprite(0.0f, 0.0f,
            mQuitButtonTextureRegion){
                @Override
                public boolean onAreaTouched(final TouchEvent
          pAreaTouchEvent, final float pTouchAreaLocalX,
          final float pTouchAreaLocalY) {
                switch(pAreaTouchEvent.getAction()) {
                case TouchEvent.ACTION_DOWN:
                    //TBD
                }
                return true;
            }
        };
        return scene;
    }
```

Here we've added an UpdateHandler to check for collisions at each update. We loop through the current set of vampires and check whether each of them has collided with the bullet, the hatchet, the cross, Miss B's front door, or the spot that the player is touching. If the answer is yes in the first three cases or the last case, we declare a nuked vampire and call `mNukeVamp()`, which we define later. If the collision occurred with Miss B's door, we declare the game over and the vampires the victors by calling the `mGameOver()` method, also defined later.

We then create the ChangeableText for the score display, plus the Sprites for the buttons that will go on the completion screens. Right now, the buttons won't do anything. We will save that work for Chapter 16.

mStartVamp

Recall that `mStartVamp()` is the method we use to start up each of the vampires. The only thing we want to do here is change what happens when the player touches a vampire. We alter that behavior in the `onAreaTouched()` override that is part of the AnimatedSprite definition for the vampire, as shown in Listing 14.11.

Listing 14.11 **Level1Activity.java Scoring Additions: mStartVamp()**

```
    private Runnable mStartVamp = new Runnable() {
        public void run() {
. . .
          asprVamp[i] = new AnimatedSprite(CAMERA_WIDTH - 30.0f,
         startY, mScrumTextureRegion.clone()) {
              @Override
```

```
     public boolean onAreaTouched(
final TouchEvent pAreaTouchEvent,
final float pTouchAreaLocalX,
final float pTouchAreaLocalY) {
        switch(pAreaTouchEvent.getAction()) {
        case TouchEvent.ACTION_DOWN:
          /* Is there a vampire close by? */
          touchRect = new Rectangle (
   pAreaTouchEvent.getX(),
   pAreaTouchEvent.getY(), 20.0f, 20.0f);
          touchActive = true;
          break;
        case TouchEvent.ACTION_UP:
          touchActive = false;
         }
        return true;
}
     };
```

Earlier, in the `onAreaTouched()` method, we checked for collision by comparing the touch point with the position of the Sprite. We've modified that a bit here so that we can make use of our `mNukeVamp()` method for touches as well as weapons. This approach also simplifies the code.

Whack-A-Vampire

The Whack-A-Vampire gamelet is simpler than the graveyard gamelet described in the last section. We do need to add the scoring mechanism and to define completion of the WAV gamelet to finish it, however.

- Ways to score: We will reuse much of the code from the last section to implement scoring based on the number of coffins the player successfully closes. We'll also say that if a coffin stays open for more than some time limit without being touched, the player loses points.
- Coffin closings: We'll set up two time constants (in anticipation of making them variables when we start adjusting game difficulty in Chapter 16). The first time constant, `OPEN_RATE`, will be the average time between coffin openings. The second constant, `OPEN_TIME`, will be the time for which a coffin stays open and the player can touch and close it. A third constant, `OPENS_PER_GAME`, will set the maximum number of coffins opened for the game.
- Gamelet completion: The gamelet will end when `OPENS_PER_GAME` coffins have been opened and closed (either by touch or by time-out). The player does not actually win or lose in this gamelet, so we'll just present the results using the "player wins" screen, as shown in Figure 14.2.

Again, we'll break down the changes by examining them in sections and refer back to the Level 1 changes where we do very similar things in Whack-A-Vampire.

Constants and Fields

In addition to the fields we introduced for the results screens in Level 1, we add some constants and fields to support the Whack-A-Vampire scoring, as shown in Listing 14.12.

Listing 14.12 **WAVActivity.java Scoring Additions: Constants and Fields**

```
. . .
   private static final int CLOSE_COFFIN_SCORE = 100;
   private static final int OPEN_RATE = 4000;
   private static final int OPENS_PER_GAME = 10;
   private static final int STAY_OPEN = 2000;
. . .
   private int mNumClosed = 0;
   private ArrayList<Integer> openCoffins = new ArrayList<Integer>();
. . .
```

We talked about the constants earlier. Here, we've introduced two variables to keep track of open coffins. The ArrayList `openCoffins` keeps a FIFO (first in, first out) list of coffins that have been opened.

We'll skip over the revisions to `onLoadEngine()` and `onLoadResources()`. The changes are very similar to what we did in Level 1, including getting the current high scores from the SharedPreferences and loading the textures needed for the results screen.

onLoadScene

The `onLoadScene()` method is where the scoring action happens, so let's take a look at the way the ArrayList is used to keep track of the open coffins in Listing 14.13.

Listing 14.13 **WAVActivity.java Scoring Additions: onLoadScene()**

```
   @Override
   public Scene onLoadScene() {
. . .
      scene.setOnSceneTouchListener(new IOnSceneTouchListener() {
         @Override
         public boolean onSceneTouchEvent(
            final Scene pScene,
            final TouchEvent pSceneTouchEvent) {
            switch(pSceneTouchEvent.getAction()) {
            case TouchEvent.ACTION_DOWN:
              /* Get the touched tile */
              tmxTile = tmxLayer.getTMXTileAt(
```

```
                pSceneTouchEvent.getX(),
                pSceneTouchEvent.getY());
              if((tmxTile != null) &&
                 (tmxTile.getGlobalTileID() ==
                  mOpenCoffinGID)) {
                mAddScore(CLOSE_COFFIN_SCORE);
                tmxTile.setGlobalTileID(
                   mWAVTMXMap, mCoffinGID);
              }
              break;
            case TouchEvent.ACTION_UP:
              break;
            }
            return true;
          }
      });
. . .
```

The `onSceneTouchEvent()` override was found in `WAVActivity.java` in earlier versions of the gamelet. We've actually added only one new line (the rest of the listing is shown for context):

```
mAddScore(CLOSE_COFFIN_SCORE);
```

openCoffin and closeCoffin

Let's look at the methods we'll use to open a coffin and close a coffin. The complete updated methods are shown in Listing 14.14.

Listing 14.14 **WAVActivity.java Scoring Additions: openCoffin() and closeCoffin()**

```
   private Runnable openCoffin = new Runnable() {
        public void run() {
           int openThis = gen.nextInt(coffinPtr);
           int tileRow = coffins[openThis]/15;
           int tileCol = coffins[openThis] % 15;
           tmxTile = tmxLayer.getTMXTileAt(tileCol*32 + 16,
         tileRow*32 + 16);
           tmxTile.setGlobalTileID(mWAVTMXMap, mOpenCoffinGID);
           openCoffins.add(openThis);
           int openTime = gen.nextInt(OPEN_RATE);
          mHandler.postDelayed(openCoffin, openTime);
          mHandler.postDelayed(closeCoffin, openTime+STAY_OPEN);
         }
        };
```

```
    private Runnable closeCoffin = new Runnable() {
        public void run() {
           int closeThis = openCoffins.get(0);
           openCoffins.remove(0);
           int tileRow = coffins[closeThis]/15;
           int tileCol = coffins[closeThis] % 15;
           tmxTile = tmxLayer.getTMXTileAt(tileCol*32 + 16,
         tileRow*32 + 16);
      tmxTile.setGlobalTileID(mWAVTMXMap, mCoffinGID);
      if (++mNumClosed > OPENS_PER_GAME) mGameOver(PLAYER_WINS);
         }
        };
```

The `openCoffin()` method is much the same as before, but now we've added the scheduling of `closeCoffin()` to close the coffin after the `STAY_OPEN` time.

The `closeCoffin()` method is new. It takes the first coffin from `openCoffins` and closes it (even if it was already "closed" by the player by touching it). We know the first coffin is the next to be closed, because all coffins have the same `STAY_OPEN` time. This is the only place where we check for the end of the game.

The changes to the rest of `WAVActivity.java` are almost exactly the same as the ones we made to `Level1Activity.java` in the last section.

Irate Villagers

For the Irate Villagers gamelet, once again we need to introduce the scoring mechanism and define game completion:

- Scoring: In IV, the player advances the score by causing a vampire head to hit the floor. We'll give the player 200 points for each successful concussion.
- Gamelet completion: The gamelet should end either when the last vampire head hits the floor (the player wins) or when a set number of stakes have been launched (the vampires win). We'll display results screens similar to the ones that we used in the other gamelets.

Constants and Fields

The constants and fields we need to add for the IV gamelet are shown in Listing 14.15.

Listing 14.15 **IVActivity.java Scoring Additions: Constants and Fields**

```
// ============================================================
// Constants
// ============================================================

. . .
```

```
   private static final int VAMPIRE_FLOORED = 200;
   private static final boolean PLAYER_WINS = true;
   private static final boolean VAMPIRES_WIN = false;
   private static final int MAX_STAKES = 5;
. . .
   // ======================================================
   // Fields
   // ======================================================

   private Texture mPopUpTexture;
   private TextureRegion mEndBackTextureRegion;
   private TextureRegion mAgainButtonTextureRegion;
   private TextureRegion mQuitButtonTextureRegion;
   private TextureRegion mNextButtonTextureRegion;
   private TextureRegion mNewHighTextureRegion;
. . .
   private Texture mFontTexture;
   private Font mFont32;
   private ChangeableText mCurrScore;
   private Sprite endBack, newHigh, againButton, quitButton,
      nextButton;
. . .
   private int numStakes = 0;
. . .
   private int numHeads = 0;
   private ArrayList<Body> deadHeads = new ArrayList<Body>();
   private SharedPreferences scores;
   private SharedPreferences.Editor scoresEditor;
   private int[] highScores = new int[5];
   private int thisScore = 0;
```

The field most worthy of mention in Listing 14.15 is the `ArrayList<Body> deadHeads`. We will use that field to keep track of which vampire heads have hit the floor. The danger is that a head will bounce, registering multiple collisions with the floor—we want to count only the first of those hits.

The `onLoadEngine()` and `onLoadResources()` methods change much the same way they did for `Level1Activity.java` and `WAVActivity.java`, so we won't dwell on them here.

onLoadScene

Recall that the `onLoadScene()` method is where we load levels into the gamelet, using the level loader routines. We need to identify the number of vampire heads we load, and note which of the physics bodies is the floor. We'll do so by using the ID tag. Vampire heads will be marked with the ID "vamp" and the floor marked with the tag "floor." Listings 14.16 and 14.17 show the needed changes.

Listing 14.16 **IVActivity.java Scoring Additions: onLoadScene(), Part 1**

```
   @Override
   public Scene onLoadScene() {
. . .
      bkLoader.registerEntityLoader(TAG_BODY,
         new IBKEntityLoader() {
         @Override
         public void onLoadEntity(final String pEntityName,
           final Attributes pAttributes,
           final String pValue) {
             if(mShape.equals(TAG_SHAPE_VALUE_SQUARE)) {
. . .
            final Body mBody =
              PhysicsFactory.createBoxBody(
               mPhysicsWorld, bodyShape,
               mBodyType,
               PhysicsFactory.createFixtureDef(
                  mDensity, mElasticity,
                  mFriction));
               mBody.setUserData(mID);
. . .
             } else if(mShape.equals(
               TAG_SHAPE_VALUE_CIRCLE)) {
. . .
            final Body mBody =
              PhysicsFactory.createBoxBody(
               mPhysicsWorld, bodyShape,
               mBodyType,
               PhysicsFactory.createFixtureDef(
                  mDensity, mElasticity,
                  mFriction));
               mBody.setUserData(mID);
. . .
             } else if(mShape.equals(
. . .
```

Listing 14.16 shows the part of `onLoadScene()` that actually creates the physics bodies from the data that have already been collected from the XML level file. Here, we simply load the ID (previously stored in `mID`) into the `UserData` for each body. We can retrieve that ID easily later when the bodies collide.

The other change in `onLoadScene()` occurs a little later, where we're setting up the loading of ID tags, as shown in Listing 14.17. The addition here is to bump the number of heads whenever we see a "vamp" ID on a tag.

Listing 14.17 **IVActivity.java Scoring Additions: onLoadScene(), Part 2**

```
      bkLoader.registerEntityLoader(TAG_PHYSICSANDID,
         new IBKEntityLoader() {
         @Override
         public void onLoadEntity(final String pEntityName,
. . .
            mID = trimQuotes(physTokens[3]);
            if (mID.equals("vamp")) numHeads++;
         }
```

onLoadComplete

The other change for the IV gamelet is to expand the `beginContact()` override in `onLoadComplete()` so that we register a score whenever a vampire head hits the floor. The new version is shown in Listing 14.18.

Listing 14.18 **IVActivity.java Scoring Additions: onLoadComplete()**

```
   @Override
   public void onLoadComplete() {

      this.mPhysicsWorld.setContactListener(
         new ContactListener() {
            @Override
            public void beginContact(Contact contact) {
             Body bodyA = contact.getFixtureA().getBody();
             Body bodyB = contact.getFixtureB().getBody();
             String idA = (String)bodyA.getUserData();
             String idB = (String)bodyB.getUserData();
             if ((idA.equals("vamp")) && (idB.equals("floor"))) {
               if (!deadHeads.contains(bodyA)) {
         deadHeads.add(bodyA);
           }
               mAddScore(VAMPIRE_FLOORED);
               if (deadHeads.size() == numHeads) }
            mGameOver(PLAYER_WINS);
           }
             }
             if ((idB.equals("vamp")) && (idA.equals("floor"))) {
               if (!deadHeads.contains(bodyB)) {
         deadHeads.add(bodyB);
           }
               mAddScore(VAMPIRE_FLOORED);
               if (deadHeads.size() == numHeads) }
```

```
            mGameOver(PLAYER_WINS);
           }
             }
             public void endContact(Contact contact) {
            }
    });
}
```

When we are notified of a physics collision, we check whether either body was a vampire head, and whether the other body was the floor. In case of a head–floor collision, we credit the player with the score and check whether all the heads have hit the floor. As mentioned at the beginning of the section, we record each head that hits the floor in the ArrayList `deadHeads`, so we won't count bounces or multiple hits of the same head.

addStake

The only other change (other than adding to IV all of the scoring code we added to the other gamelets) is to count the number of stakes launched and declare the game over after the last one is launched. That change consists of a one-line addition to the `addStake()` method, as shown in Listing 14.19.

Listing 14.19 **IVActivity.java Scoring Additions: addStake()**

```
   private void addStake(final float pX, final float pY, float velX,
         float velY) {
      /* If player has used up the stakes, game is over */
      if (numStakes++ > MAX_STAKES) mGameOver(VAMPIRES_WIN);
. . .
```

With that, the changes to include collisions and scoring are complete. V3 is now a more playable game. After we take a look at some AndEngine extensions in Chapter 15, we'll finish off the V3 game in Chapter 16.

Summary

In this chapter, we looked at two related features, scoring and collisions. There are two ways to detect collisions in AndEngine games, with the method used depending on whether we are using the Box2D physics engine. We also built a scoring structure using the display elements available in AndEngine and the SharedPreferences storage APIs from Android.

Exercises

1. You may have noticed that a sound file called `oof.ogg` is included in the downloadable code (in the `assets/mfx` folder). Change `Level1Activity.java` so that the "oof" sound is played whenever a vampire's head hits the floor.

2. Many games take special notice when the player achieves a new high score for a gamelet. Change the results screen on the three gamelets so "New High Score" will be displayed only when the player's score is higher than any other score on the high scores list. The graphic for "New High Score" is already in the downloadable code, in the `assets/gfx/scoring` folder, and an example screen is shown in Figure 14.4.

Figure 14.4 High score indication

This page intentionally left blank

15

Multimedia Extensions

AndEngine is a growing, evolving platform for building Android games. As new features are developed, they are often introduced to the platform as extensions. Extensions are not bundled into the base AndEngine code but are readily available for download. The source for all the extensions can be reached from the main AndEngine URL:

`http://code.google.com/p/andengine/`

As of this book's writing, there were seven extensions available for AndEngine, some of which we have already used earlier in the book:

- Live wallpaper: lets you use AndEngine to build Android live wallpapers
- MOD player: enables the play of MOD and other music files, using the XMP player
- Multiplayer: uses a special protocol to enable multiplayer games over IP connections
- Multi-touch: incorporates multi-touch capabilities for those Android devices that have them
- Physics Box2D: a physics engine (discussed at length in Chapter 12)
- Augmented reality: allows the overlay of AndEngine Scenes over the device's camera preview
- SVG TextureRegions: supports scalable vector graphics (discussed in Chapter 5)

This chapter devotes a section to each of the extensions that we have not yet investigated. We will develop five short example programs that show the basic features of each extension and leave it up to you to investigate further if you are interested.

Downloading Extensions

Each of the AndEngine extensions has its own source repository. The downloadable code for the book includes the `.jar` files for each of the extensions, which were current as the book was written. Later versions of the `.jar` files may be available in the

lib folder on each source repository by the time you read these words. Alternatively, you can clone the sources for the repositories and build the .jar files yourself, using the pattern covered in Chapter 12.

The source repositories for the extensions are found at the following URLs:

```
http://code.google.com/p/andenginelivewallpaperextension/
http://code.google.com/p/andenginemodplayerextension/
http://code.google.com/p/andenginemultiplayerextension/
http://code.google.com/p/andenginemultitouchextension/
http://code.google.com/p/andenginephysicsbox2dextension/
http://code.google.com/p/andengineaugmentedrealityextension/
http://code.google.com/p/andenginesvgtextureregionextension/
```

Live Wallpapers

The AndEngine Live Wallpapers extension helps us create Android live wallpapers that take advantage of all the display capabilities of AndEngine games. We can create and display Sprites, Shapes, Animations, Modifiers, Particle Effects, Text, Tilemaps, and everything else we've used in creating games, and make them part of the device wallpaper that is displayed as the background for the home screen.

You can download an example wallpaper activity from the following site:

```
http://code.google.com/p/andenginelivewallpaperextension/
```

This wallpaper simulates a lit cigarette, with copious clouds of smoke rising out of the cigarette. When you tilt the Android device, the smoke always floats "up." To run the example, download it, create a new Android Project in Eclipse using the downloaded project files, build and install the files using the Run command, and use the Android Settings > Personalize > Home wallpaper dialog to change the device wallpaper. "Cigarette Live-Wallpaper" will be one of the choices in the list of available wallpapers. Figure 15.1 shows the home screen with the example wallpaper running.

Android Live Wallpapers

Android version 2.1 added support for live wallpapers. From our point of view, that means users running previous versions of Android will not be able to load and use any live wallpapers we create. The older devices are quickly being replaced, however, so it's a temporary issue.

Live wallpapers in Android run as a Service. If you plan to create one, you should take a look at the developer blog entry at this URL:

```
http://android-developers.blogspot.com/2010/02/live-wallpapers.html
```

As the blog points out, live wallpapers can be real battery eaters, because they run all the time. In particular, you must ensure that the wallpaper stops executing when it is covered by another Activity—that is, when another Activity takes over the screen.

Figure 15.1 Example wallpaper

Otherwise, you're just wasting battery power by churning out graphics that no one will see.

Creating a Live Wallpaper for V3

You might want to create a live wallpaper out of the elements of your game. As an example, we'll create such an element for V3. We'll compose a small scene in which vampires walk through a dark graveyard, as shown in Figure 15.2.

Figure 15.2 V3 wallpaper

Listing 15.1 shows the interesting parts of the `V3LiveWallpaper.java` source file (the complete source is included with the download).

Listing 15.1 **V3LiveWallpaper.java**

```
. . .
public class LiveWallpaperService extends BaseLiveWallpaperService {
. . .
   private static final int VAMP_RATE = 2000;
   private static final int MAX_VAMPS = 10;
```

```
// ============================================================
// Fields
// ============================================================

private Handler mHandler;
private Texture mScrumTexture;
private Texture mBackgroundTexture;
private TextureRegion mBackgroundTextureRegion;
private TiledTextureRegion mScrumTextureRegion;

private AnimatedSprite[] asprVamp = new AnimatedSprite[10];
private int nVamp;

private ScreenOrientation mScreenOrientation;

private Random gen;

// ============================================================
// Methods for/from SuperClass/Interfaces
// ============================================================

@Override
public org.anddev.andengine.engine.Engine onLoadEngine() {
   mHandler = new Handler();
   gen = new Random();
   return new org.anddev.andengine.engine.Engine(
      new EngineOptions(true, this.mScreenOrientation,
      new FillResolutionPolicy(), new Camera(0, 0,
         CAMERA_WIDTH, CAMERA_HEIGHT)));
}

@Override
public void onLoadResources() {
   TextureRegionFactory.setAssetBasePath("gfx/Wallpaper/");
   mBackgroundTexture = new Texture(512, 512,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mBackgroundTextureRegion =
      TextureRegionFactory.createFromAsset(
      this.mBackgroundTexture, getApplicationContext(),
      "V3Wallpaper.png", 0, 0);
   mEngine.getTextureManager().loadTexture(
      this.mBackgroundTexture);
   mScrumTexture = new Texture(512, 256,
      TextureOptions.DEFAULT);
   mScrumTextureRegion =
      TextureRegionFactory.createTiledFromAsset(
      this.mScrumTexture, getApplicationContext(),
      "scrum_tiled.png", 0, 0, 8, 4);
```

```
    mEngine.getTextureManager().loadTexture(
        this.mScrumTexture);
    this.getEngine().getTextureManager().loadTexture(
        this.mScrumTexture);
}

@Override
public Scene onLoadScene() {
    final Scene scene = new Scene(1);

    //Load background
    Sprite bg = new Sprite(0,0,mBackgroundTextureRegion);
    scene.getLastChild().attachChild(bg);

          // Add first vampire (which will add the others)
          nVamp = 0;
    mHandler.postDelayed(mStartVamp,3000);

    scene.registerUpdateHandler(new IUpdateHandler() {
        @Override
        public void reset() { }

        @Override
        public void onUpdate(final float pSecondsElapsed) {
          for (int i=0; i<nVamp; i++){
           if (asprVamp[i].getX() < 30.0f){
           //move vampire back to right side of screen
                   float startY =
              gen.nextFloat()*
              (CAMERA_HEIGHT - 50.0f);
                   asprVamp[i].clearEntityModifiers();
                   asprVamp[i].registerEntityModifier(
                     new MoveModifier(40.0f,
                  CAMERA_WIDTH - 30.0f, 0.0f,
                  startY, 340.0f)
                     );
           }
          }
        }
    });

    return scene;
}

// ===========================================================
// Methods
// ===========================================================
 private Runnable mStartVamp = new Runnable() {
     public void run() {
         int i = nVamp;
        Scene scene = LiveWallpaperService.this.mEngine.getScene();
```

```
            float startY = gen.nextFloat()*(CAMERA_HEIGHT - 50.0f);
            asprVamp[i] = new AnimatedSprite(CAMERA_WIDTH - 30.0f,
        startY, mScrumTextureRegion.clone()) ;
            nVamp++;
         scene.registerTouchArea(asprVamp[i]);
          final long[] frameDurations = new long[26];
          Arrays.fill(frameDurations, 500);
              asprVamp[i].animate(frameDurations, 0, 25, true);
            asprVamp[i].registerEntityModifier(
                  new SequenceEntityModifier (
                    new AlphaModifier(5.0f, 0.0f, 1.0f),
                   new MoveModifier(60.0f,
             asprVamp[i].getX(), 0.0f, startY,
             340.f)
                  ));
            scene.getLastChild().attachChild(asprVamp[i]);
          if (nVamp < MAX_VAMPS){
            mHandler.postDelayed(mStartVamp,VAMP_RATE);
          }
       }. . .
   @Override
   public void onGamePaused() {
      mHandler.removeCallbacks(mStartVamp);
   }

   @Override
   public void onGameResumed() {
      mHandler.postDelayed(mStartVamp,VAMP_RATE);
   }

. . .
```

If you've been following the coding so far in the book, you can see that what we have done in Listing 15.1 is to lift and adapt code from the `Level1Activity.java` game and put it inside a class that extends BaseLiveWallpaperService. We don't try to detect collisions or create paths; instead, we simply let the vampires walk from right to left. When they reach the left side of the screen, the update handler repositions them to the right side of the screen.

Most of the changes to the code are intended to accommodate the difference in screen orientation. Unlike in our games, wallpapers usually run in portrait orientation (for some phones and most Android tablets using Honeycomb or later, however, you'll have to provide for the wallpaper shifting between the portrait and landscape orientations as the screen is reoriented). It's just a small matter of adjusting the graphics and coordinates used to match that orientation.

The manifest also needs some changes. The manifest for our V3Wallpaper is shown in Listing 15.2. Note particularly the need for `BIND_WALLPAPER` permission and the intent filter for `android.service.wallpaper.WallpaperService`.

Listing 15.2 **AndroidManifest.xml for V3Wallpaper**

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.pearson.lagp.v3livewallpaper"
      android:versionCode="4"
      android:versionName="1.0">

   <uses-sdk android:minSdkVersion="7" />

   <uses-feature android:name="android.software.live_wallpaper" />
    <uses-permission android:name=
         "android.permission.WRITE_EXTERNAL_STORAGE" />

    <application android:icon="@drawable/icon"
      android:label="@string/app_name">
      <service
         android:name="LiveWallpaperService"
         android:enabled="true"
         android:icon="@drawable/icon"
         android:label="@string/service_name"
         android:permission=
            "android.permission.BIND_WALLPAPER">
         <intent-filter android:priority="1" >
             <action android:name=
             "android.service.wallpaper.WallpaperService" />
         </intent-filter>
         <meta-data
            android:name="android.service.wallpaper"
            android:resource="@xml/wallpaper" />
      </service>
    </application>

</manifest>
```

MOD Music

I was not familiar with the MOD music format before I heard about the MOD Player extension to AndEngine. It turns out there is a whole subculture of digital artists who create "demos," which are noncommercial artistic demonstrations of what computers are capable of showing and playing. These artists compete to develop demos that are the most artistic and best show off the capabilities of the device they run on. One of

their favorite audio file formats is called MOD, which derives from the format of the same name that was popular on Amiga computers in the 1980s. The demos are created and played with programs called "trackers" or "modplayers," and these programs are available for Android devices.

Finding MOD Music

Many of the artists creating demos are part of communities. One of the more popular communities seems to be Demoscene, which you can check out at `www.demoscene.info`. The related archive of MOD files can be found at the following URL:

```
http://modarchive.org
```

Hundreds of megabytes of demos are available, ready to play on an appropriate tracker. The audio from many older games is also available in MOD or one of the other demo formats that are playable with the extension.

You must be careful if you decide to include these files in a commercial game. The licenses for most of the MOD files say they are licensed for your personal reuse and redistribution; for commercial use (such as inclusion in a game), however, you must obtain the explicit permission of the author. Some MOD files explain how to contact the author in the information that is displayed by players such as XMP (see the next section). Others do not, or the information is out-of-date. A very good discussion of the rights granted to downloads from modarchive can be found at this URL:

```
http://modarchive.org/index.php? faq-licensing
```

XMP MOD Player

The AndEngine extension for MOD files uses the player engine from a MOD player called XMP. XMP is available for free on Android Market, and it's worth installing it to get an idea of the many demo files available. Figure 15.3 shows the screen for XMP as its playing a demo; the bar graph in the middle of this screen is an animated frequency display.

XMP is actually much more than a MOD player. It can read and play more than 90 different audio file formats, including many of the formats that were used in the original electronic games. XMP does the work of decoding and playing the format. The AndEngine extension allows our games to access the XMP player via the Native Development Kit and to direct what it is doing. You don't have to install the XMP player on the target device, as the XMP code is included in the extension `.jar` library. Although MOD files are freely distributed on a number of Internet archives, that fact does not necessarily give you the right to use them in your game. As with all other intellectual property, you need to read the license agreement that is available with the file to determine whether you can include the file in a commercial game.

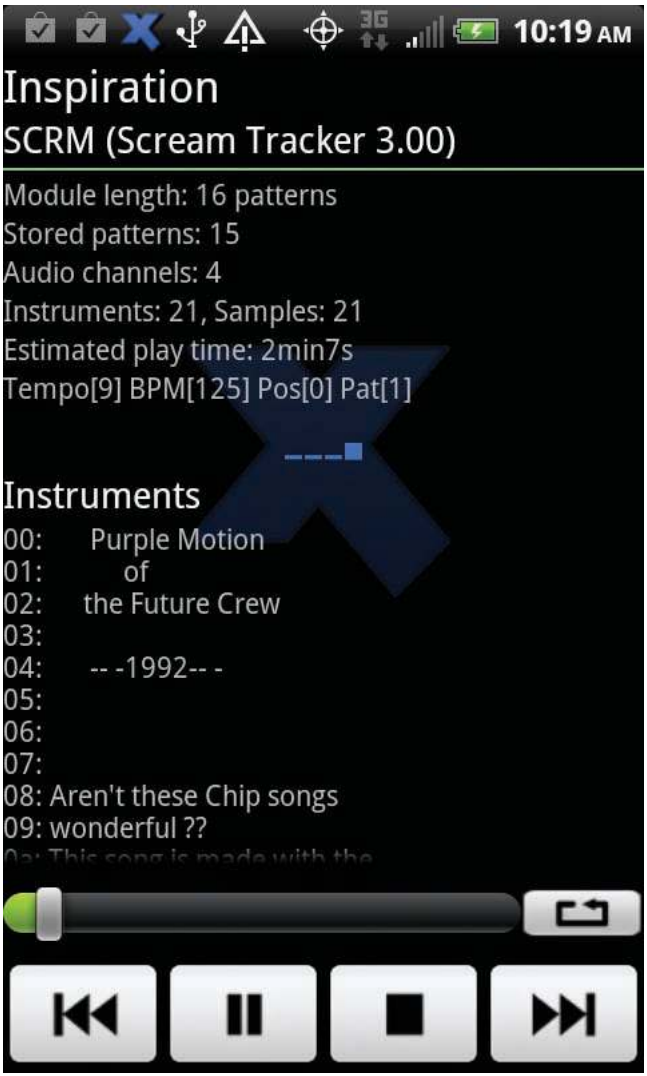


Figure 15.3 XMP screen

An interesting wrinkle is that XMP expects to find the sound files on the SD card of the Android device. If you have a `.mod` file in your `assets` folder, how do you get it to the SD card so that XMP can find it? Fortunately, Nicolas has provided some file utility functions to help with just such a case and a MOD example program to show us how to use them. The example source is located at this URL:

`http://code.google.com/p/andengineexamples/source/browse/src/org/anddev/andengine/examples/ModPlayerExample.java`

Listing 15.3 is a simplified version of the file relocation code in that example.

Listing 15.3 **Moving a File to SD Storage**

```
. . .
   private final ModPlayer mModPlayer = ModPlayer.getInstance();
. . .
   if(FileUtils.isFileExistingOnExternalStorage(this,
         "mfx/8bit.mod")) {
      this.startPlayingMod();
   } else {
      this.doAsync(R.titleResouceID, R.messageResourceID,
         new Callable<Void>() {
           @Override
           public Void call() throws Exception {
         FileUtils.ensureDirectoriesExistOnExternalStorage(
            ModPlayerExample.this, "mfx/");
         FileUtils.copyToExternalStorage(
            ModPlayerExample.this, "mfx/8bit.mod",
               "mfx/8bit.mod");
         return null;
           }
        }, new Callback<Void>() {
           @Override
           public void onCallback(final Void pCallbackValue) {
         ModPlayerExample.this.startPlayingMod();
      }
   });
. . .
   // ===========================================================
   // Methods
   // ===========================================================

   private void startPlayingMod() {
    this.mModPlayer.play( FileUtils.getAbsolutePathOnExternalStorage(
      this, "mfx/8bit.mod));
   }
. . .
```

If the file already exists on the SD card, the code just calls `startPlayingMod()` to pass the absolute filename to XMP via the ModPlayer object. Otherwise, in code that runs asynchronously to the game, the utilities make sure the target directory exists and then copy the file from `assets/mfx` to the target directory. When the copy operation is complete, the `startPlayingMod()` method is invoked to pass the filename to XMP.

Multiplayer Games

Another AndEngine extension gives us the basic capability for communicating between multiple Android devices for multiplayer games. The communication approach taken by the extension is to use stream sockets to create a client/server relationship between devices. If you're not familiar with sockets, the idea originated with TCP/IP communication, and you can think of a socket as a virtual numbered port to a device. One way sockets can be used is to exchange streams of data. The devices agree on a set of port numbers to use ("well-known" port numbers exist for common protocols such as POP3 and SMTP for email), and then send and receive messages to and from these ports.

With this multiplayer extension, one Android device takes the role of server, and the other assumes the role of client. The two devices can then freely exchange messages over Wi-Fi or WAN (3G, 3G, EDGE, GPRS), or even Bluetooth.

Nicolas has written an excellent example program showing the messaging capability, which you can find at the following location:

```
http://code.google.com/p/andengineexamples/source/browse/src/org/anddev/
andengine/examples/MultiplayerExample.java
```

This example is set up to send messages back and forth between one Android device that declares itself to be the client, and another that declares itself to be the server. In the version of the example that was available as this book was being written, the IP addresses had to be entered manually. This requirement poses a problem when using the extension, as the IP addresses for the other players must either be fixed (so they can be written into the program) or entered by hand. Right now, the plan is to improve the extension so it can perform address discovery, which will make it easier to use. This fix may have been made by the time you read these words.

Reading or writing messages from or to a socket is something that has to be done in a thread separate from your main UI thread. You don't want the UI to freeze while you're waiting for messages to be exchanged. The extension handles this requirement, as you can see in the source and example code, by creating a SocketServer that does the actual communication, connected to your game through callbacks.

Finding Your IP Address on an Android Device

If you need to find the IP address of your device, either for the multiplayer extension or for some other reason, it's easy. On the Android device, go to Settings > Wi-Fi Settings and touch the name of the network where you have a current connection. A small dialog box will display the IP address and other information, such as the channel being used and the MAC address.

Figure 15.4 shows a screenshot from the example program. Once you set up the client/server connection, a face will show up wherever you touch the screen on the server device. The server sends a message to the client, and the same face shows up in the same spot on the client screen.

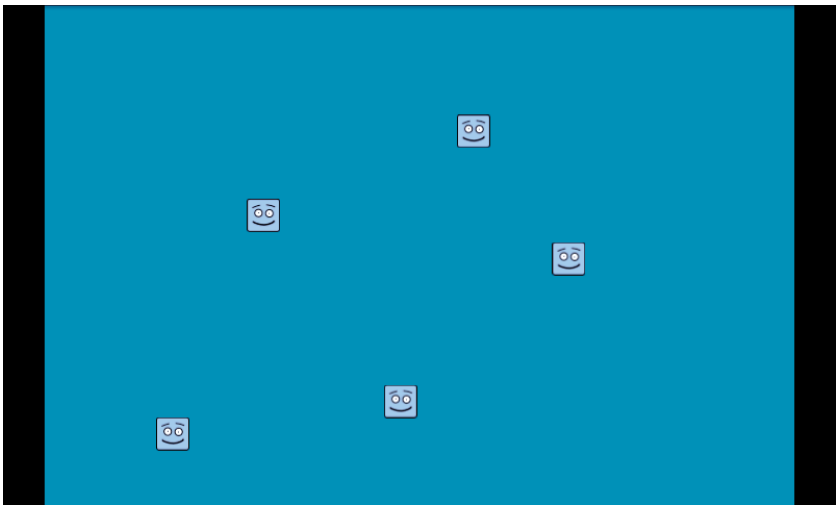


Figure 15.4 Multiplayer screen

Multi-Touch in AndEngine

The AndEngine multi-touch extension makes it possible to collect and make use of multi-touch gestures in games. Of course, it can perform this feat only if the Android device it is running on supports multi-touch capabilities. Unfortunately, there is no magic that can circumvent this requirement. Most Android devices running version 2.0 (Éclair) or later will support multi-touch.

The AndEngine extensions greatly simplify the use of multi-touch in games. As mentioned in Chapter 8 in the discussion of touch inputs, Android deals with multi-touch cases by returning additional simultaneous touch events with `ACTION_POINTER_DOWN`, `ACTION_POINTER_MOVE`, or `ACTION_POINTER_UP` as the action parameter. Android programs can then get details from the event:

- `int getPointerCount()`: returns the number of active pointers (fingers on the screen)
- `float getX(int pointerIndex)`: analogous to the `getX()` method in the single-touch case
- `float getY(int pointerIndex)`: analogous to `getY()` in the single-touch case

The pointer index is not guaranteed to remain consistent from event to event (e.g., pointer index 1 in one event may or may not refer to the same finger as pointer 1 in the next event). Pointer IDs are provided to ensure that continuity, and many other bits of information are available from the events, but the key point is that with this extension, you don't have to worry about any of those issues.

Listing 15.4 was adapted from the example activity `MultiTouchExample.java` written by Nicolas Gramlich.

Listing 15.4 **Multi-Touch Example Code: Part 1**

```
. . .
   @Override
   public Engine onLoadEngine() {
. . .
      try {
         if(MultiTouch.isSupported(this)) {
            engine.setTouchController(
               new MultiTouchController());
            if(MultiTouch.isSupportedDistinct(this)) {
               mMultiTouchDistinct = true;
            } else {
               mMultiTouchDistinct = false;
            }
         } else {
            Toast.makeText(this,
            "Sorry your device does NOT support " +
            "MultiTouch!\n\n(Falling back to " +
            "SingleTouch.)", Toast.LENGTH_LONG).show();
         }
      } catch (final MultiTouchException e) {
         Toast.makeText(this, "Sorry your Android " +
         "Version does NOT support MultiTouch!\n\n" +
         "(Falling back to SingleTouch.)",
         Toast.LENGTH_LONG).show();
      }

      return engine;
   }
. . .
```

First, assume the library file for the multi-touch extension, `andenginemultitouch-extension.jar`, has been loaded into the project in the same way we loaded the other libraries. The try-catch statement in `onLoadEngine()` tests whether multi-touch capabilities are available both on the device and on the version of Android that is running. MultiTouch is a new class that takes care of testing the underlying hardware and Android version for multi-touch support. Two types of multi-touch capabilities are currently available in Android devices:

- One type of multi-touch supports simple two-finger gestures, such as the pinch gesture for zooming, but does not track multiple finger moves.
- "Distinct" multi-touch tracks each finger's moves separately.

The code continues in Listing 15.5. In this listing, the `onAreaTouched` code for the Sprite is almost exactly like the code that we saw earlier for single-touch events. The only difference is that we have added the `mGrabbed boolean` to the Sprite definition, which allows us to track whether this Sprite has been "grabbed" with an `ACTION_DOWN`. For the case where you just want to be able to move multiple Sprites using multiple touches, using this extension is a lot easier than working with the Android pointer indexes and pointer IDs.

Listing 15.5 **Multi-Touch Example Code: Part 2**

```
. . .
   final Sprite sprite = new Sprite(pX, pY,
      this.mTextureRegion) {
      boolean mGrabbed = false;

      @Override
      public boolean onAreaTouched(final TouchEvent
         pSceneTouchEvent,
         final float pTouchAreaLocalX,
         final float pTouchAreaLocalY) {
         switch(pSceneTouchEvent.getAction()) {
            case TouchEvent.ACTION_DOWN:
               this.mGrabbed = true;
               break;
            case TouchEvent.ACTION_MOVE:
               if(this.mGrabbed) {
                  this.setPosition(
         pSceneTouchEvent.getX() - Card.CARD_WIDTH / 2,
         pSceneTouchEvent.getY() - Card.CARD_HEIGHT / 2);
               }
               break;
            case TouchEvent.ACTION_UP:
               if(this.mGrabbed) {
                  this.mGrabbed = false
                  this.setScale(1.0f);
               }
               break;
         }
         return true;
      }
```

Augmented Reality

The term "augmented reality" has come to mean several different things with respect to mobile games. In our case, it refers to games that superimpose computer-generated graphics over the view seen through the device camera preview.

Figure 15.5 Augmented reality—vampires in my backyard

Figure 15.5 shows a simple augmented reality screen in which vampires walk across my backyard.

This AndEngine extension adds two new classes when we include the `augmentedrealityextension.jar` class into our Android game project:

- BaseAugmentedRealityGameActivity: extends BaseGameActivity to include the SurfaceView for the device camera preview
- CameraPreviewSurfaceView: creates the SurfaceView itself

Using the extension couldn't be easier. The essentials of the code that produced vampires walking across my backyard are shown in Listing 15.6.

Listing 15.6 **Augmented Reality Example Code**

```
package com.pearson.lagp.vinb;

. . .

public class VampiresInBackyard extends BaseAugmentedRealityGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================
```

```
private static final int CAMERA_WIDTH = 480;
private static final int CAMERA_HEIGHT = 320;
private static final int VAMP_RATE = 2000;
private static final int MAX_VAMPS = 10;

// ===========================================================
// Fields
// ===========================================================

private Camera mCamera;
private Handler mHandler;
private Texture mScrumTexture;
private TiledTextureRegion mScrumTextureRegion;

private AnimatedSprite[] asprVamp = new AnimatedSprite[10];
private int nVamp;

private Random gen;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
  Toast.makeText(this, "If you don't see a vampire moving " +
  "over the screen, try starting this while already being in " +
  "Landscape orientation!!", Toast.LENGTH_LONG).show();
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   mHandler = new Handler();
   gen = new Random();
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
      CAMERA_HEIGHT), this.mCamera));
}
```

```
    @Override
    public void onLoadResources() {
          TextureRegionFactory.setAssetBasePath("gfx/VinB/");
          mScrumTexture = new Texture(512, 256,
             TextureOptions.DEFAULT);
          mScrumTextureRegion =
             TextureRegionFactory.createTiledFromAsset(
                this.mScrumTexture,
                getApplicationContext(),
                "scrum_tiled.png", 0, 0, 8, 4);
          mEngine.getTextureManager().loadTexture(
             this.mScrumTexture);
          this.getEngine().getTextureManager().loadTexture(
             this.mScrumTexture);
    }

    @Override
    public Scene onLoadScene() {
       final Scene scene = new Scene(1);
       scene.setBackground(
          new ColorBackground(0.0f, 0.0f, 0.0f, 0.0f));

           // Add first vampire (which will add the others)
           nVamp = 0;
       mHandler.postDelayed(mStartVamp,3000);

       scene.registerUpdateHandler(new IUpdateHandler() {
          @Override
          public void reset() { }

          @Override
          public void onUpdate(final float pSecondsElapsed) {
             for (int i=0; i<nVamp; i++){
                if (asprVamp[i].getX() < 30.0f){
                  //Move vampire back to right
                        float startY = gen.nextFloat()*
(CAMERA_HEIGHT - 50.0f);
                        asprVamp[i].clearEntityModifiers();
                        asprVamp[i].registerEntityModifier(
                          new MoveModifier(40.0f,
                   CAMERA_WIDTH - 30.0f, 0.0f,
                   startY, 340.0f)
                        );
                }
             }
          }
       });
       return scene;
    }
```

```
    // ============================================================
    // Methods
    // ============================================================
     private Runnable mStartVamp = new Runnable() {
         public void run() {
            int i = nVamp;
            Scene scene = VampiresInBackyard.this.mEngine.getScene();
               float startY = gen.nextFloat()*(CAMERA_HEIGHT - 50.0f);
               asprVamp[i] = new AnimatedSprite(CAMERA_WIDTH - 30.0f,
           startY, mScrumTextureRegion.clone()) ;
               nVamp++;
            scene.registerTouchArea(asprVamp[i]);
             final long[] frameDurations = new long[26];
             Arrays.fill(frameDurations, 500);
                 asprVamp[i].animate(frameDurations, 0, 25, true);
               asprVamp[i].registerEntityModifier(
                  new SequenceEntityModifier (
                     new AlphaModifier(5.0f, 0.0f, 1.0f),
                      new MoveModifier(40.0f,
               CAMERA_WIDTH - 30.0f, 0.0f, startY, 340.f)
                  ));
               scene.getLastChild().attachChild(asprVamp[i]);
            if (nVamp < MAX_VAMPS){
               mHandler.postDelayed(mStartVamp,VAMP_RATE);
            }
         }
      };

}
```

Most of this code was grabbed with a cut-and-paste operation from `V3LiveWallpaper.java` (which in turn gets a lot of the code from `Level1Activity.java`). Thus, when you build a game activity that extends BaseAugmentedRealityGameActivity instead of BaseGameActivity, you automatically see the device camera preview as the background of your Scene. You can layer whatever graphics you want on top of that preview to create your own augmented reality application.

Summary

This chapter reviewed the extensions available for AndEngine. As it happens, they are all related to multimedia in one way or another. As AndEngine continues to grow and improve, many other extensions will undoubtedly emerge that give us more ways to make our games fun and interactive.

New AndEngine extensions are usually announced in the AndEngine forum:

```
http://www.andengine.org/forums/extensions/
```

The source and binaries for extensions are always available from the AndEngine github site:

```
http://code.google.com/p/andengine/
```

Exercises

1. Modify `V3LiveWallpaper.java` so that the vampires disappear in an explosion when they get to the left side of the screen, as shown in Figure 15.6.

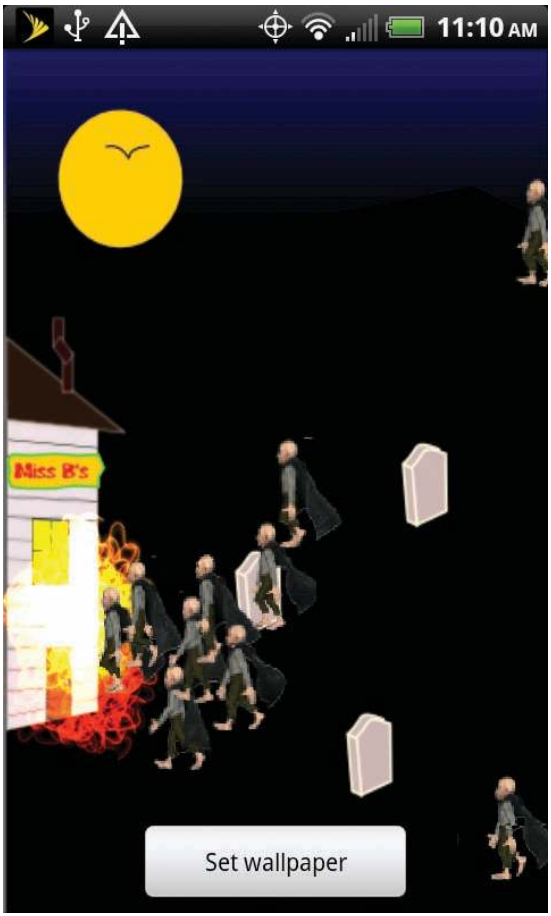


Figure 15.6 V3LiveWallpaper with explosions

2. Right now, V3's Whack-A-Vampire gamelet doesn't play any music. Modify `WAVActivity.java` so it plays a MOD file of your choice.
3. Modify `VampiresInBackyard.java` to indicate the current compass direction at the top of the screen, as shown in Figure 15.7.

Figure 15.7 Augmented reality: vampires in my backyard with compass direction (at the top of the screen)

This page intentionally left blank

16

Game Integration

At this point we've covered the basic elements of the AndEngine game engine. We've discussed and demonstrated the following topics:

- The basic game loop
- Building and displaying menus
- Creating scenes and making transitions between them
- Creating sprites and attaching modifiers
- Animating sprites
- Drawing text using fonts
- Getting input from the user
- Using tile maps as a game canvas
- Building and displaying particle effects
- Playing music and sound effects
- Using the Box2D physics engine
- Applying artificial intelligence techniques
- Tracking scores so players can measure their play
- Multimedia extensions

In the process, we created a fair bit of code around a game we've called V3, or Vampires Versus Virgins. We have also left a number of loose ends as we created that example game, however. Now we'd like to bring those ends together to create a better-integrated, more playable game.

This chapter discusses the following areas:

- Difficulty balancing: We need to be able to adjust the difficulty of each gamelet so that each is difficult enough to be challenging, yet easy enough to win. Each gamelet has its own set of adjustable parameters, and we'd like those parameters to be persistent across game-playing sessions.
- Gamelet completion: When we implemented scoring in Chapter 14, we created end Scenes for each gamelet, telling the player his or her score and showing the

five highest scores among all players. We also provided three buttons on those screens that we said we would implement later (specifically, now):

- Again: meaning "I'd like to play this same gamelet again."
- Quit: meaning "I'd like to quit the game entirely."
- Next: meaning "I'd like to play the next gamelet."

We'll devote a section to each of the generic topics of difficulty balancing and completion, and then we'll have sections for the specific changes in each gamelet.

Difficulty Balancing

Difficulty balancing is a function that can make or break your game in terms of its attractiveness to players. If a game is too easy, players will get bored very quickly and not play anymore. If a game is too difficult, players will get frustrated and stop playing. You have to get the balance just right.

To make matters worse, players usually get better as they play, so the balance point changes over time. No matter which scheme we use to adjust difficulty, we need to be able to have the level of difficulty persist across game-playing sessions. Once players get good at a game, they don't want to have to navigate the "bunny slopes" every time they play.

Difficulty Parameter Storage

We want to be able to adjust the difficulty of each gamelet, and to maintain that adjustment from run to run. We could put the parameters in a SQLite database, which may be the appropriate solution for complex games, but doing so is overkill for our game given the few parameters we need to store.

Instead, we will turn to our old friend SharedPreferences, and create a new preferences file with the name "difficulty." Each gamelet will be responsible for storing and retrieving its own parameters, and Android will take care of preserving the values between sessions.

Listing 16.1 shows how we'll handle the SharedPreferences in each gamelet.

Listing 16.1 **Difficulty SharedPreferences Generic Code**

```
. . .
   difficulty = getSharedPreferences("difficulty", MODE_PRIVATE);
   diffEditor = difficulty.edit();
   mParam1 = difficulty.getInt("GAMELET.PARAM1", 1000);
   mParam2 = difficulty.getInt("GAMELET.PARAM2", 1);
   mWins = difficulty.getInt("GAMELET.WINS", 0);
   mPlays = difficulty.getInt("GAMELET.PLAYS", 0);
. . .
    private void mSaveDifficulty() {
```

```
        diffEditor.putInt("GAMELET.PARAM1", mParam1);
        diffEditor.putInt("GAMELET.PARAM2", mParam2);
        diffEditor.putInt("GAMELET.WINS", mWins);
        diffEditor.putInt("GAMELET.PLAYS", mPlays);
    }
. . .
```

At the beginning of the gamelet [in `onLoadEngine()`], we'll pull the parameters out of the SharedPreferences. In the string names, `GAMELET` will be replaced with the name of the gamelet, and `PARAMn` with the name of the parameter. At the end of each game, we'll save the (possibly updated) parameters back to the SharedPreferences as shown in the `mSaveDifficulty()` method.

Difficulty Parameter Setting

Once we have the parameters available to adjust the difficulty, how do we know where to set them? The answer is simple, but you can spend a career implementing it properly. You have to test your game with real players. You can then use their actual game-playing results to properly set the difficulty level of your game. After gathering and analyzing these data, you can make the difficulty adjustments at several different levels of integration:

- Test the game with as many target players as possible, and have them note the results of their play. Correlate the results and set the degree of difficulty based on the "sweet spot" of the feedback.
- Adjust the difficulty as the game is played. You still have to come up with the initial difficulty settings, but the game should then adjust them based on the number of wins and losses as the player tries the game. This is the approach we will use for V3.
- Adjust the settings remotely, based on automatic feedback from a range of players. Almost all Android devices are connected to the Internet, and you can use that connection to gather information from the player's runs of the game. You can use the same connection to adjust the difficulty settings of the game. Data gathering and remote manipulation require the player to grant permission allowing you to access and tinker with the player's game, but this can be a powerful technique, particularly in social games.

To demonstrate our chosen strategy, we will track the total number of game plays and the total number of losses in our SharedPreferences. At the end of a game, we can use that information, along with whether the player just won, to adjust the difficulty of the gamelet for the next play. You might also consider data such as "recent losses in a row," although wins and losses are sufficient to demonstrate the idea. For some games, you might also want to register accelerometer feedback to note how many times a player threw the Android device across the room in frustration (that's a joke).

The code in Listing 16.1 shows the wins and plays as tracked in the SharedPreferences (losses are obviously just the difference between plays and wins). We might have made these values bigger than an integer, but the integer type allows for a play to take place every second for about 68 years, so it should be big enough.

Each gamelet will include a method called `mIncreaseDifficulty()` that will bump up that gamelet's difficulty by a set amount. We can make that increase as complicated as we like, but for our purposes, we'll keep it pretty simple. The details for each gamelet are given in the gamelet sections later in this chapter.

Completion

To be able to respond to the user pressing the Again button, we need to be able to restart our game. There are at least two ways we could do so:

- Cancel any outstanding delayed Runnables, reinitialize any counters used to keep track of the game's current status, and call the method that starts things rolling.
- Post a Runnable that starts our Activity again, and call `finish()`, so the current Activity goes away.

The first method probably makes the best use of computing resources. The textures, particle effects, and levels don't have to be reloaded, and many of the objects don't have to be re-created; instead, we just have to reinitialize everything properly. The second method is easier, as the reinitialization is handled for us by the Runnable.

We'll use the second method in the examples in this chapter, and leave the first method for you to implement in the exercises at the end of the chapter. You can always look at the code in the Exercise Solutions Appendix to see the first method.

Listing 16.2 shows the generic completion code that we will adapt for all of the gamelets in V3.

Listing 16.2 **Gamelet Completion Generic Code**

```
. . .
   againButton = new Sprite(0.0f, 0.0f, mAgainButtonTextureRegion){
          @Override
          public boolean onAreaTouched(final TouchEvent pAreaTouchEvent,
      final float pTouchAreaLocalX,
      final float pTouchAreaLocalY) {
             switch(pAreaTouchEvent.getAction()) {
             case TouchEvent.ACTION_DOWN:
                mEndCleanup();
                mHandler.post(mPlayThis);
                finish();
                break;
             }
```

```
            return true;
         }
   };
   nextButton = new Sprite(0.0f, 0.0f, mNextButtonTextureRegion){
         @Override
         public boolean onAreaTouched(final TouchEvent pAreaTouchEvent,
      final float pTouchAreaLocalX,
      final float pTouchAreaLocalY) {
            switch(pAreaTouchEvent.getAction()) {
            case TouchEvent.ACTION_DOWN:
               mEndCleanup();
               mHandler.post(mPlayNext);
               finish();
               break;
            }
            return true;
         }
   };
   quitButton = new Sprite(0.0f, 0.0f, mQuitButtonTextureRegion){
         @Override
         public boolean onAreaTouched(final TouchEvent pAreaTouchEvent,
      final float pTouchAreaLocalX,
      final float pTouchAreaLocalY) {
            switch(pAreaTouchEvent.getAction()) {
            case TouchEvent.ACTION_DOWN:
               mEndCleanup();
               finish();
               break;
            }
            return true;
         }
   };
. . .
    private void mEndCleanup() {
   mPlays++;
      if (mPlayerWon) {
         mIncreaseDifficulty();
         mWins++;
      }
      mSaveDifficulty();
    }

    private Runnable mPlayThis = new Runnable() {
         public void run() {
           Intent myIntent = new Intent(ThisActivity.this,
         ThisActivity.class);
           ThisActivity.this.startActivity(myIntent);
```

```
        }
    };

    private Runnable mPlayNext = new Runnable() {
        public void run() {
            Intent myIntent = new Intent(ThisActivity.this,
        NextActivity.class);
            ThisActivity.this.startActivity(myIntent);
        }
    };
```

The `onAreaTouched()` case for `ACTION_DOWN` for each button is similar:

- Some common end-of-game processing is done in the method `mEndCleanup()`. Alternatively, we could have put this code in the `mGameOver()` method, but this approach keeps things cleaner.
- If either this Activity or another is to run next, a Runnable is posted to start it.
- The `finish()` method is called to stop this Activity.

The `mEndCleanup()` method performs three tasks:

- It increments the number of plays.
- If the player won this gamelet, it bumps the difficulty by calling `mIncrease-Difficulty()` (see the previous section on difficulty balancing for an explanation of what goes on in that method) and increments the number of wins.
- The difficulty parameters are saved back to SharedPreferences.

The `mPlayThis` Runnable restarts the current Activity (`ThisActivity` in the code in Listing 16.2 is replaced with the actual name of the current Activity). The `mPlayNext` Runnable starts the gamelet that should run next.

Level 1: The Main Game

Two difficulty parameters are currently coded as constants for the main (Level 1) gamelet. We'll make them into adjustable variables stored as SharedPreferences and add a third parameter that can contribute a distraction:

- `mMaxVamps`: This has been the constant `MAX_VAMPS` up until now. It is just the maximum number of vampires the gamelet will put on the screen at any one time.
- `mVampRate`: This is the maximum time between launches of a new vampire. It was formerly the constant `VAMP_RATE`.

Figure 16.1 Level 1 with distracting virgin in Miss B's window

- `mDistract`: This is a new boolean type that indicates whether distractions should be enabled. We can make the distractions as complicated as we wish, but for V3 this variable will be an animated virgin calling for help periodically from Miss B's window, as shown in Figure 16.1.

The major changes to `Level1Activity.java` are shown in Listing 16.3.

Listing 16.3 **Completion and Difficulty Balance of Level1Activity.java**

```
package com.pearson.lagp.v3;

+imports

public class Level1Activity extends BaseGameActivity {
    // ===========================================================
    // Fields
    // ===========================================================
. . .
    private Texture mSarahTexture;
. . .
    private TiledTextureRegion mSarahTextureRegion;
. . .
    private AnimatedSprite[] asprVamp = new AnimatedSprite[40];
    private AnimatedSprite asprSarah;
```

```
. . .
   private Sound mSaveMeSound;
. . .
   private SharedPreferences difficulty;
   private SharedPreferences.Editor diffEditor;
. . .
   private AStar[] aStar = new AStar[40];
   private Path[] pathVamp = new Path[40];
   private int mWins, mPlays;
   private int mMaxVamps;
   private int mVampRate;
   private boolean mDistract;
   private boolean mPlayerWon;
   private boolean mActivityVisible = true;

. . .
   @Override
   public Engine onLoadEngine() {
. . .
      difficulty = getSharedPreferences("difficulty",
         MODE_PRIVATE);
      diffEditor = difficulty.edit();
      mMaxVamps = difficulty.getInt("Lvl1.MAX_VAMPS", 10);
      mVampRate = difficulty.getInt("Lvl1.VAMP_RATE", 4000);
      mDistract = difficulty.getBoolean("Lvl1.DISTRACT", true);
      mWins = difficulty.getInt("Lvl1.WINS", 0);
      mPlays = difficulty.getInt("Lvl1.PLAYS", 0);

. . .
   @Override
   public void onLoadResources() {
. . .
      mSarahTexture = new Texture(256, 64,
         TextureOptions.DEFAULT);
      mSarahTextureRegion =
         TextureRegionFactory.createTiledFromAsset(
this.mSarahTexture, getApplicationContext(),
            "sarahanim.png", 0, 0, 6, 1);
      mEngine.getTextureManager().loadTexture(
this.mSarahTexture);

. . .
      SoundFactory.setAssetBasePath("mfx/");
      try {
. . .
        this.mSaveMeSound = SoundFactory.createSoundFromAsset(
         this.mEngine.getSoundManager(),
```

```
        getApplicationContext(), "saveme.ogg");
. . .
   @Override
   public Scene onLoadScene() {
. . .

      // If distractions are enabled, start the first one
      if (mDistract) {
         mHandler.postDelayed(mStartSarah, 5000);
      }

. . .
//EndScene buttons updated as per Listing 16.2
. . .
             asprSarah = new AnimatedSprite(15.0f, 90.0f,
         mSarahTextureRegion);
             asprSarah.setVisible(false);
           scene.getLastChild().attachChild(asprSarah);
      return scene;
   }

. . .
   @Override
   public void onGamePaused() {
. . .
      mSaveMeSound.stop();
      mActivityVisible = false;
   }

   @Override
   public void onGameResumed() {
      super.onGameResumed();
      mActivityVisible = true;
   }
. . .
     private Runnable mStartSarah = new Runnable() {
         public void run() {
            final long[] frameDurations = new long[6];
            Arrays.fill(frameDurations, 200);
            asprSarah.setVisible(true);
            asprSarah.animate(frameDurations, 0, 5, false);
                 playSound(mSaveMeSound);
            mHandler.postDelayed(mStartSarah,
         (long)(gen.nextFloat()*7000.0f + 8000.0f));
            mHandler.postDelayed(mEndSarah, 2000);
         }
      };
```

```
      private Runnable mEndSarah = new Runnable() {
          public void run() {
             asprSarah.setVisible(false);
          }
      };

. . .
     private void playSound (Sound mSound){
        if ((audioOptions.getBoolean("effectsOn", false)) &&
              (mActivityVisible)){
           mSound.play();
        }
     }

. . .
     private void mIncreaseDifficulty() {
        // Make the gamelet a little harder
      if (mMaxVamps < 40) mMaxVamps += 5;
      if (mVampRate > 500) mVampRate -= 500;
      if (mWins > 5) mDistract = true;
   }

    private void mSaveDifficulty() {
       diffEditor.putInt("Lvl1.MAX_VAMPS", mMaxVamps);
       diffEditor.putInt("Lvl1.VAMP_RATE", mVampRate);
       diffEditor.putBoolean("Lvl1.DISTRACT", mDistract);
       diffEditor.putInt("Lvl1.WINS", mWins);
       diffEditor.putInt("Lvl1.PLAYS", mPlays);
    }

    private void mEndCleanup() {
            mPlays++;
               if (mPlayerWon) {
                  mIncreaseDifficulty();
                  mWins++;
               }
               mSaveDifficulty();
   }

    private Runnable mPlayThis = new Runnable() {
         public void run() {
           Intent myIntent = new Intent(Level1Activity.this,
         Level1Activity.class);
           Level1Activity.this.startActivity(myIntent);
           finish();
```

```
        }
    };

    private Runnable mPlayNext = new Runnable() {
        public void run() {
            Intent myIntent = new Intent(Level1Activity.this,
        WAVActivity.class);
            Level1Activity.this.startActivity(myIntent);
            finish();
        }
    };
}
```

The difficulty parameters are loaded and saved to the SharedPreferences, as discussed in the previous sections. The button changes are exactly as shown in Listing 16.2, so they aren't repeated here. The parameters are different, of course, but the method names—`mIncreaseDifficulty()`, `mSaveDifficulty()`—are exactly the same for all gamelets.

The virgin crying out for help is named Sarah, in case you haven't guessed. I purchased the rights to use her image from Content Paradise for $3.99.

There isn't a whole lot new in this code. We load the TiledTextureRegion for Sarah just like we did for the vampires, and we load the cry for help effect the same way we loaded the other effects. If the gamelet starts with distractions enabled, we post a Runnable to start the first Sarah, which will then wait a variable amount of time before relaunching herself. We create Sarah as an invisible Sprite, and then turn visibility (and animation) on and off as we need to show her to the player.

There is one new wrinkle in `onGamePaused()` and `onGameResumed()`. We've added a boolean variable that will tell us whether the game is the Activity that is currently visible. Based on its value, the `playSound()` method can tell whether it should play Sarah's cry. Without that kind of gate, Sarah keeps yelling for help, even when the game is no longer on the screen.

One other point to note is that we've increased the sizes for vampire-related arrays to 40 (the maximum value for `mMaxVamps`). It would be even better to make this value a constant that can be changed in one place for the whole gamelet.

The code for `mPlayThis()` and `mPlayNext()` are shown in Listing 16.3, as they are more explicit than the generic versions found in Listing 16.2. `mPlayThis()` restarts the current activity, and `mPlayNext()` starts the next one in this chain:

- Level 1
- Whack-A-Vampire
- Irate Villagers

Whack-A-Vampire

The three difficulty parameters that are currently constants in Whack-A-Vampire are summarized here. Again we make them into variables, so we can adjust them and store them in the "difficulty" SharedPreferences:

- `mOpenRate`: Formerly the constant `OPEN_RATE`, this variable is the maximum number of milliseconds between coffin openings. Increasing this rate makes the gamelet easier, while reducing it makes the gamelet harder.
- `mStayOpen`: Formerly the constant `STAY_OPEN`, this variable is the maximum number of seconds a coffin will stay open. Again, increasing this parameter makes the gamelet easier, while reducing it makes play more difficult.
- `mOpensPerGame`: Formerly the constant `OPENS_PER_GAME`, this variable dictates the number of coffins that are opened in one run of the game. Opening more coffins makes the game harder (because you have to concentrate longer), while opening fewer coffins makes it easier.

Listing 16.4 shows the changes needed to complete and add some difficulty balancing to the Whack-A-Vampire gamelet in `WAVActivity.java`:

Listing 16.4 **Completion and Difficulty Balance of WAVActivity.java**

```
package com.pearson.lagp.v3;

+imports

public class WAVActivity extends BaseGameActivity {
. . .

   // ============================================================
   // Fields
   // ============================================================

. . .
   private int mOpenRate;
   private int mOpensPerGame;
   private int mStayOpen;
   private int mWins, mPlays;
   private boolean mPlayerWon;

. . .
   // ============================================================
   // Methods for/from SuperClass/Interfaces
   // ============================================================
```

```
   @Override
   public Engine onLoadEngine() {
. . .
      difficulty = getSharedPreferences("difficulty",
         MODE_PRIVATE);
      diffEditor = difficulty.edit();
      mOpenRate = difficulty.getInt("WhAV.OPEN_RATE", 4000);
      mStayOpen = difficulty.getInt("WhAV.STAY_OPEN", 2000);
      mOpensPerGame = difficulty.getInt("WhAV.OPENS_PER_GAME",
         10);
      mWins = difficulty.getInt("WhAV.WINS", 0);
      mPlays = difficulty.getInt("WhAV.PLAYS", 0);
. . .
// EndScene buttons changes as in Listing 16.2
. . .
   private Runnable openCoffin = new Runnable() {
        public void run() {
. . .
          mHandler.postDelayed(openCoffin, openTime);
          mHandler.postDelayed(closeCoffin, openTime+mStayOpen);
        }
     };

    private Runnable closeCoffin = new Runnable() {
        public void run() {
. . .
         if (++mNumClosed > mOpensPerGame)
            mGameOver(PLAYER_WINS);
        }
     };

     private void mGameOver(boolean pWin){
. . .
        if (pWin){
           mPlayerWon = true;
           scene.setChildScene(mCreateEndScene(newTop,
         "Congratulations!!"), false, true, true);
        } else {
           mPlayerWon = false;
           scene.setChildScene(mCreateEndScene(false,
         "You Suck! \n. . . .blood"));
        }
     }

. . .
     private void mIncreaseDifficulty() {
        // Make the gamelet a little harder
```

```
      if (mOpenRate > 1000) mOpenRate -= 1000;
      if (mStayOpen > 500) mStayOpen -= 200;
      if (mOpensPerGame < 50) mOpensPerGame += 10;
   }

    private void mSaveDifficulty() {
       diffEditor.putInt("WhAV.OPEN_RATE", mOpenRate);
       diffEditor.putInt("WhAV.STAY_OPEN", mStayOpen);
       diffEditor.putInt("WhAV.OPENS_PER_GAME", mOpensPerGame);
       diffEditor.putInt("WhAV.WINS", mWins);
       diffEditor.putInt("WhAV.PLAYS", mPlays);
    }

. . .
}
```

Irate Villagers

In physics games such as Irate Villagers, the level of difficulty mostly depends on the design of each level. As a consequence, difficulty is a bit less under our control while the game is being played. There are still some things we can adjust:

- `mMaxStakes`: This variable consists of the number of stakes the player gets to throw at the pile of heads and debris before the game is declared over. In earlier versions of this gamelet, this value was given by the constant `MAX_STAKES`.
- `mMaxTime`: Right now we give players all the time in the world to complete the level, but we'll add a timer to put a little pressure on them.

The changes needed to complete the Irate Villagers gamelet and add some difficulty balancing to `IVActivity.java` are shown in Listing 16.5.

Listing 16.5 **Completion and Difficulty Balance of IVActivity.java**

```
package com.pearson.lagp.v3;

+imports

public class IVActivity extends BaseGameActivity implements
      IOnSceneTouchListener, BKConstants {
   // ===========================================================
   // Constants
   // ===========================================================

. . .
   private static final int NUM_LEVELS = 4;
```

```
. . .
   // =============================================================
   // Fields
   // =============================================================

. . .
   private ChangeableText mCurrScore, mTimeLeftTxt;
   private float mTimeLeft;
. . .
   private boolean mPlayerWon;
   private int mWins, mPlays;
   private int mMaxStakes;
   private int mMaxTime;
   private boolean [] mLevelComplete = new boolean[NUM_LEVELS];
   private int mCurrentLevel;
   private Handler mHandler;

. . .

   @Override
   public Engine onLoadEngine() {
. . .
      difficulty = getSharedPreferences("difficulty",
         MODE_PRIVATE);
      diffEditor = difficulty.edit();
      mMaxStakes = difficulty.getInt("IV.MAX_STAKES", 10);
      mMaxTime = difficulty.getInt("IV.MAX_TIME", 90);
      mTimeLeft = mMaxTime;
      for (int i=0; i<NUM_LEVELS; i++){
         mLevelComplete[i] =
            difficulty.getBoolean("IV.LEVEL"+i, false);
      }
      mWins = difficulty.getInt("IV.WINS", 0);
      mPlays = difficulty.getInt("IV.PLAYS", 0);
. . .
   }

   @Override
   public void onLoadResources() {
. . .
      // Time display
      mTimeLeftTxt = new ChangeableText(0.2f*CAMERA_WIDTH, 10.0f,
         mFont32, "Secs: 0", "Secs: XXX".length());
      mScene.getLastChild().attachChild(mTimeLeftTxt);

. . .
      mScene.registerUpdateHandler(new IUpdateHandler() {
```

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

            @Override
            public void onUpdate(final float pSecondsElapsed) {
               mTimeLeft -= pSecondsElapsed;
               if (mTimeLeft < 0){
                  mGameOver(false);
               }
               mTimeLeftTxt.setText("Time: " +
                  (int)mTimeLeft);
            }
         });

         return mScene;
      }

. . .
      private void mIncreaseDifficulty() {
         // Make the gamelet a little harder
       if (mMaxStakes > 1) mMaxStakes -= 1;
       if (mMaxTime > 20) mMaxTime -= 10;
      }

      private void mSaveDifficulty() {
     diffEditor.putInt("IV.MAX_STAKES", mMaxStakes);
     diffEditor.putInt("IV.MAX_TIME", mMaxTime);
     for (int i=0; i<NUM_LEVELS; i++){
         diffEditor.putBoolean("IV.LEVEL"+i, mLevelComplete[i]);
     }
     diffEditor.putInt("IV.WINS", mWins);
     diffEditor.putInt("IV.PLAYS", mPlays);
      diffEditor.commit();
      }
}
```

In addition to the difficulty parameters, we have introduced two other features to Irate Villagers:

- The ability to load, run, and track player success on a series of physics levels, labeled "iv0.lvl," "iv1.lvl," and so on. The maximum number of levels is set in constant `NUM_LEVELS`.
- A timer now counts down on the screen. If the player doesn't complete the level before the timer goes off, the gamelet is over and the player loses. The starting value of the timer is one of the difficulty parameters that are adjusted at the end of each run.

The first feature is implemented with a boolean array that tracks completion of the different levels, using the difficulty SharedPreferences. When the gamelet starts, it looks at this array to determine which level should be run next. When a level is successfully completed by the player, it is so marked and committed to the SharedPreferences.

The timing feature is implemented in much the same way as we implemented the process of showing scores, but now the time is updated by a newly introduced UpdateHandler. Conveniently, every UpdateHandler is provided with the time since the last update, so it can keep track of the approximate elapsed time. If the timer runs out, we call `mGameOver` with a value of `false` to indicate the player lost.

Options Menu

Since we've chained the gamelets together, we would normally delete Whack-A-Vampire and Irate Villagers from the options menu. However, for debugging purposes, it's convenient to be able to reach a gamelet without going through the whole chain, so we'll leave the menu the way it is.

Summary

Our V3 example game is now almost complete. Some housekeeping is needed to make the code neater and easier to support, but feature-wise the game is ready for beta testing with real players.

The techniques covered in this chapter aren't really unique to AndEngine. The mechanisms we used to parameterize and adjust gamelet difficulty were all Android APIs and plain old Java code. The method we used to implement the replay and play-next functionality was built entirely on Android's `startActivity()` method.

We're almost ready to publish our game. In Chapter 17, we'll discuss beta testing and the publishing process, and then we can make V3 available for the world to admire!

Exercises

1. In Level1Activity's `onPaused()` method, we set the boolean `mActivityVisible` to `false` so `playSound()` will not play sounds while we're working on something off the V3 screen. A slightly better way of handling this situation would be to cancel all outstanding Runnables while we're away, and then restore them in `onResume()`. With this approach, the compute resources that would otherwise be devoted to posting and running the Runnables will be saved. Change `Level1Activity.java` so Runnables don't run when V3 is off the screen.

2. Change `OptionsActivity.java` so the menu items for Whack-A-Vampire and Irate Villagers are no longer visible. Instead, hide two invisible buttons on the right side of the splash screen that will trigger those two gamelets.
3. Add a Help screen that appears when the player selects "Help" from the main menu in `MainMenuActivity.java`.

17

Testing and Publishing

The V3 game (and your own game, we hope) is now almost feature complete, and we'd like to get it ready to publish to the world. There is one more feature we'd like to add to V3—the ability for the user to make in-app purchases. Once we've added this feature, we need to test our game thoroughly, finding and resolving as many bugs as we possibly can. When users download our game and try it out, we want them to be pleasantly surprised by how much fun our game is to play and how well it works. It takes only a few "force close" errors to turn a player off of a game permanently.

Application Business Models

Before we dive into the technical details of implementing different ways to collect money, let's take a minute to talk about how you plan to market your game. We'll dig into the topic of promoting the game in a later section, but here we need to decide if and how you might get repaid for your diligent effort in developing the world's best mobile game.

Table 17.1 lists a few well-known business models and summarizes their chief tradeoffs. There's no right model for everyone, so take a look at the pros and cons and decide which one you'd like to use (or come up with your own business model). As noted in Table 17.1, some of the models require you to add code to your game, but others don't.

Table 17.1 does not represent an exhaustive list of business models, as new ones are thought up daily. Most of today's applications, however, use one or more of the models listed in the table. It's very common, for example, to offer a basic game for free and an enhanced game for a download fee.

Our V3 game isn't really about making money, so we'll make it available for free download. If you are interested in finding out more about adding advertisements to your game, a list of mobile advertising companies appears toward the end of this chapter, and Google provides an overview of integrating its AdMob service at the following URL:

`http://code.google.com/mobile/ads/docs/android/fundamentals.html`

Table 17.1 **Android Game Business Models**

Model	Benefit	Cost
Free	Reaches the widest possible audience. Can be used to build your reputation.	Low—just the cost of publication.
Mobile advertising	A small fee goes into your account for every player who clicks on a mobile ad.	Moderate. Code to present mobile ads goes into the game. The ads take up some screen real estate.
Freemium: free plus in-app purchases	Can reach a broad audience by giving away the basic game. Some revenue can be collected by offering add-ons for a fee.	Moderate. Code to manage the in-app purchasing process has to be added to game.
Pay for play	You get revenue immediately when the user downloads and installs the game.	Low—just the cost of publication.

If you are interested in providing in-app purchases in your game, Google also provides a convenient example application for the Android SDK, and an overview. You can find these items at this address:

```
http://developer.android.com/guide/market/billing/billing_overview.html
```

Testing and Getting Ready

We won't dodge the issue: Testing can be tedious. In fact, most software developers look at testing as a bore. Playing the part of a user and entering the same commands over and over to ferret out problems and prove the corrections have fixed the problem gets old in a hurry.

Of course, it's also the source of a lot of the fun in software development. At the beginning of the book we noted that solving puzzles was fun and that software development is fun for many of us because it gives us interesting puzzles to solve. The testing process is one way we discover those puzzles, and devising tests is an interesting problem in itself.

I know what you're thinking, and I won't argue with you: It's still boring to repeat the same test processes over and over—but it's absolutely essential to the success of your game. You have to find as many bugs as you can before you hand your game over to beta testers, and you have to get your beta testers to find as many bugs as they can before you release your game to the world.

Google's Android developer site offers a very good checklist for preparing any application for publication:

```
http://developer.android.com/guide/publishing/preparing.html
```

We will paraphrase this checklist here, adapting it for games and adapting it for use with application stores other than Android Market.

Test the Game on Actual Devices

This point can't be overemphasized: You need to test your game on the actual devices on which users will run it. The emulators that come with the Android SDK are great, but emulation on a PC or Mac using a mouse for touch gestures just isn't the real thing. If you're short of Android devices and need to do unit testing using the emulator, that's fine—but I don't even do that. I leave an Android device connected to my development machine and always test and debug on a real Android device.

However, one Android device isn't nearly enough. Devices can be vastly different, running different versions of Android, with different processors, including some with graphics processors and some without; some devices with large, high-resolution screens and others with small, low-resolution screens; and some with only single-touch capabilities and others with advanced multi-touch support. You need to test your game on as many Android devices as you can reach. The good news is that, unlike with some other platforms such as iOS and Windows Phone 7, you can sideload your game to as many Android devices as you want to before committing it to an app store.

Most of us don't have unlimited funds for buying one of each Android device. Now is a good time to leverage your friends who have Android devices and ask them to be beta testers of your new game. You'll need to send them the APK, of course, but it doesn't even have to be signed, as long as they enable sideloads (Settings > Applications > Unknown Sources) and know how to use adb to load the APK.

Consider Adding an End User License Agreement

For legal reasons, you should include an End User License Agreement (EULA) with your game that limits your liability. Players who download the game see this agreement when they install the game and have the option of agreeing or declining the agreement. We aren't lawyers, and we're not in the business of offering legal advice, but for example EULAs for Android applications, you can take a look at this site:

```
http://code.google.com/p/apps-for-android/source/browse/trunk/
DivideAndConquer/assets/EULA?r=93
```

You can also look at the EULA we used for Vampires Versus Virgins, which is included with the downloadable source code for this chapter.

We need some code that displays the EULA when players first start the game. They can accept or decline the EULA. If they accept, the game starts, and they are never asked again. If they decline, the game immediately exits. The EULA acceptance dialog is shown in Figure 17.1. The code that presents this agreement (and records the fact that the EULA has been presented in SharedPreferences) is shown in Listing 17.1.

Figure 17.1 EULA dialog

Listing 17.1 **StartActivity.java showEULA Method**

```
. . .
   @Override
   public void onCreate(Bundle pSavedInstanceState){
      super.onCreate(pSavedInstanceState);
      showEULA();
   }
. . .
     public void showEULA() {
         eula = getSharedPreferences("eula", MODE_PRIVATE);
         boolean eulaShown = eula.getBoolean("shown", false);
         if(eulaShown == false){
             String title = this.getString(R.string.app_name);
             String message = this.getString(R.string.updates) + "\n\n" +
         getString(R.string.eula);

             AlertDialog.Builder builder = new AlertDialog.Builder(this)
                     .setTitle(title)
                     .setCancelable(false)
                     .setMessage(message)
                     .setPositiveButton(getString(R.string.accept),
             new Dialog.OnClickListener() {
                         @Override
                         public void onClick(
                DialogInterface dialogInterface, int i) {
                             // Mark this version as read.
                             eulaEditor = eula.edit();
                             eulaEditor.putBoolean("shown", true);
                             eulaEditor.commit();
                             dialogInterface.dismiss();
                         }
                     })
```

```
                .setNegativeButton(getString(R.string.decline),
            new Dialog.OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialog,
            int which) {
                              finish();
                        }
                    });
             builder.create().show();
        }
    }
```

The `onCreate()` override calls the `showEULA()` method, after calling the `super.onCreate()` method so it can do its thing. The `showEULA()` method creates an AlertDialog that contains the update message and the EULA text that we've added to `strings.xml` and displays the dialog. We've used `setCancelable(false)` for the AlertDialog to ensure that the user can't use the back button to get around it.

Add an Icon and a Label to the Manifest

We'll need an icon for V3 to display on the Application Launcher's menu. Let's use the same one we used for the live wallpaper in Chapter 15. We created the icon in three sizes, for use in the three Android dpi classes. The icon images all have the name `icon.png` and are loaded into the three resource folders under `res/`, as shown in Table 17.2. These icons all look like the image in Figure 17.2.

Table 17.2 **V3 Icon Images**

Folder	Dimensions
`drawable-hdpi`	128 × 128 pixels
`drawable-mdpi`	64 × 64 pixels
`drawable-ldpi`	32 × 32 pixels

Figure 17.2 V3 icon

Turn Off Logging and Debugging

We didn't make extensive use of logging and debug statements in V3 (at least not in the versions that are shown in the book). If you do use logging and debug statements as you're developing your own game, it's a good idea to conditionalize them, which makes it easy to turn the statements off or on. We don't want the player to be confronted with debug statements, and we shouldn't waste log space by filling it with messages that won't be read.

In the downloaded V3 code, you may have noticed one place where we do need to turn off debugging—in `AndroidManifest.xml`. The Android device we have been using for debugging purposes is a commercial Android smartphone that requires the `debuggable` attribute in the manifest file to run the gdb debugger on the game. To turn off this debugging, we need to change just one line in `AndroidManifest.xml`. The new line reads as follows:

```
<application android:icon="@drawable/icon" android:label= "@string/app_
name" android:debuggable="false">
```

Add a Version Number to the Game

In case you haven't already been sold on this concept, versioning is of critical importance in Android applications, including games. Android itself doesn't look at the version number, but Android Market will use this information to let users know that updates are available for applications you have installed.

In the manifest for your game, you should define two identifiers:

- `android:versionCode`: an integer of your choice that monotonically increases with each version of the game you release. It is suggested that you start with 1 and increment this number by 1 for every major and minor release.
- `android:versionName`: a String that can be displayed to the user that describes this release of your game. You can use whatever versioning system you want, but most developers stick with the `major.minor` format to indicate how many major and minor releases have been made.

The versioning part of V3's manifest file is shown in Listing 17.2.

Listing 17.2 **V3's Manifest File: Versioning Information**

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.pearson.lagp.v3"
      android:versionCode="1"
      android:versionName="1.0">
. . .
```

Obtain a Crypto Key

While we've been developing our game, the Android SDK has been providing us with a debug cryptographic (crypto) key for all of our APKs. That arrangement worked well for development, but now we want to distribute our game to other users, so we need our own private crypto key that we can use to sign the `.apk` file.

If you haven't been through this process before, relax: It's just a matter of walking through a few steps. The Android SDK comes with a tool called Keytool that knows how to generate crypto keys, and you just use that tool to create the key you need. You run Keytool from a Command window. Listing 17.3 shows something like what you'll see on the screen. This listing was obviously created on a Windows PC, but the tool runs the same way on Linux or OS X.

Listing 17.3 **Running Keytool to Generate a Key**

```
C:\Users\rick>keytool -genkey -v -keystore V3-release-key.keystore -alias
        V3_alias -keyalg RSA -keysize 2048 -validity 10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Rick]: Rick Rogers
What is the name of your organizational unit?
  [Unknown]: Portmobile Software
What is the name of your organization?
  [Unknown]: Portmobile Software
What is the name of your City or Locality?
  [Unknown]: Harvard
What is the name of your State or Province?
  [Unknown]: MA
What is the two-letter country code for this unit?
  [Unknown]: US
Is CN=Rick, OU=Portmobile Software, O=Portmobile Software, L=Harvard,
ST=MA, C=US correct?
  [no]: yes

Generating 2,048 bit RSA key pair and self-signed certificate
(SHA1withRSA) with a validity of 10,000 days
        for: CN=Rick Rogers, OU=Portmobile Software, O=Portmobile Software,
L=Harvard, ST=MA, C=US
Enter key password for <V3_alias>
        (RETURN if same as keystore password):
[Storing V3-release-key.keystore]
```

After you run Keytool, the resulting private crypto key is stored in the keystore, with both the key and the keystore being protected by the password. The alias name is the handle you use to refer to a specific key in the keystore; it will be used a little later,

when we sign the final .apk file. Don't forget the password, and don't forget the alias that you used when creating the key. The recommendation is that you sign every release of your game with this same certificate throughout its life, which means you may need the password and alias for a long time. The number after "validity" is the number of days for which the generated key will remain good. Google requires that all private keys used for application signing be good at least until October 22, 2033.

Compile and Sign the Final .apk File

Generating a signed .apk file is just a matter of working your way through some dialog boxes once you have a key saved in the keystore. When you right-click on the Android project for your game, left-click on Android Tools, and left-click on Export Signed apk, a series of dialog boxes open that ask you for the following information:

- Location of the keystore: Keytool will have created the keystore in whatever directory you were in when you ran it. In the preceding example, the keystore V3-release-key.keystore is in C:\Users\rick.
- Password: The dialogs will ask for the password twice—once for the keystore and once for the key.
- Alias: In the preceding example, the alias is V3_alias. To help you determine the appropriate alias, the dialog box lists all the aliases it finds in the keystore.
- Destination file: This option allows you to choose where you want to put the signed .apk file.

You can now install your .apk file on Android devices using adb. For example, if there is only one Android device connected to your computer via USB, you can use this command:

```
adb -d install v3.apk
```

Your game will be installed on the device. If you change the game and want to reinstall the signed package, you have to uninstall it first:

```
adb -d uninstall com.pearson.v3
```

Test the Final .apk File

It's entirely possible that something in your game's code may have changed in the process of generating the final .apk file for release to Android Market. Because you're going to put your game in front of the world soon, it's worth a final sanity check to make sure things still work the way they are supposed to. Download your signed .apk file to as many Android devices as you can lay your hands on, and make sure the game still works the way it did when you finished the earlier testing cycle. If you find any new problems, it's easy enough to go through the signing process again, now that you have your private key ready to use.

Publishing

The number of app stores available for publishing Android applications and games grows every day. Two of the more popular sites for downloads are these options:

- Android Market: the original Android app store from Google
- Amazon App Store: a newer entry into the app store race from Amazon, currently available only in the United States

At the time of writing, Android Market is by far the larger venue. Just about every consumer Android device on the planet comes with Android Market pre-installed, so if you put your game anywhere, you should put it on Android Market.

That said, many Android device owners prefer downloading applications and games from other app stores, such as Amazon's site. Whether it's the convenience of purchasing from Amazon or the lure of Amazon's "Free App of the Day," the Amazon store appears to be gaining in popularity. Users do have to take the trouble to install the Amazon App Store application on their Android device to find and download apps—a consideration that will probably always be an impediment for alternative stores.

From the developer's point of view, signing up to peddle games on both stores is easy (see the details in the next two subsections). There is a difference in fees for developers, however. Android Market charges a one-time $25 fee "to encourage higher-quality products." Amazon App Store charges a $99 yearly fee, but currently waives the fee for the first year.

Android Market

Google tells us that more than half a million Android devices are activated every day, and more than 100 million devices have already been activated. Almost every one of those Android devices has the Android Market application running on it, making it easy for users to find and download applications and games. By providing developers with access to this enormous audience, Android Market is an incredible bargain at $25 to publish all the Android applications you can write.

The developer's introduction to Android Market can be found at the following site:

```
https://market.android.com/publish/signup
```

This document tells developers that they must do three things to begin publishing on Android Market:

- Create a developer's profile.
- Pay the registration fee to Google.
- Agree to the Android Market Developer Distribution Agreement.

The developer's profile is very simple, as you aren't even asked for your name and address (you do have to enter that information when you pay the registration fee). You

are asked for an email and phone number in case Android Market needs to contact you, but the vendor promises not to give that contact information to others.

The registration fee ($25) is payable only through Google Checkout, which accepts all the normal credit cards.

The Android Market Developer Distribution Agreement is a binding legal document. I am not a lawyer and would never offer you legal advice. It's best to have someone with legal knowledge review anything you are going to commit yourself to, but in the end, if you want to distribute through Android Market, you have to find a way to agree to it.

Once you've completed these three steps, approval of your developer status is generally granted in minutes, and you gain access to the Developer's Console. From here, you can upload new applications and games and manage the ones you've already uploaded.

When you upload your signed `.apk` file, you will be asked to edit the information about your application, and provide some screenshots, an icon, and optional graphics, including an optional promotional video. Google has changed the information collected several times, and will likely change it again, but for the game listing on Android Market you are currently asked for the following items:

- The language used by this version of the game.
- The title of the game (up to 30 characters).
- A description of the game (in English, up to 4000 characters).
- Any recent changes to the game (in English, up to 500 characters).
- A bit of promotional text (in English, up to 80 characters).
- An application type of either "Applications" or "Games." This choice determines in which of those categories your app will be listed on Android Market.
- Assuming you chose "Games," a game category. You have six choices:
  - Arcade & Action
  - Brain & Puzzle
  - Cards & Casino
  - Casual
  - Racing
  - Sports Games
- A content rating for your game:
  - High maturity
  - Medium maturity
  - Low maturity
  - Everyone
- Pricing.
- The countries where your game will be offered.

- Contact information, such as a website, email, and phone number.
- Two items of consent:
  1. That your game meets the Android Content Guidelines, which are reasonable. To reiterate, I am not a lawyer, so check them out for yourself (`http://www.android.com/us/developer-content-policy.html`).
  2. That you accept responsibility for the export of your game under U.S. law.

Once all of that information is entered, you are ready to click the Publish button and have your game served up to the masses of Android gamers.

If you want to charge for the game or enable in-app purchasing, you must also have a Google Merchant account. This is not a difficult process (a link to sign up is available on the Edit Application web form), and it involves yet another of those legal contracts that it is best to have a professional review.

Amazon App Store

The Amazon App Store is the up-and-coming new kid on the block when it comes to distribution of Android applications. The bad news is that Amazon currently has a much smaller audience of potential game players who could download your game.

The upside is that there are many fewer applications and games on the Amazon site, so it is easier for your new game to get noticed there. Another upside is that Amazon seems to be a bit more flexible in the content of applications it accepts. Whereas Google's Developer Program Policies explicitly forbid things like nudity and gambling with real money, Amazon asks that you note these items in the content rating.

The process of publishing on Amazon is similar to the process for Android Market:

- Go to the App Store Developer's portal. The portal has a huge, long URL that will probably change by the time you read this book, so just search for "Amazon app store developer portal," and you should be able to find it.
- Sign in using an existing Amazon account, or create a new one.
- You'll be asked to accept yet another legal agreement. (It seems as if the lawyers are the ones making the real money in this business.)
- If you plan to charge money for your game, you can enter bank account information so Amazon can send you your money.
- On the MyApps tab, you can add your application. Again, the fields are likely to change over time, but currently you are asked for the following information:
  - A title
  - The form factor: either Phone, Tablet, or both
  - An optional application SKU (Stock Keeping Unit—in case you want to track your application sales like the big boys do)
  - Supported languages (English is the default)
  - Optional support contact information: email, phone, website URL

- A Merchandising Section includes the following items:
  - A category (e.g., Games) and subcategories (Games currently has 17 subcategories to choose from)
  - Keywords to help searchers find your game
  - A description (up to 4000 characters)
  - List price (or free)
  - A set of dates for original release, visibility, availability, and discontinuation
- A Content Rating Section includes these selections:
  - Advertisements
  - Culture Intolerance
  - Dynamic Content
  - Nudity
  - Sexual Content
  - Alcohol, Tobacco, or Drugs
  - Designed for Children
  - Gambling
  - Profanity
  - Violence
- The Upload Multimedia Section is where you can upload screenshots, icons, and other images.
- The Upload Binary Section is where you upload your signed game `.apk` file.

When you have filled in all of the necessary information, click the "Submit App" button. Amazon will review your information and most likely approve your game and put it up on its site.

Promoting Your Game

For players to download and enjoy playing your game, they have to know it exists. The word "promotion" often brings to mind carnival barkers and snake oil salesmen, but you really do have to promote your game so people will know about it.

There are many ways to promote mobile applications, and entire books have been written on the subject. We'll cover a few of the basics but highly recommend that you seek out as many of those books and articles as you have time to read. The ones written for other mobile platforms apply in principle to Android games, too. It is also recommended that you "think outside the box" when it comes to promoting your game. Which clever ways can you imagine to get your game in front of potential players?

App Store Promotion

When we went through the app store submission processes for Android Market and Amazon App Store, we saw that both provide you with opportunities to make your game easy for players to find and download. Let's look at each of those opportunities in a bit more detail.

Feature Apps

Both app stores offer lists of featured apps and games for their users. Getting onto one of the "Top" charts is an obvious benefit—but it means your game has to be one of the Top Paid, Top Free, Top Grossing, Top New Paid, or Top New Free games. Android Market also has Featured lists for Trending apps (including games) and Best-Selling Games. The lists that are not quantitative (e.g., Featured Apps) are developed by the editors of the respective websites. The best thing you can do is make your game noticeable and hope to hook one of the editors into playing your game and putting it on the list. There is no way to buy your way onto this list.

Title

Spend some time coming up with a catchy title that people will remember. V3 isn't really intended to become the next blockbuster game, but if it were, having memorable words like "virgin" and "vampire" in the title would be good for sales. The best names have emotional content that draw potential players' attention and are hard to forget. The title is limited to 30 characters on Android Market, so make sure you get the most out of them. Test potential titles with your friends just the way you test the actual game.

Keywords

Most players will find your game download from a keyword search. Pick keywords that are related to your game and its title, and pick keywords that are commonly used. Unfortunately, mobile app stores haven't yet developed all of the analytic tools available to website developers, but don't let that shortcoming stop you. Google AdWords has a free keyword tool, available at the following URL:

```
https://adwords.google.com/o/Targeting/Explorer?__u=1000000000&__c=1000000
000&ideaRequestType=KEYWORD_IDEAS#search.none
```

The keyword statistics produced by this tool are for the Web, not for mobile game searches, but they can help you identify popular keywords that might help promote your game.

Screenshots and Icons

You will need to create an attractive icon that makes players want to download and try out your game. Should you be fortunate enough to get recognition for your game on an app store review list, your icon is likely to be all that is displayed on the top page of the list. Make it a good one.

The screenshots you upload for your game should be the most interesting screens you can muster. You want potential customers to see those screenshots and say, "Wow, that game looks like it could be fun. I think I'll download it." The order in which you load the screenshots is also important, because not all are displayed on the front page of your game entry. Pick the most engaging screenshot to be the one searchers see first.

Description

The description posted on the app store is where you really get to put on your carnival barker's hat. One trick that advertising writers use is to focus on the words, sentence by sentence. For example, the only goal of the first sentence in the copy is to get the reader to read the second sentence. Write, rewrite, and hone that first sentence until anyone reading it cannot help but read the next sentence. By the time they've read the second sentence, they should be anxious to complete the first paragraph. The first paragraph makes them want to read the rest of the copy. Before it ends, you give the potential players a "call to action," asking them to download the game.

Content Rating

You might not think of content ratings as a vehicle for marketing your game, but they are. You should have a target audience in mind for your game, and the content ratings should reflect the needs of that target audience. The app stores allow search filtering, and you want your audience's filters to select *for* your game, not *against* it.

Player Reviews

For some reason, potential customers tend to give considerable weight to reviews written by complete strangers on a website. You want to make sure your game is reviewed favorably, so don't release it until you've thoroughly beta tested it with a variety of users. Those users should feel free to provide their review comments on the app store sites, and you should encourage them to do so. Don't worry too much about the occasional poor review. Instead, focus on having the majority of the reviews give a positive impression of your game.

Price

Pricing is an art. Even if you read all the books that have been written on this topic, you still wouldn't know the right price for your game. Free is always great, and there are alternative ways to make money other than charging for your game download. If you do plan to charge for your game, consider offering a "lite" version for free that can introduce your game to a wider audience.

Aside from free games, mobile games seem to range in price from $0.99 to $4.99 or so, with the lower-priced games selling better than the higher-priced ones. App store searchers can filter searches based on price, and again you don't want your game to be filtered out.

Whatever price you charge, be aware that you won't get 100% of the money. The current Android Market agreement calls for Google to pay you 70% of each sale, and

the Amazon App Store agreement calls for the company to pay the greater of 70% of the sale or 20% of the list price (in case it's a discounted sale). If you choose to give your game away, you can't go back and charge users for it later, but you can offer a limited free edition and a full-featured edition for a price.

Game Review Sites

Mobile game review websites are an excellent way to get your game noticed and reviewed. As mentioned earlier, each of the app stores has its own review section, but there are also sites dedicated to reviewing games. Each site has its own process for submitting your game. If your game is reviewed, however, it can make a whole new audience aware that it exists, which is the first step toward them actually downloading and playing the game.

Here are a few sites specializing in mobile games:

- `http://www.pocketgamer.co.uk/`
- `http://www.gamespot.com/mobile/index.html`
- `http://www.pocket-arcade.com/`

Mobile Advertising

We discussed mobile advertising as a possible way of making money on your game earlier in this chapter, but it can also be an effective way of marketing your game. In this case, however, you are paying for the advertising and using it to promote your game. For example, in the unlikely event that someone does a Google search for "virgins" and "vampires," I could pay Google a small amount to present the user with a small ad stating that a game involving both exists for Android devices.

Here are some of the advertising services aimed at mobile users:

- AdMob (`www.admob.com`): the largest service, owned by Google
- inMobi (`www.inmobi.com`): a strong competitor for AdMob on Android devices
- JumpTap (`www.jumptap.com`): claims to maximize return by carefully targeting ads at mobile users
- Millennial Media (`www.millennialmedia.com`): another strong competitor for AdMob

All of these are multiple-platform services (Android, iOS, Window Phone 7), but you can select the target audience you are after.

Word of Mouth

The dream of every game developer is to have his or her game "go viral"—that is to have people talking about your game as they stand around the water cooler or eat lunch, spreading awareness to new customers, and telling each other how much fun

it is to play. There's no way to ensure that outcome happens, but fundamentally you want your game to be played by people who like to spread the news about new things, and you want to make sure they have fun when they play it.

Tell everybody you know about your new game. Then tell them again, as they may have forgotten about your news bulletin in the crush of everyday life.

Social Networking

If your game doesn't have a Facebook site, it should. Why not? Facebook is another place where people can find out about your game and share their experiences in playing it. Twitter is another way to bring your game to people's attention, particularly as you make updates available and want people to download them and try them out.

Summary

If you had a person with an MBA on your staff (or maybe you have an MBA of your own), that individual would tell you that marketing is all about the four P's:

- Product: You have to have a product that fills a need for your customers. In the ideal scenario, you've written a game that is engaging and will entertain players, help them pass unused time, and maybe even educate them a little.
- Price: Products have to be priced at what customers are willing to pay. As we've seen, pricing is tricky and involves more than the original purchase price of your game. This area represents an opportunity to be creative.
- Place: Place in this sense comprises the marketplace where your customer purchases the product. In our case, that is almost certainly one (or more) of the app stores that huge companies such as Google and Amazon operate.
- Promotion: Customers have to be aware of your game, and they have to realize why it would be fun to download and play your game. By promoting your game on the app stores, on game review sites, through mobile advertising, by word of mouth, and through social networking, you can be sure that it reaches the widest possible audience.

Congratulations! If you've read through this book and followed along with the examples and exercises, you know everything necessary to create and publish your very own mobile game for the Android platform using the AndEngine game platform. Create the game of your dreams, offer it to the world, and, most of all, remember to have a good time while you're doing it.

Appendix

Exercise Solutions

Chapter 1

The exercises in this chapter don't have generic solutions. They ask you to use the tools we've talked about to build artwork, music, and plans for your own game. If you have trouble completing any of the exercises, refer back to the related section in the text and you should be able to work through the problems.

Chapter 2

The exercises in this chapter don't have generic solutions. Instead, they ask you to use the tools we've talked about to build or otherwise obtain specific artwork, music, and plans for your own game. If you have trouble completing any of the exercises, refer back to the related section in the text and you should be able to work through the problems.

Chapter 3

1. The static menu scene is added to the main scene in `onLoadScene()` before anything is displayed. Adding it at that time doesn't trigger any animations to run. The pop-up scene is added later, after the main scene is already being displayed, so its animations run, and it slides in from the left.
2. The color of the menu items can be set in two places—either when the font texture regions are loaded in `onLoadResources()` or with a ColorMenuItemDecorator. Because we're already using ColorMenuItemDecorators in `MainMenuActivity.java`, the following code shows separate fonts being loaded in red, white, and blue and their use as the menu items are added.

MenuActivity.java Changes for Multicolored Menu

```
. . .
   @Override
   public void onLoadResources() {
      /* Load Font/Textures. */
      this.mFontTexture = new Texture(256, 256,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);

      FontFactory.setAssetBasePath("font/");
      this.mFont = FontFactory.createFromAsset(
         this.mFontTexture, this, "Flubber.ttf", 32,
         true, Color.RED);
      this.mEngine.getTextureManager().loadTexture(
         this.mFontTexture);
      this.mEngine.getFontManager().loadFont(this.mFont);
. . .
   protected void createStaticMenuScene() {
      this.mStaticMenuScene = new MenuScene(this.mCamera);

      final IMenuItem playMenuItem =
         new ColorMenuItemDecorator(
         new TextMenuItem(MENU_PLAY, mFont, "Play Game"),
         0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
      playMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mStaticMenuScene.addMenuItem(playMenuItem);

      final IMenuItem scoresMenuItem =
         new ColorMenuItemDecorator(
         new TextMenuItem(MENU_SCORES, mFont, "Scores"),
         0.5f, 0.5f, 0.5f, 0.0f, 1.0f, 0.0f);
      scoresMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mStaticMenuScene.addMenuItem(scoresMenuItem);

      final IMenuItem optionsMenuItem =
         new ColorMenuItemDecorator(
         new TextMenuItem(MENU_OPTIONS, mFont, "Options"),
         0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f);
      optionsMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mStaticMenuScene.addMenuItem(optionsMenuItem);

      final IMenuItem helpMenuItem =
         new ColorMenuItemDecorator(
         new TextMenuItem(MENU_HELP, mFont, "Help"), 0.5f,
```

```
         0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
      helpMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mStaticMenuScene.addMenuItem(helpMenuItem);
. . .
```

The font loading code stays the same, and the ColorMenuItemDecorators are changed just slightly, so "Play Game" appears in red, "Scores" in blue, "Options" in green, and "Help" in red.

3. The ScaleMenuItemDecorator is used much like the ColorMenuItemDecorator. The following code shows an excerpt using it with one of the menu items in the main menu.

MenuActivity.java Changes for Blooming Menu Items

```
. . .
   protected void createStaticMenuScene() {
      this.mStaticMenuScene = new MenuScene(this.mCamera);

      final IMenuItem playMenuItem = new ScaleMenuItemDecorator(
         new TextMenuItem(MENU_PLAY, mFont, "Play Game"),
         1.2f, 1.0f);
      playMenuItem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE_MINUS_SRC_ALPHA);
      this.mStaticMenuScene.addMenuItem(playMenuItem);
. . .
```

The last two parameters to ScaleMenuItemDecorator are the scales for selected and unselected options, respectively. In this case, the item will appear 20% larger when selected, then go back to its original size when deselected.

Chapter 4

1. A simple Activity for trying out different modifiers is shown in the following listing. The complete Android project is found in the downloads as project Modifier:

StartActivity.java for Trying Modifiers

```
package com.pearson.lagp.modex;

import org.anddev.andengine.engine.Engine;
import org.anddev.andengine.engine.camera.Camera;
import org.anddev.andengine.engine.options.EngineOptions;
```

```
import org.anddev.andengine.engine.options.EngineOptions
   .ScreenOrientation;
import org.anddev.andengine.engine.options.resolutionpolicy
   .RatioResolutionPolicy;
import org.anddev.andengine.entity.modifier.ScaleModifier;
import org.anddev.andengine.entity.scene.Scene;
import org.anddev.andengine.entity.sprite.Sprite;
import org.anddev.andengine.entity.util.FPSLogger;
import org.anddev.andengine.opengl.texture.Texture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.opengl.texture.region.TextureRegion;
import org.anddev.andengine.opengl.texture.region.TextureRegionFactory;
import org.anddev.andengine.ui.activity.BaseGameActivity;

import android.os.Handler;

public class StartActivity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;

   // ===========================================================
   // Fields
   // ===========================================================

   private Camera mCamera;
   private Texture mTexture;
   private TextureRegion mFaceTextureRegion;
   private Handler mHandler;

   // ===========================================================
   // Constructors
   // ===========================================================

   // ===========================================================
   // Getter and Setter
   // ===========================================================

   // ===========================================================
   // Methods for/from SuperClass/Interfaces
   // ===========================================================

   @Override
   public Engine onLoadEngine() {
```

```
      mHandler = new Handler();
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH, CAMERA_HEIGHT);
      return new Engine(new EngineOptions(true, ScreenOrientation
.LANDSCAPE, new RatioResolutionPolicy(CAMERA_WIDTH, CAMERA_HEIGHT), this
.mCamera).setNeedsMusic(true));
   }

   @Override
   public void onLoadResources() {
      TextureRegionFactory.setAssetBasePath("gfx/");
      this.mTexture = new Texture(512, 1024,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      this.mFaceTextureRegion = TextureRegionFactory
.createFromAsset( this.mTexture, this,
         "mathead.png", 0, 0);
      this.mEngine.getTextureManager().loadTexture(this.mTexture);
   }

   @Override
   public Scene onLoadScene() {
      this.mEngine.registerUpdateHandler(new FPSLogger());

      final Scene scene = new Scene(1);

      /* Center the face on the camera. */
      final int centerX = (CAMERA_WIDTH - this.mFaceTextureRegion
.getWidth()) / 2;
      final int centerY = (CAMERA_HEIGHT - this.mFaceTextureRegion
.getHeight()) / 2;

      /* Create the face sprite and add it to the scene. */
      final Sprite face = new Sprite(centerX, centerY, this
.mFaceTextureRegion);

      face.registerEntityModifier(new ScaleModifier(10.0f, 0.0f, 1.0f));

      scene.getLastChild().attachChild(face);
      return scene;

   }

   @Override
   public void onLoadComplete() {
   }

   // ===========================================================
   // Methods
   // ===========================================================
```

```
    // ============================================================
    // Inner and Anonymous Classes
    // ============================================================
}
```

2. This sequence requires a combination of SequenceEntityModifiers and ParallelEntityModifiers. One combination that works is shown here (and can be tested with the code shown for Exercise 1). This code is also included in project Modifier.

Combination of Modifiers

```
. . .
    face.registerEntityModifier(new ParallelEntityModifier(
        new MoveModifier(3.0f, 0.0f, CAMERA_WIDTH/2,
                0.0f, CAMERA_HEIGHT/2),
        new SequenceEntityModifier(
                        new ColorModifier(2.0f, 1.0f,
                                0.0f, 1.0f, 0.0f,
                                1.0f, 1.0f),
                        new ColorModifier(2.0f, 0.0f,
                                1.0f, 0.0f, 0.0f,
                                1.0f, 0.0f),
                        new ColorModifier(2.0f, 1.0f,
                                0.0f, 0.0f, 1.0f,
                                0.0f, 0.0f)
                        ),
        new SequenceEntityModifier(
                       new ScaleModifier(1.0f, 1.0f,
                               1.0f),
                       new ScaleModifier(2.0f, 1.0f,
                               0.5f),
                       new ScaleModifier(2.0f, 0.5f,
                               1.0f)
                       ),
        new SequenceEntityModifier(
                       new DelayModifier(2.0f),
                       new RotationModifier(2.0f,
                               0.0f, 720.0f)
                      )
        )
    );
. . .
```

3. This is a one-line addition to `onLoadScene()`. Sometimes after you create mMainScene, you just need to add the following line:

```
mMainScene.setScaleCenter(centerX, centerY);
```

4. One implementation of `EaseWiggle.java` is shown here.

EaseWiggle.java

```
public class EaseLinear implements IEaseFunction {
        private static EaseLinear INSTANCE;

        // ===========================================================
        // Constructors
        // ===========================================================

        private EaseLinear() {
        }

        public static EaseLinear getInstance() {
                if(INSTANCE == null) {
                        INSTANCE = new EaseLinear();
                }
                return INSTANCE;
        }

        // ===========================================================
        // Methods for/from SuperClass/Interfaces
        // ===========================================================

        @Override
        public static float getPercentageDone(final float pSecondsElapsed,
                final float pDuration, final float pMinValue,
                final float pMaxValue) {
                return (float) (pMaxValue * pSecondsElapsed / pDuration +
                        pMinValue + 4.0f * Math.sin(Math.PI *
                        pSecondsElapsed * 10.0f/pDuration));
        }

}
```

Chapter 5

1. One solution for the rotating red star is shown here:

StarActivity.java

```
package com.pearson.lagp.v3;

+imports

public class StarActivity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 480;
   private static final int CAMERA_HEIGHT = 320;
   private String tag = "SpriteTestActivity";

   // ===========================================================
   // Fields
   // ===========================================================

   protected Camera mCamera;

   protected Scene mMainScene;

   // ===========================================================
   // Constructors
   // ===========================================================

   // ===========================================================
   // Getter and Setter
   // ===========================================================

   // ===========================================================
   // Methods for/from SuperClass/Interfaces
   // ===========================================================

   @Override
   public Engine onLoadEngine() {
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
            CAMERA_HEIGHT), this.mCamera));
   }
```

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

    @Override
    public Scene onLoadScene() {
        this.mEngine.registerUpdateHandler(new FPSLogger());

        final Scene scene = new Scene(1);
        scene.setBackground(new ColorBackground(0.0f, 0.0f, 0.0f));

        /* Center the camera. */
        final int centerX = CAMERA_WIDTH  / 2;
        final int centerY = CAMERA_HEIGHT / 2;

        /* Draw the star */
        Line star1 = new Line(centerX-100, centerY-40, centerX+100,
            centerY-40, 3.0f);
        star1.setColor(1.0f, 0.0f, 0.0f);
        Line star2 = new Line(200, 0, 40, 125, 3.0f);
        star2.setColor(1.0f, 0.0f, 0.0f);
        Line star3 = new Line(-160, 125, -100, -75, 3.0f);
        star3.setColor(1.0f, 0.0f, 0.0f);
        Line star4 = new Line(-100, -75, -30, 125, 3.0f);
        star4.setColor(1.0f, 0.0f, 0.0f);
        Line star5 = new Line(-30, 125, -200, 0, 3.0f);
        star5.setColor(1.0f, 0.0f, 0.0f);
        star1.attachChild(star2);
        star1.getLastChild().attachChild(star3);
        star1.getLastChild().attachChild(star4);
        star1.getLastChild().attachChild(star5);
        star1.setRotationCenter(100, 40);
        star1.registerEntityModifier(new RotationModifier(5.0f,
            0.0f, 360.0f));
        scene.getLastChild().attachChild(star1);

        return scene;
    }

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

Notice that the coordinates for the Lines and the Rotation Center are relative to star1.

2. The only changes needed (other than adding the SVG extension .jar library and the SVG graphics files) are in onLoadResources() in SpriteTest-Activity.java, which is shown here. This code is also included as V305SVG in the downloadable code for this chapter.

StarActivity.java

```
. . .
   @Override
   public void onLoadResources() {
      /* Load Textures. */
      TextureRegionFactory.setAssetBasePath("gfx/SpriteTest/");
      mTestTexture = new Texture(512, 256,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);
      this.mHatchetTextureSource= new SVGAssetTextureSource(
         this, "svg/hatchet40.svg", 1.0f);
      this.mMadMatTextureSource= new SVGAssetTextureSource(this,
         "svg/Mat.svg", 1.0f);

      mHatchetTextureRegion =
         TextureRegionFactory.createFromSource(mTestTexture,
            mHatchetTextureSource, 0,0);
      mMadMatTextureRegion =
         TextureRegionFactory.createFromSource(mTestTexture,
            mMadMatTextureSource, 50,0);
      this.mEngine.getTextureManager().loadTexture(
         this.mTestTexture);
   }
. . .
```

3. Each reader will have his or her own solution.

Chapter 6

1. The changes needed to make the bat fly back and forth are shown in this excerpt for onLoadScene() in StartActivity.java.

StartActivity.java onLoadScene() with Moving Bat

```
. . .
   @Override
   public Scene onLoadScene() {
      this.mEngine.registerUpdateHandler(new FPSLogger());

      final Scene scene = new Scene(1);
```

```
        /* Center the splash on the camera. */
        final int centerX = (CAMERA_WIDTH -
           this.mSplashTextureRegion.getWidth()) / 2;
        final int centerY = (CAMERA_HEIGHT -
           this.mSplashTextureRegion.getHeight()) / 2;

        /* Create the background sprite and add it to the scene. */
        final Sprite splash = new Sprite(centerX, centerY,
           this.mSplashTextureRegion);
        scene.getLastChild().attachChild(splash);

        /* Create the animated bat sprite and add to scene */
        final AnimatedSprite bat = new AnimatedSprite(350, 100,
           this.mBatTextureRegion);
        bat.animate(100, true);
        bat.registerEntityModifier(new LoopEntityModifier(
           new SequenceEntityModifier (
              new MoveXModifier(2.0f, bat.getX(),
                 bat.getX()-60),
              new MoveXModifier(2.0f, bat.getX()-60,
                 bat.getX()))));
        scene.getLastChild().attachChild(bat);
        return scene;
    }
```

We've registered a LoopEntityModifier to keep the bat flying back and forth, and used a sequence of MoveXModifiers to do the actual movement. To make the bat fly behind the tombstone, you'd have to extract the tombstone from the background and place it on another Layer above the AnimatedSprite.

2. This answer is reader specific.
3. This answer is reader specific.
4. A solution is shown here. AndEngine allows us to register an UpdateHandler method that will be executed before each screen update. We register such a method in `onLoadScene()` just before we return the Scene to the Engine.

Level1Activity.java onLoadScene() with Disappearing Vampires

```
. . .
   @Override
   public Scene onLoadScene() {
. . .
      scene.registerUpdateHandler(new IUpdateHandler() {
         @Override
         public void reset() { }
```

```
         @Override
         public void onUpdate(final float pSecondsElapsed) {
            for (int i=0; i<nVamp; i++){
               if (asprVamp[i].getX() < 35.0f){
                  asprVamp[i].setVisible(false);
               }
            }
         }
      });
. . .
```

We check each of the displayed vampires to see if it has reached Miss B's. If it has, we make it invisible.

Chapter 7

1. This is a one-line change. Look in `onLoadResources()` and find the StrokeFont constructor call. We just need to change the last parameter (`boolean pStrokeOnly`) to false:

```
this.mStrokeFont = new StrokeFont(this.mStrokeFontTexture,

   Typeface.create(Typeface.DEFAULT, Typeface.BOLD), 32, true,
   Color.RED, 2.0f, Color.WHITE, false);
```

2. The "i" disappeared because the Texture ran out of room to hold the entire font. The font images are stored as a matrix, so it's difficult to predict which characters will drop out, but if you're missing characters, the first thing to try is enlarging the font Texture size. In this case, increasing this size to 512 × 512 pixels is enough to fix the problem.
3. This is a matter of adding the `MENU_OPTION` constant, and adding the option to both the menu builder and the menu's `onClickMenuItem()` method, as shown here:

OptionsActivity.java Changes to Support Help Option

```
package com.pearson.lagp.v3;

+imports

public class OptionsActivity extends BaseGameActivity implements
IOnMenuItemClickListener {
   // ===========================================================
   // Constants
   // ===========================================================
. . .
   protected static final int MENU_MUSIC = 0;
```

```
   protected static final int MENU_EFFECTS = MENU_MUSIC + 1;
   protected static final int MENU_HELP = MENU_EFFECTS + 1;

   // ===========================================================
   // Fields
   // ===========================================================
. . .
   private TextMenuItem mHelp;
   private IMenuItem helpMenuItem;

. . .

   // ===========================================================
   // Constructors
   // ===========================================================

   // ===========================================================
   // Getter and Setter
   // ===========================================================

   // ===========================================================
   // Methods for/from SuperClass/Interfaces
   // ===========================================================

. . .
   @Override
   public void onLoadResources() {
      mHelp = new TextMenuItem(MENU_EFFECTS, mFont, "Help");
   }

. . .
   @Override
   public boolean onMenuItemClicked(final MenuScene pMenuScene, final
IMenuItem pMenuItem, final float pMenuItemLocalX, final float
pMenuItemLocalY) {
      switch(pMenuItem.getID()) {
. . .
         case MENU_HELP:
            Toast.makeText(MainMenuActivity.this,
            "Help selected", Toast.LENGTH_SHORT).show();
            return true;
         default:
            return false;
      }
   }

   // ===========================================================
   // Methods
   // ===========================================================
```

```
   protected void createOptionsMenuScene() {
. . .
      helpMenuItem = new ColorMenuItemDecorator( mHelp, 0.5f,
         0.5f, 0.5f, 1.0f, 0.0f, 0.0f);
      this.mOptionsMenuScene.addMenuItem(helpMenuItem);

      this.mOptionsMenuScene.buildAnimations();

      this.mOptionsMenuScene.setBackgroundEnabled(false);
      this.mOptionsMenuScene.setOnMenuItemClickListener(this);
   }
. . .
```

4. Add the Toasts to `OptionsMenuActivity.java`, as shown here. This is actually the way I debugged the original code, as keeping track of Boolean flips can be an error-prone process.

OptionsActivity.java Changes to Show Boolean Flips

```
Toast.makeText(context, text, duration).show();

. . .
   @Override
   public boolean onMenuItemClicked(final MenuScene pMenuScene, final
IMenuItem pMenuItem, final float pMenuItemLocalX, final float
pMenuItemLocalY) {
      switch(pMenuItem.getID()) {
         case MENU_MUSIC:
            if (isMusicOn) {
               isMusicOn = false;
               Toast.makeText(this,
               "Music turned off",
               LENGTH_SHORT).show();
            } else {
               isMusicOn = true;
               Toast.makeText(this,
               "Music turned on",
               LENGTH_SHORT).show();
            }
            createOptionsMenuScene();
            mMainScene.clearChildScene();
            mMainScene.setChildScene(mOptionsMenuScene);
            return true;
         case MENU_EFFECTS:
            if (isEffectsOn) {
               isEffectsOn = false;
               Toast.makeText(this,
               "Effects turned off",
```

```
                LENGTH_SHORT).show();
            } else {
                isEffectsOn = true;
                Toast.makeText(this,
                "Effects turned on",
                LENGTH_SHORT).show();
            }
            createOptionsMenuScene();
            mMainScene.clearChildScene();
            mMainScene.setChildScene(mOptionsMenuScene);
            return true;
         default:
            return false;
      }
   }
. . .
```

Chapter 8

1. The following code shows a possible solution:

DCtrlMove.java

```
package com.pearson.lagp.example;

+imports

public class DCtrlMove extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 720;
   private static final int CAMERA_HEIGHT = 480;

   // ===========================================================
   // Fields
   // ===========================================================

   private Camera mCamera;
   private Texture mTexture;
   private TextureRegion mTextureRegion;
   private float centerX, centerY;
   private Sprite face;
   private Texture mOnScreenControlTexture;
   private TextureRegion mOnScreenControlBaseTextureRegion;
```

```
private TextureRegion mOnScreenControlKnobTextureRegion;


// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   centerX = CAMERA_WIDTH/2;
   centerY = CAMERA_HEIGHT/2;
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
      CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
   this.mTexture = new Texture(256, 256,
         TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   mTextureRegion = TextureRegionFactory.createFromAsset(
      mTexture, getApplicationContext(),
      "gfx/mathead.png", 0, 50); // 32x32

   this.mEngine.getTextureManager().loadTexture(
          this.mTexture);
   this.mOnScreenControlTexture = new Texture(256, 128,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
   this.mOnScreenControlBaseTextureRegion =
      TextureRegionFactory.createFromAsset(
         this.mOnScreenControlTexture, this,
         "gfx/onscreen_control_base.png", 0, 0);
   this.mOnScreenControlKnobTextureRegion =
      TextureRegionFactory.createFromAsset(
         this.mOnScreenControlTexture, this,
```

```
            "gfx/onscreen_control_knob.png", 128, 0);
      this.mEngine.getTextureManager().loadTextures(
         this.mTexture, this.mOnScreenControlTexture);
   }

   @Override
   public Scene onLoadScene() {
      final Scene scene = new Scene(1);
      scene.setBackground(new ColorBackground(0.1f, 0.6f, 0.9f));

      face = new Sprite(centerX, centerY, mTextureRegion);
      scene.getLastChild().attachChild(face);
      final DigitalOnScreenControl digitalOnScreenControl =
         new DigitalOnScreenControl(0,
         CAMERA_HEIGHT -
         this.mOnScreenControlBaseTextureRegion.getHeight(),
         this.mCamera,
         this.mOnScreenControlBaseTextureRegion,
         this.mOnScreenControlKnobTextureRegion, 0.1f,
         new IOnScreenControlListener() {
         @Override
         public void onControlChange(
           final BaseOnScreenControl pBaseOnScreenControl,
            final float pValueX, final float pValueY) {
              face.setPosition(face.getX() + pValueX *
               10, face.getY() + pValueY * 10);
         }
      });
      digitalOnScreenControl.getControlBase().setBlendFunction(
         GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
      digitalOnScreenControl.getControlBase().setAlpha(0.5f);
      digitalOnScreenControl.getControlBase().setScaleCenter(0,
         128);
      digitalOnScreenControl.getControlBase().setScale(1.25f);
      digitalOnScreenControl.getControlKnob().setScale(1.25f);
      digitalOnScreenControl.refreshControlKnobPosition();

      scene.setChildScene(digitalOnScreenControl);

      return scene;
   }

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

2. The example program included with the download ACtrlMove is very similar to DCtrlMove. The following code shows the changes:

ACtrlMove.java

```
. . .
public class ACtrlMove extends BaseGameActivity {
. . .
   @Override
   public Scene onLoadScene() {
. . .
      final AnalogOnScreenControl analogOnScreenControl =
         new AnalogOnScreenControl(0, CAMERA_HEIGHT -
         this.mOnScreenControlBaseTextureRegion.getHeight(),
         this.mCamera,
         this.mOnScreenControlBaseTextureRegion,
         this.mOnScreenControlKnobTextureRegion, 0.1f, 200,
         new IAnalogOnScreenControlListener() {
         @Override
         public void onControlChange(
           final BaseOnScreenControl pBaseOnScreenControl,
            final float pValueX, final float pValueY) {
            face.setPosition(face.getX()+pValueX * 20,
               face.getY()+pValueY * 20);
         }

         @Override
         public void onControlClick(
            final AnalogOnScreenControl
                  pAnalogOnScreenControl) {
            face.registerEntityModifier(
               new SequenceEntityModifier(
               new ScaleModifier(0.25f, 1, 1.5f),
               new ScaleModifier(0.25f, 1.5f, 1)));
         }
      });
      analogOnScreenControl.getControlBase().setBlendFunction(
         GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
      analogOnScreenControl.getControlBase().setAlpha(0.5f);
      analogOnScreenControl.getControlBase().setScaleCenter(0,
         128);
      analogOnScreenControl.getControlBase().setScale(1.25f);
      analogOnScreenControl.getControlKnob().setScale(1.25f);
      analogOnScreenControl.refreshControlKnobPosition();

      scene.setChildScene(analogOnScreenControl);

      return scene;
   }
. . .
```

3. The following code for `AccelColor.java` shows a simple application that uses AndEngine's extensions of the Android accelerometer APIs to detect changes in position. If not obvious, it works because the acceleration in the *Y* direction (usually the long axis of a smartphone) is near 0 when the device is held flat, whereas the acceleration of gravity (approximately 9.8 m/sec) applies when the device is upright.

AccelColor.java

```
package com.pearson.lagp.example;

+imports

public class AccelColor extends BaseGameActivity implements
      IAccelerometerListener {
   // ===========================================================
   // Constants
   // ===========================================================

   private static final int CAMERA_WIDTH = 720;
   private static final int CAMERA_HEIGHT = 480;

   // ===========================================================
   // Fields
   // ===========================================================

   private Camera mCamera;
   private Scene scene;

   // ===========================================================
   // Methods for/from SuperClass/Interfaces
   // ===========================================================

   @Override
   public Engine onLoadEngine() {
      this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
         CAMERA_HEIGHT);
      return new Engine(new EngineOptions(true,
         ScreenOrientation.LANDSCAPE,
         new RatioResolutionPolicy(CAMERA_WIDTH,
         CAMERA_HEIGHT), this.mCamera));
   }

   @Override
   public void onLoadResources() {
      this.enableAccelerometerSensor(this);
   }
```

```
    @Override
    public Scene onLoadScene() {
        scene = new Scene(1);
        scene.setBackground(new ColorBackground(0.0f, 1.0f, 0.0f));
        return scene;
    }

    @Override
    public void onLoadComplete() {
    }

    @Override
    public void onAccelerometerChanged(
        final AccelerometerData pAccelerometerData) {
        String message = "X= "+pAccelerometerData.getX()+
            "; Y="+pAccelerometerData.getY()+
            "; Z="+pAccelerometerData.getZ()+";";
        //Toast.makeText(this, message, Toast.LENGTH_SHORT).show();
        if (pAccelerometerData.getY() > 5.0f){
            scene.setBackground(new ColorBackground(1.0f, 0.0f,
                0.0f));
        } else {
            scene.setBackground(new ColorBackground(0.0f, 1.0f,
                0.0f));
        }
    }
}
```

Chapter 9

1. A revised `.tmx` file, `WAVTilesetEx.tmx`, is included with the downloadable code in project V309Ex. This file includes the assignment of the property, and the new tile set from Exercise 2.
2. The "Boo!" tile is also included in project V309Ex. It is part of the new tile set `WAVTileSetEx,png`. It isn't assigned anywhere in the tile map, but `WAVActivity.java` is changed as shown here:

WAVmap.tmx with Edits for Subfolders

```
package com.pearson.lagp.v3;

+imports

public class WAVActivity extends BaseGameActivity {
. . .
```

```
   // ============================================================
   // Fields
   // ============================================================
. . .
   private int mTombstoneGID = -1;
   private int mBooGID = -1;
   private int mOpenCoffinGID = 1;
. . .
   @Override
   public Scene onLoadScene() {
      this.mEngine.registerUpdateHandler(new FPSLogger());

      final Scene scene = new Scene(1);
      try {
         final TMXLoader tmxLoader = new TMXLoader(this,
            this.mEngine.getTextureManager(),
            TextureOptions.BILINEAR_PREMULTIPLYALPHA,
            new ITMXTilePropertiesListener() {
            @Override
            public void onTMXTileWithPropertiesCreated(
               final TMXTiledMap pTMXTiledMap,
               final TMXLayer pTMXLayer,
               final TMXTile pTMXTile,
               final TMXProperties<TMXTileProperty>
                 pTMXTileProperties) {
            if(pTMXTileProperties.containsTMXProperty(
              "coffin", "true")) {
               coffins[coffinPtr++] =
               pTMXTile.getTileRow() * 15 +
               pTMXTile.getTileColumn();
              if (mCoffinGID<0){
                mCoffinGID = pTMXTile.getGlobalTileID();
            }
              }

            if(pTMXTileProperties.containsTMXProperty(
               "tombstone", "true")) {
               if (mTombstoneGID<0){
                 mTombstoneGID =
                  pTMXTile.getGlobalTileID();
               }
            }

            if(pTMXTileProperties.containsTMXProperty(
               "boo", "true")) {
               if (mBooGID<0){
                 mBooGID =
```

```
                pTMXTile.getGlobalTileID();
                }
            }
        }
    });
    this.mWAVTMXMap = tmxLoader.loadFromAsset(this,
        "gfx/WAV/WAVmapEx.tmx");
} catch (final TMXLoadException tmxle) {
    Debug.e(tmxle);
}

tmxLayer = this.mWAVTMXMap.getTMXLayers().get(0);
scene.getFirstChild().attachChild(tmxLayer);
scene.setOnSceneTouchListener(new IOnSceneTouchListener() {
    @Override
    public boolean onSceneTouchEvent(final Scene
    pScene, final TouchEvent pSceneTouchEvent) {
      switch(pSceneTouchEvent.getAction()) {
       case TouchEvent.ACTION_DOWN:
       /* Get the touched tile */
       tmxTile = tmxLayer.getTMXTileAt(
          pSceneTouchEvent.getX(),
          pSceneTouchEvent.getY());
       if((tmxTile != null) &&
          (tmxTile.getGlobalTileID() ==
             mOpenCoffinGID)) {
          tmxTile.setGlobalTileID(
             mWAVTMXMap, mCoffinGID);
       } else {
       if((tmxTile != null) &&
          (tmxTile.getGlobalTileID() ==
          mTombstoneGID)) {
          tmxTile.setGlobalTileID(mWAVTMXMap,
             mBooGID);
       }}
             break;
          case TouchEvent.ACTION_UP:
             break;
       }
       return true;
    }
});

mHandler.postDelayed(openCoffin,gen.nextInt(2000));
. . .
}
```

The patterns are much the same as were discussed in the text. We use the tile properties to find the global IDs for the coffin and tombstone tiles, and then use those global IDs to go directly to the tile image we want. We can't find the "Boo!" tile global ID that way (there are no "Boo!" tiles in the starting map, so none will ever be loaded), but that's okay. We know the global IDs from reading the TSX file, which tells us the first global ID number; we can then just count them off in the Tiled view of the tile set. It's safer to look for a property that will move with the tile if things change, but in this case we have no choice.

Chapter 10

1. The changed lines of code are shown here. Specific lines that changed are in bold.

Changes to Level1Activity.java for Smoke Effect

```
package com.pearson.lagp.v3;
. . .
   @Override
   public Scene onLoadScene() {
. . .
      particleEmitter = new CircleParticleEmitter(
        CAMERA_WIDTH * 0.5f, CAMERA_HEIGHT * 0.5f + 20, 40);
      particleSystem = new ParticleSystem(particleEmitter,
         100, 100, 500, this.mParticleTextureRegion);

      particleSystem.addParticleInitializer(
         new ColorInitializer(1, 1, 1));
      particleSystem.addParticleInitializer(
         new AlphaInitializer(0));
      particleSystem.setBlendFunction(GL10.GL_SRC_ALPHA,
         GL10.GL_ONE);
      particleSystem.addParticleInitializer(
         new VelocityInitializer(-2, 2, -2, -2));
      particleSystem.addParticleInitializer(
         new RotationInitializer(0.0f, 360.0f));

      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-
          .ScaleModifier(1.0f, 2.0f, 0, 5));
      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-
          .ColorModifier(0, 0.5f, 0, 0.5f, 0, 0.5f, 0, 2));
      particleSystem.addParticleModifier(
         new org.anddev.andengine.entity.particle.modifier-
          .ColorModifier(0.5f, 0, 0.5f, 0, 0.5f, 1, 2, 4));
      particleSystem.addParticleModifier(
```

```
            new org.anddev.andengine.entity.particle.modifier-
              .AlphaModifier(0, 1, 0, 1));
         particleSystem.addParticleModifier(
            new org.anddev.andengine.entity.particle.modifier-
              .AlphaModifier(1, 0, 3, 4));
         particleSystem.addParticleModifier(
            new ExpireModifier(2, 4));

         particleSystem.setParticlesSpawnEnabled(false);
         scene.getLastChild().attachChild(particleSystem);

         return scene;
      }
. . .
}
```

2. The PX file for a smoke effect is shown here:

smoke.px

```
<ParticleConfig>
   <emitter
      shape="circle"
      center_x="0.0"
      center_y="0.0"
      radius_x="40.0"
      radius_y="40.0">
   </emitter>
   <system
      texture="particle_fire.png"
      min_rate="100"
      max_rate="100"
      max_particles="500">
      <init_color
         min_red="1"
         max_red="1"
         min_green="1"
         max_green="1"
         min_blue="1"
         max_blue="1">
      </init_color>
      <init_alpha
         min_alpha="0"
         max_alpha="0">
      </init_alpha>
      <init_velocity
         min_velocity_x="-2"
```

```
   max_velocity_x="2"
   min_velocity_y="-2"
   max_velocity_y="-2">
</init_velocity>
<init_rotation
   min_rotation="0.0"
   max_rotation="360.0">
</init_rotation>
<mod_scale
   from_scale_x="1.0"
   to_scale_x="1.0"
   from_scale_y="1.0"
   to_scale_y="1.0"
   to_scale="2.0"
   from_time="0"
   to_time="5">
</mod_scale>
<mod_color
   from_red="0"
   to_red="0.5"
   from_green="0"
   to_green="0.5"
   from_blue="0"
   to_blue=".5"
   from_time="0"
   to_time="2">
</mod_color>
<mod_color
   from_red=".5"
   to_red="0"
   from_green="0.5"
   to_green="0"
   from_blue=".5"
   to_blue="1"
   from_time="2"
   to_time="4">
</mod_color>
<mod_alpha
   from_alpha="0"
   to_alpha="1"
   from_time="0"
   to_time="1">
</mod_alpha>
<mod_alpha
   from_alpha="1.0"
   to_alpha="0"
   from_time="3"
```

```
            to_time="4">
         </mod_alpha>
         <mod_expire
            min_lifetime="2"
            max_lifetime="4">
         </mod_expire>
      </system>
</ParticleConfig>
```

3. The PX file `rain.px` is shown here:

rain.px

```
<ParticleConfig>
   <emitter
      shape="rectangle"
      center_x="0.0"
      center_y="0.0"
      width="480"
      height="2.0">
   </emitter>
   <system
      texture="particle_point.png"
      min_rate="100"
      max_rate="100"
      max_particles="500">
      <init_color
         min_red="1"
         max_red="1"
         min_green="1"
         max_green="1"
         min_blue="1"
         max_blue="1">
      </init_color>
      <init_alpha
         min_alpha="0"
         max_alpha="0">
      </init_alpha>
      <init_velocity
         min_velocity_x="-2"
         max_velocity_x="2"
         min_velocity_y="50"
         max_velocity_y="100">
      </init_velocity>
      <init_rotation
         min_rotation="0.0"
```

```
            max_rotation="0.0">
        </init_rotation>
        <init_gravity
            gravity_on="true">
        </init_gravity>
        <mod_scale
            from_scale_x="0.2"
            to_scale_x="0.2"
            from_scale_y="0.3"
            to_scale_y="0.3"
            from_time="0"
            to_time="5">
        </mod_scale>
        <mod_alpha
            from_alpha="0"
            to_alpha="1"
            from_time="0"
            to_time="1">
        </mod_alpha>
        <mod_alpha
            from_alpha="1.0"
            to_alpha="0"
            from_time="3"
            to_time="4">
        </mod_alpha>
        <mod_expire
            min_lifetime="2"
            max_lifetime="4">
        </mod_expire>
    </system>
</ParticleConfig>
```

Chapter 11

1. All the changes needed to add music to the cross weapon are in `Level1Activity.java`:
   - Record the song. I saved mine as an Ogg/Vorbis file, `OCS.ogg`, and imported it into `assets/mfx`.
   - Add a new Music object to `Level1Activity.java`: `Music mOCSMusic`.
   - Add `.setNeedsMusic()` to the Engine options for `Level1Activity.java`.
   - In `onLoadResources()`, add the lines to load the new song:
     `MusicFactory.setAssetBasePath("mfx/");`

```
try {
   this.mOCSMusic =
         MusicFactory.createMusicFromAsset(
           this.mEngine.getMusicManager(),
           this, "OCS.ogg");
         this.mOCSMusic.setLooping(true);
} catch (final IOException e) {
         Debug.e(e);
}
```

- In the switch for `cross TouchEvents`, in the case for `ACTION_UP`, add the lines:

```
if (audioOptions.getBoolean("musicOn", false)) {
    mOCSMusic.play();
}
```

- In `onGamePaused()`, add the line:

```
mOCSMusic.stop();
```

2. The change needed to switch to using the MIDI file is in the `onLoad-Resources()` method of `StartActivity.java`. In the try/catch statement that uses MusicFactory to load the music file, change the line as follows:

```
StartActivity.mMusic = MusicFactory.createMusicFromAsset(
this.mEngine.getMusicManager(), getApplicationContext(),
        "bachfugue2.mid");
```

3. To add scores to SharedPreferences and the Scores screen, first change `StartActivity.java` to add the following lines:

Changes to StartActivity.java for Exercise 3

```
   private SharedPreferences scores;
   private SharedPreferences.Editor scoresEditor;
. . .
   @Override
   public Engine onLoadEngine() {
. . .
      scores = getSharedPreferences("scores", MODE_PRIVATE);
      scoresEditor = scores.edit();
      if (!scores.contains("WAV")){
         scoresEditor.putInt("WAV", 0);
         scoresEditor.putInt("Level1", 0);
         scoresEditor.commit();
      }
```

In `MainMenuActivity.java`, in the `MenuItemClicked()` method, change the `MENU_SCORES` case to read as shown here. Also add the Runnable shown after the ellipsis.

MainMenuActivity.java for Exercise 3

```
. . .
   @Override
   public boolean onMenuItemClicked(final MenuScene pMenuScene,
      final IMenuItem pMenuItem, final float pMenuItemLocalX,
      final float pMenuItemLocalY) {
      switch(pMenuItem.getID()) {
. . .
         case MENU_SCORES:
            mMainScene.registerEntityModifier(
               new ScaleAtModifier(0.5f, 1.0f,
               0.0f, CAMERA_WIDTH/2,
               CAMERA_HEIGHT/2));
            mStaticMenuScene.registerEntityModifier(
               new ScaleAtModifier(0.5f, 1.0f,
               0.0f, CAMERA_WIDTH/2,
               CAMERA_HEIGHT/2));
            mHandler.postDelayed(mLaunchScoresTask,
               500);
            return true;
. . .
      private Runnable mLaunchScoresTask = new Runnable() {
          public void run() {
            Intent myIntent = new Intent(MainMenuActivity.this,
         ScoresActivity.class);
            MainMenuActivity.this.startActivity(myIntent);
          }
       };
```

Then add a new class, `ScoresActivity.java`, as shown here. Don't forget to add it to your manifest, so Android will know the new activity is there.

Scores.java for Exercise 3

```
import org.anddev.andengine.opengl.texture.Texture;
import org.anddev.andengine.opengl.texture.TextureOptions;
import org.anddev.andengine.ui.activity.BaseGameActivity;

import android.content.SharedPreferences;
import android.graphics.Color;

public class ScoresActivity extends BaseGameActivity {
   // ===========================================================
   // Constants
   // ===========================================================
```

```
private static final int CAMERA_WIDTH = 480;
private static final int CAMERA_HEIGHT = 320;

// ===========================================================
// Fields
// ===========================================================

protected Camera mCamera;

protected Scene mScoresScene;
private Text mTitle, mWAV, mLevel1;;
private Texture mFontTexture;
private Font mFont;
private SharedPreferences scores;
private SharedPreferences.Editor scoresEditor;

// ===========================================================
// Constructors
// ===========================================================

// ===========================================================
// Getter and Setter
// ===========================================================

// ===========================================================
// Methods for/from SuperClass/Interfaces
// ===========================================================

@Override
public Engine onLoadEngine() {
   this.mCamera = new Camera(0, 0, CAMERA_WIDTH,
      CAMERA_HEIGHT);
   scores = getSharedPreferences("scores", MODE_PRIVATE);
   scoresEditor = scores.edit();
   return new Engine(new EngineOptions(true,
      ScreenOrientation.LANDSCAPE,
      new RatioResolutionPolicy(CAMERA_WIDTH,
      CAMERA_HEIGHT), this.mCamera));
}

@Override
public void onLoadResources() {
   /* Load Font/Textures. */
   this.mFontTexture = new Texture(256, 256,
      TextureOptions.BILINEAR_PREMULTIPLYALPHA);
```

```
        FontFactory.setAssetBasePath("font/");
        this.mFont = FontFactory.createFromAsset(this.mFontTexture,
            this, "Flubber.ttf", 32, true, Color.RED);
        this.mEngine.getTextureManager().loadTexture(
            this.mFontTexture);
        this.mEngine.getFontManager().loadFont(this.mFont);
    }

    @Override
    public Scene onLoadScene() {
        /* Center the background on the camera. */
        final int centerX = (CAMERA_WIDTH) / 2;
        final int centerY = (CAMERA_HEIGHT) / 2;

        this.mScoresScene = new Scene(1);
        /* Add the background and scores */
        mScoresScene.setBackground(new ColorBackground(0.0f, 0.0f,
            0.0f));
        mTitle = new Text( centerX - 200, centerY - 100, mFont,
            "Scores");
        mWAV = new Text( centerX - 100, centerY - 50, mFont,
            "WAV\t\t" + scores.getInt("WAV", -1));
        mLevel1 = new Text( centerX - 100, centerY, mFont,
            "Level1\t\t" + scores.getInt("Level1", -1));
        mScoresScene.getLastChild().attachChild(mTitle);
        mScoresScene.getLastChild().attachChild(mWAV);
        mScoresScene.getLastChild().attachChild(mLevel1);
        return this.mScoresScene;
    }

    @Override
    public void onLoadComplete() {
    }

    // ===========================================================
    // Methods
    // ===========================================================

    // ===========================================================
    // Inner and Anonymous Classes
    // ===========================================================

}
```

Chapter 12

All Exercise solutions are included as part of Project V312 Exercises, in the downloadable code.

1. A second example level is shown here:

iv2.lvl

```
<level>
  <completeShape>
    <xprop>240.0000</xprop>
    <yprop>1.0000</yprop>
    <height>2.00</height>
    <width>480.00</width>
    <rotation>0.0000</rotation>
    <isDynamic>false</isDynamic>
    <shape>SQUARE</shape>
    <physicsandID>0.5,0.5,0.5,'stone'</physicsandID>
    <verts>'()'</verts>
  </completeShape>
  <completeShape>
    <xprop>240.0000</xprop>
    <yprop>319.0000</yprop>
    <height>2.00</height>
    <width>480.00</width>
    <rotation>0.0000</rotation>
    <isDynamic>false</isDynamic>
    <shape>SQUARE</shape>
    <physicsandID>0.5,0.5,0.5,'stone'</physicsandID>
    <verts>'()'</verts>
  </completeShape>
  <completeShape>
    <xprop>165.0000</xprop>
    <yprop>298.0000</yprop>
    <height>40.00</height>
    <width>40.00</width>
    <rotation>0.0000</rotation>
    <isDynamic>true</isDynamic>
    <shape>CIRCLE</shape>
    <physicsandID>0.5,0.5,0.5,'glass'</physicsandID>
    <verts>'()'</verts>
  </completeShape>
  <completeShape>
    <xprop>260.0000</xprop>
    <yprop>300.0000</yprop>
    <height>40.00</height>
    <width>40.00</width>
```

```
  <rotation>0.0000</rotation>
  <isDynamic>true</isDynamic>
  <shape>CIRCLE</shape>
  <physicsandID>0.5,0.5,0.5,'wood'</physicsandID>
  <verts>'()'</verts>
</completeShape>
<completeShape>
  <xprop>207.0000</xprop>
  <yprop>276.0000</yprop>
  <height>6.00</height>
  <width>186.00</width>
  <rotation>0.0000</rotation>
  <isDynamic>true</isDynamic>
  <shape>SQUARE</shape>
  <physicsandID>0.5,0.5,0.5,'wood'</physicsandID>
  <verts>'()'</verts>
</completeShape>
<completeShape>
  <xprop>155.0000</xprop>
  <yprop>249.0000</yprop>
  <height>50.00</height>
  <width>50.00</width>
  <rotation>0.0000</rotation>
  <isDynamic>true</isDynamic>
  <shape>SQUARE</shape>
  <physicsandID>0.5,0.5,0.5,'wood'</physicsandID>
  <verts>'()'</verts>
</completeShape>
<completeShape>
  <xprop>480.2494</xprop>
  <yprop>161.5000</yprop>
  <height>-4.50</height>
  <width>319.00</width>
  <rotation>1.5708</rotation>
  <isDynamic>false</isDynamic>
  <shape>SQUARE</shape>
  <physicsandID>0.5,0.5,0.5,'stone'</physicsandID>
  <verts>'()'</verts>
</completeShape>
<completeShape>
  <xprop>234.0000</xprop>
  <yprop>262.0000</yprop>
  <height>24.00</height>
  <width>50.00</width>
  <rotation>0.0000</rotation>
  <isDynamic>true</isDynamic>
  <shape>CIRCLE</shape>
```

```
    <physicsandID>0.5,0.5,0.5,'vamp'</physicsandID>
    <verts>'()'</verts>
  </completeShape>
  <completeShape>
    <xprop>277.6250</xprop>
    <yprop>268.5000</yprop>
    <height>9.00</height>
    <width>13.25</width>
    <rotation>0.0000</rotation>
    <isDynamic>true</isDynamic>
    <shape>SQUARE</shape>
    <physicsandID>0.5,0.5,0.5,'iq'</physicsandID>
    <verts>'()'</verts>
  </completeShape>
</level>
```

2. The changes to `IVActivity.java` are shown here:

IVActivity Changes to Use Vector Graphics

```
. . .
  @Override
  public void onLoadResources() {
          /* Textures. */
          this.mTexture = new Texture(512, 512,
                  TextureOptions.BILINEAR_PREMULTIPLYALPHA);
          TextureRegionFactory.setAssetBasePath("gfx/IV/");

          /* TextureRegions. */
          ITextureSource mSlingTextureSource =
          new SVGAssetTextureSource(this,  "gfx/IV/sling.svg",  1.0f);
          ITextureSource mStakeTextureSource =
          new SVGAssetTextureSource(this,  "gfx/IV/stake.svg",  1.0f);
          ITextureSource mGlassTextureSource =
          new SVGAssetTextureSource(this,  "gfx/IV/glass.svg",  1.0f);
          ITextureSource mStoneTextureSource =
          new SVGAssetTextureSource(this,  "gfx/IV/stone.svg",  1.0f);
          ITextureSource mMatHeadTextureSource =
          new SVGAssetTextureSource(this,  "gfx/IV/mathead.svg",  1.0f);
          mSlingTextureRegion =
                  TextureRegionFactory.createFromSource(this.mTexture,
                  mSlingTextureSource, 0, 0);
          mStakeTextureRegion = TextureRegionFactory.createFromSource(
                  this.mTexture, mStakeTextureSource, 0, 40);
          mGlassTextureRegion = TextureRegionFactory.createFromSource(
```

```
            this.mTexture, mGlassTextureSource, 0, 80);
        mStoneTextureRegion = TextureRegionFactory.createFromSource(
            this.mTexture, mStoneTextureSource, 0, 120);
        mMatHeadTextureRegion = TextureRegionFactory.createFromSource(
            this.mTexture, mMatHeadTextureSource, 0, 160);
        mWoodTextureRegion = TextureRegionFactory.createFromAsset(
            this.mTexture, getApplicationContext(), "wood.png",
                    0, 210);
        this.mEngine.getTextureManager().loadTexture(this.mTexture);
```

Chapter 13

1. The change needed is in the `mStartVamp()` Runnable, as shown here:

Changes to mStartVamp()

```
    private Runnable mStartVamp = new Runnable() {
. . .
            asprVamp[i].animate(frameDurations, 0, 25, true);
            float lagTime = gen.nextFloat()*20.0f;
            float startX = asprVamp[i].getX() -
                  (lagTime/60.0f) * (asprVamp[i].getX() - 30.0f);
            pathVamp[i] = aStar[i].getPath(startX, 1, asprVamp[i].getY(),
            10, asprVamp[i].getWidth(), asprVamp[i].getHeight());
            asprVamp[i].registerEntityModifier(
                 new SequenceEntityModifier (
                    new AlphaModifier(5.0f, 0.0f, 1.0f),
                   new MoveXModifier(lagTime, asprVamp[i].getX(), startX),
                  new PathModifier(60.0f - lagTime, pathVamp[i])
            ));
                 scene.getLastChild().attachChild(asprVamp[i]);
    };
. . .
```

   The variable `lagTime` predicts where the Sprite will be located when it is ready to find a path to Miss B's. The Path is found in advance of setting the Modifiers. Why couldn't we use an IEntityModifierListener on the MoveModifier?

2. The changes needed to `Level1Activity.java` are shown here. You can argue that the implementation does little to help the poor vampires dodge a bullet—it goes too fast. The alternative would be to warn the vampires as the player is moving the weapon, but that would generate a lot of unnecessary path finding. Maybe you can think of a better way.

Adding Vampire Warnings to Level1Activity.java

```
. . .
      @Override
      public Scene onLoadScene() {
. . .
            bullet = new Sprite(20.0f, CAMERA_HEIGHT - 40.0f,
                  mBulletTextureRegion){
              @Override
              public boolean onAreaTouched(final TouchEvent
                  pAreaTouchEvent, final float pTouchAreaLocalX,
                  final float pTouchAreaLocalY) {
                  switch(pAreaTouchEvent.getAction()) {
                  case TouchEvent.ACTION_DOWN:
                    break;
                  case TouchEvent.ACTION_UP:
                    mWarnVampires(pAreaTouchEvent.getY());
                    fireBullet(pAreaTouchEvent.getX(),
                              pAreaTouchEvent.getY());
                    break;
                  case TouchEvent.ACTION_MOVE:
                    this.setPosition( pAreaTouchEvent.getX() -
                      this.getWidth() / 2,
                      pAreaTouchEvent.getY() - this.getHeight() / 2);
                    break;
                  }
                  return true;
              }
              };
. . .
     private void mWarnVampires(float pThreatY){
           // There's a potential threat to vampires at pThreatY
           Scene scene = Level1Activity.this.mEngine.getScene();

           for (int i=0; i<nVamp; i++){
                 if (Math.abs(asprVamp[i].getY() - pThreatY) < 10.0f) {
                       asprVamp[i].clearEntityModifiers();
                      pathVamp[i] = aStar[i].getPath(asprVamp[i].getX(), 1,
                  asprVamp[i].getY() - 20.0f, 10,
                  asprVamp[i].getWidth(), asprVamp[i].getHeight());
                      asprVamp[i].registerEntityModifier(
                           new SequenceEntityModifier (
                                new MoveYModifier(5.0f, asprVamp[i].getY(),
                                asprVamp[i].getY() - 20.0f),
                                         new PathModifier(60.0f, pathVamp[i])
                     ));
                     scene.getLastChild().attachChild(asprVamp[i]);
```

```
                }
            }
        }
. . .
```

Chapter 14

1. The needed code changes are shown here:

IVActivity Changes

```
. . .
   private SharedPreferences audioOptions, scores;
. . .
   public void onLoadResources() {
. . .
      SoundFactory.setAssetBasePath("mfx/");
      try {
         this.mOofSound = SoundFactory.createSoundFromAsset(
            this.mEngine.getSoundManager(), this,
            "oof.ogg");
      } catch (final IOException e) {
         Debug.e(e);
      }
. . .
   @Override
   public void onLoadComplete() {

      this.mPhysicsWorld.setContactListener(
         new ContactListener() {
            @Override
            public void beginContact(Contact contact) {
               Body bodyA = contact.getFixtureA().getBody();
               Body bodyB = contact.getFixtureB().getBody();
               String idA = (String)bodyA.getUserData();
               String idB = (String)bodyB.getUserData();
               if ((idA.startsWith("vamp")) &&
            (idB.equals("floor"))) {
                    playSound(mOofSound);
                    int vampID = Integer.parseInt(
               idA.substring(4, 5));
                    if (!deadHeads.contains(vampID)) {
             deadHeads.add(vampID);
            }
                    mAddScore(VAMPIRE_FLOORED);
```

```
                if (deadHeads.size() == numHeads) {
              mGameOver(PLAYER_WINS);
           }
              }
              if ((idB.startsWith("vamp")) &&
           (idA.equals("floor"))) {
                 playSound(mOofSound);
                 int vampID = Integer.parseInt(
              idB.substring(4, 5));
                 if (!deadHeads.contains(vampID)){
               deadHeads.add(vampID);
            }
                 mAddScore(VAMPIRE_FLOORED);
                 if (deadHeads.size() == numHeads) {
              mGameOver(PLAYER_WINS);
            }
              }
           }
            public void endContact(Contact contact) {
           }
     });
  }
```

The sound file is loaded and the ContactListener is modified so the sound plays (if enabled) when either of the colliding bodies is a vampire head.

2. The following code shows the changes needed in `Level1Activity.java`, `IVActivity.java`, and `WAVActivity.java` to add the "New High Score" message to the final screen for those gamelets.

Changes Made in Level1Activity.java, IVActivity.java, and WAVActivity.java

```
. . .
   private TextureRegion mNewHighTextureRegion;
. . .
   public void onLoadResources() {
. . .
      mNewHighTextureRegion =
         TextureRegionFactory.createFromAsset(
            this.mPopUpTexture, getApplicationContext(),
            "newhigh.png", 100, 400);
. . .
   public Scene onLoadScene() {
. . .
      newHigh = new Sprite(0.0f, 0.0f, mNewHighTextureRegion);
. . .
     private void mGameOver(boolean pWin){
```

```
        // Called when gamelet is over - pWin=true if player won
        Scene scene = WAVActivity.this.mEngine.getScene();
        boolean newTop = false;
        int[] newHighScores = {0,0,0,0,0};
        for (int i=4; i>-1; i--){
           if (thisScore > highScores[i]){
              newHighScores[i] = thisScore;
              for (int j=i-1; j>-1; j--){
                 newHighScores[j] = highScores[j+1];
              }
              if (i==4) newTop = true;
              break;
           } else {
              newHighScores[i] = highScores[i];
           }
        }
        for (int i=0; i<5; i++) highScores[i] = newHighScores[i];
       scoresEditor.putInt("WhAV-4", highScores[4]);
       scoresEditor.putInt("WhAV-3", highScores[3]);
       scoresEditor.putInt("WhAV-2", highScores[2]);
       scoresEditor.putInt("WhAV-1", highScores[1]);
       scoresEditor.putInt("WhAV-0", highScores[0]);
       scoresEditor.commit();

        if (pWin){
           scene.setChildScene(mCreateEndScene(newTop, true,
         "Congratulations!!"), false, true, true);
        } else {
           scene.setChildScene(mCreateEndScene(false, false,
         "You Suck! \n . . . .blood"));
        }
     }

     private Scene mCreateEndScene(boolean pNewHigh, boolean pWin,
      String pTitle){
        Scene endScene = new Scene(2);
        endScene.getLastChild().attachChild(endBack);
        Text mTitle = new Text( 50.0f, 50.0f, mFont32, pTitle);
        endScene.getLastChild().attachChild(mTitle);
        if (pNewHigh) {
           newHigh.setPosition(300.0f, 50.0f);
           endScene.getLastChild().attachChild(newHigh);
        }
        Text mYourScore = new Text( 50.0f, 150.0f, mFont32,
      "Your Score: " + thisScore);
. . .
```

A Sprite is created from the "New High Score" texture. As we go through the high scores at the end of the game, we check whether the current score is higher than all of the scores in the array. If the new score is the highest, we pass a flag to `mCreateEndScene()`, which includes the Sprite.

Chapter 15

1. This exercise again demonstrates how easy it is to reuse game assets for other purposes like live wallpaper. The following code shows the changes required to `V3LiveWallpaper.java`:

V3LiveWallpaper.java with Explosions

```
package com.pearson.lagp.vinb;
+ imports
. . .
   // ===========================================================
   // Fields
   // ===========================================================

   private ParticleSystem particleSystem;
   private BaseParticleEmitter particleEmitter;
. . .
   @Override
   public Scene onLoadScene() {
. . .
      scene.registerUpdateHandler(new IUpdateHandler() {
         @Override
         public void reset() { }

         @Override
         public void onUpdate(final float pSecondsElapsed) {
            for (int i=0; i<nVamp; i++){
               if (asprVamp[i].getX() < 30.0f){
                //Show explosion
                particleEmitter.setCenter(
                      asprVamp[i].getX(),
                  asprVamp[i].getY());
                particleSystem.setParticles-
                    SpawnEnabled(true);
                   mHandler.postDelayed(
                  mEndPESpawn,2000);
                //Move vampire back to right
                        float startY = gen.nextFloat() *
                   (CAMERA_HEIGHT - 50.0f);
```

```
                        asprVamp[i].clearEntityModifiers();
                        asprVamp[i].registerEntityModifier(
                          new MoveModifier(40.0f,
                    CAMERA_WIDTH - 30.0f, 0.0f,              startY, 340.0f)
                        );
                  }
               }
            }
         });
         try {
            final PXLoader pxLoader = new PXLoader(this,
               this.mEngine.getTextureManager(),
               TextureOptions.BILINEAR_PREMULTIPLYALPHA );
            particleSystem = pxLoader.createFromAsset(this,
               "gfx/particles/explo.px");
         } catch (final PXLoadException pxle) {
            Debug.e(pxle);
         }
         particleSystem.setBlendFunction(GL10.GL_SRC_ALPHA,
            GL10.GL_ONE);
         particleSystem.setParticlesSpawnEnabled(false);
         particleEmitter =
          (BaseParticleEmitter) particleSystem.getParticleEmitter();

         scene.getLastChild().attachChild(particleSystem);
. . .

    private Runnable mEndPESpawn = new Runnable() {
        public void run() {
          particleSystem.setParticlesSpawnEnabled(false);
        }
    };
```

These changes take the explosion particle effect from `Level1Activity.java` and add it to the live wallpaper service. The effect is triggered when any vampire reaches the left side of the screen (Miss B's). Besides changing this code, there are some things we need to add to the project:

- The particle effect loading files must be added to the `src` folder:
  - `PXLoader.java`
  - `PXParser.java`
  - `PXConstants.java`
  - `PXLoaderException.java`
  - `PXParserException.java`

- The particle effect XML file `explo.px` must be placed in the folder `assets/gfx/particles`.
- The PNG file for the particle effect, `particle_fire.png`, must be placed in `assets/gfx/Wallpaper`.

2. We don't have the rights to redistribute any MOD files for this book, but you can freely obtain files for your own use at sites such as `http://modarchive.org`. The code needed to move your `.mod` file from assets to the SD card is given in Listing 15.3. For extra credit, change the method `startPlayingMod()` so that it respects the audio SharedPreferences Boolean values for music being turned on and off.
3. The resulting code is shown here. Note these significant changes:
   - Add `implements IOrientationListener` to the class declaration.
   - Add the ChangeableText to display the heading.
   - Add the `onOrientationChanged()` listener.
   - Override `onPause()` and `onResume()` methods to stop and start the compass readings.
   - Use NumberFormat to limit the length of the heading value.

Vampires in Backyard with Compass

```
package com.pearson.lagp.vinb;

+imports

public class VampiresInBackyard extends BaseAugmentedRealityGameActivity
implements IOrientationListener {
. . .
   // ============================================================
   // Fields
   // ============================================================

. . .
   private Texture mFontTexture;
   private Font mFont32;
   private ChangeableText mCurrHeading;

. . .
   @Override
   public Engine onLoadEngine() {
. . .
      nf = NumberFormat.getInstance();
```

```
      nf.setMaximumFractionDigits(2);
      nf.setMinimumFractionDigits(2);
. . .
   }

   @Override
   public void onLoadResources() {
. . .
         this.mFontTexture = new Texture(256, 512,
            TextureOptions.BILINEAR_PREMULTIPLYALPHA);
         FontFactory.setAssetBasePath("font/");
         mFont32 = FontFactory.createFromAsset(
            this.mFontTexture, this, "Flubber.ttf",
            32, true, Color.RED);
         mEngine.getTextureManager().loadTexture(
            this.mFontTexture);
         mEngine.getFontManager().loadFont(this.mFont32);
   }

   @Override
   Public void onLoadScene()
. . .
      mCurrHeading = new ChangeableText(0.5f*CAMERA_WIDTH, 10.0f,
         mFont32, "Heading: 0", "Heading: XXXXXX".length());
      scene.getLastChild().attachChild(mCurrHeading);
. . .
   @Override
   protected void onResume() {
      super.onResume();
      this.enableOrientationSensor(VampiresInBackyard.this);
   }

   @Override
   protected void onPause() {
      super.onPause();
      this.mEngine.disableOrientationSensor(this);
   }
   public void onOrientationChanged(OrientationData pOD) {
      // Update compass display
      float head = pOD.getYaw();
           mCurrHeading.setText("Heading: " + nf.format(head));
   }
}
```

Chapter 16

1. The changes needed to `Level1Activity.java` are shown here:

Changes to Level1Activity.java to Stop Runnables

```
. . .
   @Override
   public void onGamePaused() {
      super.onGamePaused();
      mGunshotSound.stop();
      mExploSound.stop();
      mOCSMusic.stop();
      mSaveMeSound.stop();
      mActivityVisible = false;
      mHandler.removeCallbacks(mStartSarah);
      mHandler.removeCallbacks(mStartVamp);
   }

   @Override
   public void onGameResumed() {
      super.onGameResumed();
      mActivityVisible = true;
      mHandler.removeCallbacks(mStartSarah);
      mHandler.removeCallbacks(mStartVamp);
      mHandler.postDelayed(mStartVamp,mVampRate);
      if (mDistract) mHandler.postDelayed(mStartSarah,5000);
   }
. . .
```

   The `removeCallbacks()` calls in `onGamePaused()` are fairly obvious, but why are they repeated in `onGameResumed()`? The answer is that `onGameResumed()` is also called as part of normal Activity startup. When the gamelet is first starting, we don't want Sarah and the vampires to appear until we get to that point in `onLoadScene()`.

2. The changes to `OptionsActivity.java` are fairly obvious—just delete the `menuItems` for Whack-A-Vampire and Irate Villagers. We've commented them out in the downloadable code in the `V316Exercises` folder. The changes to `StartActivity.java` are shown here:

Changes to StartActivity.java to Add Invisible Buttons

```
. . . // ============================================================
   // Fields
   // ============================================================
. . .
```

```
   private TextureRegion mBlackButtonTextureRegion;
. . .
   @Override
   public void onLoadResources() {
. . .
      this.mBlackButtonTextureRegion =
         TextureRegionFactory.createFromAsset(this.mTexture,
         this, "blackbutton.png", 0, 330);
. . .
   @Override
   public Scene onLoadScene() {
. . .
      /* Create buttons for WAV and IV */
      final Sprite WAVButton = new Sprite(CAMERA_WIDTH - 32,
         CAMERA_HEIGHT - 64, mBlackButtonTextureRegion){
         @Override
         public boolean onAreaTouched(
            final TouchEvent pAreaTouchEvent,
            final float pTouchAreaLocalX,
            final float pTouchAreaLocalY) {
             mHandler.removeCallbacks(mLaunchMenuTask);
             mHandler.post(mLaunchWAVTask);
             return true;
         }
      };
      mScene.registerTouchArea(WAVButton);
      mScene.setTouchAreaBindingEnabled(true);
      mScene.getLastChild().attachChild(WAVButton);

      final Sprite IVButton = new Sprite(CAMERA_WIDTH - 32,
         CAMERA_HEIGHT - 32, mBlackButtonTextureRegion){
         @Override
         public boolean onAreaTouched(
            final TouchEvent pAreaTouchEvent,
            final float pTouchAreaLocalX,
            final float pTouchAreaLocalY) {
             mHandler.removeCallbacks(mLaunchMenuTask);
             mHandler.post(mLaunchIVTask);
             return true;
         }
      };
      mScene.registerTouchArea(IVButton);
      mScene.getLastChild().attachChild(IVButton);
. . .
   @Override
   public void onDestroy() {
      super.onDestroy();
```

```
      mHandler.removeCallbacks(mLaunchMenuTask);
      mHandler.removeCallbacks(mLaunchWAVTask);
      mHandler.removeCallbacks(mLaunchIVTask);
   }
. . .
     private Runnable mLaunchWAVTask = new Runnable() {
      public void run() {
         Intent myIntent = new Intent(StartActivity.this,
         WAVActivity.class);
         StartActivity.this.startActivity(myIntent);
      }
     };

   private Runnable mLaunchIVTask = new Runnable() {
      public void run() {
         Intent myIntent = new Intent(StartActivity.this,
            IVActivity.class);
         StartActivity.this.startActivity(myIntent);
      }
   };
. . .
```

3. We can use an Alert Dialog box to display the help string, which we've added to `strings.xml` in the `res` folder. The modifications to `MainMenuActivity.java` are shown here:

Changes to MainMenuActivity.java to the Help Screen

```
. . .
   @Override
   public boolean onMenuItemClicked(final MenuScene pMenuScene,
      final IMenuItem pMenuItem, final float pMenuItemLocalX,
      final float pMenuItemLocalY) {
. . .
         case MENU_HELP:
            mShowHelp();
            return true;
. . .
    private void mShowHelp() {
        String title = this.getString(R.string.app_name);
        String message = this.getString(R.string.help);

        AlertDialog.Builder builder = new AlertDialog.Builder(this)
                .setTitle(title)
                .setCancelable(false)
                .setMessage(message)
```

```
            .setPositiveButton(R.string.help_dialog_play,
        new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int whichButton) {
                mMainScene.registerEntityModifier(
        new ScaleAtModifier(0.5f, 1.0f, 0.0f, CAMERA_WIDTH/2,
        CAMERA_HEIGHT/2));
                mStaticMenuScene.registerEntityModifier(
        new ScaleAtModifier(0.5f, 1.0f, 0.0f, CAMERA_WIDTH/2,
        CAMERA_HEIGHT/2));
                mHandler.postDelayed(mLaunchLevel1Task,500);
           }
        })
        .setNegativeButton(R.string.help_dialog_cancel,
      new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
               dialog.dismiss();
            }
        });
        builder.create().show();
    }
```

This page intentionally left blank

Index

A

B

C

D

E

F

G

M

N

O

P

Q

R

S

T

U

V

W

X

Z