Community Experience Distilled

LibGDX Game Development Essentials

Make the most of game development features powered by LibGDX and create a side-scrolling action game, Thrust Copter

Juwal Bose

[PACKT] PUBLISHING open source* community experience distilled

LibGDX Game Development Essentials

Make the most of game development features powered by LibGDX and create a side-scrolling action game, Thrust Copter

Juwal Bose

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

LibGDX Game Development Essentials

First published: December 2014

Production reference: 1181214

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-929-0

www.packtpub.com

Credits

Author
Juwal Bose

Reviewers
Pavel Czempin
Lévêque Michel
Sudarshan Shetty
Tom Wojciechowski

Commissioning Editor
Dipika Gaonkar

Acquisition Editor
Meeta Rajani

Content Development Editor
Anila Vincent

Technical Editor
Madhunikita Sunil Chindarkar

Copy Editors
Roshni Banerjee
Gladson Monteiro

Project Coordinator
Neha Bhatnagar

Proofreaders
Simran Bhogal
Maria Gould
Ameesha Green
Paul Hindle

Indexer
Hemangini Bari

Production Coordinators
Komal Ramchandani
Sushma Redkar

Cover Work
Komal Ramchandani

About the Author

Juwal Bose is a game developer, game designer, and technology consultant from the incredibly beautiful state of Kerala in India. He is an active figure in social media and game development SIGs, and he never misses a chance to speak at technical conferences and BarCamps. He conducts technical workshops for engineering students at professional colleges as part of open source initiatives. Juwal is the Director at Csharks Games and Solutions Pvt. Ltd., where he manages research and development, training, and pipeline integration.

He has been developing games since 2004 using different technologies, such as ActionScript, Objective-C, Java, Unity, LibGDX, Cocos2D, OpenFL, Unity, and Starling. His team has created more than 400 games to date, and many of the job management games are listed at the top of leading portals worldwide. He was involved in the development of more than 20 LibGDX games, primarily for the Android platform.

Juwal writes game development tutorials for GameDevTuts+ and manages the blog of Csharks. His isometric tutorial for GameDevTuts+ was well received and is considered a thorough guide for developing tile-based isometric games. This is his second book and he aims to keep writing and sharing his 10 years of game development experience through more books. His first book, *Starling Game Development Essentials*, *Packt Publishing*, was on another exceptional cross-platform game development framework—Starling.

Juwal is a voracious reader and likes to travel. His future plans also include writing fiction.

Acknowledgments

I would like to thank my parents and my wife, Dr. Nidhina Haridas, for inspiring me to aim higher and try harder. My heartfelt thanks go to Eldhose P. Mathew, CEO of Csharks, for allowing me to devote my professional time to work on this book. I would also thank Anila Vincent, Meeta Rajani, and Madhunikita Chindarkar for supporting me and all the other team members at Packt Publishing for giving me this opportunity. A special thanks to Suryakumar for those brainstorming sessions and Kenney for the open source art.

This book would not have been possible without the expertise of the technical reviewers Tomasz Wojciechowski, Lévêque Michel, Pavel Czempin, and Sudarshan Shetty. Thank you guys.

I would also like to express my deep gratitude to Mario Zechner, without whom such an incredible game framework would not have existed. Mario, you rock. Long live LibGDX!

About the Reviewers

Pavel Czempin is currently completing school in Germany and plans to study at a university. He has also joined a program to complete some of the advanced courses of computer science. There, among other things, he learned programming in Java. In his free time, he works on programming projects and is interested in developing computer games.

You can find some of his projects on his GitHub page at `https://github.com/Valep42`.

I thank my father for encouraging me to review this book.

Lévêque Michel has a Bachelor's degree in Information Technology, and he worked in Java development for 7 years. He is currently working on a LibGDX point-and-click game as a core programmer.

I would like to thank the author of this book and the team at Packt Publishing for giving me the opportunity to review this great book.

Sudarshan Shetty is a cofounder of Neurolinx Software Technologies along with his wife, Anuradha. He completed his engineering from IIT (BHU) and IIT Bombay. He worked as a sound recordist before getting into programming. He has also worked on Wall Street, developing trading applications. In his free time, he experiments with organic farming and loves to play the guitar with his kids, Shameen and Shamika. To find out more about him, check out his LinkedIn profile at `http://in.linkedin.com/pub/sudarshan-shetty/12/756/`.

Tom Wojciechowski is an independent Java developer currently working on a cross-platform multiplayer mobile game (`www.asidik.com/blobrun`). He holds a Bachelor's degree in Mathematics and Physics, and he has been a core developer for the popular and award-winning LibGDX, a cross-platform Java game development framework.

His current projects and blog can be found at `www.asidik.com`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Game development is one of the coolest things to do right now, be it as a hobby or as a full-fledged career option. With the arrival of smartphones, tablets, and other smart devices, games are leading all the app stores worldwide in terms of the number of downloads and revenue. Games have opened up revenue streams worth millions of dollars for not only game companies, but also for indies and single developers. App stores support self-publishing and they are very indie-friendly, which means anyone capable of creating a game can go ahead and publish their game and reap the benefits. This book will help you with just that.

Developing a game is not an easy task, but with the right tools at our disposal it becomes easier to develop a new game. Depending on the type of game, the platform, and prior knowledge of the developer, there are different types of tools that can be used. This book explores LibGDX, a Java-based game development framework that is primarily used to develop 2D Android games. However, LibGDX is also capable of deploying to Windows, Mac, Linux, iOS, Blackberry, and HTML5. Unity, Starling, Cocos2D, Phaser, and OpenFL are some of the alternatives that you need to be aware of.

LibGDX (`http://libgdx.badlogicgames.com`) is an open source, Java-based game development framework. It provides a setup for rapid prototyping and faster iterations, with the capability to develop and debug on our desktop. It is the brainchild of Mario Zechner, the author of *Beginning Android Games*. The team of LibGDX now has 15 people (`https://github.com/orgs/libgdx/people`). LibGDX is the undisputed leader when it comes to creating Android 2D games that are performant and universal.

This book aims to help you get started with game development using LibGDX. In this book, you will create a simple game that is similar to *Flappy Bird* and learn about the different aspects of game development. This book is bundled with the complete source code of the working game Thrust Copter, along with other sample code relevant to individual chapters. This book will help you to learn about game development, UI creation, and data persistence. Once the game is ready, a chapter is dedicated to convert it to a physics-based game using Box2D. Another chapter introduces basic 3D concepts, tile maps, and other tools of the trade. The final chapters detail how to integrate various third-party features and how to publish the game on various platforms.

The book does not try to teach you everything, but makes you ready to explore LibGDX further and encourages you to explore the source code on your own. I would appreciate it if you would support LibGDX once you are aware of its tremendous potential (`http://www.patreon.com/libgdx`). Once you are done with this book, I would urge you to read *LibGDX Cross-platform Game Development Cookbook* and *Learning LibGDX Game Development*, both by Packt Publishing, to take your knowledge to the next level.

What this book covers

Chapter 1, *Wiring Up*, explains how to set up the LibGDX development environment. We will install Eclipse, Android SDK, and LibGDX via Gradle, which helps us to run our Hello World project.

Chapter 2, *Let There Be Graphics!*, explains how to add game graphics and discusses the Texture, TextureRegion, Sprite, and Animation classes. Texture packing and SpriteBatch are also introduced in this chapter.

Chapter 3, *Thou Shall Not Pass!*, explains how to add game controls for the player to interact with the game and discusses how to add sound effects and music loop.

Chapter 4, *Bring in the Extras!*, explores how to add additional game play elements, such as pickups, GUI, and effects. Bitmap fonts and particle effects are added and their respective tools are also explained in this chapter.

Chapter 5, *Scene 2 – the Menu*, introduces Scene2D and explains the creation of the menu scene along with a loading scene. In this chapter, you will learn about Stage, Actor, and Nine Patch images. This chapter also deals with how to handle Android back button.

Chapter 6, *Physics with Box2D*, explains how to convert the completed game logic to one that uses physics simulation with the help of Box2D. In this chapter, you will learn about the basics of Box2D and use it to create an alternate implementation of the Thrust Copter game.

Chapter 7, *The Amazing World of 3D*, explores the 3D capabilities of LibGDX. With the use of sample code, we will render 3D primitives, import models, and animate them. We will also discuss 3D particle effects and 3D bullet physics in brief.

Chapter 8, *Saving Our Data*, discusses the methods of file access that will help you to preserve persistent data. We will implement a local leaderboard system with this new knowledge. The TMX tile map is also introduced in this chapter.

Chapter 9, *Finishing Our Android Game*, adds analytics tracking, Google Ads, Google Play services, leaderboard, and achievements to the game.

Chapter 10, *Time to Publish*, explains how to publish the game to different platforms and app stores. Overlap2D, a great visual scene designer for LibGDX is also introduced in this chapter.

What you need for this book

LibGDX is a cross-platform game development framework. For development, you will need a computer running either Windows, Linux (for example, Ubuntu), or Mac OS X.

Additionally, you will need to download the LibGDX framework. The Integrated Development Environment (IDE) used in this book is Eclipse. You can download the Eclipse IDE from `http://www.eclipse.org/`.

To develop games for the Android platform, you need the official Android Software Development Kit (SDK) that can be downloaded from `http://developer.android.com/sdk/index.html`.

Who this book is for

If you are a Java developer who wants to learn LibGDX and create great games, this book is for you. Even if you have already worked with LibGDX, I am sure there are sections in this book that can add to your expertise. If you do not know Java but have experience with any other object-oriented language and have created games with them, then I would urge you to try this book as you will be able to learn Java easily. Experience of using Eclipse will also be very useful.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "On a Windows machine, you may need to set the value of the environment variable `JAVA_HOME` to the installation path of the JDK after installation."

A block of code is set as follows:

```
SpriteBatch batch;
Texture img;
create ();
render();
```

Any command-line input or output is written as follows:

```
C:\Users\admin>java -version
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the Windows Start button, right-click on **Computer**, select **Properties**, and click on **Control Panel**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/9290OS_ColoredImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Wiring Up

In this chapter, we will lay the ground work to facilitate our plan of creating a game like Flappy Bird for multiple platforms, including Android, iOS, PC, and the Web. You might wish to jump right in and start designing the game, but that is not the case with LibGDX. There are lots of things to be done before we could start to code. This chapter deals with the setting up of LibGDX for a cross-platform project. This is done on a development environment configured for Android development. We will cover the following topics:

- Installing **Java Development Kit** (**JDK**)
- Installing Eclipse for Java
- Installing Android SDK
- Installing **Google Web Toolkit** (**GWT**) SDK
- Installing all relevant Eclipse plugins, such as Gradle, **Android Development Tool** (**ADT**), **Google App Engine** (**GAE**), GWT, and RoboVM
- Using the LibGDX Gradle setup app to create our first LibGDX project
- Exploring LibGDX demos and tests

Getting started

Hope I didn't scare you with all those abbreviations. The majority of what we will do in this chapter will only need to be done once, as it is required to set up your development environment. Once it is done, we can forget about it and focus on creating wonderful games. I will be using a Mac-based development setup throughout this book, but the process is similar for Windows as well. The added benefit that Mac provides is the ability to deploy to iOS as well. In the first part, we will set up a Java development environment that will be very simple for all our Java developers. This can be accomplished by setting up Eclipse, IntelliJ, or NetBeans IDEs, but we will be focusing on Eclipse throughout this book. This does not mean that this way is superior to the others but we've used it just because Eclipse is the most widely used Java IDE and the majority of our potential readers should have worked with Eclipse one way or another.

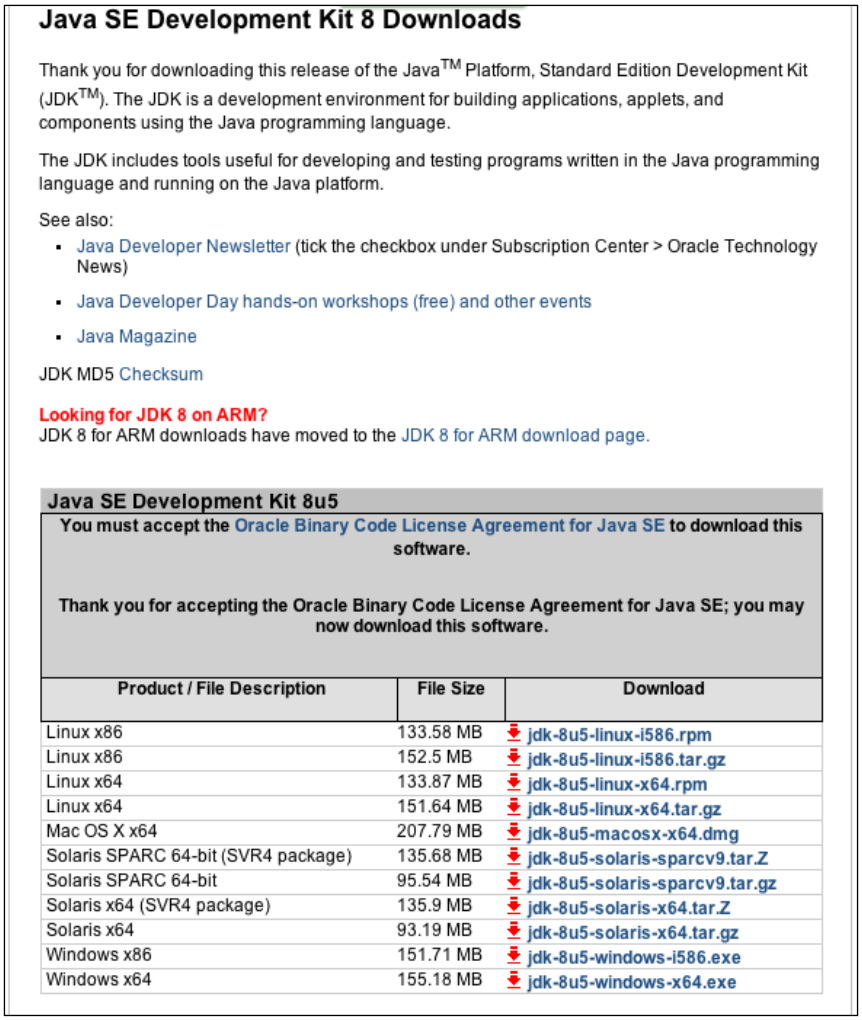
If you need to use another IDE, please check the LibGDX wiki page at `http://libgdx.badlogicgames.com/documentation.html#gettingstarted`.

Setting up the Java development environment

Our first step will be to install the **Java Development Kit** (**JDK**) if the development PC does not have it. For all of the tools, applications, and SDKs that we will use, there are different versions for Windows and Mac. Your browser will automatically take you to the relevant download in most cases, but be sure to double check that you are indeed downloading the right version. Also, there are different versions for 32-bit and 64-bit processors; make sure that you download the right one for your development PC.

You can download the latest version of JDK from Oracle's site at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

At the time of writing this book, the latest version is JDK 8u5 and the following screenshot shows how the different versions are listed:



Java SE Development Kit 8 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

See also:

- Java Developer Newsletter (tick the checkbox under Subscription Center > Oracle Technology News)
- Java Developer Day hands-on workshops (free) and other events
- Java Magazine

JDK MD5 Checksum

Looking for JDK 8 on ARM?
JDK 8 for ARM downloads have moved to the JDK 8 for ARM download page.

Java SE Development Kit 8u5

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux x86	133.58 MB	jdk-8u5-linux-i586.rpm
Linux x86	152.5 MB	jdk-8u5-linux-i586.tar.gz
Linux x64	133.87 MB	jdk-8u5-linux-x64.rpm
Linux x64	151.64 MB	jdk-8u5-linux-x64.tar.gz
Mac OS X x64	207.79 MB	jdk-8u5-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	135.68 MB	jdk-8u5-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	95.54 MB	jdk-8u5-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	135.9 MB	jdk-8u5-solaris-x64.tar.Z
Solaris x64	93.19 MB	jdk-8u5-solaris-x64.tar.gz
Windows x86	151.71 MB	jdk-8u5-windows-i586.exe
Windows x64	155.18 MB	jdk-8u5-windows-x64.exe

Once you download the relevant file, go ahead and install it on your machine. Now, your machine is ready to be used for Java-based application development.

On a Windows machine, you may need to set the value of the environment variable `JAVA_HOME` to the installation path of the JDK after installation. We can check this by running the following command in the command prompt:

```
C:\Users\admin>java -version
```

If the system displays the version details as follows, then Java is installed correctly:

```
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

Otherwise, we need to set up the Java environment variable. To find the correct path, go to `C:\Program Files (x86)\Java\`. You should see a folder with the `jdk` extension in its name. The complete path should be similar to the following one:

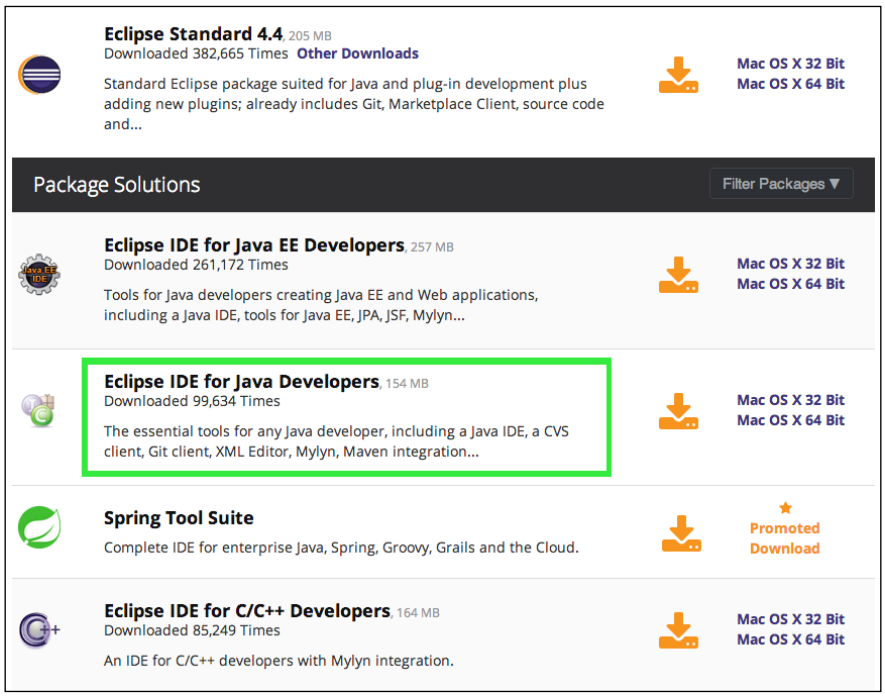
```
C:\Program Files (x86)\Java\jdk1.8.0_05
```

Follow the given steps on a 64-bit Windows machine with 32-bit Java installed to set the environment variable:

1. Click on the Windows Start button, right-click on **Computer**, select **Properties**, and open control panel system window.
2. Click on **Advanced system settings** on the left-hand side of the Control Panel window to open the **System Properties** window.
3. Next, click on the **Environment Variables…** button and click on the **New...** button at the top that corresponds to `User variables for <USERNAME>`. A window with the title **New User Variable** will appear.
4. Now, fill in the two text fields. Use `JAVA_HOME` in the **Variable name** field and the JDK path you found out earlier in the **Variable value** field.

Installing Eclipse

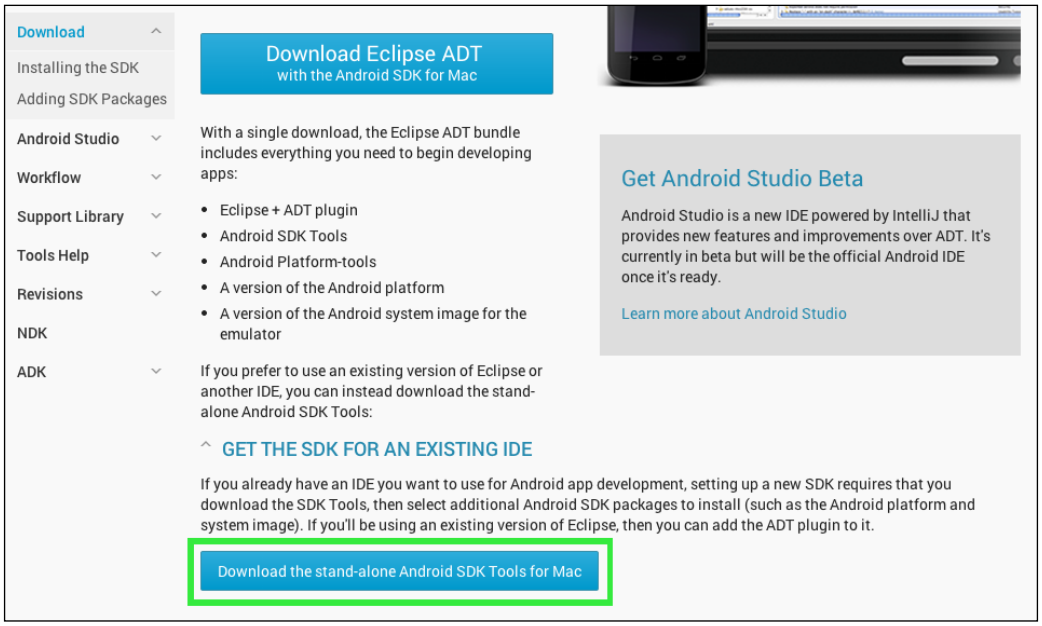
Next, we will need to install our IDE of choice: Eclipse. The latest version of Eclipse can be found at `http://www.eclipse.org/downloads/`. We need to select Eclipse for Java developers from the multitudes of flavors in which it is available. At the time of writing, the latest Eclipse build is **Luna**, as shown in the following screenshot:



Once you have downloaded the relevant IDE, go ahead and install it on your machine. Now, we are all set to write and compile the Java code. It's time to set up the Android development environment.

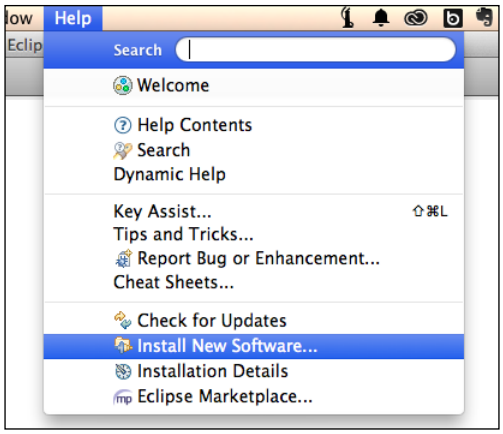
Setting up the Android development environment

To use our Eclipse IDE for Android development, we need to install two things: the Android SDK and the Android ADT plugin. Android SDK can be downloaded as a compressed archive from `http://developer.android.com/sdk/index.html`. It will be listed under **GET THE SDK FOR AN EXISTING IDE**. The archive can be extracted to a location on your hard drive which needs to be linked from within Eclipse. The following screenshot shows the download page where you can see a link to download Eclipse ADT, which is a complete package for Android development. If you are setting up Eclipse for the first time, then downloading Eclipse ADT is the way to go; however, in this chapter, we are assuming that Eclipse is already installed.

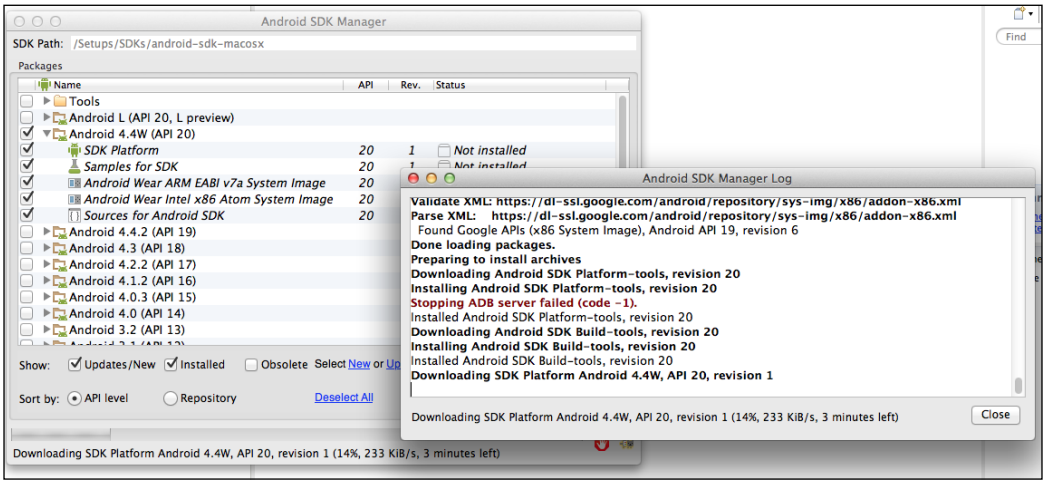


Installing the ADT plugin and linking the Android SDK with Eclipse

The ADT plugin connects with our Android SDK and keeps it up to date. It also has the SDK Manager and Android Virtual Devices that are used to emulate Android devices. We need to fire up Eclipse so that we can install the plugin. For those who are not aware of the process of installing plugins in Eclipse, the following screenshot will help. We need to select **Install New Software...** from the **Help** section.



In the new window that pops up, enter the URL `https://dl-ssl.google.com/android/eclipse/` in the section that says **type or select a site** and press *Enter*. We need to select all available items and proceed with the installation. Once the plugin is successfully installed, we need to restart Eclipse. After restarting, Eclipse will ask you for the Android SDK location; you can also set it up by navigating to **Window | Preferences | Android**. Once the Android SDK location is set, the SDK Manager will check our installation to find missing items. It will prompt you to download the latest Android SDK platform and Android platform tools. Check out the selected items in the **Download** dialog box. We need to have the recommended Android build tools and SDK platform. It is safe to not alter the recommended setting and allow the downloader of Eclipse to download all the required files. The following screenshot shows the update in progress:



Now we are all set to develop Android applications.

Installing the LibGDX support plugins

LibGDX uses GWT plugin to publish HTML5/JavaScript, which is the web platform. GWT includes the GWT SDK and Google App Engine. The other support plugins that we need are RoboVM and Gradle.

RoboVM is used to compile the LibGDX project on the iOS platform. Gradle is a dependency management and build system that wires our LibGDX game together.

Setting up GWT

From within Eclipse, launch the **Install New Software...** window, input the following link, and install the GWT plugin:

```
https://dl.google.com/eclipse/plugin/4.4
```

The last part of the link is actually the Eclipse version and the preceding link is for Version 4.4.x. Select the checkboxes to install **Google Plugin for Eclipse**, **Google App Engine SDK**, and **Google Web Toolkit SDK**. Once the plugin is installed, we need to restart Eclipse.

Installing the RoboVM plugin

iOS development is only possible if we are using a Mac machine for development. We will also need Xcode, the Mac IDE for support. RoboVM is the brainchild of Niklas Therning, a Swedish developer and co-founder of Trillian Mobile AB. RoboVM is used to make Java work on iOS via a Java to Objective-C bridge. RoboVM is open source and stable. Let's all take a few minutes to appreciate this wonderful effort by learning more about RoboVM (`http://www.robovm.com/`).

Mac users can go ahead and install the RoboVM plugin from `http://download.robovm.org/eclipse/`. After restarting, we are now all set to deploy our games on the Apple iOS platform as well.

Installing the Gradle plugin

Gradle is a dependency management system and is an easy way to pull in third-party libraries into your project without having to store the libraries in your source tree. Instead, the dependency management system relies on a file in your source tree that specifies the names and versions of the libraries you need to be including in your application.

Adding, removing, and changing the version of a third-party library is as easy as changing a few lines in that configuration file. The dependency management system will pull in the libraries you specified from a central repository and store them in a directory outside of your project.

If you want to read more about Gradle and get to know how it benefits the LibGDX setup, visit `https://github.com/libgdx/libgdx/wiki/Improving-workflow-with-Gradle`.

Gradle also has a build system that helps with building and packaging your application, without being tied to a specific IDE. This is especially useful if you use a build or continuous integration server, where IDEs aren't readily available. Instead, the build server can call the build system, providing it with a build configuration so it knows how to build your application for different platforms. More information can be found at `http://www.gradle.org/`.

At this point, we are very comfortable adding new plugins to Eclipse. Fire up the new software window to add the Gradle plugin from the link `http://dist.springsource.com/release/TOOLS/gradle`. After installation, restart Eclipse and that's the end of our setup.

The LibGDX Gradle combo

I know that you are wondering how we reached the end of our setup without installing anything related to LibGDX but everything else out there. The LibGDX installation will be handled by Gradle automatically, and we will be using a helper application to create all the dependencies and the project structure. Such a LibGDX Gradle combo requires an Internet connection while creating the project, as Gradle needs to load all the necessary dependencies, files, and libraries on the fly while setting up the project for the first time. Personally, I am not a fan of this as I come from a country where a good Internet connection is still a luxury. For those of you who are in a similar situation, the alternative way will be to download all dependencies and wire them all up as required. This is a complicated task although we used to do it during the initial days of LibGDX.

Let's start creating our first Gradle-based LibGDX project. We will be following the steps explained in the LibGDX wiki page, which can be found at `https://github.com/libgdx/libgdx/wiki/Project-Setup-Gradle`.

Using the Gradle setup application

We need to use a Java application with the Gradle setup to create Gradle-based LibGDX projects. This setup file can be found at `http://libgdx.badlogicgames.com/download.html`.

A direct link to the setup application is `http://bitly.com/1i3C7i3`.

The setup application has to be stored for easy access, as we will need it when we create new LibGDX projects. The purpose of the application is to create all platform-specific Eclipse projects, such as desktop, Android, iOS, and JavaScript applications. It also links with all the necessary dependency libraries and the latest version of LibGDX. The following screenshot shows the Gradle project structure that clearly explains how the different projects are created under the relevant folders:

Project layout

```
settings.gradle           <- definition of sub-modules. By default core, desktop, androic
build.gradle              <- main Gradle build file, defines dependencies and plugins
gradlew                   <- script that will run Gradle on Unix systems
gradlew.bat               <- script that will run Gradle on Windows
gradle                    <- local gradle wrapper
local.properties          <- Intellij only file, defines android sdk location

core/
    build.gradle          <- Gradle build file for core project*
    src/                  <- Source folder for all your game's code

desktop/
    build.gradle          <- Gradle build file for desktop project*
    src/                  <- Source folder for your desktop project, contains Lwjgl launc

android/
    build.gradle          <- Gradle build file for android project*
    AndroidManifest.xml   <- Android specific config
    assets/               <- contains for your graphics, audio, etc.  Shared with other p
    res/                  <- contains icons for your app and other resources
    src/                  <- Source folder for your Android project, contains android lau

gwt/
    build.gradle          <- Gradle build file for the html project*
    src/                  <- Source folder for your html project, contains launcher and h
    webapp/               <- War template, on generation the contents are copied to war.


ios/
    build.gradle          <- Gradle build file for the ios project*
    src/                  <- Source folder for your ios project, contains launcher
```

A Hello World project

Let's launch the `gdx-setup.jar` Gradle setup application. The following screenshot shows how I have populated the options for our Hello World project:

We need to specify a name for our project, which is `Hello World` in this case. Then, we need to specify a package name for our project. In my case, I am using my company's package name but you can use any unique package name. It's time to specify our main class name; take care that there are no spaces in between. While setting the destination project, make a new folder within your `Eclipse Workspace` folder so that there are no possible conflicts between Gradle files and Eclipse workspace metadata files. In this case, I have specified `FirstGradleProject`. Link the Android SDK and select the version of LibGDX that you will use.

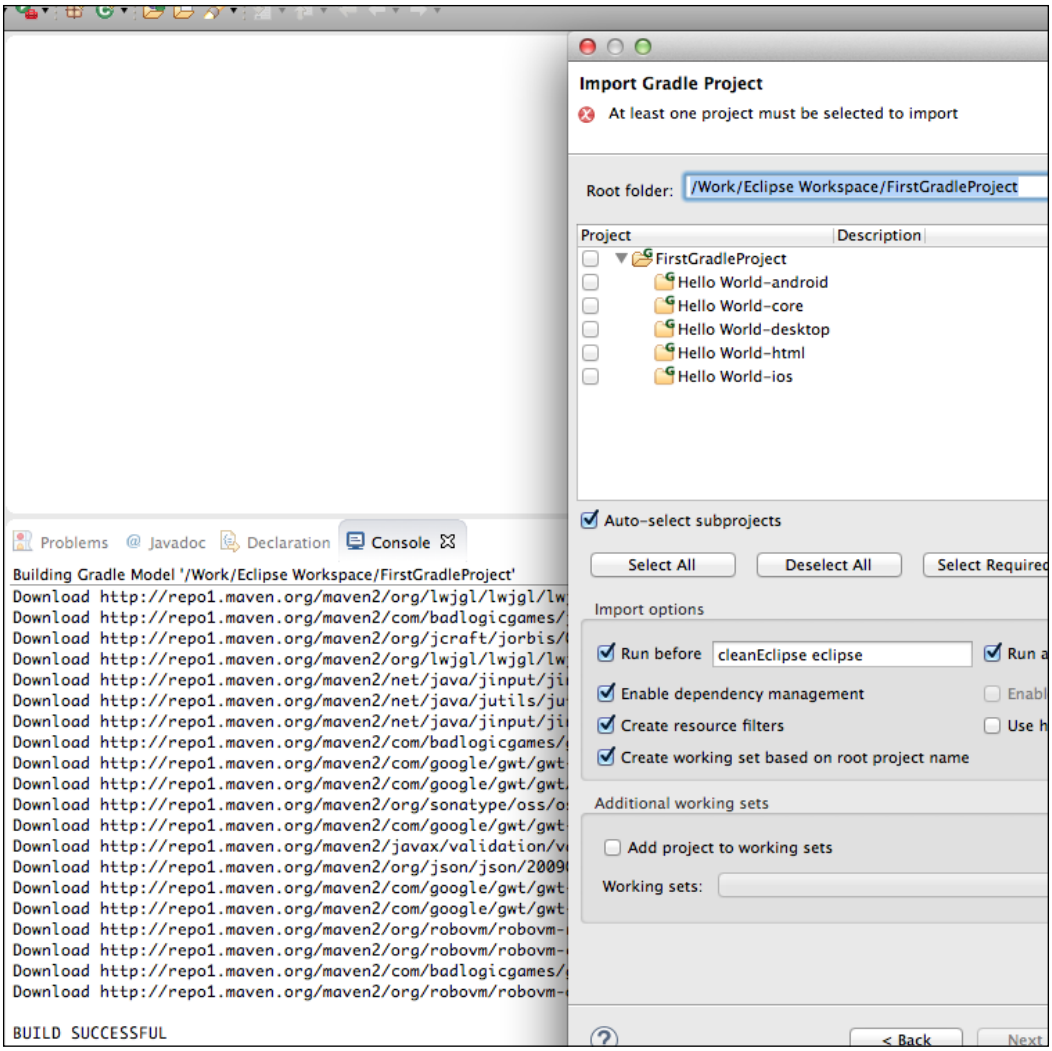
In the **Sub Projects** section, we need to select the platforms that we want the project to target. Note that we will need a Mac to compile an iOS build.

The **Extensions** section will link any of the standard LibGDX libraries or packages available. It is always safe to select the options in this stage rather than hacking into the project later on. For our Hello World app, we do not need any of these extensions. Hit the **Generate** button and your LibGDX Gradle project structure will be created.

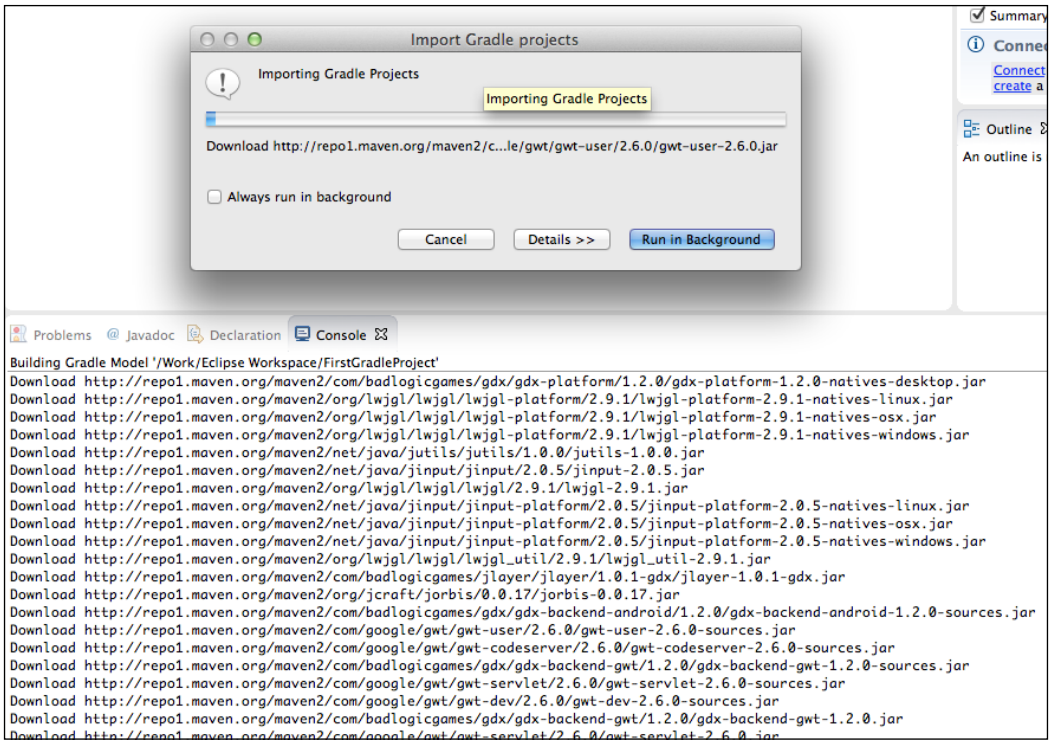
Alternatively, we can create an Eclipse-specific project structure by clicking on the **Advanced** button, enabling the Eclipse checkbox, and clicking on **Save**. If the project is created this way, then we need to import it as a normal Eclipse project by navigating to **File** | **Import** | **General** | **Existing Projects to Workspace**. This process will need an active Internet connection to load the files needed by the setup application.

Importing Gradle projects to Eclipse

It's time to import our projects into Eclipse by navigating to **File** | **Import** | **Gradle** | **Gradle Project**. Browse the root folder, `FirstGradleProject`, and hit the **Build Model** button. This step is very important, as you may not see your projects without this. Gradle will download a few necessary files based on our project, and it will take a while before you see your projects listed as shown in the following screenshot:

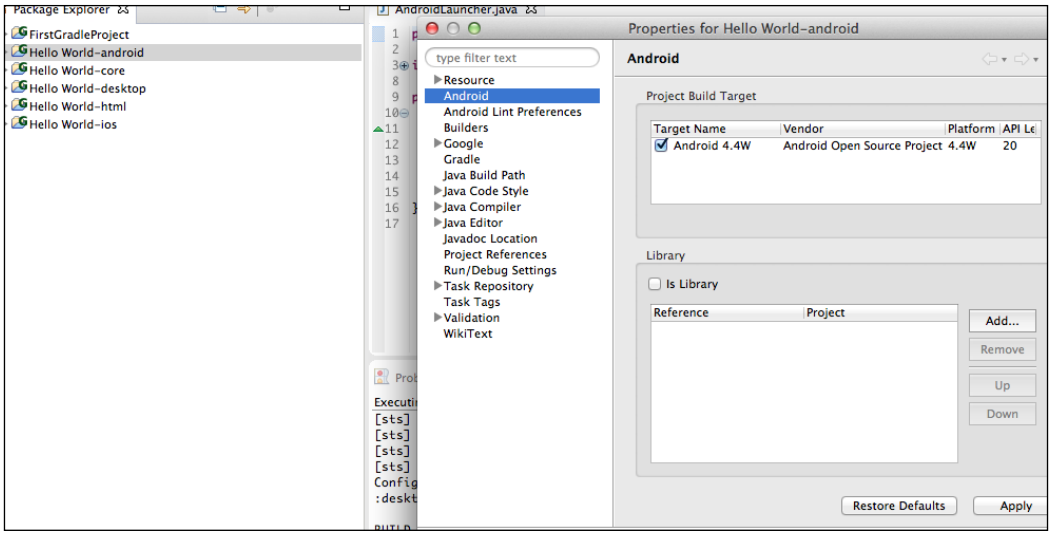

Import Gradle Project
At least one project must be selected to import
Root folder: /Work/Eclipse Workspace/FirstGradleProject
Project
Description
FirstGradleProject
Hello World-android
Hello World-core
Hello World-desktop
Hello World-html
Hello World-ios
Auto-select subprojects
Select All
Deselect All
Import options
Run before
cleanEclipse eclipse
Enable dependency management
Create resource filters
Create working set based on root project name
Additional working sets
Add project to working sets
Working sets:
< Back
Problems
Javadoc
Declaration
Console
Building Gradle Model '/Work/Eclipse Workspace/FirstGradleProject'
BUILD SUCCESSFUL

Go ahead and select all the projects and click on **Finish** to load them to Eclipse. At this point Gradle will load all the dependencies as per our selection in the setup application. This is where Gradle will actually load the LibGDX packages and extensions. The following screenshot shows the process:

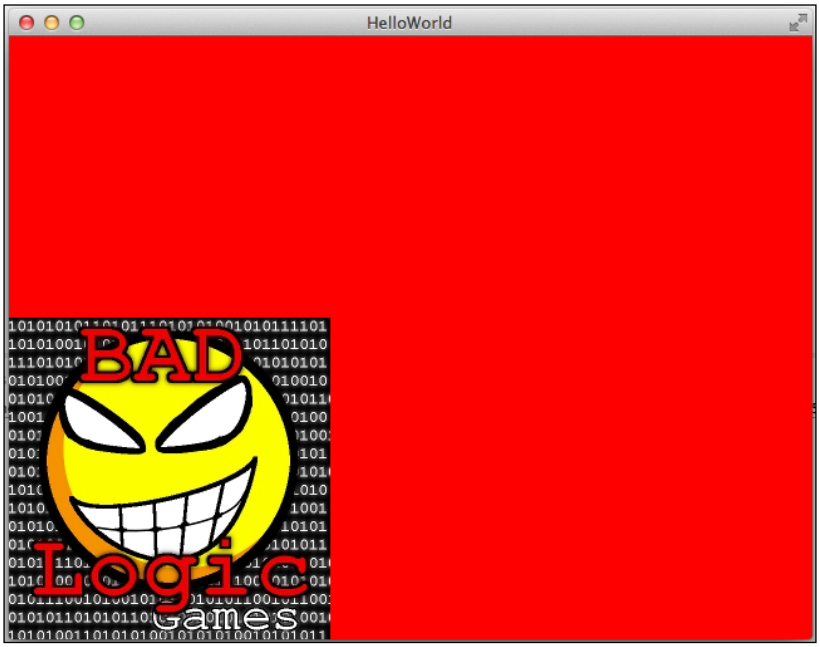


Running the project

Everything should be wired properly and all projects are good to go at this point. One issue that may pop up is that the Android project may have a red cross indicating that the proper Android SDK is missing. This happens all the time, but the fix is straightforward. Right-click on the `Android Project` folder to select its properties. Then, select the proper Android target version under the **Android** section in the window that pops up. This will remove the error. Congratulations! You can now run your projects. The following screenshot shows how to remove the Android error:



In order to run the desktop project, right-click on the Hello World desktop project, select **Run As**, and click on **Java Application**. A popup may ask you to select the application class and you should select **DesktopLauncher** to run the app. The following window will pop up, which means we have successfully created and run our first Gradle-based LibGDX application:

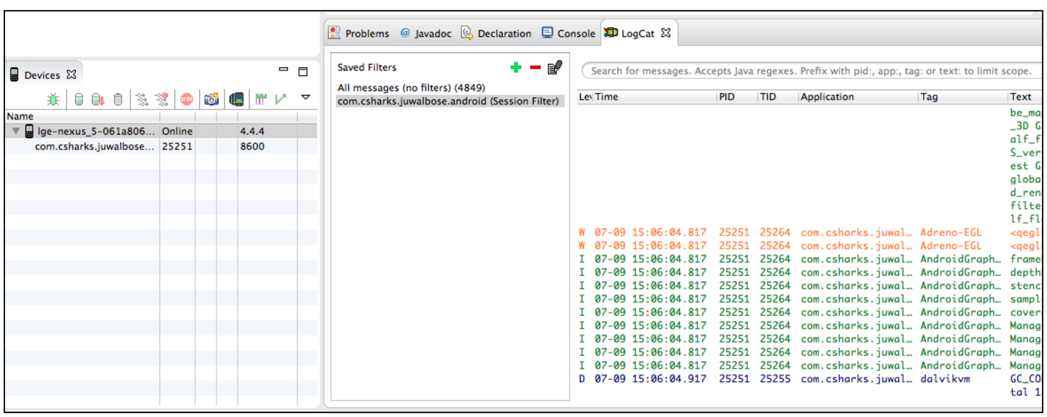


Running the application on your Android device is also very easy. Connect your Android device to the development PC via USB and make sure USB debugging is enabled in the device's settings.

> USB debugging can be enabled by navigating to **Settings** | **Developer options** | **USB debugging**.

In order to see the connected device from within Eclipse, you need to enable the **Devices** view. In Eclipse, go to **Window** | **Show View** | **Other**. Then, select **Devices** from the **Android** section. A new tab showing the connected device will be added to Eclipse. For Windows, we need to install the respective drivers for the connected phone to show up, but on Mac the connected device would be automatically detected.

Once you see your device listed in the **Devices** view, right-click on the `Hello World-android` project, select **Run as**, and click on **Android Application**. Eclipse will prompt you to select how to launch the Android app. You can select to launch on a device or on a **Android Virtual Device** (**AVD**) if you have set one up already. Select to launch the application on the device and then the application should show up on your Android device. Eclipse will show the **LogCat** view, which shows the application status:



In order to run the iOS project, you need to be on a Mac and should have Xcode Version 5 or above installed. Xcode can be downloaded free from the Mac App Store. It provides the necessary frameworks and the iOS Simulator tool. Right-click the `Hello World-ios` project, select **Run As**, and select **iOS Simulator App**. You can select either of the simulator options, iPad or iPhone. This will start the RoboVM cross compilation and would take some time for entire classes to be compiled. Eventually, you will see the app running on the iOS Simulator.

In some cases, you may run out of memory while RoboVM works on the compiling. You may need to increase the memory heap sizes for Eclipse in the `eclipse.ini` file, as follows:

```
-Xms512m
-Xmx2048m
```

Getting the HTML project to run is the trickiest part of them all. We need to follow exactly what the LibGDX wiki tell us to do. Right-click on the `Hello World-html` project, select **Run As**, and click on **External Tools Configuration**. Create a new configuration by double-clicking the **Program** entry in the left sidebar. Give the configuration a name, for example, `GWT SuperDev`. Set the location field to the `gradlew.bat` (Windows) or `gradlew` (Linux, Mac) file. Set the working directory to the root folder of your project and specify `html:superDev` as the argument. Click on **Apply** and then click on **Run**.

Wait until you see the message **The code server is ready in the console view**. After that, go to the URL `http://localhost:8080/html`. You can leave the server running. If you change the code, simply click on the **SuperDev Refresh** button in the browser. This will recompile your app and reload the site.

Check out this screenshot to see the app running on a browser:

Exploring the project

That was some heavy lifting; now we can take a break to explore the project in detail. You are seeing an image in the bottom-left corner of the app window. Let's see how we got this output. The core code logic will always be present in the project's `src` folder, which is the `Hello World-core src` folder in our case. We can find the `HelloWorld.java` main class inside the specified package; open it up. This class extends the `ApplicationAdapter` class, which in turn implements the `ApplicationListener` interface. These classes have a few methods that align with the Android application life cycle methods: `create`, `pause`, `resume`, `dispose`, `render`, and `resize`.

The `HelloWorld` class is a very simple class with only two variables and two overridden functions:

```
SpriteBatch batch;
Texture img;
create ();
render();
```

The `SpriteBatch` class is a class that facilitates the efficient drawing of images on the screen. The `create` function, which will be called when the application is launched, just creates a new `SpriteBatch` class and a `Texture` class named `img` that loads a texture from the external asset `badlogic.jpg`. The `render` function is called continuously and can be considered as our game loop. It clears the screen and draws the texture onto screen at the coordinates (0,0). This means the origin of the coordinate system in LibGDX is the bottom-left corner of the screen, as we can see that the image is placed there. We will revisit the graphics package in the next chapter.

The following code clears the screen before each draw call:

```
Gdx.gl.glClearColor(1, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

The next bit of code starts the batch drawing, draws the texture at the specified coordinates, and closes the batch drawing:

```
batch.begin();
batch.draw(img, 0, 0);
batch.end();
```

The only thing missing is the external image file, as we won't be able to find it among the core project files. LibGDX follows the convention to store all asset files within the `assets` folder in the Android project. Hence, we can find `badlogic.jpg` within the `assets` folder in the `Hello World-android` project. The `assets` folder is shared among all the other projects. All the other projects are simple wrapper projects that have platform-specific code and data that will launch the code in the core project. You can easily explore the projects on your own, but we will revisit specific platforms in the final chapter.

Displaying the Hello World text

Let's alter the code to display our `Hello World` text thereby officially declaring the start of our quest of mastering LibGDX. The easiest way to do this will be to draw some text on the screen using the `BitmapFont` class. A bitmap font is created using an external tool like **Hiero** (`https://github.com/libgdx/libgdx/wiki/Hiero`) where a font is converted into a bitmap with all the letters in a fixed size along with a `.fnt` file that stores the presentation data. To make things easier, let's use fonts from the LibGDX test project. You can download `verdana39.fnt` and `verdana39.png` from `https://github.com/libgdx/libgdx/tree/master/tests/gdx-tests-android/assets/data`. As explained, we need to place these files in the `assets` folder within the `Hello World-android` project.

Now, let's edit the code in `HelloWorld.java` in the core project to remove the texture drawing and to add our text display. The code is as follows:

```
public class HelloWorld extends ApplicationAdapter
{
  SpriteBatch batch;
  BitmapFont font;

  @Override
  public void create ()
  {
    batch = new SpriteBatch();
    font = new BitmapFont(Gdx.files.internal("verdana39.fnt"),
    Gdx.files.internal("verdana39.png"), false);
    font.setColor(Color.RED);
  }

  @Override
```

```
    public void render ()
    {
      Gdx.gl.glClearColor(0, 0, 0, 1);
      Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
      batch.begin();
      font.draw(batch, "Hello World", 200, 200);
      batch.end();
    }
    @Override
    public void dispose()
    {
      batch.dispose();
      font.dispose();
    }
}
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The `BitmapFont` class requires the `.fnt` and `.png` file to create the font. In the `render` function, we draw the **Hello World** text at the coordinates (200,200). I have also added an overridden `dispose` function that releases the memory by destroying the created instances. Congratulations! You have our shining `Hello World` text.

This method may not always be suitable to display text, as we may need to show text in multiple sizes. In such cases, we will use the `Freetype` extension that enables us to use a `ttf` font to create bitmap fonts of different sizes dynamically at runtime. We will revisit fonts in detail in *Chapter 4*, *Bring in the Extras!*

Running demos

For the adventurous among you, this is the chance to dive deeper into LibGDX by checking out the fully functional game demos. You can find the links for these games at `https://github.com/libgdx/libgdx/wiki/Running-Demos`. The most interesting one is the example project similar to Flappy Bird, **The plane that couldn't fly good**. The link to this project is `https://github.com/badlogic/theplanethatcouldntflygood`. In order to explore this project, we need to clone the repository in Git or download it on our PC. Once downloaded, we can import it to Eclipse as a Gradle project.

The `local.properties` file may be missing from the downloaded project's folder. Just copy the file from our `FirstGradleProject` root folder. This file sets the Android SDK location in our PC.

This is a very simple project to get you started and can be used as a starter code for your Flappy Bird game. The whole code is in the single file `PlaneGame.java` in the core project.

Running tests

The LibGDX tests project is a treasure trove of goodies. There are numerous small tests exploring the different features of LibGDX. These tests are the ideal way to get started with LibGDX and they also let you learn new things in an easy way. Alternatively, they act as a source of functional sample code that we can simply copy and paste for our own use cases. The LibGDX wiki entry for running these tests advises us to use Ant to set up these tests on our PC. The link is `https://github.com/libgdx/libgdx/wiki/Running-Tests`. The tests can be found at `https://github.com/libgdx/libgdx/tree/master/tests/gdx-tests/src/com/badlogic/gdx/tests`.

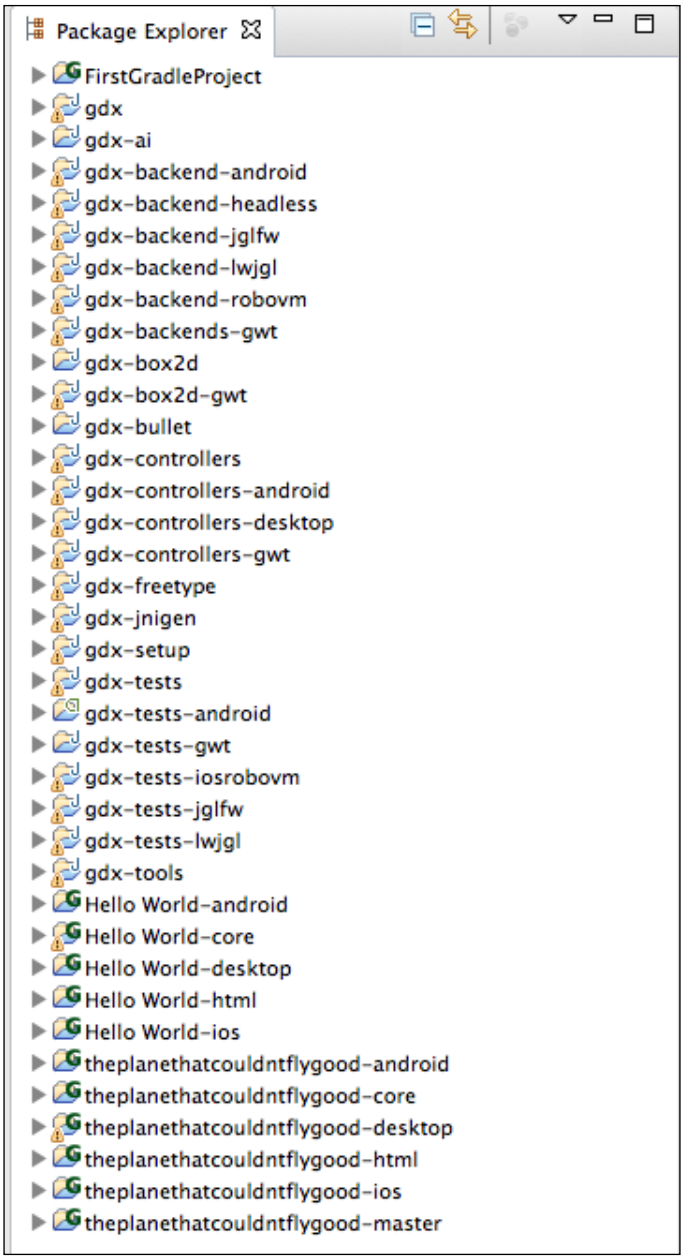
You can temporarily set up Ant after downloading it from `http://ant.apache.org/bindownload.cgi`. Assuming that it is extracted to `/Setups/apache-ant-1.9.4`, we can temporarily install ANT using the terminal on a Mac. In the terminal, enter the following code:

```
export ANT_HOME=/Setups/apache-ant-1.9.4
export PATH=${PATH}:${ANT_HOME}/bin
```

In the terminal, navigate to the folder where LibGDX Git project is cloned. Run the Ant command to fetch dependencies:

```
ant -f fetch.xml
```

This will download all the LibGDX dependencies for all the projects. Later, we can import all these projects into Eclipse by navigating to **Import | Existing Projects into Workspace**. After importing the `Demo` and `Tests` projects, our **Package Explorer** in Eclipse will look something like this:



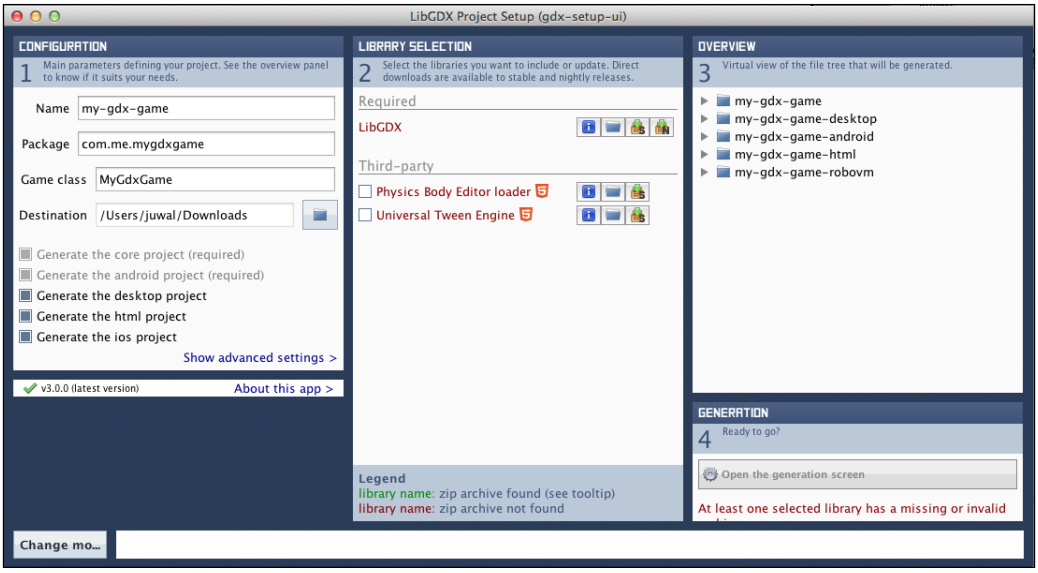
The test examples are contained in the `gdx-tests` project. This project only contains the source code. To actually start the tests on the desktop, you have to run the `LwjglTestStarter` class contained in the `gdx-tests-lwjgl` project. To run the tests on Android, simply fire up the `gdx-tests-android` project in the emulator or on a connected device! To run the tests in your browser, fire up the `gdx-tests-gwt` project. For iOS, you can start the `gdx-tests-robovm` project. Play with them.

The alternate LibGDX setup

Previously, LibGDX had to be manually wired up and was a complicated process. Later, the French developer Aurelien Ribon created a gdx-setup-ui tool that automated this process. Aurelien Ribon is a rock star Java developer who has released lots of goodies that can be accessed on his blog at `http://www.aurelienribon.com/blog/`.

This non-Gradle-based system still works but is not recommended or supported at present. The blog post with all the details of Version 3.0.0 can be found at `http://www.aurelienribon.com/blog/2012/09/libgdx-project-setup-v3-0-0/`. The `gdx-setup-ui` interface can be downloaded from `https://github.com/libgdx/libgdx-old-setup-ui`.

The app will need an Internet connection to download the LibGDX files; this screenshot shows it in action:



Summary

This chapter explained a lot of theory that lays the solid foundation for all our development. This chapter will serve as a reference to set up your development environment if you move to a new workplace or get a new laptop. You learned to set up Eclipse-based Android and Java development environments. We used the gdx-setup tool to create a Gradle-based LibGDX project, which was successfully imported to Eclipse with the help of associated support plugins. We explored the project structure of a typical LibGDX cross-platform project.

We successfully executed the Hello World project on desktop, Android, iOS, and the browser. Some minor editing of the code helped us display the text **Hello World** using the `BitmapFont` class. It is very important to successfully import the LibGDX `Demo` and `Tests` and we used Ant to get them running.

In the next chapter, we will start with the graphics package in LibGDX and get started with our game.

2
Let There Be Graphics!

In this chapter, we will start creating our game. The most important part of any game is graphics, that is, the visual representation. In this chapter, we will explore the LibGDX graphics package and create the game scene of our Thrust Copter game. We will explore the following topics:

- Creating Thrust Copter LibGDX project
- Learning about the LibGDX graphics package and the g2d package
- Adding relevant images to game scene
- Learning about `Texture`, `TextureRegion`, and `Sprite` classes
- Learning about `TextureAtlas` and `TexturePacking`
- Creating an animated `Sprite` instance
- Learning about `SpriteBatch`
- Implementing time-based animation and moving our plane

The ThrustCopter game

Creating a game has to start with proper planning. In game development circles, there is something called a **Game Design Doc** (**GDD**) that details everything about the game before we start developing it. In our case, we will need to start with a clear idea of what we are going to implement. The name of our game is Thrust Copter and the objective is to navigate a helicopter/plane across an infinite landscape. The game is inspired by Flappy Bird mechanics but is a bit different in its gameplay.

The plane is in a free fall due to gravity and will crash unless we control it. The game world moves sidewise as in Flappy Bird. There will be terrain at the top and bottom and obstacles on the way that the plane needs to avoid. The plane also needs to collect pickups like fuel, shield and stars. The plane will be navigated using taps. When we tap anywhere on screen, a thrust force is applied on the plane. The power of the force will depend on the distance of the tap position from the plane. If the tap position is closer to the plane, the force will be greater. The angle at which the thrust force is applied will also depend on the tap position and plane's position. The thrust force will always be applied away from the tap position, that is, the plane is forced to move away from the tap position. So, we need to navigate the plane using multiple taps at precise positions and fly it for as long as possible. We also need to make sure that we pick up enough fuel to survive. There will be a star pickup for score and shield pickup that makes us indestructible for a short time.

Hope the game's idea is interesting enough and the following screenshot shows how the game will look:

A LibGDX app's life cycle

A LibGDX application has a well-defined life cycle defining the various stages of the game, such as creation, pause, resume, and dispose. By implementing the `ApplicationListener` interface or extending the `ApplicationAdapter` class, we can hook into these life cycle events. When the application launches, the `create` method is called where we can add the initialization code. The `resize` method is called when the game screen is resized. To perform rendering, the `render` method is called 60 times per second, depending on the realizable **Frames Per Second** (**FPS**). We add the game logic in this method. The `pause` method is called when the application loses focus, for example, when a call comes through. In order to get back to action, we have the `resume` method. When the application is destroyed, the `dispose` method is called. We add the code to destroy created items and free the memory using this method. More details can be found at `http://bitiotic.com/blog/2013/05/23/libgdx-and-android-application-lifecycle/`.

Creating the Thrust Copter project

Let's recall what we discussed in *Chapter 1*, *Wiring Up*, to create a new LibGDX project. Here is the list of things that you should do:

1. Fire up the `gdx-setup` application.
2. Provide relevant details, point to a new folder in workspace, and generate a project.
3. Fire up Eclipse and navigate to the newly created project folder from **File** | **Import** | **Gradle** | **Gradle Project**.
4. Select the folder and click on **Build Model**.
5. Once done, select the project and click on **Import**.

In the following screenshot, you can see the settings I used to create the Thrust Copter project:

Planning art assets

To create games, we will obviously need a lot of art. I expect you guys to be programmers and not artists, which leads to the obvious problem of how to procure art for our game. Unless you are the exceptionally rare blend of programmer plus artist, you need to seek the help of an artist or need to look for free art assets. For the purpose of this book, we will use free art available at `http://opengameart.org/users/kenney`. For our purpose, we need the following art assets:

- Animated helicopter art
- Scrollable terrain art
- Sky backdrop art
- Fuel collectible art
- Star collectible art
- Shield collectible art
- Shield effect animation art

- Tap indicator art
- Obstacle rocks
- Pillar art
- Fuel indicator art

We will need some more art for effects and **Heads Up Display** (**HUD**), but for the time being we can start with the preceding list. You will find the necessary assets in the `chapter2.zip` file provided along with this book. It also has the source files for this chapter. We'll go through the source files once we finish the chapter.

The game scene

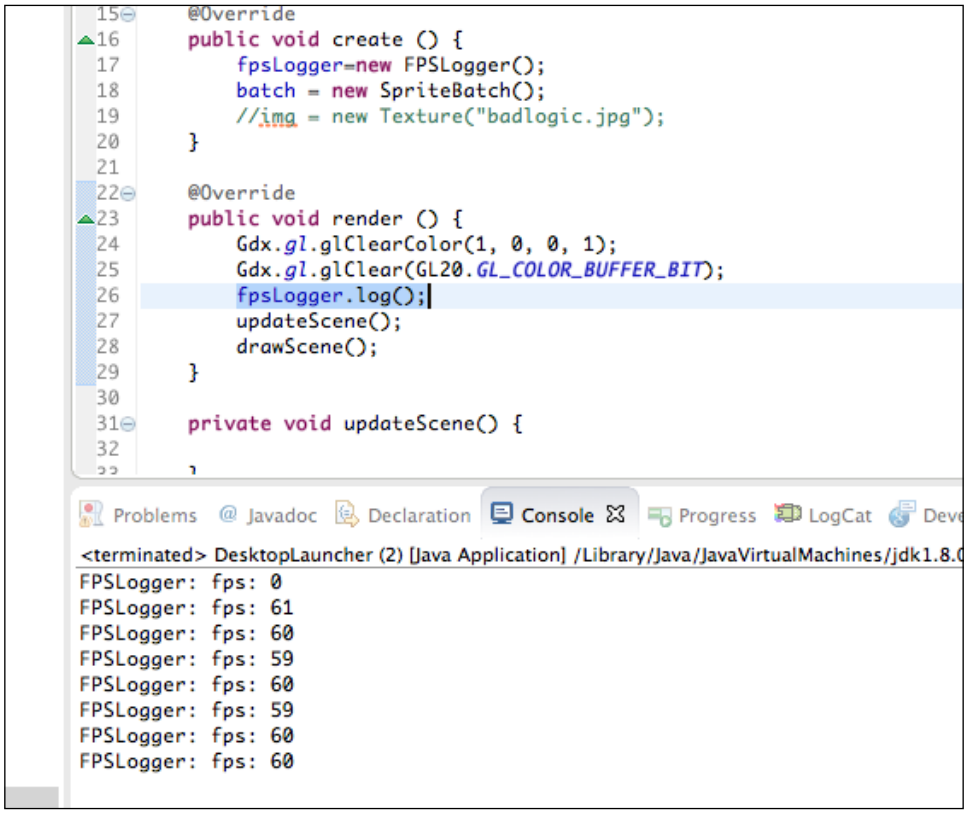
Let's create our game scene by placing all the relevant art. Fire up Eclipse and open our project. The `Thrust Copter-android` project will have the `assets` folder. We need to copy all our art images to this folder. We are using `.png` files as this is the most efficient format that handles transparency with low file size. Remove the `badlogic.jpg` image as we won't be using it anymore. It's time to add items to our game scene, so go ahead and open up the `ThrustCopter.java` file in the `Thrust Copter-core` project. Remove the draw calls from the `render` function and add two new functions, `updateScene()` and `drawScene()`, as shown in the following code. This can be used as a convention where the `updateScene` function applies the game logic there by updating the properties of items in scene, whereas the `drawScene` function draws everything on the screen.

At this point, you can create these functions and leave them empty. Also, it is advisable to add the `FPSLogger` instance to our scene to measure the FPS. This will help us optimize our code if we are not getting a solid 60 fps while running it on mobile devices. Adding an `FPSLogger` instance is as easy as creating a variable of that type and calling its `log()` function from within the `render` method. This is shown in the following code:

```
public void render ()
{
  Gdx.gl.glClearColor(1, 0, 0, 1);
  Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
  fpsLogger.log();
  updateScene();
  drawScene();
}
```

As we are into active development, we will require the help of the LibGDX API documentation that can be found at `http://libgdx.badlogicgames.com/nightlies/docs/api/`.

The `FPSLogger instance` will print our FPS within the Eclipse console when we run the application, as shown in the following screenshot:



Populating the game scene

The first step in populating a scene will be adding a `Camera` class, which acts as a window to our game world. The `Camera` class makes it easier to navigate the game world and most of the complicated matrix projection operations are handled automatically. For our purpose, we need to add an `OrthographicCamera` class. This particular kind of camera is to be used for 2D games where we need orthographic projection. An orthographic projection lacks a vanishing point. All items placed will have the same scale no matter how far or close they are to the camera. Let's create the `OrthographicCamera` instance and set its window to 800 x 480. We also need to add our first image, which can be the background sky image, to the scene. The code to do so is as follows:

```
camera = new OrthographicCamera();
```

```
camera.setToOrtho(false, 800, 480);
background = new Texture("background.png");
```

A camera can be set to any width or height and does not relate to the pixel dimensions of the screen. A typical way to initialize a camera will be as follows, where we take the window size and create a camera that respects the aspect ratio:

```
float w = Gdx.graphics.getWidth();
float h = Gdx.graphics.getHeight();
cam = new OrthographicCamera(30, 30 * (h / w));
```

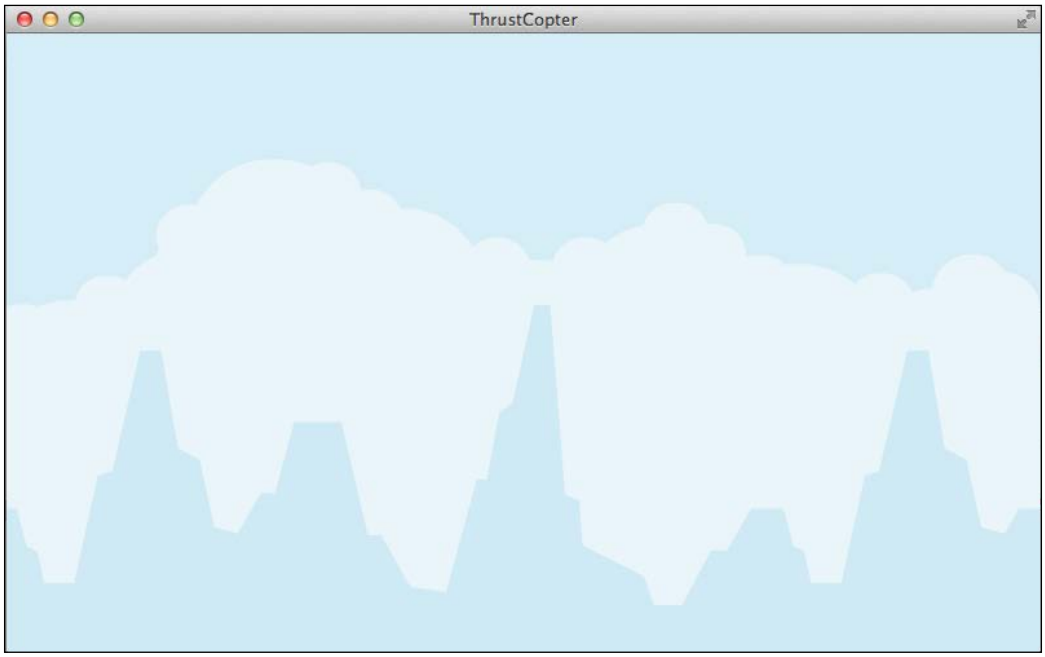
Now let's add some `draw` code in the `drawScene` function, as shown in the following code:

```
private void drawScene()
{
  camera.update();
  batch.setProjectionMatrix(camera.combined);
  batch.begin();
  batch.draw(background, 0, 0);
  batch.end();
}
```

Please note that all the drawing has to take place within the `begin` and `end` methods of the `SpriteBatch` instance, thereby facilitating optimized batching. We can run the desktop project as a Java application to see the background displayed in the window. As we will be developing and viewing the desktop project, we need to set its window size as well. Update the `DesktopLauncher.java` file within the `Thrust Copter-desktop` project as shown in the following code:

```
public static void main (String[] arg)
{
  LwjglApplicationConfiguration config = new
  LwjglApplicationConfiguration();
  config.width=800;
  config.height=480;
  new LwjglApplication(new ThrustCopter(), config);
}
```

This will launch the desktop project in a 800 x 480 window, which is the size of our art and the right aspect ratio. You can see the backdrop in the following screenshot:



So, we successfully added our first image as a `Texture` class. There are other ways to populate the game scene, which we will explore now.

Displaying the graphics

Texture is how an image is uploaded to the **Graphics Processing Unit** (**GPU**), which is later used by the OpenGL context to draw onto the screen. Usually textures are drawn mapped to a rectangular geometry. Texture drawing is an expensive process and any trick that can help make this faster should be employed. This is where the `SpriteBatch` class comes in.

The `SpriteBatch` class can make texture drawing more efficient by collecting all the positions where a specific texture is to be drawn and drawing it in one pass. It batches geometry that we want to send to the GPU to draw. This is effective when the same texture is being drawn at multiple positions. But if the texture changes then it becomes ineffective. The solution is to pack all textures into one big texture and use pieces from this big texture for drawing. This process is called texture packing and we will discuss it later in this chapter.

Another way of drawing graphics is using the `TextureRegion` class. The `TextureRegion` class is a rectangular portion of a bigger texture, but can also be used on single textures. We can alter our current project to draw a `TextureRegion` class instead of a `Texture` class by making some simple changes. Create a `TextureRegion` class named `bgRegion` from the background `Texture` with the size of the image and alter the drawing call to draw it instead of the `Texture` class, which is shown as follows:

```
background = new Texture("background.png");
backgroundRegion=new TextureRegion(background, 800, 480);
```

Here, 800 and 480 are the dimensions of the image that can also be acquired using the `getWidth` and `getHeight` methods of `Texture`. Then, in the `drawScene` method, include the following code:

```
batch.draw(bgRegion, 0, 0);
```

We will get the same output as before.

Another way of displaying graphics is by using a `Sprite` class. A `Sprite` class combines `Texture` with its position and size data, which means we do not need to specify where it should be drawn in the draw code. This makes using `Sprite` the most convenient option in most cases. Let's alter our code to draw a `Sprite` class instead of `TextureRegion`, which follows the same logic as described in the previous code:

```
background = new Texture("background.png");
backgroundSprite=new Sprite(background);
backgroundSprite.setPosition(0,0);
```

In the `drawScene` method, we use the following line of code:

```
backgroundSprite.draw(batch);
```

Please note that we need to explicitly set the position information of the `Sprite` class and we are using the `Sprite.draw(batch)` method instead of using the `batch.draw(graphic)` method.

On checking, we can see that the background image does not need too much flexibility and we can just use a `Texture` class to represent it. We will need the obstacle items to be sprites as they will be moving based on the gameplay. The terrain area on the top and bottom will also need to move and we will be using `TextureRegion` for this, as we need to draw them multiple times to create the seamless scrolling effect.

The final game scene

Let's add the terrain area on the top and bottom as our next step. One important performance tip will be to disable blending for the background texture. Blending is the process of adding translucent pixel values when one texture is drawn over another texture in the scene. So, we do not need any blending for the first texture, which will be the case with the background texture. Once this texture is drawn, we need to enable blending because we need other textures to be overlaid on top of this one. Refer to the following code:

```
batch.begin();
batch.disableBlending();
batch.draw(background, 0, 0);
batch.enableBlending();
//other draw calls
```

To add the terrain area at the top and bottom, we can add the following code to the `create` and `render` methods respectively:

```
terrainBelow=new TextureRegion(new Texture("groundGrass.png"));
terrainAbove=new TextureRegion(terrainBelow);
terrainAbove.flip(true, true);
```

In the `drawScene` method, add the following lines of code:

```
batch.draw(terrainBelow, terrainOffset, 0);
batch.draw(terrainBelow, terrainOffset + terrainBelow.
getRegionWidth(), 0);
batch.draw(terrainAbove, terrainOffset, 480 - terrainAbove.
getRegionHeight());
batch.draw(terrainAbove, terrainOffset + terrainAbove.
getRegionWidth(), 480 - terrainAbove.getRegionHeight());
batch.end();
```

The `terrainAbove` instance of the `TextureRegion` class is flipped in the *x* and *y* axis so that it looks inverted and doesn't look like the `terrainBelow` instance along its width when it scrolls. We draw both `TextureRegion` classes twice so that we can create the illusion of seamless scrolling. The second region is drawn at the point where the first region ends. It will be clear if you consider the value of the `terrainOffset` instance to be zero in the previous draw code.

This `terrainOffset` instance is a float value used to scroll the terrain. We will be increasing or decreasing this value to make the terrain scroll right or left, respectively. To check this, you can add the following code in `updateScene` and run the desktop application:

```
private void updateScene()
{
  float deltaTime = Gdx.graphics.getDeltaTime();
  terrainOffset-=200*deltaTime;
}
```

This will make the terrain scroll left, and I am sure you can figure out how to make it scroll right with a simple change in the code. Please remove the demo code once you are satisfied with the result, as we will be implementing the scrolling based on user input in the next chapter.

In this game, we are trying to move the items right to left in order to give the feel of the plane flying from left to right. When the scene has a lot of items, it becomes complicated to keep track of and move all the items. The alternative is to place everything else static except the plane and move the camera from left to right. This is very easy, as there are only a few items to update per frame.

Another thing that we need to discuss is the value `deltaTime`. The previous code would have worked the same way without this value, as follows:

```
private void updateScene()
{
  terrainOffset-=200;
}
```

This is frame-based animation, where the animation is directly controlled by the frame rate or FPS. In an ideal case, we will get a stable FPS, but practically this might not be the case. Frame rates tend to vary based on the speed of the device, heavy code execution, or other processor load. This means that on a device with a better frame rate, the animation will play better and faster, whereas on a slow device it will play slower. This will break the gameplay experience. The solution to this is time-based animation. The `deltaTime` value represents the time elapsed after the last frame was drawn. On a fast device, this will be less and this value will be more on a slow device, which essentially means that multiplying a value with this gives a result that is the same on both devices.

For example, in a device giving 60 fps, the `deltaTime` value will be 1/60, that is, 0.0167. On a slower device with 40 fps, it will be 1/40, that is, 0.025. For every frame, this value multiplied by 200 is 3.34 and 5 respectively for fast and slow devices. So, for 1 second, it will be 60 x 3.34 and 40 x 5 for these devices respectively, which is approximately 200 (the original value) in both cases. So, the movement in 1 second is 200 on both devices. Did I lose you here? Just go through the math one more time and you will get it. The important point here is that we need to use time-based animation to get the same result on different devices. As we are targeting cross-platform devices, this is a must. The following screenshot shows the terrains added to the backdrop:

Adding the plane animation

So far, we've only been dealing with still images or non-animated items. However, our plane is an animated item that has three separate images. An animation is really a series of still images shown one after the other quickly to make us believe that it is a moving graphic. Based on the speed at which it is shown, our eyes feel the illusion of continuous motion. Hence, we need to play these three images one after the other to make the players feel that the propeller of the plane is actually rotating. For this, we will use the 2D `Animation` class.

Plane animation can be created by providing the frame delay and relevant frames, as shown in the following code:

```
plane = new Animation(0.05f, new TextureRegion(new Texture("planeRed1.
png")),
```

```
    new TextureRegion(new Texture("planeRed2.png")),
    new TextureRegion(new Texture("planeRed3.png")),
    new TextureRegion(new Texture("planeRed2.png")));
    plane.setPlayMode(PlayMode.LOOP);
```

As the plane has a continuous looping animation, we can set `PlayMode` to `LOOP`. To update the animation, we need to explicitly get new frames after the provided frame delay is elapsed. For this purpose, we can initialize a variable named `planeAnimTime` with a value of zero and add the `deltaTime` value to it in the `updateScene` method, as shown in the following code:

```
    float deltaTime = Gdx.graphics.getDeltaTime();
    planeAnimTime+=deltaTime;
```

Then, in the `drawScene` method, we need to get the current frame of the animation by providing the `planeAnimTime` value and draw it as follows:

```
    batch.draw(plane.getKeyFrame(planeAnimTime), 350, 200);
```

The `getKeyFrame` method returns the current frame of the animation based on the time that has passed. Now, we should see the animated plane in the middle of our scene. We can alter the frame delay value to make the animation faster or slower. Play around with it to create the correct feeling. Here is the plane added to the scene, as shown in the following screenshot:

Moving the plane

It is no fun having a plane standing still in the middle of the screen. Let's move our plane by applying some gravity to it. We will need to create some new `Vector2` variables to store the position and velocity of the plane and the value for gravity. The code is as follows:

```
Vector2 planeVelocity= new Vector2();
Vector2 planePosition= new Vector2();
Vector2 planeDefaultPosition= new Vector2();
Vector2 gravity= new Vector2();
```

In the `create` method, let's call a new function `resetScene()`, which actually does what the name says; it resets the scene to the initial state. This function can be called once our plane crashes and we need to restart the game, as shown in the following code:

```
private void resetScene()
{
  terrainOffset=0;
  planeAnimTime=0;
  planeVelocity.set(0, 0);
  gravity.set(0, -2);
  planeDefaultPosition.set(400-88/2, 240-73/2);
  planePosition.set(planeDefaultPosition.x,
  planeDefaultPosition.y);
}
```

The plane is placed in the center of the screen in the previous code, where the dimensions of the plane's image are 88 x 73. In the `updateScene` method, let's apply gravity to the plane using the following code:

```
planeVelocity.add(gravity);
planePosition.mulAdd(planeVelocity, deltaTime);
```

The `Vector2` class has the method `mulAdd`, which multiplies a scalar value with `Vector2` and adds it to the other `Vector2` class. In the previous code, the `mulAdd` method sufficiently performs the time-based animation update that was discussed earlier by multiplying it with `deltaTime`. The plane should be drawn at `planePosition` in the `drawScene` method as shown in the following code:

```
batch.draw(plane.getKeyFrame(planeAnimTime), planePosition.x,
planePosition.y);
```

Finally, our plane is now falling down due to gravity. If you want to move your plane forward as well, then set `planeVelocity` in `resetScene` to values higher than (0,0). Having fun yet? We will have to leave our falling plane for the time being; we will revisit this in the next chapter where we will control the plane's motion using touch inputs.

Feel free to explore the `Vector2` class at `http://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/math/Vector2.html`.

The code so far

Let's review the complete code at this stage:

```
public class ThrustCopter extends ApplicationAdapter
{
  SpriteBatch batch;
  FPSLogger fpsLogger;
  OrthographicCamera camera;
  Texture background;
  TextureRegion terrainBelow;
  TextureRegion terrainAbove;
  float terrainOffset;
  Animation plane;
  float planeAnimTime;
  Vector2 planeVelocity= new Vector2();
  Vector2 planePosition= new Vector2();
  Vector2 planeDefaultPosition= new Vector2();
  Vector2 gravity= new Vector2();
  private static final Vector2 damping= new Vector2(0.99f,0.99f);

  @Override
  public void create ()
  {
    fpsLogger=new FPSLogger();
    batch = new SpriteBatch();
    camera = new OrthographicCamera();
    camera.setToOrtho(false, 800, 480);
    background = new Texture("background.png");
    terrainBelow=new TextureRegion(new
      Texture("groundGrass.png"));
    terrainAbove=new TextureRegion(terrainBelow);
```

```
    terrainAbove.flip(true, true);

    plane = new Animation(0.05f, new TextureRegion(new
      Texture("planeRed1.png")),
      new TextureRegion(new Texture("planeRed2.png")),
      new TextureRegion(new Texture("planeRed3.png")),
      new TextureRegion(new Texture("planeRed2.png")));
    plane.setPlayMode(PlayMode.LOOP);

    resetScene();
  }

  @Override
  public void render ()
  {
    Gdx.gl.glClearColor(1, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    fpsLogger.log();
    updateScene();
    drawScene();
  }
  private void resetScene()
  {
    terrainOffset=0;
    planeAnimTime=0;
    planeVelocity.set(400, 0);
    gravity.set(0, -4);
    planeDefaultPosition.set(400-88/2, 240-73/2);
    planePosition.set(planeDefaultPosition.x,
    planeDefaultPosition.y);
  }
  private void updateScene()
  {
    float deltaTime = Gdx.graphics.getDeltaTime();
    planeAnimTime+=deltaTime;
    planeVelocity.scl(damping);
    planeVelocity.add(gravity);
    planePosition.mulAdd(planeVelocity, deltaTime);
    terrainOffset-=planePosition.x-planeDefaultPosition.x;
    planePosition.x=planeDefaultPosition.x;
    if(terrainOffset*-1>terrainBelow.getRegionWidth())
    {
      terrainOffset=0;
    }
```

```
        if(terrainOffset>0)
        {
          terrainOffset=-terrainBelow.getRegionWidth();
        }
      }
      private void drawScene()
      {
        camera.update();
        batch.setProjectionMatrix(camera.combined);
        batch.begin();
        batch.disableBlending();
        batch.draw(background, 0, 0);
        batch.enableBlending();
        batch.draw(terrainBelow, terrainOffset, 0);
        batch.draw(terrainBelow, terrainOffset +
          terrainBelow.getRegionWidth(), 0);
        batch.draw(terrainAbove, terrainOffset, 480 -
          terrainAbove.getRegionHeight());
        batch.draw(terrainAbove, terrainOffset +
          terrainAbove.getRegionWidth(), 480 -
          terrainAbove.getRegionHeight());
        batch.draw(plane.getKeyFrame(planeAnimTime), planePosition.x,
          planePosition.y);
        batch.end();
      }
    }
```

A new variable which is used is `damping`, which acts as a friction value to reduce the velocity of the plane. In the `updateScene` function the value of `terrainOffset` is checked to see if we need to reset the value. This makes sure that the illusion of seamless terrain is maintained, else we will see the end of the terrain texture.

Texture packing

Earlier, you learned about `SpriteBatch` and how it improves performance when you are drawing the same texture at multiple positions. When we are using individual images as our art asset, we cannot get much advantage from sprite batching. The solution is to draw all the images into one big image and use it as a texture. This process is called texture packing.

Texture packing uses many algorithms that essentially perform concise packing of rectangles. You can learn more about this topic at `http://clb.demon.fi/projects/even-more-rectangle-bin-packing`.

There is a `TexturePacker` class in the gdx-tools project that can be used via command line to pack the images in a folder into texture pages, as shown in the following code:

```
//*NIX (OS X/Linux)
java -cp gdx.jar:gdx-tools.jar
com.badlogic.gdx.tools.texturepacker.TexturePacker inputDir
outputDir packFileName

//WINDOWS
java -cp gdx.jar;gdx-tools.jar
com.badlogic.gdx.tools.texturepacker.TexturePacker inputDir
outputDir packFileName
```

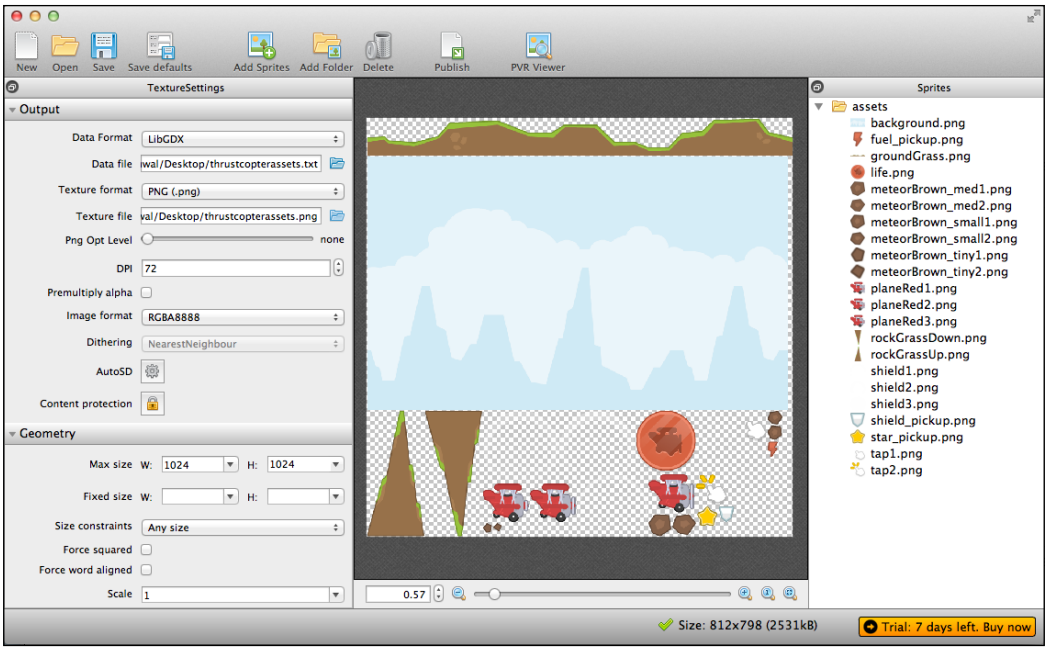
Also, there are commercial products that can be used to create the `TextureAtlas` pages that are the packed texture; one such tool is TexturePacker from CodeAndWeb (`http://www.codeandweb.com/texturepacker`). Rock star Java developer Aurelien Ribon has a free tool exclusively for LibGDX texture packing called TexturePacker GUI, which is available at `http://www.aurelienribon.com/blog/2012/06/texturepacker-gui-v3-0-0/`. We will use this tool as it is free and easy to use. We will use this tool only once to pack the assets and it won't be required during runtime.

To directly download the tool, visit `https://code.google.com/p/libgdx-texturepacker-gui/`.

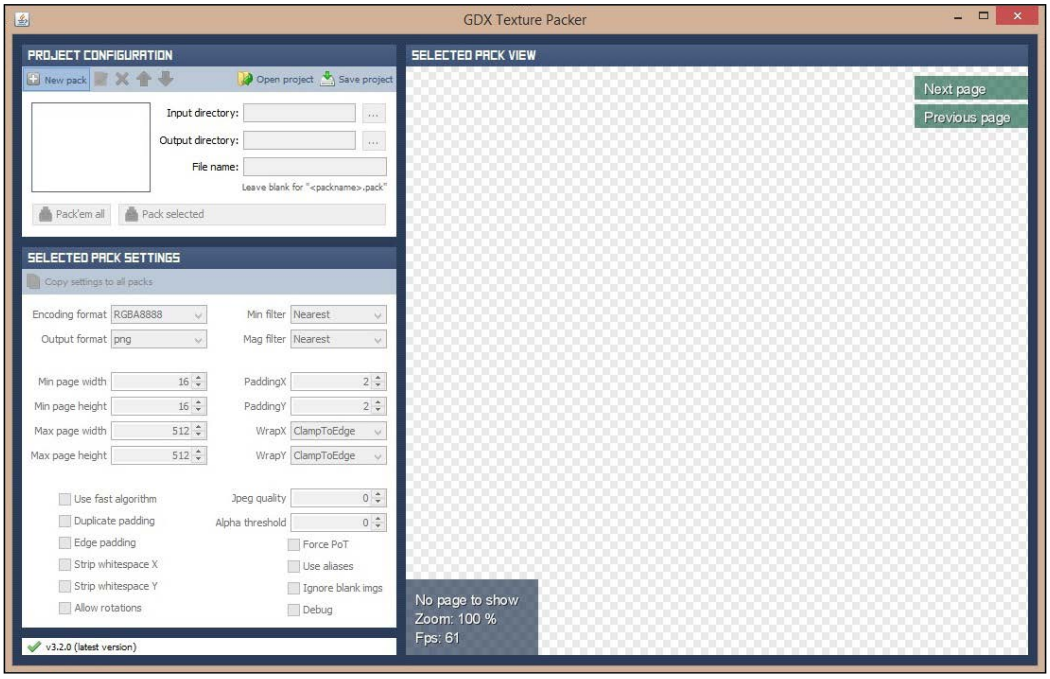
Texture packer outputs multiple pages of images and a text file with the packing information that has the coordinates and other information that helps retrieve each image. Multiple pages are created only if the images cannot be packed into a single image based on the maximum resolution provided. While packing images, a few things need to be taken care of:

- Place all images of a scene in the same folder
- Place all image frames of an animation in the same folder
- Name all images properly before packing
- Trim whitespaces in images before packing

We do not want `Spritebatch` rebinding textures unnecessarily; hence, all images that can appear together need to be in one the `TextureAtlas` pack. If all images cannot be packed into one `TextureAtlas` pack, then organize them intelligently so that performance is not hit badly. Please note that the `TexturePacker` class within LibGDX can be used to dynamically pack assets at runtime. Take a look at the following screenshot for the interface of the commercial `TexturePacker` class:



Here is the interface for the free texturepacker-gui-v3.2:



We need to select the **New Pack** button and provide a name for the pack. Then, we can set the input and output folders. The input folder should be the folder with all our art assets. Then, click on the **Pack'em All** button to create the packed files. I created `ThrustCopter.pack` using this tool and `thrustcopterassets.txt` using the commercial tool—both of these will work. Each of these files has a `.png` file with the same name.

The latest Mac systems running Java 1.8 64-bit might not be able to run the free `TexturePacker` class due to some Java incompatibility.

Once we create our `TextureAtlas` pack, we do not need the individual images. We can remove all of them and keep only the two `TextureAtlas` files. To load the `TextureAtlas` pack, we need to execute either of the following code snippets depending on which `TextureAtlas` we are using:

```
TextureAtlas atlas = new
TextureAtlas(Gdx.files.internal("ThrustCopter.pack"));
```

Or we can use the following code:

```
TextureAtlas atlas = new
TextureAtlas(Gdx.files.internal("thrustcopterassets.txt"));
```

Once the `atlas` instance is loaded, we can access `TextureRegion` with the following code:

```
terrainBelow=atlas.findRegion("groundGrass");
```

Note that we have omitted the `.png` part in the file name. If you open up the `.pack` file, you can see that the references are named without the `.png` extension.

The revised code

Now that we have a `TextureAtlas` class, we can change our code to use it in place of the earlier code. We need to change the background texture to a `TextureRegion` class as a `TextureAtlas` class will be providing regions. The following changes are made in the `create` method:

```
//atlas = new TextureAtlas(Gdx.files.internal("ThrustCopter.pack"));
atlas = new TextureAtlas(Gdx.files.internal("thrustcopterassets.
txt"));
bgRegion = atlas.findRegion("background");
terrainBelow=atlas.findRegion("groundGrass");
terrainAbove=new TextureRegion(terrainBelow);
```

```
terrainAbove.flip(true, true);

plane = new Animation(0.05f, atlas.findRegion("planeRed1"),
  atlas.findRegion("planeRed2"),
  atlas.findRegion("planeRed3"),
  atlas.findRegion("planeRed2"));
plane.setPlayMode(PlayMode.LOOP);
```

The rest of the code remains the same, as the only change is in the way we loaded our graphics.

Handling multiple screen sizes and aspect ratios

You must be excited to run the game on your Android or iOS device. Go ahead and try it, as we have already discussed how to do that in *Chapter 1, Wiring Up*. At this point, your game will fit the screen of the device you have connected to. Such a solution is not ideal as the aspect ratio of the game might actually be different and stretching to fit the device's screen might distort the art and spoil the experience. In order to handle multiple screens, LibGDX provides something called `Viewports`. We can assign `camera` with `viewport` as per our discretion. We will be designing our game for a virtual viewport, which is 800 x 480 in our case because our art is made for that resolution. The different viewports are as follows:

- `StretchViewport`: This will stretch the virtual viewport to fit the screen. The aspect ratio can be anything depending on the device, and scaling is possible.
- `FitViewport`: This will fit the virtual viewport within the screen with black bars that fill the additional area. The aspect ratio will remain the same as our virtual viewport and there is no distortion.
- `FillViewport`: This is similar to `FitViewport` but there won't be any black bars, as it will always fill the screen with the fixed aspect ratio. This means some part of the game will get cut off or fall outside the screen.
- `ScreenViewport`: This will always match the window size without any scaling or black bars. However, this is not advisable because it gives unnecessary advantage to devices with a larger screen.

More details can be found at `https://github.com/libgdx/libgdx/wiki/Viewports`.

Let's add a `FitViewport` class to our game. To do so, we need to declare a `Viewport` variable first, as shown in the following code:

```
Viewport viewport;
```

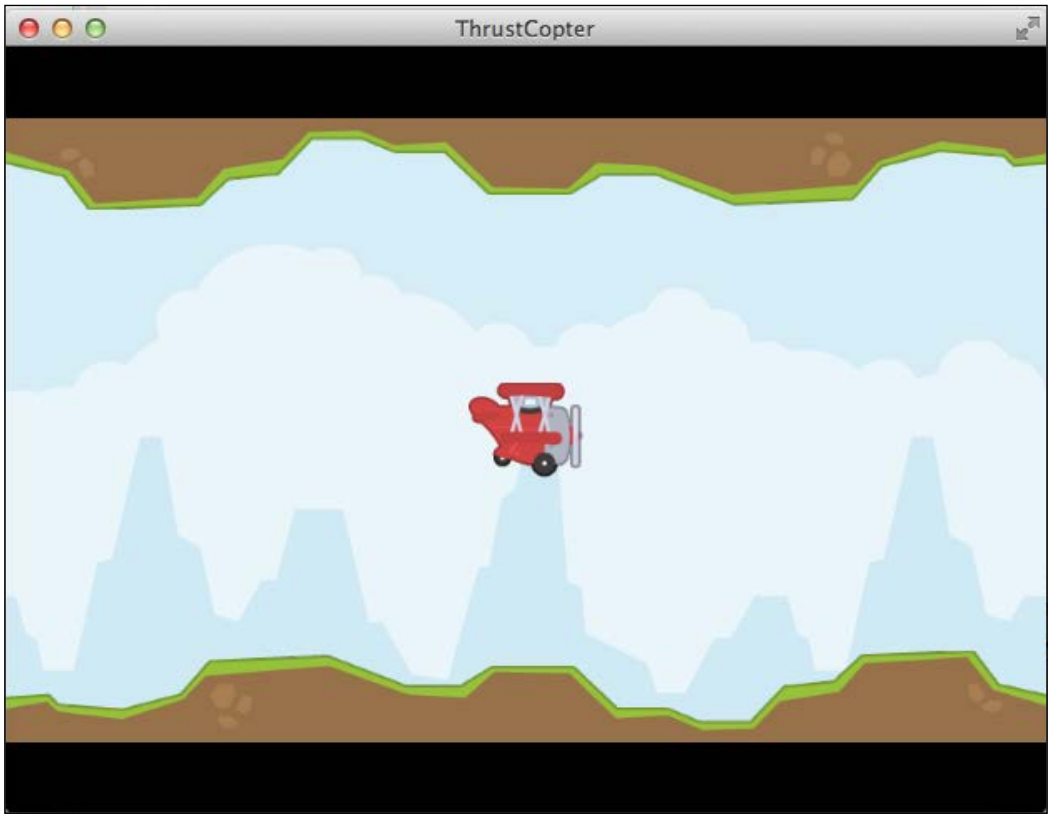
Then edit the following code:

```
camera = new OrthographicCamera();
//camera.setToOrtho(false, 800, 480);
camera.position.set(400,240,0);
viewport = new FitViewport(800, 480, camera);
```

Whenever the window is resized, we need to update the viewport so that it can recalculate. So, add one more method, as shown in the following code:

```
@Override
public void resize (int width, int height)
{
  viewport.update(width, height);
}
```

In order to see this code in action, you can run the desktop application and drag the corner of the window to resize it, as shown in the following screenshot:



It is time to review the source code provided. In the upcoming chapters, the graphics addition code might not be explained, as everything is similar to how we set it up in this chapter.

Summary

We have come to the end of an interesting chapter. We created the Thrust Copter LibGDX project and added graphics to the game scene. We discussed `Texture`, `TextureRegion`, and `Sprite`. You also learned that sprite batching increases performance and speeds up rendering. We used the `FPSLogger` instance to log our game's FPS. Finally, we created our plane animation and moved our plane across the screen.

Once we learned about texture packing, we created `TextureAtlas` to replace all our art. We now know how to use a `Viewport` class to support multiple screens without distorting our game's graphics. Finally, we successfully executed our game on our Android and iOS devices as per the instructions from *Chapter 1*, *Wiring Up*.

In the next chapter, we will try to take the touch input from the user to control our plane and we will also add some sound effects.

3
Thou Shall Not Pass!

Currently, we have our animated plane moving across a seamlessly scrolling sky backdrop. There is no fun in seeing our plane crash-land each time we run the game. In this chapter, we will add player input to control the plane so the player could try to keep the plane from crashing. To make things more interesting, we will add some sound effects and a music loop. The name of the chapter also points to the fact that we will add many obstacles that the plane will need to dodge in order to avoid a crash. We will explore the following in this chapter:

- Polling various user input
- Adding mouse and touch-based user control
- Learning about input processors
- Adding obstacles for our plane to dodge
- Adding game logic to control our plane and other aspects of the gameplay
- Learning about game controllers
- Implementing sound effects and looping background music

The source files for this chapter can be found in `chapter3.zip` provided along with the book. You can open it up as a reference while you proceed through this chapter.

Piloting our plane

Our idea is to let the user control the plane. There are different input methods that can be used to control the plane, such as the following:

- Mouse input
- Keyboard input
- Touch or tap input

- Gestures
- Tilt or accelerometer input
- Game controller input

The first two are exclusive to desktop computers, whereas the next two are exclusive to handheld devices such as mobiles and tablets. In some new generation laptops, we can find the touch input enabled as well. The last input method is exclusive to micro consoles that are Android-based gaming consoles with a dedicated game controller unit, for example, **OUYA**. I will try to familiarize you with all of these input methods as you will definitely need most of these for your future game development ventures. Most of these inputs can be detected in two ways: polling and event handling. Polling is the method in which we constantly check the status of a particular input. This is easier but not very fast or accurate when it comes to tracking the sequence of input. In such cases, we can use event handling, which provides very precise details about the input. We will need a listener that will keep on looking for any input and report it to us to take the necessary action.

Note that you need to update LibGDX often because it gets updated frequently; the people behind it are very active. While writing this chapter, LibGDX got updated to 1.3.0. Updating existing projects is very easy as all you will need to do is edit two files. Find and open the `build.gradle` file in the root of your project.

Edit the following line to reflect the latest LibGDX version:

```
gdxVersion = "1.3.0"
```

Now select all the associated projects in Eclipse and right-click on them and navigate to **Gradle** | **Refresh All**. This will make Gradle get all the associated files and wire them up. Also, the `pom.xml` file will have the version tag for `com.badlogicgames.gdx`, which can be edited to use the most recent LibGDX. Refer to the following code:

```
<version>1.3.1-SNAPSHOT</version>
```

For the Thrust Copter project, we will use touch input through which we will move our plane away from the point of touch; that is, if we touch on the left-hand side of the plane, it will move to the right.

Navigating using touch input

First, let's try to implement our touch control based on the more simpler input-polling technique. First we need to check whether there has been a tap, which can be detected using the `Gdx.input.justTouched()` method. We will create a new variable called `touchPosition`, which is a `Vector3` instance to store the point of touch. The following code shows how to get the *x* and *y* coordinates of the touch point:

```
touchPosition.set(Gdx.input.getX(),Gdx.input.getY(),0);
```

This sets the *x* and *y* coordinates of `touchPosition` with the screen coordinates of the point where the touch has occurred. These coordinates will be based on the screen resolution, and as our camera has a different but fixed resolution set, we will need to convert the coordinates to the camera's coordinates. This is done by the following:

```
camera.unproject(touchPosition);
```

The `unproject` method only accepts `Vector3` values for which we had created `touchPosition` as the `Vector3` instance. While assigning values, we had set the *z* value as 0. Now we have the coordinates based on our camera's resolution which we need to compare with the plane's current position to determine the angle and force to be applied. Let me first show you the code that is added to the top of the `updateScene` method, which is as follows:

```
if(Gdx.input.justTouched())
{
  touchPosition.set(Gdx.input.getX(),Gdx.input.getY(),0);
  camera.unproject(touchPosition);
  tmpVector.set(planePosition.x,planePosition.y);
  tmpVector.sub( touchPosition.x, touchPosition.y).nor();
  planeVelocity.mulAdd(tmpVector,
    TOUCH_IMPULSE-MathUtils.clamp(Vector2.dst(touchPosition.x,
    touchPosition.y, planePosition.x, planePosition.y), 0,
    TOUCH_IMPULSE));
  tapDrawTime=TAP_DRAW_TIME_MAX;
}
tapDrawTime-=deltaTime;
```

We are using a few new variables, which are as follows:

```
Vector3 touchPosition=new Vector3();
Vector2 tmpVector=new Vector2();
private static final int TOUCH_IMPULSE=500;
TextureRegion tapIndicator;
float tapDrawTime;
private static final float TAP_DRAW_TIME_MAX=1.0f;
```

The `tapDrawTime` variable is used to track whether we need to draw an indicator that would show where the last tap occurred. We set this variable to `TAP_DRAW_TIME_MAX`, as shown in the second-to-last line of the `updateScene` method in the preceding code. We will be deducting `deltaTime` from this in each frame and will only draw it on screen when `tapDrawTime` is greater than 0. In `drawScene`, this is done using the following code:

```
if(tapDrawTime>0)
{
  batch.draw(tapIndicator, touchPosition.x-29.5f,
    touchPosition.y-29.5f);
    //29.5 is half width/height of the image
}
```

In the preceding code, the value of the `tapIndicator` variable is as follows:

```
tapIndicator = atlas.findRegion("tap2");
```

A `Vector2` instance value pointing from Vector2 A to Vector2 B can be found by subtracting A from B. As we need the `Vector2` instance to point from `touchPosition` to `planePosition`, we will subtract `touchPosition` from `planePosition`. We just need the direction vector, which has to be a unit vector; hence, we will normalize it using the `nor()` method. The lines that will do this are given in the following code:

```
tmpVector.set(planePosition.x,planePosition.y);
tmpVector.sub( touchPosition.x, touchPosition.y).nor();
```

The next line is a complicated mix of a lot of operations. We are first finding out the distance between the plane and our touch point. Then, we clamp the resulting value so that it always stays within our set minimum and maximum values:

```
MathUtils.clamp(Vector2.dst(touchPosition.x, touchPosition.y,
planePosition.x, planePosition.y), 0, TOUCH_IMPULSE)
```

As per our game logic, the farther you touch, the lesser should be the force applied on the plane. Hence, we need to subtract the resulting value from the maximum value, which is `TOUCH_IMPULSE`. Once the value of the force is obtained, we will multiply it with the unit vector we found previously and add it to the `planeVelocity` variable, thereby affecting the plane's motion. Run the game and tap/click anywhere to control the movement of the plane. The game output can be seen in the following screenshot:

Tap below the plane to make it move up, above to make it move down, and similarly on the sides as well. You can even make your plane move backward at this point by tapping in the front. The interesting thing is that LibGDX has mapped our mouse input to touch input. LibGDX considers mouse input as a special case of touch input where there is only one touch pointer. This makes testing very easy as we can quickly test the desktop project for major changes.

Be aware that a mouse provides a precise input, whereas your fingers are not very precise. A finger tip covers a much bigger area than a mouse pointer and sometimes this might be a problem. Also, the majority of the touchscreens can take multitouch input, which means there can be more than one touch happening at the same time.

Dealing with other input methods

Now that you have something to play with, let's explore the other input options and how LibGDX deals with them. The mouse input is essentially the same as the touch input when it comes to tracking position. However, in order to get the state of the mouse buttons, we will need to use other methods as given in the following code:

```
boolean leftPressed =
  Gdx.input.isButtonPressed(Input.Buttons.LEFT);
boolean rightPressed =
  Gdx.input.isButtonPressed(Input.Buttons.RIGHT);
boolean middlePressed =
  Gdx.input.isButtonPressed(Input.Buttons.MIDDLE);
```

Polling keyboard keys

The `Gdx.input` package also provides a very convenient method to get the states of keyboard keys as well:

```
boolean isSpacePressed = Gdx.input.isKeyPressed(Keys.SPACE);
```

We could also use a keyboard to control our plane, where the arrow keys provide the force instead of the tap for a PC game. We will explore how specific phases of a key event can also be detected using event handling later in this chapter.

Accessing accelerometer data

Most handheld devices such as mobiles and tablets have an accelerometer that can report the motion and orientation of the device. It is always safe to first look for the availability of accelerometer in the device in the context of Android's fragmentation. This can be polled using `Gdx.input.isPeripheralAvailable(Peripheral.Accelerometer)`. Using LibGDX, the orientation of a device can be easily polled using the following code:

```
int orientation = Gdx.input.getRotation();
```

This will return the difference or orientation from the native orientation of the device, which is usually portrait for mobiles and landscape for tablets. Getting accelerometer values along the three axes is also straightforward in LibGDX:

```
float accelX = Gdx.input.getAccelerometerX();
float accelY = Gdx.input.getAccelerometerY();
float accelZ = Gdx.input.getAccelerometerZ();
```

We can use `accelX` and `accelY` values to set the position of our plane. This is something you can try by yourselves as an exercise. In such a case, we can to disable other forces, such as gravity, and use the accelerometer alone to set the plane's position directly.

Event handling for inputs

An alternative to polling input is to use event handling to pick up precise and detailed data related to input. We need to implement an interface called `InputProcessor` that has methods to deal with touch, mouse, and keys. There are methods named `keyDown()`, `keyUp()`, and `keyTyped()` that can be used to execute specific code when this event occurs. These methods will get the key code of the key that has spawned the event. Similarly, `touchDown()`, `touchUp()`, and `touchDragged()` methods can be used to track touch events. These methods are valid for mouse input as well with the addition of `mouseMoved()` and `scrolled()`.

If we implement the `InputProcessor` interface with a new class named `MyInputTracker`, then we can create its instance and set it as our `InputProcessor` interface using the following code:

```
MyInputTracker inputProcessor = new MyInputTracker();
Gdx.input.setInputProcessor(inputProcessor);
```

Alternatively, you can use the game scene class itself to implement the `InputProcessor` interface, in which case we can simply set it using the following code:

```
Gdx.input.setInputProcessor(this);
```

Using the InputAdapter class

Alternatively, we can use the `InputAdapter` class that already implements the `InputProcessor` interface. Our class needs to just extend the `InputAdapter` class and implement the required methods only. The difference here is that we do not need to implement all the unwanted methods as in the earlier method. Alternatively, the following code also works:

```
InputAdapter inputAdapter= new InputAdapter ()
{
  public boolean touchDown (int x, int y, int pointer, int button)
  {
    // add your touch down code
    return true; // return true to indicate the event was handled
```

```
    }

    public boolean touchUp (int x, int y, int pointer, int button)
    {
      // add your touch up code
      return true; // return true to indicate the event was handled
    }
  };
  Gdx.input.setInputProcessor(inputAdapter);
```

I will encourage you to change the code to use `InputAdapter` to get the same result. Make sure you extend `InputAdapter` as we need to make use of various local variables from our game scene while processing the touch input.

Capturing gestures

Gestures are an intuitive way of providing input that is possible using touchscreens. Some famous examples of a gesture are the swipe and pinch zoom. LibGDX provides a class named `GestureDetector` that implements `InputProcessor` to help detect gestures. We need to implement an interface named `GestureListener` and use it as a parameter to create a `GestureDetector` object:

```
Gdx.input.setInputProcessor(new GestureDetector(new
CustomGestureListener()));
```

The `GestureDetector` class helps us to detect the following gestures:

- `touchDown`
- `longPress`
- `tap`
- `pan`
- `panStop`
- `fling`
- `zoom`
- `pinch`

Fling can be used to detect swipe gestures.

Game controllers controller

Android-based micro consoles have come out in numbers, but picking a winner from among them is still impossible. Some interesting consoles are OUYA, GameStick, GamePop, MOJO, and nVidia's Shield. These devices have external game controllers that are used as input devices. These controllers might have different features such as buttons, axes, hat switches, sliders, accelerometers, touchpads, and so on.

LibGDX has a controllers extension class that can be used to detect input from these game controllers. As there can be multiple controllers attached to a console, we will need to poll them and assign an instance using `Controllers.getControllers()`. Once we have a controller instance, we can poll the different input as shown in the following code:

```
boolean buttonPressed = controller.getButton(buttonCode);
float axisValue = controller.getAxis(axisCode);
```

Alternatively, for event-based tracking, we can implement the `ControllerListener` interface or extend the `ControllerAdapter` class, which is similar to how we implement event-based input handling.

OUYA is the most talked about micro console out there and LibGDX already supports it. We can detect an OUYA device using the following code:

```
if(Ouya.runningOnOuya)
{
  // Your code!
}
```

More details on how to develop games for OUYA can be found at `https://devs.ouya.tv/developers/docs/libGDX`.

Adding the different game states

As our plane can now be controlled via user input, we can add the obstacles that would make the game fun and challenging. There are three obstacles we need to avoid that include the terrain's top and bottom, triangular land portions, and meteor rocks of different sizes. If the plane hits any of these, the game is over. In order to implement the game over state, we can follow the game development standards and try to implement game states.

A game is essentially defined as a loop where the program switches between different game states based on the game logic and user input. At any point of time, the game can be in one of these predefined game states. For our game, we need to use only three game states, which are `INIT`, `ACTION`, and `GAME_OVER`. The `INIT` state is when the game starts and we initialize all the variables as per the initial conditions. The `ACTION` state is when the game is running and the `GAME_OVER` state is when the plane has crashed.

For our purposes, we can use enums for the implementation of our game state. Add the following to the game class:

```
static enum GameState
{
  INIT, ACTION, GAME_OVER
}
```

Then, initialize the game state in `INIT`, as follows:

```
GameState gameState = GameState.INIT;
```

While in the `INIT` state, we need to skip the logic in the `updateScene` method but render an indicator that lets the player know that he or she needs to tap to start the game play. The changes are as follows:

```
private void updateScene()
{
  if(Gdx.input.justTouched())
  {
    if(gameState == GameState.INIT)
    {
      gameState = GameState.ACTION;
      return;
    }
    if(gameState == GameState.GAME_OVER)
    {
      gameState = GameState.INIT;
      resetScene();
      return;
    }
    touchPosition.set(Gdx.input.getX(),Gdx.input.getY(),0);
    camera.unproject(touchPosition);
    tmpVector.set(planePosition.x,planePosition.y);
    tmpVector.sub( touchPosition.x, touchPosition.y).nor();
```

```
        planeVelocity.mulAdd(tmpVector,
          TOUCH_IMPULSE-MathUtils.clamp(Vector2.dst(touchPosition.x,
          touchPosition.y, planePosition.x, planePosition.y), 0,
          TOUCH_IMPULSE));
        tapDrawTime=TAP_DRAW_TIME_MAX;
      }
      if(gameState == GameState.INIT || gameState ==
        GameState.GAME_OVER)
      {
        return;
      }
      float deltaTime = Gdx.graphics.getDeltaTime();
    }
```

We skip the execution of game logic when the game state is either the `INIT` or `GAME_OVER` state. In the `drawScene` method, we need to add the code to draw the `tap1` graphic, which is as follows:

```
if(gameState == GameState.INIT)
{
  batch.draw(tap1, planePosition.x, planePosition.y-80);
}
```

Run the game to see the screen stay motionless with the tap indicator. The game will start only when the player taps the screen:

The top and bottom terrain are already placed, but the collision logic is not yet implemented. Let's fix that now. We just need to check whether the plane overlaps these terrains significantly; to do this, we can simply check whether the vertical position of the plane is below the ground level or above the top terrain level. Towards the end of `updateScene`, we need to add the following code:

```
if(planePosition.y < terrainBelow.getRegionHeight() - 35 ||
  planePosition.y + 73 > 480 - terrainBelow.getRegionHeight() +
35)
{
  if(gameState != GameState.GAME_OVER)
  {
    gameState = GameState.GAME_OVER;
  }
}
```

All we are doing is setting up the game state to `GAME_OVER` when the overlap occurs. We will draw a `gameOver` graphic to indicate the game over state in the `drawScene` method:

```
if(gameState == GameState.GAME_OVER)
{
  batch.draw(gameOver, 400-206, 240-80);
}
```

Run the game to check whether you are getting the game over display:

Adding the pillar rocks

Now it's time to add the main obstacle, which is the vertical pillar-like triangular rocks. Dodging them is going to be very hard. One thing we need to avoid first is our plane moving backward. So let us add a constant velocity so that the scene always scrolls from right to left, thereby making the plane always move forward. To do this, we need to create a new variable called `scrollVelocity` and set its value in the `resetScene` method, as follows:

```
scrollVelocity.set(4, 0);
```

Then, in the `updateScene` method, we add it to `planeVelocity`, just below the line where we would add gravity:

```
planeVelocity.add(scrollVelocity);
```

You can run the game to see the scene scrolling which in turn makes the game more lively. To add the pillar rocks, we need to create and store positions to draw the rocks. These position values are `Vector2` variables where the *x* value will denote their *x* position and *y* value will denote whether the pillar is pointing up or down. We can set the *y* value to 1 to indicate an upward pillar whereas a value of -1 to indicate a downward pillar. In order to store these values, we need to create a typed array `pillars` using the following code:

```
Array<Vector2> pillars = new Array<Vector2>();
```

We add a new function named `addPillar` to add a new pillar to the array, as follows:

```
private void addPillar()
{
  Vector2 pillarPosition=new Vector2();
  if(pillars.size==0)
  {
    pillarPosition.x=(float) (800 + Math.random()*600);
  }
  else
  {
    pillarPosition.x=lastPillarPosition.x+(float) (600 +
    Math.random()*600);
  }
  if(MathUtils.randomBoolean())
  {
    pillarPosition.y=1;
  }
  else
```

```
    {
      pillarPosition.y=-1;//upside down
    }
    lastPillarPosition=pillarPosition;
    pillars.add(pillarPosition);
  }
```

Instead of `Math.random`, we can also use `MathUtils.Random`.

We store a reference to the last pillar's position to the `lastPillarPosition` variable. This enables us to check whether this particular pillar has reached the middle of the screen to add another pillar. In the `updateScene` method, we also move all the pillar positions in the same way we move the terrains using the following code:

```
for(Vector2 vec: pillars)
{
  vec.x-=deltaPosition;
  if(vec.x+pillarUp.getRegionWidth()<-10)
  {
    pillars.removeValue(vec, false);
  }
}
if(lastPillarPosition.x<400)
{
  addPillar();
}
```

In the preceding code, the value of `deltaPosition` is as follows:

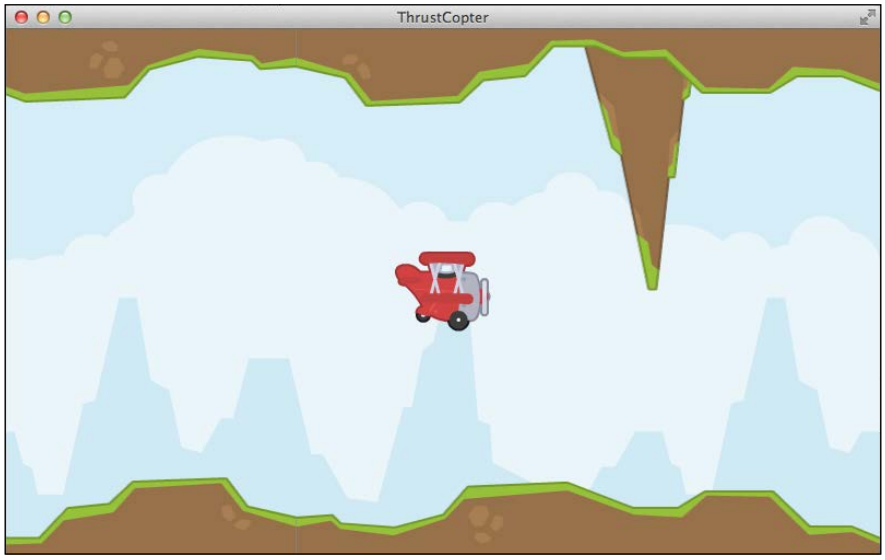
```
deltaPosition=planePosition.x-planeDefaultPosition.x;
```

As you can see, we remove the pillars that go out of our screen as well. In order to draw these pillars, we add the following code to the `drawScene` method, just below the line where we enable blending:

```
for(Vector2 vec: pillars)
{
  if(vec.y==1)
  {
    batch.draw(pillarUp, vec.x, 0);
  }
  else
  {
```

```
        batch.draw(pillarDown, vec.x,
          480-pillarDown.getRegionHeight());
      }
    }
```

Here, `pillarUp` and `pillarDown` are the respective texture regions. Run the code to see the moving pillar obstacles:



Collision with the pillars

It's time to add collision with the pillars. We need to specifically check the overlap between the plane and pillars to determine the collision. We will use two rectangles to store the bounding box of the plane and the obstacle pillar, as follows:

```
Rectangle planeRect=new Rectangle();
Rectangle obstacleRect=new Rectangle();
```

Now in `updateScene`, we set the values for these rectangles and look for any overlap, as detailed in the following code:

```
planeRect.set(planePosition.x + 16, planePosition.y, 50, 73);
for(Vector2 vec: pillars)
{
  vec.x-=deltaPosition;
  if(vec.x+pillarUp.getRegionWidth()<-10)
  {
```

```
        pillars.removeValue(vec, false);
      }
      if(vec.y==1)
      {
        obstacleRect.set(vec.x + 10, 0, pillarUp.getRegionWidth()-20,
          pillarUp.getRegionHeight()-10);
      }
      else
      {
        obstacleRect.set(vec.x + 10,
          480-pillarDown.getRegionHeight()+10,
          pillarUp.getRegionWidth()-20, pillarUp.getRegionHeight());
      }
      if(planeRect.overlaps(obstacleRect))
      {
        if(gameState != GameState.GAME_OVER)
        {
          gameState = GameState.GAME_OVER;
        }
      }
    }
```

We are using the convenient `overlaps` method of the `Rectangle` class to detect collision. If you run the game now, you will see how hard it is to dodge each pillar. It will take timely and skilled tapping to pass each pillar. The game can be seen in the following screenshot:

Adding meteor rocks

Our final obstacle is meteor rocks. The difference between these and the pillar rocks is that these have their own velocity and move across the screen, hence the name. We will use the same approach to implement meteor rocks, but we do not need an array as there will only be one meteor rock on screen at a time. We will add a few new variables for this purpose:

```
Array<TextureAtlas.AtlasRegion> meteorTextures = new
Array<TextureAtlas.AtlasRegion>();
TextureRegion selectedMeteorTexture;
boolean meteorInScene;
private static final int METEOR_SPEED=60;
Vector2 meteorPosition= new Vector2();
Vector2 meteorVelocity= new Vector2();
float nextMeteorIn;
```

We need to store the six different meteor textures in an array and use one of those randomly for the current meteor. We use a countdown to release meteors using the `nextMeteorIn` instance. In the `create` method, we will add all the meteor textures:

```
meteorTextures.add(atlas.findRegion("meteorBrown_med1"));
meteorTextures.add(atlas.findRegion("meteorBrown_med2"));
meteorTextures.add(atlas.findRegion("meteorBrown_small1"));
meteorTextures.add(atlas.findRegion("meteorBrown_small2"));
meteorTextures.add(atlas.findRegion("meteorBrown_tiny1"));
meteorTextures.add(atlas.findRegion("meteorBrown_tiny2"));
```

In `resetScene`, we need to set the initial values for these variables, as follows:

```
meteorInScene=false;
nextMeteorIn=(float)Math.random()*5;
```

In `updateScene`, we add the following code to set the position of the meteor if it is present in the scene and reduce the countdown to launch the next meteor when the countdown hits 0:

```
if(meteorInScene)
{
  meteorPosition.mulAdd(meteorVelocity, deltaTime);
  meteorPosition.x-=deltaPosition;
  if(meteorPosition.x<-10)
  {
    meteorInScene=false;
  }
```

```
}
nextMeteorIn-=deltaTime;
if(nextMeteorIn<=0)
{
  launchMeteor();
}
```

The meteor is launched using a new function named `launchMeteor`, as follows:

```
private void launchMeteor()
{
  nextMeteorIn=1.5f+(float)Math.random()*5;
  if(meteorInScene)
  {
    return;
  }
  meteorInScene=true;
  int id= (int)(Math.random()*meteorTextures.size);
  selectedMeteorTexture=meteorTextures.get(id);
  meteorPosition.x=810;
  meteorPosition.y=(float) (80+Math.random()*320);
  Vector2 destination=new Vector2();
  destination.x=-10;
  destination.y=(float) (80+Math.random()*320);
  destination.sub(meteorPosition).nor();
  meteorVelocity.mulAdd(destination, METEOR_SPEED);
}
```

We find a random texture from `meteorTextures` and set it to the `selectedMeteorTexture` variable. We set the initial position of the meteor just out of the right edge of screen. We find a unit vector toward the left extreme and multiply it with the `METEOR_SPEED` value to set its velocity. In the `drawScene` method, we do the following to draw the meteor:

```
if(meteorInScene)
{
  batch.draw(selectedMeteorTexture,meteorPosition.x,
    meteorPosition.y);
}
```

Collision with the meteor rock

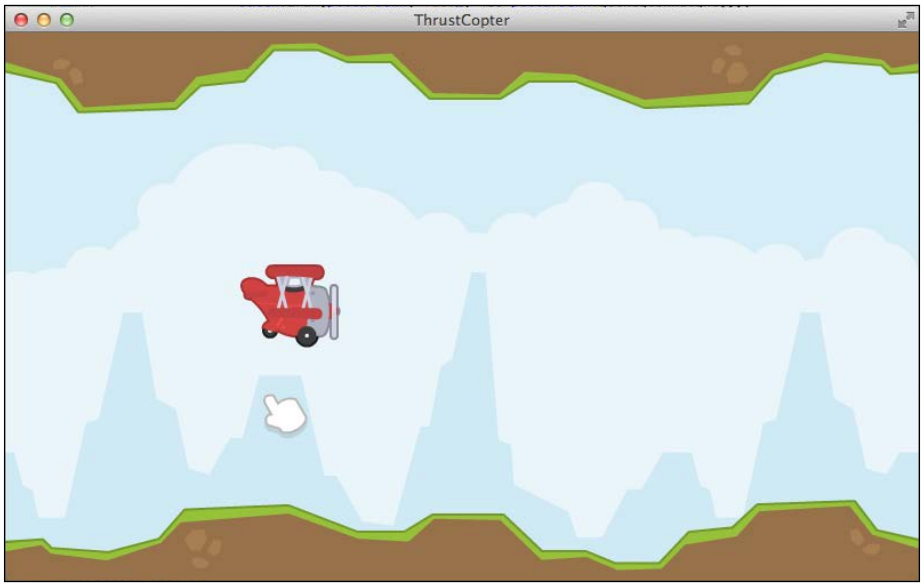
We have our meteor rocks rushing across the screen, but they are not yet colliding with our plane. Let's fix that now. This is the same as what we did for collision with the pillars. The following code goes in the `updateScene` method:

```
if(meteorInScene)
{
  obstacleRect.set(meteorPosition.x + 2, meteorPosition.y + 2,
    selectedMeteorTexture.getRegionWidth()-4,
    selectedMeteorTexture.getRegionHeight()-4);
  if(planeRect.overlaps(obstacleRect))
  {
    if(gameState != GameState.GAME_OVER)
    {
      gameState = GameState.GAME_OVER;
    }
  }
}
```

Now you should see the meteor crashing our plane upon collision, as shown in the following screenshot:

Making the game easier

Yes, our game is very hard to play now and that is something we should fix! But, we had it planned along the lines of the *Flappy Bird* game, which is again very hard and frustrating to play. How can we make the game easier? We can play with the values of the variables, thereby making the game play easier. I will leave it to you to figure it out. We can give a warning to the player that a meteor is coming, which lets him or her anticipate if an obstacle is approaching. This can be in the form of an audio SFX, which we will add in the next section. Another easy thing to do is to move our plane to the left thereby giving more time to take evasive measures and letting us see the obstacles much earlier. This can be seen in the following screenshot:



This is done by changing the `planeDefaultPosition.x` value.

Playing with audio

It's time to add life to the game with some background music and sound effects. For our game, we will use royalty free music from `http://opengameart.org/content/journey`. The `Music` instance is streamed from the disk rather than loading it onto the RAM. Dealing with long music files is a heavy process, and it is highly recommended that you do not use more than one `Music` object at a time. First we need to place the music file in the assets folder of our Android project. Once done, we can load the music and play it in the following manner:

```
Music music;
```

Then, in the `create` method, we make the following changes:

```
music =
Gdx.audio.newMusic(Gdx.files.internal("sounds/journey.mp3"));
music.setLooping(true);
music.play();
```

This will set the music in loop and start playing it. The `Music` class has methods to poll its status and pause or stop it:

```
music.stop();
music.pause();

music.isPlaying();
music.isLooping();
```

Adding sound effects

Let's add a few sound effects to our game. We need sound for tapping and crashing and an indicator sound when a new meteor is launched. LibGDX supports `MP3`, `OGG`, and `WAV` files for audio where `OGG` is not supported on iOS. Adding and playing sounds is straightforward as music, but the difference here is that it gets loaded into the RAM rather than streamed from the disk. We can create a `Sound` object as follows:

```
Sound tapSound;
```

Then, in the `create` method, we can load the sound as follows:

```
tapSound =
Gdx.audio.newSound(Gdx.files.internal("sounds/pop.ogg"));
```

Later, it can be played whenever needed using following code:

```
tapSound.play();
```

We will create three `Sound` objects named `tapSound`, `crashSound`, `spawnSound` as required. We will play `tapSound` when we detect a tap within the `updateScene` method. The `spawnSound instance` will be played in the `launchMeteor` function and `crashSound` will be played whenever we change the game state to `GAME_OVER`. Make sure to dispose off the `Music` and `Sound` objects when removing the scene:

```
@Override
public void dispose ()
{
  tapSound.dispose();
  crashSound.dispose();
```

```
        spawnSound.dispose();
        music.dispose();
        batch.dispose();
        pillars.clear();
        atlas.dispose();
        meteorTextures.clear();
    }
```

Summary

We have come to the end of another interesting chapter. The core game play is completely implemented except for the pickups collection logic. We learned to deal with the various kinds of input and implemented tap-based navigation for the plane. We added various obstacles to the game, which were the pillars and meteor rocks. The process of adding collisions with the terrain and obstacles were also implemented. Different game states were added to facilitate effective game play. Also, sounds and music bought the game to a much more polished level.

In the next chapter, we will play with some particle effects and add the game UI. We will also collect the different pickups and implement their logic as well.

4
Bring in the Extras!

The core game play mechanics of Thrust Copter are now complete. We can move on to implement the necessary visual additions and some game logic to add depth to our game play. We will add the GUI and implement pickup logic. We will also discuss few new tools that will help us in creating bitmap fonts and particle effects. We will explore the following topics in this chapter:

- Refactoring our code and project structure to use the `Game` and `ScreenAdapter` classes
- Learning to use the `AssetManager` class
- Implementing pickups such as stars and shield
- Adding particle effects
- Implementing scoring and adding it as part of GUI
- Implementing a fuel meter based on fuel pickup
- Using tools to create bitmap fonts and font generation
- Using the LibGDX particle designer tool

The source files for this chapter can be found in `chapter4.zip` provided along with the book. You can keep it open as reference while you proceed through the chapter.

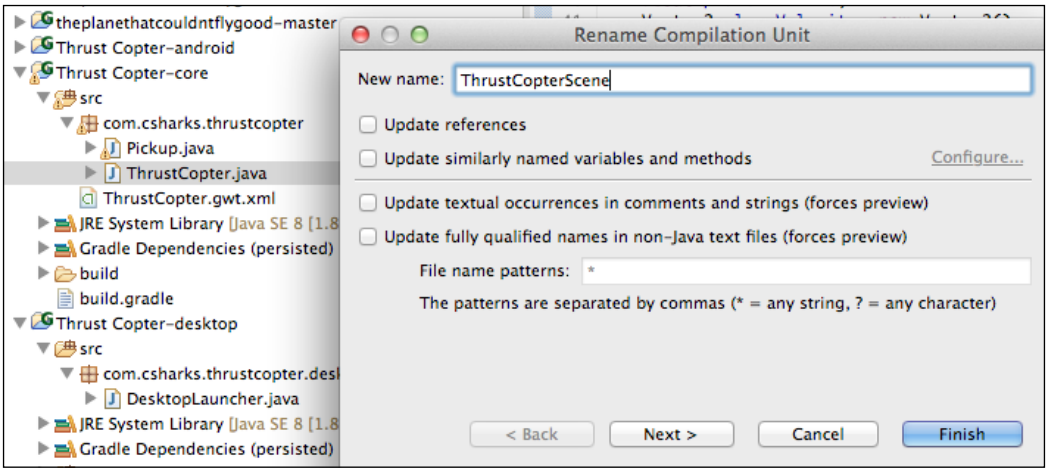
Refactoring time

Time to get our hands dirty and refactor our code to implement the standard project structure followed in LibGDX projects. When working on a complicated project we will have multiple scenes which are essentially different classes for menu, game, game over, level up, and so on. There are many common functionalities that need to be available for all those scenes. Also, many variables can be reused across all of these scenes as well, for example, the `TextureAtlas` instance, the `OrthographicCamera` instance, values for screen size, and so on.

We didn't do it this way earlier as we needed to begin in the easiest possible way. Now that we are all accustomed with LibGDX development, we should start following the standards. The main class where the execution begins, which in our case is `com.csharks.thrustcopter.ThrustCopter`, should never be any scene. This class needs to extend the LibGDX `Game` class that has the inherent functionality to set screens. Every scene, such as menu, game, and game over needs to extend the `ScreenAdapter` class that implements the `Screen` interface.

Creating a ThrustCopterScene class

Let's start refactoring by first renaming our main class to `ThrustCopterScene`. Right-click on the `ThrustCopter.java` file in Eclipse inside the `Thrust Copter-core` project. Click on **Refactor** and select **Rename**. Enter the new name `ThrustCopterScene`, deselect the **Update references** checkbox, and click on **Finish**. We do not want the references to the old name to change, as we will be creating another class of that name to act as our landing class, as shown in the following screenshot:



Edit the `ThrustCopterScene` class to extend `ScreenAdapter` instead of `ApplicationListener`, as shown in the following code:

```
public class ThrustCopterScene extends ScreenAdapter
```

The `ScreenAdapter` class does not have a `create` method, so we will change it to our constructor after removing `@Override`:

```
public ThrustCopterScene()
{
…
}
```

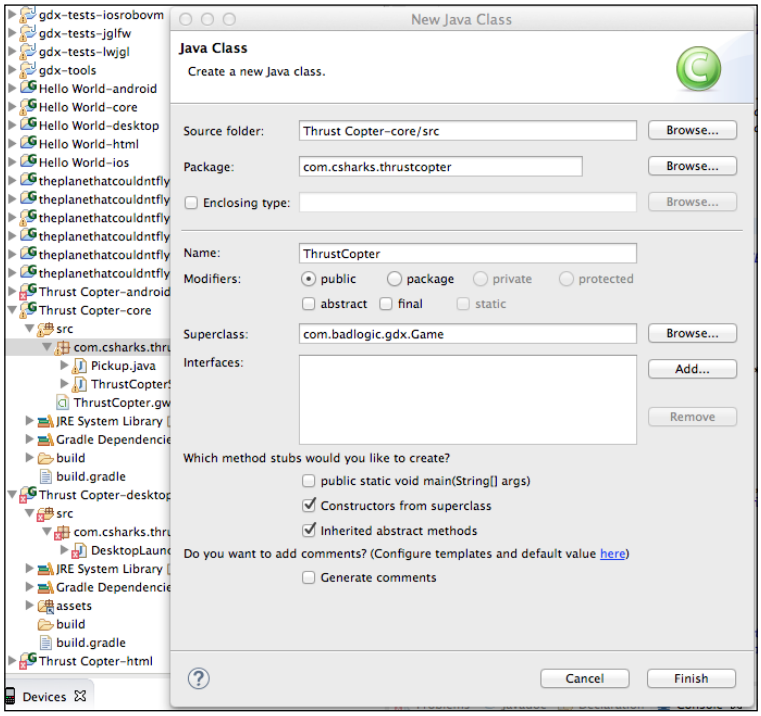
One more minor edit and we are set with this class. The `render` method receives a `float` value as `delta` that needs to be added to the code:

```
public void render (float delta)
{
…
}
```

Now, we have the `ThrustCopterScene` class that implements the `Screen` interface that can be set as active screen using the `Game` class's `setScene` method.

Creating our Game class instance

Eclipse will show a few red cross marks, which means that the original `ThrustCopter` class that many other classes are referring to is missing. Let's fix that by creating a new class with the name `ThrustCopter`, but this time extending the `Game` class that actually implements `ApplicationListener`. Within the `Thrust Copter-core` project, right-click on the `com.csharks.thrustcopter` package to select the **Class** option under **New**. Name the class `ThrustCopter`, select `com.badlogic.gdx.Game` as the super class, and click on **Finish**. This is shown in the following screenshot. Once this class is created, the red cross marks will go but the game is not wired yet.



This new class just has a constructor (if you opted for it) and an overridden `create` method. Add a new line of code to set the screen to a new `ThrustCopterScene` instance:

```
public void create()
{
  setScreen(new ThrustCopterScene());
}
```

Run the desktop game and voila! We have our game back, but this time in a much more standard and extendable manner.

More refactoring

We can easily move some of the objects from our `Screen` class to our `Game` class object so that those can be reused across all screens. We will move the `TextureAtlas`, `SpriteBatch`, `FPSLogger`, and `OrthographicCamera` instances and a few other values to the `ThrustCopter` class from the `ThrustCopterScene` class. We will pass the reference to the `ThrustCopter` instance to all the scenes via their constructors while using the `setScene` method. Check out the following updated code for `ThrustCopter`:

```
public static final int screenWidth=800;
public static final int screenHeight=480;

public ThrustCopter()
{
  fpsLogger=new FPSLogger();
  camera = new OrthographicCamera();
  camera.position.set(screenWidth/2,screenHeight/2,0);
  viewport = new FitViewport(screenWidth, screenHeight, camera);
}
@Override
public void create()
{
  batch=new SpriteBatch();
  atlas = new
    TextureAtlas(Gdx.files.internal("thrustcopterassets.txt"));
  setScreen(new ThrustCopterScene(this));
}
@Override
public void render()
{
  fpsLogger.log();
```

```
    super.render();
  }
  @Override
  public void resize (int width, int height)
  {
    viewport.update(width, height);
  }
  @Override
  public void dispose ()
  {
    batch.dispose();
    atlas.dispose();
  }
```

We will need to make necessary changes in the `ThrustCopterScene` class by removing the `FPSLogger` and `Viewport` instances completely and referencing instances of classes such as `SpriteBatch`, `Camera`, and so on:

```
ThrustCopter game;

public ThrustCopterScene(ThrustCopter thrustCopter)
{
  game=thrustCopter;
  batch=game.batch;
  camera=game.camera;
  atlas=game.atlas;
}
```

We can remove the `resize` method and clean up the `dispose` method as `batch` and `atlas` are disposed elsewhere. From now on, we will be creating common reusable objects in the `ThrustCopter` class as public variables that can be accessed in the `ScreenAdapter` objects via the passed reference.

Using AssetManager

The `AssetManager` class is the class that can help us to efficiently manage our assets. It can load all kinds of files, such as `Texture`, `BitmapFont`, `TextureAtlas`, `Music`, `Sound`, `ParticleEffect`, and so on. It can also help manage context loss on Android devices. Let's create a public `AssetManager` object in the `ThrustCopter` class that will be available to all `Screen` instances:

```
AssetManager manager = new AssetManager();
```

Loading an asset via `AssetManager` is straightforward, as shown in the following code:

```
manager.load("Texture.png", Texture.class);
manager.load("Font.fnt", BitmapFont.class);
manager.load("Music.ogg", Music.class);
```

Assets are loaded asynchronously; hence, we will need to complete the loading before we move on with our code. To do this, we need to call the `finishLoading` method, as shown in the following code:

```
manager.finishLoading();
```

Once the assets are loaded, we get access our assets at any time, as shown in the following code

```
Texture tex = manager.get("Texture.png", Texture.class);
BitmapFont font = manager.get("Font.fnt", BitmapFont.class);
```

The `AssetManager` class also has an unload method to dispose of specific assets that are of no further use. Let's use it to load our `TextureAtlas`, `Music` and `Sound` files. Let's change the `create` method of `ThrustCopter` to load everything, and I believe you will do the necessary changes in the `ThrustCopterScene` class to get them from `game.manager`:

```
public void create()
{
  manager.load("gameover.png", Texture.class);
  manager.load("sounds/journey.mp3", Music.class);
  manager.load("sounds/pop.ogg", Sound.class);
  manager.load("sounds/crash.ogg", Sound.class);
  manager.load("sounds/alarm.ogg", Sound.class);
  manager.load("thrustcopterassets.txt", TextureAtlas.class);
  manager.finishLoading();

  batch=new SpriteBatch();
  atlas=manager.get("thrustcopterassets.txt", TextureAtlas.class);

  setScreen(new ThrustCopterScene(this));
}
```

Time for pickups

There is no fun in the game without any collectibles. For `ThrustCopter`, we have three different collectibles that are as follows:

- The star collectible that adds to the score.
- The fuel collectible that is needed to refill our plane's fuel.
- The shield collectible that when collected makes our plane invincible for a short time. We will start the game with shield enabled.

Collectibles can be released randomly as game progresses, but fuel and shield need to be time-based. Shield has to be a rare collectible that comes after a longer time. The fuel collectible needs to be released based on the time the plane's fuel takes to completely deplete. We will need to release at least two fuel pickups in that time, thereby giving the player a fair chance of survival—once the fuel is depleted, the game is over. While releasing collectibles, we need to make sure that we should be able to collect them. They should not fall inside the pillars or the terrains.

Using a pickup class

Up until now, we were not really using much of Java's OOP methodology, so let's go ahead and create a new `Pickup` class that will represent all our pickups. The code is as follows:

```
public class Pickup
{

  public static final int STAR =1;
  public static final int SHIELD =2;
  public static final int FUEL =3;
  TextureRegion pickupTexture;
  Vector2 pickupPosition = new Vector2();
  int pickupType;
  int pickupValue;
  Sound pickupSound;

  public Pickup(int type, AssetManager manager)
  {
    TextureAtlas atlas=manager.get("thrustcopterassets.txt",
      TextureAtlas.class);
    pickupType=type;
    switch(pickupType){
```

```
        case STAR:
          pickupTexture=atlas.findRegion("star_pickup");
          pickupValue=5;
          pickupSound = manager.get("sounds/star.ogg", Sound.class);
          break;
        case SHIELD:
          pickupTexture=atlas.findRegion("shield_pickup");
          pickupValue=15;
          pickupSound = manager.get("sounds/shield.ogg", Sound.class);
          break;
        case FUEL:
          pickupTexture=atlas.findRegion("fuel_pickup");
          pickupValue=100;
          pickupSound = manager.get("sounds/fuel.ogg", Sound.class);
          break;
        }
      }
    }
```

The class sets the necessary variables in its constructor based on the parameter type passed in.

Adding pickup logic

As we have three different pickups to be released in a timely manner, we will need countdown variables to track their timing as we did with the meteor rocks. We can use a `Vector3` variable such as `pickupTiming`, which has three variables (*x*, *y*, and *z*) to reduce the clutter. Each value in *x*, *y*, and *z* will track the pickup release timing for star, fuel, and shield, respectively. The code is provided here, but I am not going to go through the movement, drawing, and collision of pickups because it is the same as the others explained earlier. We use a `pickupsInScene` array to keep track of all the pickups added to scene. A new function, `checkAndCreatePickup`, is called from `updateScene` to check whether it is time to add a new pickup. When our plane collides with a pickup, the `pickItUp` function is called. Please go through the source file if there are any doubts. The code is as follows:

```
    private void checkAndCreatePickup(float delta)
    {
      pickupTiming.sub(delta);
      if(pickupTiming.x<=0)
      {
        pickupTiming.x=(float)(0.5+Math.random()*0.5);
        if(addPickup(Pickup.STAR))
```

```
      pickupTiming.x=1+(float)Math.random()*2;
    }
    if(pickupTiming.y<=0)
    {
      pickupTiming.y=(float)(0.5+Math.random()*0.5);
      if(addPickup(Pickup.FUEL))
      pickupTiming.y=3+(float)Math.random()*2;
    }
    if(pickupTiming.z<=0)
    {
      pickupTiming.z=(float)(0.5+Math.random()*0.5);
      if(addPickup(Pickup.SHIELD))
      pickupTiming.z=10+(float)Math.random()*3;
    }
  }
  private boolean addPickup(int pickupType)
  {
    Vector2 randomPosition=new Vector2();
    randomPosition.x=820;
    randomPosition.y=(float) (80+Math.random()*320);
    for(Vector2 vec: pillars)
    {
      if(vec.y==1)
      {
        obstacleRect.set(vec.x , 0, pillarUp.getRegionWidth(),
          pillarUp.getRegionHeight());
      }
      else
      {
        obstacleRect.set(vec.x , 480-pillarDown.getRegionHeight(),
          pillarUp.getRegionWidth(), pillarUp.getRegionHeight());
      }
      if(obstacleRect.contains(randomPosition))
      {
        return false;
      }
    }
    tempPickup=new Pickup(pickupType, game.manager);
    tempPickup.pickupPosition.set(randomPosition);
    pickupsInScene.add(tempPickup);
    return true;
  }
  private void pickIt(Pickup pickup)
```

```
{
  pickup.pickupSound.play();
  switch(pickup.pickupType){
  case Pickup.STAR:
    starCount+=pickup.pickupValue;
    break;
  case Pickup.SHIELD:
    shieldCount=pickup.pickupValue;
    break;
  case Pickup.FUEL:
    fuelCount=pickup.pickupValue;
    break;
  }
  pickupsInScene.removeValue(pickup, false);
}
```

Note that we avoid placing pickups within a pillar. We have also added new variables to track fuel, star, and shield statuses. When `fuelCount` reaches 0, we need to disable the tap input code so that the plane falls down to crash, as it should when a plane's fuel runs out, as shown in the following screenshot:

Yes, as you can see I am not an expert in the game.

Heads Up Display (HUD) UI

We need to show the fuel status, shield status, and stars collected at the top of the game screen as our HUD. Fuel status is to be shown as a progressively filled graphic in the top-left corner, whereas the shield status needs to display a shield over our plane and show a countdown at the top of screen in the center that will show how much time is left. Stars get added to the score, where score is actually the time we survive. This needs to be displayed in top-right corner as text.

Adding a fuel bar is just a matter of drawing the `fuelIndicator` texture in a black tint first and then drawing a portion of it again in the default white tint. The portion to draw depends on the `fuelPercentage` variable, which can be easily calculated in `updateScene` as follows:

```
fuelCount-=6*deltaTime;
fuelPercentage=(int) (114*fuelCount/100);
//114=fuelIndicator.getRegionWidth()
```

Then, add the following code in the `drawScene` method:

```
batch.setColor(Color.BLACK);
batch.draw(fuelIndicator, 10, 350);
batch.setColor(Color.WHITE);
batch.draw(fuelIndicator,10,350,0,0,fuelPercentage,119);
//119=fuelIndicator.getRegionHeight()
```

Now, we have a nice fuel bar that gradually decreases as fuel runs out. To display the shield, we will add a new `Animation` named `shield` and draw it around with the plane when the `shieldCount` value is positive. Now, there won't be any collisions with pillars or meteor rocks. However, the plane will still crash if it hits the top or bottom of the terrain, and the `ShieldCount` value needs to reduce in the `updateScene` method. Now, it's time to learn how to put texts on screen to show our `score` and `shieldCount` values.

Displaying text

There are multiple ways to display text in LibGDX, but the fact is that most of these are unsuitable to show text that changes rapidly. In the game, we will need to show text values that change rapidly, such as score, bonus, multipliers, and so on. The ideal solution is to use a `BitmapFont` class. You have already learned what `BitmapFont` is and what it does in *Chapter 1*, *Wiring Up*. Let's try to create a `BitmapFont` class.

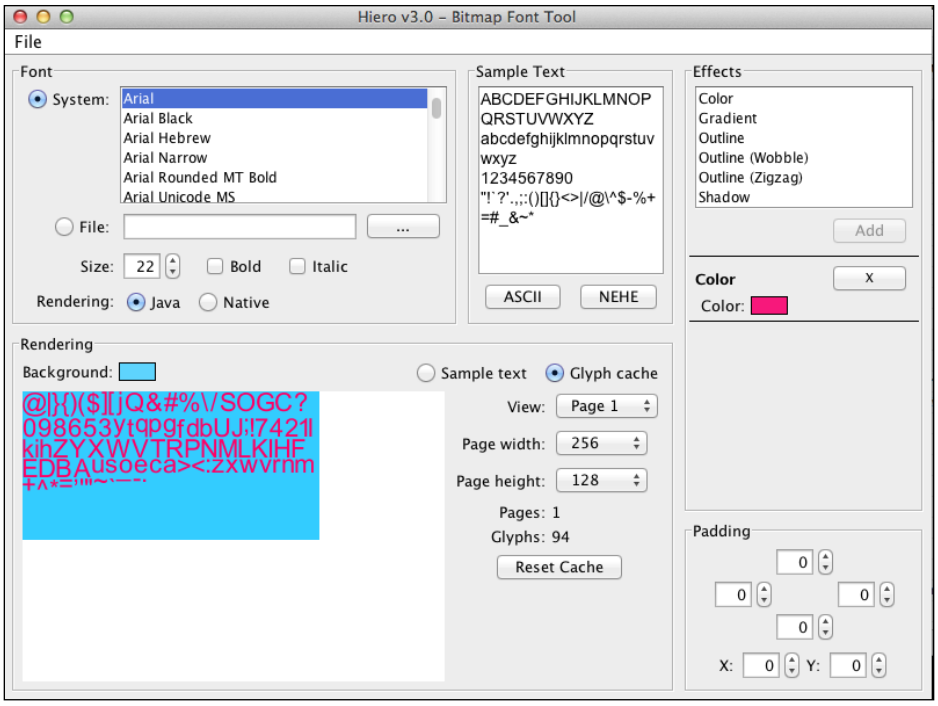
Hiero – the BitmapFont creator tool

Hiero is a free Java application that can be used to create bitmap fonts. You can find this application at `http://wiki.libgdx.googlecode.com/git/jws/hiero.jnlp`. It needs to be run as a web application, which sometimes might cause some Java security concerns. If that happens, get additional help from the official Java website at `http://www.java.com/en/download/help/java_blocked.xml`. Hiero can also be run from within Eclipse if you have the LibGDX source projects, including the `gdx-tools` project.

The following are the other alternatives:

- ShoeBox: `http://renderhjs.net/shoebox/`
- BMFont: `http://www.angelcode.com/products/bmfont/`
- Glyph Designer: `http://www.71squared.com/glyphdesigner`

The UI of Hiero is as shown in the following screenshot:



We need to select the font, font size, font color, and background color and apply any styling available in the right panel. Once you have done so, export the `BitmapFont` file by going to **File | Save BMFont files**. Hiero will create two files that we need to copy and paste in our assets folder in the `Thrust Copter-android` project. We already know how to use the `BitmapFont` class, but this time use the `AssetManager` class to load and get the font files instead of loading directly. The font size I used is `40` and I have provided the `.fnt` file in the source. The relevant code will be as follows:

```
manager.load("impact-40.fnt", BitmapFont.class);

font=manager.get("impact-40.fnt", BitmapFont.class);

font.draw(batch, ""+((int)shieldCount), 390, 450);
font.draw(batch, ""+(int)(starCount+score), 700, 450);
```

Conversion of `shieldCount` and `starCount+score` to string can also be done as follows:

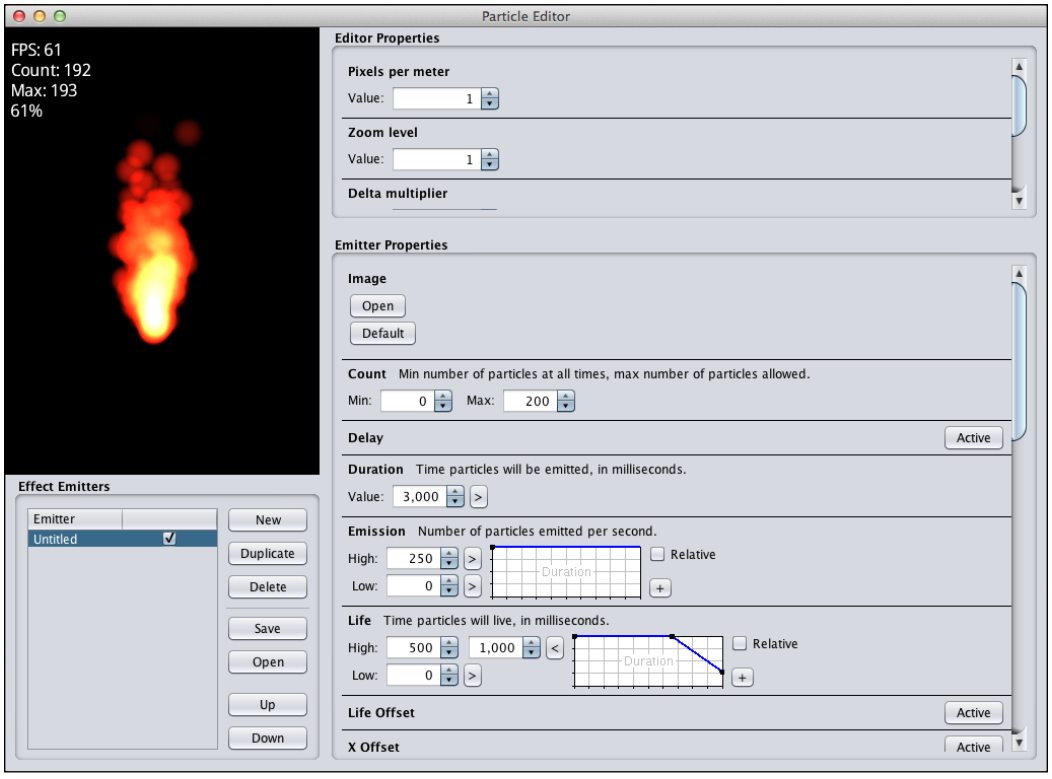
```
String.format("%d", (int)(starCount+score)).
```

With all the UI in place, our game now looks like this:

Special effects with particles

Particle systems are basically a cluster of images that behave in a specific manner in order to simulate graphical effects such as fire, smoke, explosion, and so on. Each particle system will consist of hundreds or may be thousands of small images that blend together to create the illusion of the special effect. Each of these images is referred to as a particle and will have numerous properties that can be set or controlled to create unlimited permutations and combinations resulting in infinite different special effects.

Manually creating a particle system or hand coding all the properties is no easy task. Hence, we have a LibGDX Particle Editor tool. You can download it from `libgdx.googlecode.com/svn/jws/particle-editor.jnlp`. However, it would be better to run it from the LibGDX source. It is acceptable to be overwhelmed at the first sight of the tool's interface, but at the same time I am sure you are totally impressed by the fire effect that is already playing in the editor's window:



You can see a single emitter at the bottom of the left panel. An emitter is the source for a particle system. We can have multiple emitters with different settings to have composite special effects. There is an image setting where you can change the image used for the particles. You are welcome to play with different images to see how the effect changes. The value in the **Duration** field determines the time for which particles will be emitted. The value in the **Count** field sets the minimum and maximum number of particles that will be present. The value in the **Emission** field controls the number of particles that will be emitted per second. The value in the **Life** field controls the time for which a particle lives. The value of **Spawn** sets the shape which emits particles. Other settings have self explanatory names that you can figure out.

The **Additive** option sets the blending to additive and the **Continuous** option makes the particle effect loop infinitely. For the purpose of our game, we need to create two different particle effects: one `Smoke` effect to add as trail for our plane and one `Explosion` effect to add when plane crashes. Let's create the smoke effect first by entering the following values in the respective fields:

- **Count**: In this field, enter **Min** as `0` and **Max** as `200`.
- **Duration**: In this field, enter **Value** as `1000`.
- **Emission**: In this field, enter **Low** as `100` and **High** as `250`.
- **Life**: In this field, enter **Low** as `100` and **High** from `500` to `1000`. Remove the nodes in life graph so that only first node remains.
- **Particle size**: In this field, enter **Low** as `0` and **High** as `10`.
- **Velocity**: In this field, enter **Low** as `0`, and **High** from `150` to `300`.
- **Angle**: In this field, enter **Low** as `180` and **High** from `170` to `210`.
- **Tint**: In this field, enter the value `Grey`.
- **Options**: In this field, disable the **Additive** option and enable **Continuous**.

Save your `Smoke` effect with the name `Smoke` by clicking on the **Save** button. Creating an explosion is not that simple. Click on the **Open** button in the Particle Editor and open the `Explosion` file provided with source to explore how it is done. In order to add these effects, copy the `Smoke` and `Explosion` files to the `assets` folder of the `Thrust Copter-android` project. We also need to copy the `particle.png` file to the `assets` folder. Add the particle effects with the following code:

```
manager.load("Smoke", ParticleEffect.class);
manager.load("Explosion", ParticleEffect.class);

smoke=game.manager.get("Smoke", ParticleEffect.class);
explosion=game.manager.get("Explosion", ParticleEffect.class);
```

Then, in the `updateScene` method, we need to update the effect using the following code:

```
smoke.setPosition(planePosition.x+20, planePosition.y+30);
smoke.update(deltaTime);
```

In `drawScene`, write the following code:

```
smoke.draw(batch);
```

The explosion effect is to be conditionally drawn when the plane crashes. We have a new function that gets called when this happens, in which we reset the explosion effect so that it starts again:

```
private void endGame()
{
  if(gameState != GameState.GAME_OVER)
  {
    crashSound.play();
    gameState = GameState.GAME_OVER;
    explosion.reset();
    explosion.setPosition(planePosition.x+40, planePosition.y+40);
  }
}
```

We get the following output when we run the code:

Pooling particle effects

Particle effects are resource-hungry in nature and using a lot of particles on your scene takes a toll on performance. One of the concepts used in game development to reduce runtime performance hit is pooling. Pooling is the method of creating a fixed number of instances on initialization to be used throughout the game. Each time we need an instance, we get one from the pool. Once it has served its purpose, we return it to the pool for later use. This makes sure no new instances are created at runtime. LibGDX has specialist classes to handle pooling for particles, `ParticleEffectPool` and `PooledEffect`. A `ParticleEffectPool` class can be created by providing a particle effect, initial size of the pool, and maximum size of the pool. It has an `obtain` method that returns a `PooledEffect` class that can be used as a new particle effect. Once we are done, the `PooledEffect` class can be returned to the pool using the `free` method. I would advise you to use it wherever relevant.

Summary

Things got more serious in this chapter and we refactored our code to follow standards. Now, you know how to create new screens and set them from our `Game` instance. You learned about the standardized way of loading assets via the `AssetManager` class. We created the UI for our game and explored the `BitmapFont` creator tools, especially Hiero. I believe the highlight of the chapter was Particle Editor and hope you had fun playing with it. There is no easier way to master it other than experimenting with it.

In the next chapter, you will learn about `Scene 2d` and use it to create other screens, such as the menu and loading screen.

5
Scene 2 – the Menu

Our game is almost complete but there is no menu scene. LibGDX provides Scene2D to create menus and the UI. We will use it to create our menu and loading scenes. We will do the following in this chapter:

- Explore Scene2D
- Learn how to use the `Actor`, `Group`, and `Action` classes
- Learn how to use the `Stage` class
- Create our menu scene
- Integrate a standard loading scene
- Learn how to handle Android's back and menu buttons
- Learn how to create a Nine Patch image
- Explore the layout options in Scene2D

The source files for this chapter can be found in `chapter5.zip` provided along with the book. You can open it up as a reference while you proceed through the chapter.

Introducing Scene2D

Scene2D is a 2D scene graph that helps you create UIs and applications. It can handle the laying out, input processing, and drawing processes for many standard UI elements. At the core, Scene2D comprises the following:

- `Actor`: This is a node in the graph with a position, size, origin, scale, rotation, and color. Each `Actor` node has its own coordinate system.
- `Group`: This is an `Actor` node with many child actors. The rotation and scale of a group is added to all the child actors.

- `Stage`: This handles the drawing and input handling of all the actors through a `camera` instance, a `SpriteBatch` class, and a root group holding all the actors.
- `Actions`: This is a simple system to animate actors over time.

Although Scene2D is ideally suited for a UI, it has been used to create complete games as well. You can create board games with it, but for more complicated games, I won't recommend it as it may limit your freedom of expression and the flexibility of the game mechanics.

The stage for actors

The `Stage` class can be considered as conventional stage for a drama where different artists or actors will act. Based on this analogy, `Stage` will be what we will see in a scene. We will add different actors to the `Stage` instance. It is also an `InputProcessor` interface that detects different inputs and delivers them to the relevant actors. `Stage` is initialized with a `Viewport` parameter:

```
Stage stage = new Stage(new FitViewport(800, 480));
Gdx.input.setInputProcessor(stage);
```

It has an `act` method that receives the delta time value, which in turn calls the `act` method on all the actors on `Stage`:

```
public void render (float delta) {
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    stage.act(delta);
    stage.draw();
}
```

The `Stage` class also has a `draw` method that draws everything on the stage. If we need to make any actor invisible, we can use a `setVisible` method with a `boolean` parameter. This will also toggle the input-handling capability of the actor as well. `Stage` can also be thought of as a container that holds all the Scene2D items and makes them work.

Actors and their actions

An actor can be partially thought of as the Scene2D version of LibGDX's `Sprite`. If an actor is required to receive events, then it will need to have listeners attached to it. For example, the following code shows how we can attach an `InputListener` event to an `actor` instance to handle input events. Note that we can override only the relevant functions in the listener for our purposes:

```
actor.setBounds(0, 0, texture.getWidth(), texture.getHeight());

actor.addListener(new InputListener() {
    public boolean touchDown (InputEvent event, float x, float y, int
pointer,   int button) {
        System.out.println("down");
        return true;
    }

    public void touchUp (InputEvent event, float x, float y, int
pointer, int   button) {
        System.out.println("up");
    }
});
```

Any actor can be assigned actions to perform some standard animations. Once an `Action` class is applied, it gets updated via the `act` method. The action will get removed once it is complete. Some of the available actions are as follows:

- `MoveToAction`
- `MoveByAction`
- `RotateByAction`
- `RotateToAction`
- `ScaleByAction`
- `ScaleToAction`
- `SizeByAction`
- `SizeToAction`

The functionality of these `Action` classes can be understood from their names. Adding an `Action` class to an actor is easy:

```
MoveToAction action = new MoveToAction();
action.setPosition(x, y);
action.setDuration(duration);
actor.addAction(action);
```

Complicated actions can be done by running multiple actions in parallel or in sequence. An `Action` class can also have tweening curves applied to it:

```
action.setInterpolation(Interpolation.bounceOut);
```

Widgets

The `scene2d.ui` package provides a collection of widgets and other classes that are built and based on Scene2D. There are layout widgets such as the following:

- `Table`
- `Container`
- `Stack`
- `ScrollPane`
- `SplitPane`
- `Tree`
- `VerticalGroup`
- `HorizontalGroup`

The normal widgets available are as follows:

- `Label`
- `Image`
- `Button`
- `TextButton`
- `ImageButton`
- `CheckBox`
- `ButtonGroup`
- `TextField`
- `TextArea`
- `List`
- `SelectBox`
- `ProgressBar`
- `Slider`

- `Window`
- `TouchPad`
- `Dialog`

Widgets do not set their size or position; their respective parent widgets will set these for them. We need to call the `invalidate` method of a widget if its state changes. If the state change affects the size of the parent widget, then we need to call the `invalidateHierarchy` method that calls the `invalidate` method of each parent widget. In a typical case, we will just add a `Table` widget class that will fill the entire `Stage` class where all the other widgets will be added. An example of this is as follows:

```
Table table = new Table();
table.setFillParent(true);
stage.addActor(table);
```

The `Table` class is a complicated topic in itself. You will find more details at `https://github.com/libgdx/libgdx/wiki/Table`.

To create any widget, we will need to provide a `style` instance that will determine how it will be drawn. Different widgets have different style classes that we can use. The following code shows a typical use case:

```
TextButtonStyle style = new TextButtonStyle();
style.up = new TextureRegionDrawable(upTexture);
style.down = new TextureRegionDrawable(downTexture);
style.font = buttonFont;

TextButton playButton = new TextButton("Play", style);
table.add(playButton);
```

Adding a loading scene

Let's use the information we have learned so far to implement a loading scene. We will use a standard one available at `https://github.com/Matsemann/libgdx-loading-screen/tree/libgdx1.2.0-EyeOfMidas`. This `LoadingScreen` class is created by Mats Svensson. Go ahead and download the class and its support files. The purpose of a loading scene is to show a progress bar while the assets are being loaded. This is not relevant for our simple game, but once you have heavier assets and multiple texture atlas classes, this becomes inevitable. From the Android project of the downloaded file, copy the `loading.pack` and `loading.png` files to the `assets` folder in the `Thrust Copter-android` project. Copy the `matsemann` folder from the downloaded core project to our `com` folder within the `Thrust Copter-core src` folder. We need to do some cleanup now, so go ahead and fire up Eclipse. Remove the `SomeCoolGame`, `AbstractScreen`, and `MainMenuScreen` classes as we do not need them. Edit the `LoadingScreen` class to extend `ScreenAdapter` instead of `AbstractScreen`. Also, change the constructor to receive the instance of our `Game` class as shown in the following code:

```
public class LoadingScreen extends ScreenAdapter {
  ThrustCopter game;

  public LoadingScreen(ThrustCopter thrustCopter) {
    game=thrustCopter;
  }

}
```

There will be an error in the `render` method where it was trying to set a scene to a class we already removed. Replace it to set `ThrustCopterScene` instead:

```
game.setScreen(new ThrustCopterScene(game));
```

Now we need to move all the assets' loading code to the `LoadingScreen` class. Move all of the code from the `create` method of the `ThrustCopter` class to the end of the `show` method of `LoadingScreen`. The `create` method of `ThrustCopter` now looks like this:

```
  public void create() {
  batch=new SpriteBatch();
  setScreen(new LoadingScreen(this));
}
```

The `render` method of `LoadingScreen` needs to be edited as we do not need to wait for the user to tap to proceed with our game:

```
if (game.manager.update()) {
  game.atlas=game.manager.get("thrustcopterassets.txt", TextureAtlas.
class);
  game.font=game.manager.get("impact-40.fnt", BitmapFont.class);
  game.setScreen(new ThrustCopterScene(game));
}
```

The output of the preceding code can be seen as follows:

I would also advise you to add a `dispose` method that disposes the instances used in `LoadingScreen`. Run the desktop project to see a beautiful LibGDX-branded progress bar.

Investigating the LoadingScreen class

Let's see how the `LoadingScreen` class is wired. It will be interesting to note that the initialization code is actually in the `show` method and not in the constructor. The `show` method gets automatically called when a `Screen` instance is initialized. We load the assets needed for the loading bar and force the `AssetManager` class to load it instantly. A `Stage` instance is created along with multiple `Image` widgets and a `LoadingBar` instance. The `LoadingBar` class actually extends `Actor` with a little bit of additional code to draw the bar animation. All of these are added to `Stage` via the `addActor` method.

The `resize` method is where all the widgets and actors are positioned properly. The `render` method checks whether the `AssetManager` class has loaded the assets completely by executing the following code:

```
if (game.manager.update()) {
...
}
```

We also find the percentage of completion and update the relevant graphics using the following code:

```
// Interpolate the percentage to make it more smooth
percent = Interpolation.linear.apply(percent, game.manager.
getProgress(), 0.1f);

// Update positions (and size) to match the percentage
loadingBarHidden.setX(startX + endX * percent);
loadingBg.setX(loadingBarHidden.getX() + 30);
loadingBg.setWidth(450 - 450 * percent);
loadingBg.invalidate();
```

The `hide` method unloads the `pack` file used to load bar graphics, which is no more necessary. The `LoadingScreen` class demonstrates how `Stage`, `Widgets`, and `Actors` can be used in the minimal way possible.

Adding the menu scene

The Thrust Copter menu scene needs to have the **PLAY GAME**, **SOUND OPTIONS**, **LEADERBOARD**, and **EXIT GAME** buttons. We need additional graphics for these and we will also learn how to make those scalable graphics skins. Go ahead and create a new class named `MenuScene` that extends `ScreenAdapter` and accepts the `ThrustCopter` instance. We will use the `uiskin.json`, `uiskin.png`, `uiskin.atlas`, and `default.fnt` files from the assets of the LibGDX tests project. We will make use of two tables in this scene: one for the menu buttons and another for the options display. The code in the constructor is as follows:

```
stage = new Stage(game.viewport);
Gdx.input.setInputProcessor(stage);
skin = new Skin(Gdx.files.internal("uiskin.json"));

screenBg = new Image(game.atlas.findRegion("background"));
title= new Image(game.manager.get("title.png", Texture.class));
helpTip=new Label("Tap around the plane to move it!",skin);
helpTip.setColor(Color.NAVY);
```

```
table=new Table().debug();
playButton=new TextButton("PLAY GAME", skin);
table.add(playButton).padBottom(10);
table.row();
optionsButton=new TextButton("SOUND OPTIONS", skin);
table.add(optionsButton).padBottom(10);
table.row();
table.add(new TextButton("LEADERBOARD", skin)).padBottom(10);
table.row();
exitButton=new TextButton("EXIT GAME", skin);
table.add(exitButton);
table.setPosition(400, -200);

options=new Table().debug();
Label soundTitle=new Label("SOUND OPTIONS",skin);
soundTitle.setColor(Color.NAVY);
options.add(soundTitle).padBottom(25).colspan(2);
options.row();
muteCheckBox = new CheckBox(" MUTE ALL", skin);
options.add(muteCheckBox).padBottom(10).colspan(2);
options.row();
options.add(new Label("VOLUME ",skin)).padBottom(10).padRight(10);
volumeSlider = new Slider(0, 2, 0.2f, false, skin);
options.add(volumeSlider).padTop(10).padBottom(20);
options.row();
backButton=new TextButton("BACK", skin);
options.add(backButton).colspan(2).padTop(20);
options.setPosition(400, -200);
muteCheckBox.setChecked(!game.soundEnabled);
volumeSlider.setValue(game.soundVolume);

stage.addActor(screenBg);
stage.addActor(title);
stage.addActor(helpTip);
stage.addActor(table);
stage.addActor(options);

playButton.addListener(new ClickListener(){
  @Override
  public void clicked(InputEvent event, float x, float y) {
           game.setScreen(new ThrustCopterScene(game));
      }
});
optionsButton.addListener(new ClickListener(){
      @Override
      public void clicked(InputEvent event, float x, float y) {
          showMenu(false);
```

```
        }
    });
    exitButton.addListener(new ClickListener(){
          @Override
          public void clicked(InputEvent event, float x, float y) {
              Gdx.app.exit();

          }
    });
    volumeSlider.addListener(new ChangeListener() {
      public void changed (ChangeEvent event, Actor actor) {
        game.soundVolume=volumeSlider.getValue();
      }
    });
    muteCheckBox.addListener(new ChangeListener() {
      public void changed (ChangeEvent event, Actor actor) {
        game.soundEnabled=!muteCheckBox.isChecked();
      }
    });
    backButton.addListener(new ClickListener(){
          @Override
          public void clicked(InputEvent event, float x, float y) {
            showMenu(true);
          }
    });
```

Do check out the `uiskin.json` file to see how it drives the skinning.

The backdrop is added as an image named `screenBg`. The game title is also an image named `title`. There is a small label named `helpTip` that displays the instructions at the bottom. The table named `table` contains all the menu buttons. The table named `options` has a slider named `volumeSlider`, a checkbox named `muteCheckBox`, and a text button named `backButton`. Check how the items are added and laid out in these tables. As these tables are created with the `debug` method that is invoked, we can call `Table.drawDebug(stage)` in the `render` method to see how the tables are laid out. We have created a couple of public variables in the `ThrustCopter` class to track the volume of sounds through `soundVolume` and whether the sound is enabled through `soundEnabled`. Sound is enabled by default and volume is set to 1. The `show` method will position the `title` and `helpTip` instances and add an `Action` instance to the `title` instance, which will make it fly in when the scene is loaded:

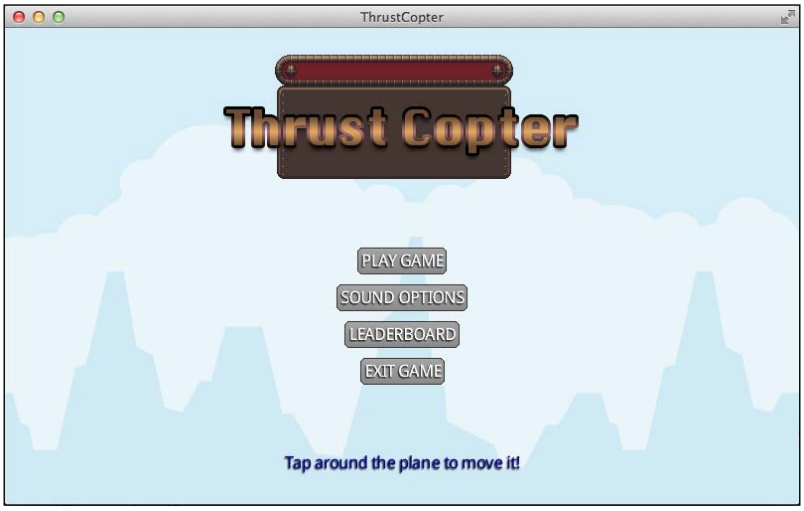
```
@Override
public void show() {
  title.setPosition(400-title.getWidth()/2, 450);
  helpTip.setPosition(400-helpTip.getWidth()/2, 30);

  MoveToAction actionMove = Actions.action(MoveToAction.class);
  actionMove.setPosition(400-title.getWidth()/2, 320);
  actionMove.setDuration(2);
  actionMove.setInterpolation(Interpolation.elasticOut);
  title.addAction(actionMove);

  showMenu(true);
}
```

There is a `showMenu` method that toggles the display of the two tables, thereby showing the menu or options. Here as well, we will use `Action` to tween these tables on and off the screen. An example of this is as follows:

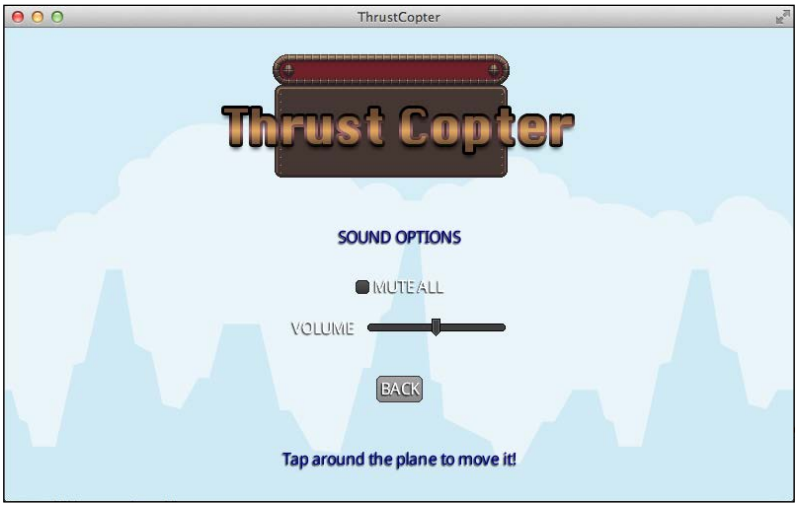
```
private void showMenu(boolean flag) {
  MoveToAction actionMove1 = Actions.action(MoveToAction.class);//out
  actionMove1.setPosition(400, -200);
  actionMove1.setDuration(1);
  actionMove1.setInterpolation(Interpolation.swingIn);

  MoveToAction actionMove2 = Actions.action(MoveToAction.class);//in
  actionMove2.setPosition(400, 190);
  actionMove2.setDuration(1.5f);
  actionMove2.setInterpolation(Interpolation.swing);

  if(flag){
    table.addAction(actionMove2);
    options.addAction(actionMove1);
  }else{
    options.addAction(actionMove2);
    table.addAction(actionMove1);
  }
}
```

Remember to remove the `InputProcessor` interface in the `hide` method so that it does not hijack the input processing when proceeding to another scene.

Make sure that `LoadingScreen` loads `MenuScene` rather than the `ThrustCopterScene` class now. Check out the menu scene with the menu buttons displayed in the following screenshot:



Check it out with the options displayed in the following screenshot:



We also need to make relevant changes to our `ThrustCopterScene` class to ensure all sounds are controlled by `soundEnabled` and `soundVolume`:

```
if(game.soundEnabled){
  music = game.manager.get("sounds/journey.mp3", Music.class);
```

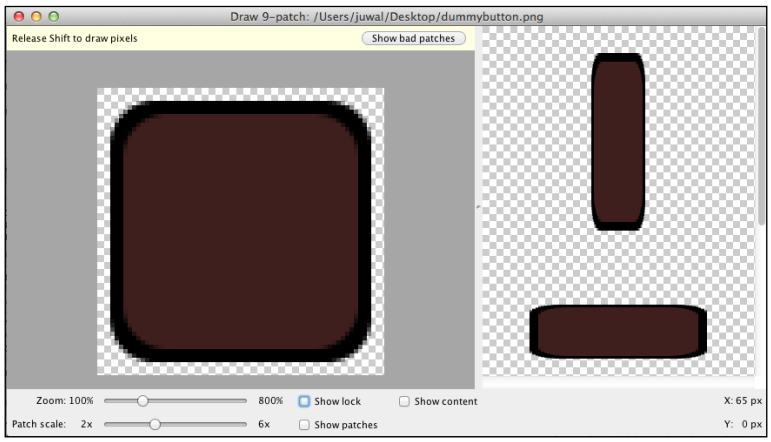
```
    music.setLooping(true);
    music.play();
    music.setVolume(game.soundVolume);
    ...
  }
  ...
  if(game.soundEnabled)spawnSound.play(game.soundVolume);
```

Creating Scene2D layouts is not a simple straightforward task as it may seem. You will definitely need more help. Please refer to `https://github.com/EsotericSoftware/tablelayout`.

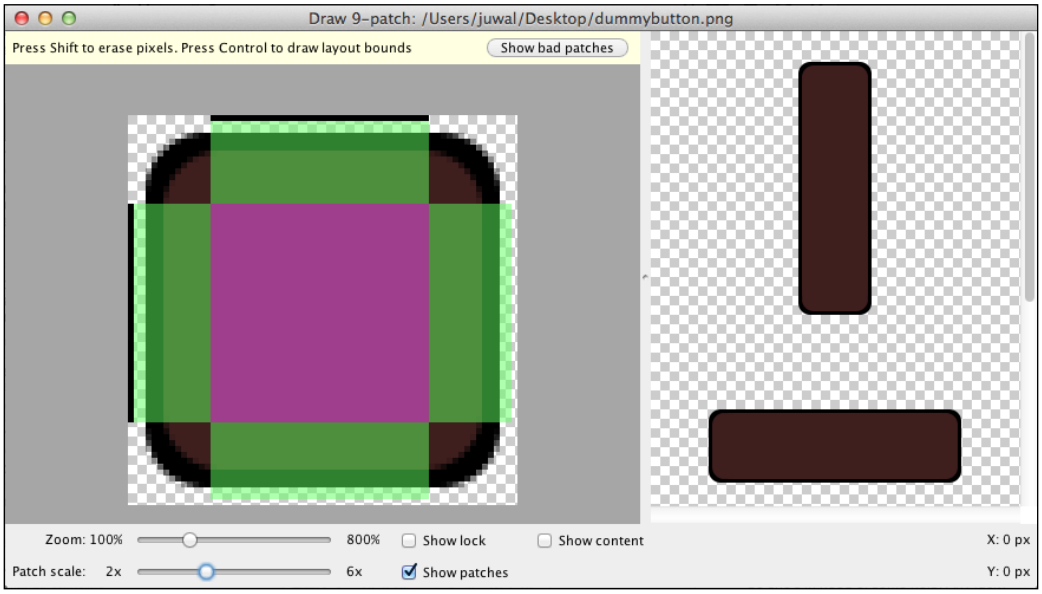
Creating scalable skins using the 9-patch tool

We have created many buttons with varying sizes, and there seems to be no distortion in the button's base image used. This is so because we are making use of a Nine Patch image. A Nine Patch image is drawn in nine sections. This enables it to stretch only the middle section while scaling, thereby not stretching those areas that may make it look pixelated. The original `.png` file when made into a Nine Patch will be saved as a `.9.png` file. More details on Nine Patch can be found at `http://developer.android.com/guide/topics/graphics/2d-graphics.html#nine-patch`.

The Android SDK has a Draw 9-Patch tool that we can use to create Nine Patch images. More details on this tool can be found at `http://developer.android.com/tools/help/draw9patch.html`. The `draw9patch` application can be found in the `tools` folder within the Android SDK folder. Run it from the command line or terminal. I have provided a `dummybutton.png` file with the source, which is a rounded rectangle image that can be used as the button's base. You can drag-and-drop it inside the `draw9patch` application window to see the following result that will show how distortion happens when it is stretched:



We can draw lines outside the image to show the area where the stretching should occur. Press *Ctrl/command* and draw the lines on the top and left borders as shown in the following screenshot. Enable the **Show patches** checkbox to see the patches being displayed. The pink area will be the one that will get stretched. Check how the display will be updated to show no distortion. This image can be saved via **File | Save 9-patch** and the screenshot is as follows:



Such an image can be used to create a `TextButton` instance by supplying a `TextButtonStyle` class manually:

```
textButtonStyle = new TextButtonStyle();
textButtonStyle.font = font;
textButtonStyle.up = skin.getPatch("buttonup");
textButtonStyle.down = skin.getPatch("buttondown");
button = new TextButton("Play", textButtonStyle);
```

The `Skin` class can be created from our texture atlas instead of being initialized from a JSON file as we had done. A 9 patch image can also be created dynamically, which can be explored via `https://gist.github.com/briangriffey/4391807`.

Handling the Android back button

Android devices have a few additional buttons along with the home button, which is found also on iOS devices. There is a back button, a menu button, and sometimes a search button. The standard way of handling them is to capture and disable all the buttons other than the back button. When the back button is pressed, the game will respond in a context-sensitive way, depending on the scene being displayed. For example, when we are in the game scene, it will pause to show the options to resume, go back to the menu, and so on. Pausing should also happen when the application gets interrupted as in the case of a call coming through. On the menu scene, the user should be prompted with an exit dialog box.

As each screen needs to implement the exit dialog box in a different way, it is better to capture it on a new class that gets extended by all our scenes. Then, each scene can override the specific function to implement the context-sensitive result. For the game screen, we can just call the `pause` and `resume` methods. The `pause` method gets called when the game gets interrupted by another app or call. For the menu, we can open an exit dialog box. Let's create a `BaseScene` class that extends the `ScreenAdapter` class, which `MenuScene` and `ThrustCopterScene` will extend. We will capture the menu key and back key as we had learned in input handling section:

```
protected ThrustCopter game;
private boolean keyHandled;
public BaseScene(ThrustCopter thrustCopter) {
  game=thrustCopter;
  keyHandled=false;
  Gdx.input.setCatchBackKey(true);
  Gdx.input.setCatchMenuKey(true);
}
@Override
public void render(float delta) {
  super.render(delta);
  if(Gdx.input.isKeyPressed(Keys.BACK)){
    if(keyHandled){
      return;
    }
    handleBackPress();
    keyHandled=true;
  }else{
```

```
      keyHandled=false;
    }
  }
  protected void handleBackPress() {
    System.out.println("back");
  }
```

Now in our scenes, we can override the `handleBackPress` method to add the functionality. Check how the exit dialog is implemented in `MenuScene` and the game is paused in `ThrustCopterScene`.

Do not forget to call `super.render(delta)` from within the `render` methods of all the scenes that extend the `BaseScene` class.

Summary

We entered a whole new world of Scene2D in this chapter. Although we have only touched the surface of this incredible package, we have learned how to implement a simple menu scene. We also used a Scene2D-based `LoadingScreen` to effectively load assets and show a progress bar. We learned to capture the Android back button and create 9 patch images using the `draw9patch` tool in the Android SDK. If you explore further, you will find that complete games have been made using Scene2D, hence consider this as a good starting point to you exploring much more.

In the next chapter, we will learn about the `Box2D` physics library and convert our game to use it for many physics simulations.

6
Physics with Box2D

Almost all of us have played the *Angry Birds* game. Such games involve physics calculations where we need to find out realistic results of how the different items in the scene collide and interact with each other. Real-world physics equations need to be used to calculate these results, and it is indeed a complicated affair. Thanks to Erin Catto, we have the exceptional yet free and open source physics library Box2D to save the day.

In this chapter, we will explore Box2D and cover the following topics:

- An introduction to Box2D
- Creating a new game scene, this time based on Box2D physics
- Learning how to create different types of Box2D bodies
- Learning how to detect collisions between Box2D bodies
- Learning how to disable certain collisions
- Learning how to draw the Box2D world using the debug renderer
- Learning how to map the Box2D world with our Camera world
- Learning how to implement fixed time updates for our Box2D world

The incredible world of Box2D

Box2D is an open source C++ engine for simulating rigid body physics in 2D. Box2D was developed by Erin Catto and has the zlib license. It was developed in a platform-independent way, so it can be ported to many other programming languages. In most cases, the naming conventions and methodologies were left unchanged or were drastically similar so that the existing knowledge of Box2D in any language could be used in any other language as well. Here are the features of Box2D:

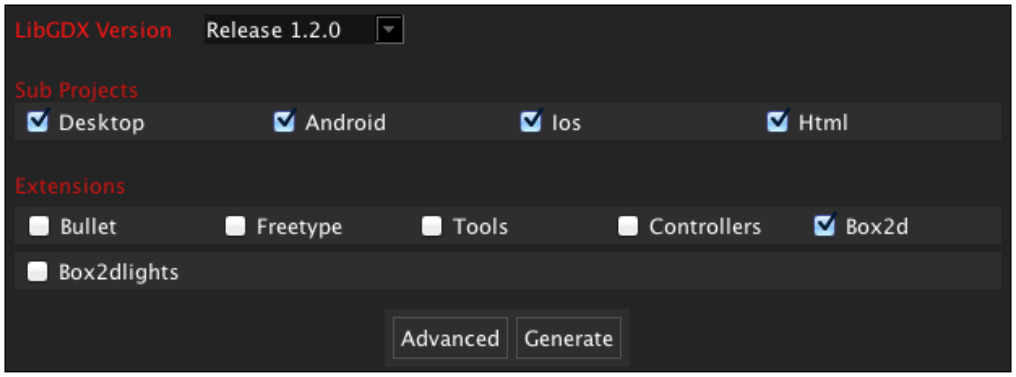
- Collision:
    - Continuous collision detection
    - Contact callbacks (begin, end, pre-solve, post-solve)
    - Convex polygons and circles
    - Multiple shapes per body
    - One-shot contact manifolds
    - Dynamic tree broadphase
    - Efficient pair management
    - Fast broadphase AABB queries
    - Collision groups and categories
- Physics:
    - Continuous physics with time of impact solver
    - Persistent body-joint-contact graph
    - Island solution and sleep management
    - Contact, friction, and restitution
    - Stable stacking with a linear-time solver
    - Revolute, prismatic, distance, pulley, gear, mouse joint, and other types of joints
    - Joint limits, motors, and friction
    - Momentum-decoupled position correction
    - Fairly accurate reaction forces/impulses

All the documentation that can be found in the official Box2D manual (available at `http://box2d.org/manual.pdf`) is valid for LibGDX as well. Keep this PDF document as a handy reference, as you are going to need it for serious Box2D experimentation.

In simple terms, Box2D can be considered as a system where real-world physics can be simulated to find realistic results. We can create objects; apply gravity, force, friction, restitution, density, and joints; and then make different rigid bodies interact with each other in a way real objects interact. The difference is that in a real-world situation, the interaction is in 3D, but here in Box2D, it is 2D only.

LibGDX with Box2D

LibGDX uses a thin Java wrapper over the original Box2D software that makes use of **Java Native Interface** (**JNI**). Currently, Box2D is a separate extension that needs to be manually included. As we will need to try out different experiments as we proceed through the chapter, we can add Box2D to our existing project now. We can also create another project just to try out Box2D, in which case you just need to check the option in the `gdx-setup.jar` file (refer to the following screenshot) to enable Box2D while creating your project:



However, in our Thrust Copter project, we did not enable Box2D when we created the project. It's time to learn how we can add new extensions after the project is created. We will need to edit the `build.gradle` file present in the root folder of our project. Every project folder will have a file with this name, so make sure you open the one in the root folder, which is the top-most folder containing all the different platforms' project folders. The different sections in this file list the dependencies each platform has. We will need to add the following additional lines of code in the relevant sections.

Core dependency:

```
compile "com.badlogicgames.gdx:gdx-box2d:$gdxVersion"
```

Desktop dependency:

```
compile "com.badlogicgames.gdx:gdx-box2d-platform:
$gdxVersion:natives-desktop"
```

Android dependency:

```
compile "com.badlogicgames.gdx:gdx-box2d:$gdxVersion"
natives "com.badlogicgames.gdx:gdx-box2d-platform:
$gdxVersion:natives-armeabi"
```

```
natives "com.badlogicgames.gdx:gdx-box2d-platform:
$gdxVersion:natives-armeabi-v7a"
natives "com.badlogicgames.gdx:gdx-box2d-platform:
$gdxVersion:natives-x86"
```

iOS dependency:

```
compile "com.badlogicgames.gdx:gdx-box2d:$gdxVersion"
natives "com.badlogicgames.gdx:gdx-box2d-platform:
$gdxVersion:natives-ios"
```

HTML dependency:

```
compile "com.badlogicgames.gdx:gdx-box2d-gwt:$gdxVersion:sources"
compile "com.badlogicgames.gdx:gdx-box2d:$gdxVersion:sources"
```

These dependencies are specific to the extension we are adding. After saving the file, we need to select all our projects in Eclipse, click on the right mouse button, select **Gradle**, and click on **Refresh Dependencies**. This will make Gradle download all the missing libraries and files to set up our Box2D environment.

Creating a Box2D world

We will need to set up a Box2D `World` instance first to start playing with Box2D. This instance holds all the Box2D objects and runs the physics simulations within it. Remember that Box2D deals with meters for scale, that is, 1 unit in Box2D is 1 meter. Let's see how the `world` instance is created:

```
World world = new World(new Vector2(0, -9.8), true);
```

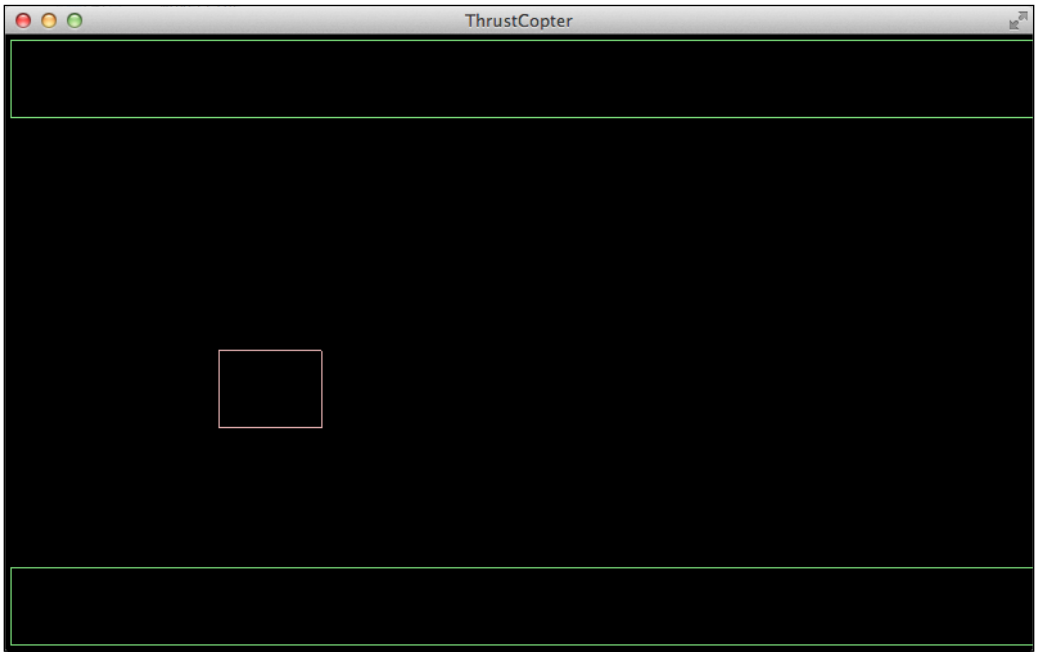
We used a gravity vector and a Boolean value that decides whether objects should sleep within the world. It is suggested that you enable sleeping of objects in order to save CPU cycles, as physics calculations tend to be very CPU-intensive. Box2D can only simulate physics; we will need to use LibGDX graphics for rendering. The gravity value can be anything, but ideally it should be close to 9.8—the real-world value for acceleration due to gravity. Once the world is created, we can add rigid bodies to it.

Drawing the Box2D world

Box2D does not have rendering capability; it is a library used to simulate physics. The game graphics need to be rendered manually using the LibGDX graphics functionality. However, in order to see what is happening in the Box2D `world` instance, we can use a `Box2DDebugRenderer` class provided in LibGDX. This renderer actually draws wireframe primitives of the `world` instance as shown in the following code:

```
world = new World(new Vector2(0, -10), true);
debugRenderer = new Box2DDebugRenderer();
box2dCam=new OrthographicCamera(8, 4.8);
box2dCam.position.set(4, 2.4, 0);
...
if(DRAW_BOX2D_DEBUG){
  box2dCam.update();
  debugRenderer.render(world, box2dCam.combined);
}
```

Box2D works perfectly within a world size of 10 meters, and care should be taken to ensure that all Box2D items are smaller than this or have a size corresponding to their real-world size. For example, a hero can be 1 meter tall and should therefore have a Box2D body size of 1 unit. Note that we are using a new `OrthographicCamera` instance named `box2dCam`, which is of size 8 x 4.8. Box2D deals with meters and each unit is a meter. This essentially means an 8 meter x 4.8 meter space. This camera is used by the `debugRenderer` instance. Remember that our game camera is set to 800 x 480, which means that the relation between the Box2D world and our game world is 1:100. This means 1 Box2D unit will make 100 units in the game scene. The `DRAW_BOX2D_DEBUG` flag is a Boolean value used to toggle the debug rendering, as we won't need it for the final game; it is required only in the development stage. Once you add rigid bodies in the `world` instance, your world may be rendered as shown in the following screenshot:



The screenshot actually shows the start of our game where the middle box is our plane and the top and bottom boxes are the terrains drawn using `Box2DDebugRenderer`.

We created our game camera in pixel sizes, which meant we could easily place our art based on the size values. We could have created that camera in 8 x 4.8 size, in which case we would not have to convert between the game world and Box2D world coordinates.

Simulating the Box2D world

To calculate the physics, we need to tell the `world` instance to run its `step` method. We will call the `step` method from within our `render` method, as physics needs to be calculated as many times as possible for accurate results:

```
world.step(deltaTime, 8, 3);
```

The arguments passed to the `step` method are `velocityIterations` and `positionIterations`, which are the number of times velocity and position of a rigid body is calculated by Box2D. The higher these numbers, the better the results. However, the performance will be affected. The recommended values are `8` and `3`, but a lower value can be used if it works fine. The `step` method calculates the new velocity and position of all the rigid bodies in the `world` instance based on their current movement, interaction, collision, and forces in action.

Fixing the time step

Ideally, when the FPS is stable, the `deltaTime` value will remain constant. In reality, this won't be the case, and we will have different `deltaTime` values within the `render` method. This happens when the device is slower, or the processor is multitasking with some resource-hungry operations, or if your code is on the heavier side. It is recommended that constant values should be used to step through the physics calculations. We must try to make sure that physics stepping is done based on the same value as much as possible for correct results. The solution is to step based on a lower fixed time irrespective of the `deltaTime` value, but repeat as many times as needed. Refer to the following code:

```
private float tempValue = 0;

private void fixedTimeStep(float deltaTime) {
   float frameTime = Math.min(deltaTime, 0.25f);
    tempValue += frameTime;
    while (tempValue >= 0.1) {
```

```
            world.step(0.1, 8, 3);
            tempValue -= 0.1;
        }
    }
```

This sample code loops through to call the `step` method using a fixed value of `0.1` as many times as possible within the current `deltaTime` value. Such an approach is highly recommended in a complicated physics game, but we will skip this in our game for the sake of simplicity.

You can find out more about this topic at `http://gafferongames.com/game-physics/fix-your-timestep/`.

Box2D rigid bodies

A body is just a collection of matter with some attributes assigned to it, such as its position and orientation. It is what we usually call an object in the real world. A rigid body describes an idealized body, which is assumed to be solid and thus incapable of being deformed by the exerting forces. Box2D supports rigid bodies, but the 3D physics library `bullet` has support for soft bodies as well. There are three types of bodies:

- **Static**: This type of body cannot move and is not affected by forces. However, they will affect a dynamic body (the ground).
- **Kinematic**: This type of body is similar to a static body that can move. It is not affected by forces and can affect dynamic bodies (moving platforms).
- **Dynamic**: This type of body will be affected by forces and is capable of moving around and interacting with other static, dynamic, or kinematic bodies (bouncing balls).

Creating a body requires a bit of ground work, as this translates directly from the C++ implementation. Refer to the following code:

```
BodyDef bodyDef = new BodyDef();
bodyDef.type = BodyType.DynamicBody;
bodyDef.position.set(10, 30);

Body body = world.createBody(bodyDef);

CircleShape circle = new CircleShape();
circle.setRadius(6f);

FixtureDef fixtureDef = new FixtureDef();
```

```
fixtureDef.shape = circle;
fixtureDef.density = 0.5f;
fixtureDef.friction = 0.4f;
fixtureDef.restitution = 0.6f;

Fixture fixture = body.createFixture(fixtureDef);
circle.dispose();
```

The preceding code is a bit too much just to create a `Dynamic` body. Let's try to understand what is happening here. We use a body definition `BodyDef` to define the type of body along with its position in the Box2D `world` instance. Various shapes are available to create the shape of our body, for example, `ChainShape`, `PolygonShape`, `CircleShape`, and `EdgeShape`. Fixtures store the properties of the body, which is defined using a fixture definition `FixtureDef`. Fixtures can have density, which sets the mass, friction, and restitution. This in turn sets the bounciness. Go ahead and create a few bodies to see whether they show up in the debug renderer.

We can also create custom shapes by providing vertex data:

```
PolygonShape trianglePoly = new PolygonShape();
float[] vertices = {-5.4f, -11.95f, 1.1f, 11.95f,
5.4f, -11.95f};
trianglePoly.set(vertices);
boxBody.createFixture(trianglePoly, 1);
trianglePoly.dispose();
```

Interactions in the Box2D world

A kinematic body needs to be assigned a velocity for it to move. It won't be affected by forces or static bodies. We will use it for our meteor rocks:

```
meteorBody.setLinearVelocity(2.0f, 0.0f);
```

We will need to add forces to the applicable dynamic bodies to move them or they need to be affected by other `static`, `dynamic`, or `kinematic` bodies. If the Box2D world has constant force at play, then we can set the resultant of that force as the final force when creating the `world` instance. For example, the following code says that there is a horizontal force (8,0) in the `world` instance along with gravity of (0,-10), resulting in an effective force of (8,-10). The code is as follows:

```
world = new World(new Vector2(8, -10), true);
```

Such a force will affect all dynamic bodies in this world in every physics step. Alternatively, we can apply force or impulse to move a dynamic body. Force is applied over time and needs to be applied at a position. Refer to the following code:

```
planeBody.applyForce(1.0f, 0.0f, pos.x, pos.y, true);
```

Impulse has an immediate effect than force which acts over time. Refer to the following code:

```
planeBody.applyLinearImpulse(1.0f, 0, pos.x, pos.y, true);
```

For our game, we will use impulse to provide the force when we tap near the plane.

Linking the Box2D and game worlds

Till now, we are only able to see wireframe drawings using the debug renderer. So, how will we relate our game graphics with the Box2D bodies? One way is to store the graphics data using the `setUserData` method for the corresponding body as in the following code:

```
heroBody.setUserData(heroTexture);
```

Then, later in the `render` method, we can poll the position and rotation of the body as well as the texture using `getUserData` and draw it as in the following code:

```
position= heroBody.getPosition();
rotation=(MathUtils.radiansToDegrees * heroBody.etAngle();
heroTexture= (TextureRegion)heroBody.getUserData();
batch.draw(heroTexture, position.x, position.y, 0, 0, 88, 73, 1,
1, rotation);
```

Care has to be taken that the position is converted from the Box2D space to game space, which is in the ratio of 1:100 in our case. So, we will need to multiply the position value by 100:

```
position.scl(100);
```

Detecting collisions

We will need to detect collisions as they happen to execute some additional code pertaining to our game logic. For this purpose, we need to use a `ContactListener` instance as follows:

```
world.setContactListener(new ContactListener() {
  @Override
  public void beginContact (Contact contact) {
    // System.out.println("begin contact");
  }

  @Override
```

```
    public void endContact (Contact contact) {
      // System.out.println("end contact");
    }

    @Override
    public void preSolve (Contact contact, Manifold oldManifold) {
      System.out.println(contact.getFixtureA().getBody());
      System.out.println(contact.getFixtureB().getBody());
    }

    @Override
    public void postSolve (Contact contact, ContactImpulse impulse) {
    }
  });
```

The listener methods `beginContact` and `endContact` are fired when a collision happens and ends, respectively. There will always be two fixtures with contact names A and B. These fixtures can be retrieved and their related bodies are identified as follows:

```
contact.getFixtureA().getBody();
```

Listener methods `preSolve` and `postSolve` are fired just before the collision happens and just after the collision has ended. These can be used in specific use cases where we may need to disable the collision or enable something else after the collision.

There is a Physics Body Editor tool by Aurelien Ribon that can help you create complex shapes based on graphics data. It can be found at `http://www.aurelienribon.com/blog/projects/physics-body-editor/`.

So much for the theory, let's get on with converting our game.

Box2D version of Thrust Copter

With what we have learned so far, let's try to completely redo our game scene to use Box2D instead of our old code. First, check whether the Box2D library is retrieved with Gradle, as explained earlier in this chapter. We will retain all the old source as well as that will still be our real game. Create a new folder Box2D within the current package and create a new `ThrustCopterSceneBox2D` class, extending `BaseScene` within this new folder. So, all our Box2D-related classes will be placed in this new package, `com.csharks.thrustcopter.box2d`. We will reuse some of the code from the original `ThrustCopterScene` class, but we will change the code for movement, rendering, and creation of items.

Creating and placing objects

Our approach with the original game is to move all items from right to left while keeping the plane at a specific *x* position to create the illusion of scrolling. In our Box2D version, we will let our plane fly through the world, thereby moving the plane and not the other objects. All other items will be part of the world and may remain stationary as the pillars and pickups or move on their own as the meteor rocks. We will use a variable to store the ratio between Box2D units (meter) and game world units (pixels). Refer to the following code:

```
private static final int BOX2D_TO_CAMERA =100;
```

We have a new method called `initPhysics` where the Box2D variables are initialized. We create rigid bodies for the plane, the top and bottom terrains, and the meteors. The `meteorBody` instance is placed away from our viewable area until it is used. Refer to the following code:

```
private void initPhysics() {
  world = new World(new Vector2(5f, -8), true);
  debugRenderer = new Box2DDebugRenderer();
  box2dCam=new OrthographicCamera(8, 4.8);
  box2dCam.position.set(4, 2.4, 0);
  previousCamXPos=4;

  planeBody=createPhysicsObjectFromGraphics(plane.getKeyFrame(0),
planePosition,BodyType.DynamicBody);
  terrainBodyUp=createPhysicsObjectFromGraphics(terrainAbove,
new   Vector2(terrainAbove.getRegionWidth()/2,480-  terrainAbove.
getRegionHeight()/2), BodyType.StaticBody);
  terrainBodyDown=createPhysicsObjectFromGraphics(terrainBelo
w, new   Vector2(terrainBelow.getRegionWidth()/2,   terrainBelow.
getRegionHeight()/2),BodyType.StaticBody);
  meteorBody=createPhysicsObjectFromGraphics(selectedMeteorTexture,
new   Vector2(800,500),BodyType.KinematicBody);
}
```

You will notice that the world has a gravity of `-8` and a force of `5` in positive *x* direction. If we are creating a realistic simulation, then the value of gravity has to be -10. However, our game is not a realistic one. For example, our plane flies at `2.4` meters from the ground and has a size of 0.88 m x 0.73 m. With a realistic gravity, we will have a hard time keeping it above ground.

A new function, `createPhysicsObjectFromGraphics`, helps us to handle all the necessary boilerplate code to create rigid bodies. This can be given a texture region with a position and body type to create a rigid body, as shown in the following code:

```
private Body createPhysicsObjectFromGraphics(TextureRegion region,
Vector2 position, BodyType bodyType) {
  BodyDef boxBodyDef = new BodyDef();
  boxBodyDef.type = bodyType;
  boxBodyDef.position.x = position.x/BOX2D_TO_CAMERA;
  boxBodyDef.position.y = position.y/BOX2D_TO_CAMERA;
  Body boxBody = world.createBody(boxBodyDef);
  PolygonShape boxPoly = new PolygonShape();
  boxPoly.setAsBox(region.getRegionWidth()/(2*BOX2D_TO_CAMERA),
region.getRegionHeight()/(2*BOX2D_TO_CAMERA));

  FixtureDef fixtureDef = new FixtureDef();
  fixtureDef.shape = boxPoly;
  fixtureDef.density=1;
  fixtureDef.restitution=0.2f;
  boxBody.createFixture(fixtureDef);

  boxPoly.dispose();
  boxBody.setUserData(region);
  return boxBody;
}
```

The `render` method calls the Box2D variant of `updateScene` and `drawScene`, and it draws the Box2D `world` instance using the debug renderer. Refer to the following code:

```
updateSceneBox2D(delta);
drawSceneBox2D();
if(DRAW_BOX2D_DEBUG){
  box2dCam.update();
  debugRenderer.render(world, box2dCam.combined);
}
```

In the `updateSceneBox2D` method, we find the `deltaPosition` value based on the change in the position of `box2dCam`. Although the code to move the terrain graphic remains the same, as follows:

```
deltaPosition=(box2dCam.position.x-previousCamXPos)*BOX2D_TO_CAMERA;
previousCamXPos=box2dCam.position.x;
...
world.step(deltaTime, 8, 3);

box2dCam.position.x=planeBody.getPosition().x+1.94f;
```

```
terrainBodyUp.setTransform(box2dCam.position.x+0.04f, 4.45f, 0);
terrainBodyDown.setTransform(box2dCam.position.x+0.04f, 0.35f, 0);
```

We are moving the `box2dCam` instance along with the plane to get the same display as our original game. We need to keep the rigid bodies of the top and bottom terrain always within the scene, or else we will need to create them very wide or multiple times. So, we also move them along with the camera. There are changes to the code that detects tap and applies force, as shown in the following code:

```
if(Gdx.input.justTouched()){
  ...
  if(fuelCount>0){
    touchPosition.set(Gdx.input.getX(),Gdx.input.getY(),0);
    touchPositionBox2D.set(touchPosition);
    box2dCam.unproject(touchPositionBox2D);
    tmpVector.set(planeBody.getPosition());
    tmpVector.sub( touchPositionBox2D.x, touchPositionBox2D.y).nor();

    tmpVector.scl(TOUCH_IMPULSE - MathUtils.clamp       (2.0f*Vector2.
dst (touchPositionBox2D.x,         touchPositionBox2D.y, planeBody.
getPosition().x,       planeBody.getPosition().y), 0.0f, TOUCH_
IMPULSE));
    planeBody.applyLinearImpulse(tmpVector, planeBody.getPosition(),
true);
    tapDrawTime=TAP_DRAW_TIME_MAX;
    camera.unproject(touchPosition);
  }

}
```

A tap needs to be detected and converted based on `box2dCam` and not the game camera.

Creating obstacles

We need to change the way the pillars and meteors are added. The terrains rigid bodies are already added. See how the `launchMeteor` function is altered to use Box2D:

```
private void launchMeteor() {
  nextMeteorIn=1.5f+(float)Math.random()*5;
  if(meteorInScene){
    return;
  }
  tmpVector.set(box2dCam.position.x+4.2,0);
```

```
    if(game.soundEnabled)spawnSound.play(game.soundVolume);
    meteorInScene=true;
    tmpVector.y=(float)(80+Math.random()*320)/BOX2D_TO_CAMERA;
    meteorBody.setTransform(tmpVector,0);
    Vector2 destination=new Vector2();
    destination.x=box2dCam.position.x-4.2;
    destination.y=(float) (80+Math.random()*320)/BOX2D_TO_CAMERA;
    destination.sub(tmpVector).nor();
    destination.scl(METEOR_SPEED);
    meteorBody.setLinearVelocity(destination);
  }
```

The `setTransform` method of a body sets a new position and rotation without affecting physics. Positioning is done based on the *x* position of the `box2DCam` instance, which remains at the center of the viewable scene. Pillar rocks are added in a similar manner using a vertex-based shape that helps us to create a triangular rigid body to match with the shape of the rock. Refer to the following code:

```
  private void addPillar() {
    if(pillars.size==0){
      tmpVector.x=(float) (800 + Math.random()*400);
    }else{
      tmpVector.x=lastPillarBody.getPosition().x*BOX2D_TO_CAMERA+(float)
  (600 + Math.random()*400);
    }
    Body pillar;
    if(MathUtils.randomBoolean()){
      pillar=createPillarBody(pillarUp, new     Vector2(tmpVector.
  x+pillarUp.getRegionWidth()/2,   pillarUp.
  getRegionHeight()/2),BodyType.StaticBody);
    }else{
      pillar=createPillarBody(pillarDown, new   Vector2(tmpVector.
  x+pillarDown.getRegionWidth()/2, 480-  pillarDown.
  getRegionHeight()/2),BodyType.StaticBody);
    }
    lastPillarBody=pillar;
    pillars.add(pillar);
  }
  private Body createPillarBody(TextureRegion region, Vector2 position,
  BodyType bodyType) {
    BodyDef boxBodyDef = new BodyDef();
    boxBodyDef.type = bodyType;
    boxBodyDef.position.x = position.x/BOX2D_TO_CAMERA;
    boxBodyDef.position.y = position.y/BOX2D_TO_CAMERA;
    Body boxBody = world.createBody(boxBodyDef);
```

```
    PolygonShape trianglePoly = new PolygonShape();

    if(region == pillarUp){
      float[] vertices = {-.54f, -1.195f, .11f, 1.195f, .54f, -1.195f};
      trianglePoly.set(vertices);
    }else{
      float[] vertices = {-.54f, 1.195f, .54f, 1.195f, .11f, -1.195f};
      trianglePoly.set(vertices);
    }
    boxBody.createFixture(trianglePoly, 1);
    trianglePoly.dispose();
    boxBody.setUserData(region);
    return boxBody;
  }
```

The vertices are provided in Box2D units, where (0,0) is the center of the body that falls at the center of the triangle and not in a corner. To find the vertices, we need to add or subtract *width/2* or *height/2* in Box2D units. After adding all this, you can play the game by setting the scene to our new class in `MenuScene`. Interestingly, you will see everything in the debug renderer but no other game graphics. Also, our plane body will collide with everything in the scene to bounce off and rotate, which may not be an expected behavior for our game.

Drawing the scene

For most of the rigid bodies, we store the related texture in the `userData` method. So, rendering them is easy, as the the following code in the `drawSceneBox2D` method shows:

```
for(Body vec: pillars) {
  tmpVector.set(vec.getPosition());
  tmpVector.scl(BOX2D_TO_CAMERA);
  tmpVector.x-=(box2dCam.position.x-4)*BOX2D_TO_CAMERA;
  toDraw=(TextureRegion) vec.getUserData();
  batch.draw(toDraw, tmpVector.x-toDraw.getRegionWidth()/2,
tmpVector.y-  toDraw.getRegionHeight()/2);
}
planePosition=planeBody.getPosition();
planePosition.scl(BOX2D_TO_CAMERA);
smoke.setPosition(planePosition.x+20-(box2dCam.position.x-4)*BOX2D_TO_
CAMERA-44, planePosition.y-7);
smoke.draw(batch);
batch.draw(plane.getKeyFrame(planeAnimTime), planePosition.x-
(box2dCam.position.x-4)*BOX2D_TO_CAMERA-44, planePosition.y-36.5f);
if(shieldCount>0){
```

```
  batch.draw(shield.getKeyFrame(planeAnimTime), planePosition.x-20-
(box2dCam.position.x-4)*BOX2D_TO_CAMERA-44, planePosition.y-36.5f);
  font.draw(batch, ""+((int)shieldCount), 390, 450);
}
if(meteorInScene){
  batch.draw(selectedMeteorTexture, meteorPosition.x-
(box2dCam.position.x-4)*BOX2D_TO_CAMERA-selectedMeteorTexture.
getRegionWidth()/2, meteorPosition.y-selectedMeteorTexture.
getRegionHeight()/2);
}
```

We need to offset by half of the TextureRegion instances' width and height as the position returned using a rigid body's `getPosition` falls in the middle of the body.

Note that `box2dCam` is set at 4 units in *x* and `box2dCam.position.x-4` is 0 in *x*.

Handling collisions

By default, the Box2D world is simulating the right response for our game at this point. The plane will bounce and tumble all over the place if it collides with meteor rocks or pillars. Such a response would have been excellent if our game was *Angry Birds* or *Cut the Rope*. If we think about it such a response is fine as our game will end on such a collision. In order to detect collisions, we will use `ContactListener`. In the `initPhysics` method, we add the code in the `endContact` method as follows:

```
world.setContactListener(new ContactListener() {
  @Override
  public void endContact (Contact contact) {
    bodyA=contact.getFixtureA().getBody();
    bodyB=contact.getFixtureB().getBody();
    boolean planeFound=false;
    if(bodyA.equals(planeBody)){
      planeFound=true;
      unknownBody=bodyB;
    }else if(bodyB.equals(planeBody)){
      planeFound=true;
      unknownBody=bodyA;
    }
    if(planeFound){
      ItemType itemType=getItemType(unknownBody);
      if(itemType==ItemType.Terrain){
```

```
            endGame();
          }else if(shieldCount<=0 && (itemType==ItemType.Meteor||
    itemType==ItemType.Pillar)){
            endGame();
          }else if(itemType==ItemType.Pickup){
            pickIt((PickupBox2D) unknownBody.getUserData());
          }
        }
      }
      ...
    });
```

This will end the game with any collision involving the plane, even those with pickups if you implemented them. This is shown in the following screenshot:

To restart the game, we will need another function to reset the physics-related values as well, which gets called from the `resetScene` function when the game is over. Refer to the following code:

```
private void resetPhysics() {
  for(Body vec: pillars) {
    world.destroyBody(vec);
  }
```

```
    pillars.clear();
    tmpVector.set(800,500);
    meteorBody.setTransform(tmpVector,0);
    tmpVector.set(planePosition);
    planeBody.setTransform(tmpVector.x/BOX2D_TO_CAMERA,   tmpVector.y/
  BOX2D_TO_CAMERA, 0);
    planeBody.setAwake(true);
    box2dCam.position.set(4.0, 2.4, 0);
    previousCamXPos=4;

    terrainBodyUp.setTransform(box2dCam.position.x+0.04f, 4.45f,   0);
    terrainBodyDown.setTransform(box2dCam.position.x+0.04f, 0.35f,   0);
    lastPillarBody=null;
    addPillar();
  }
```

Collision filtering is a complicated feature, using which you can set collisions between specific items only. You can find out more on this topic at `http://www.aurelienribon.com/blog/2011/07/box2d-tutorial-collision-filtering/`.

Ignoring collisions with shield

Our game logic demands that when the plane has shield, it should be invincible against pillars and meteor rocks. In such a case, we will need to ignore those collisions by detecting them before the actual collision happens. This can be done using the `preSolve` method in the `ContactListener` event. Refer to the following code:

```
  public void preSolve (Contact contact, Manifold oldManifold) {
    bodyA=contact.getFixtureA().getBody();
    bodyB=contact.getFixtureB().getBody();
    boolean planeFound=false;
    if(bodyA.equals(planeBody)){
      planeFound=true;
      unknownBody=bodyB;
    }else if(bodyB.equals(planeBody)){
      planeFound=true;
      unknownBody=bodyA;
    }
    if(planeFound){
      ItemType itemType=getItemType(unknownBody);
      if(shieldCount>0 && (itemType==ItemType.Meteor||
  itemType==ItemType.Pillar)){
```

```
        contact.setEnabled(false);
      }else if(itemType==ItemType.Pickup){
        contact.setEnabled(false);
      }
    }
  }
```

Collision for pickups

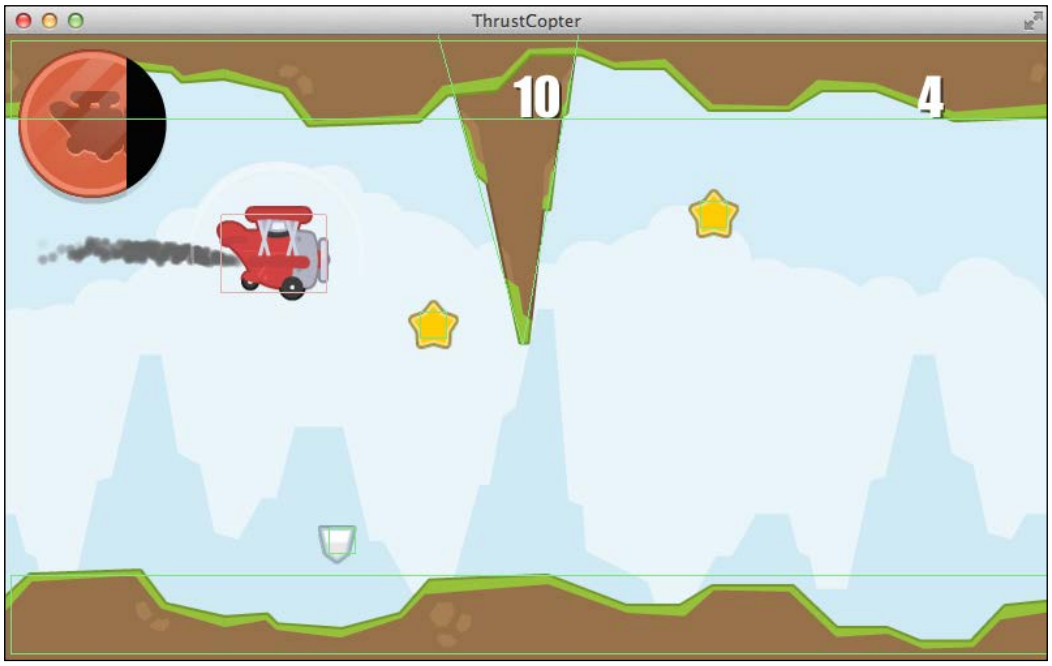
The previous code also disabled collision with pickups, as we don't want the plane to bounce off pickups. Even though the collisions are disabled, the `endContact` method will still be called and to collect the pickup we call the `pickIt` function. One of the changes in `addPickup` is the way we detect whether the new pickup will overlap a pillar. We use a `QueryAABB` method of the `world` instance, which finds out any overlapping fixtures based on the provided rectangular area. Refer to the following code:

```
testPoint.x=box2dCam.position.x+4.2f;
testPoint.y=(float)(80+Math.random()*320)/BOX2D_TO_CAMERA;

hitBody = null;
world.QueryAABB(callback, testPoint.x - 1.9f, testPoint.y - 1.9f,
testPoint.x + 1.9f, testPoint.y + 1.9f);

if (hitBody != null) {
  return false;
}
```

The `QueryAABB` method is actually used to select items using touch or a mouse, but serves our purpose just as well. Refer to the following code:

```
Vector3 testPoint = new Vector3();
  QueryCallback callback = new QueryCallback() {
    @Override
    public boolean reportFixture (Fixture fixture) {
      if (fixture.testPoint(testPoint.x, testPoint.y)) {
        hitBody = fixture.getBody();
        return false;
      } else
        return true;
    }
  };
```

The final output can be seen in the following screenshot:



Please go through the code to get the complete picture of the implementation. Note that this is not as polished as our original game, but is a good start nonetheless. You may struggle with the various values used while converting units for rendering. Most of those values would be the size of the textures either used directly or converted to Box2D units. Also, do not forget to disable debug rendering when releasing your game.

Summary

Box2D is an immensely capable library and can be used to implement very complex physics simulations. In this chapter, we had an overview of Box2D and discussed the different rigid bodies. We also converted our game to use Box2D physics for gameplay mechanics. We successfully rendered the Box2D world using `Box2DdebugRenderer`. The importance of fixing our time step was also explained. You also learned to use `ContactListener` to detect collisions and how to disable collisions using the `preSolve` method.

In the next chapter, we will explore the third dimension with LibGDX 3D and learn to use bullet physics.

7
The Amazing World of 3D

What would be better than a 2D game? A 3D game of course! LibGDX is famous for its 2D capabilities, but it also packs an arsenal of 3D features. Creating simple 3D games is easy using LibGDX, but creating a feature-rich 3D game can be a very complex process. We do not need help with writing game logic as it remains the same, but I will be explaining the concepts of 3D in this chapter. In this chapter, we will explore LibGDX 3D and cover the following topics:

- Exploring 3D in LibGDX
- Introducing the `PerspectiveCamera`, `Environment`, `ModelBatch`, `Model`, and `ModelInstance` classes
- Creating and rendering 3D primitives
- Loading 3D models
- Using the `fbx-conv` application to convert to the `.g3db` or `.g3dj` formats
- Loading and rendering animated 3D models
- Animating via code
- Interacting with 3D objects
- Creating 3D particle effects using Flame
- Learning bullet physics

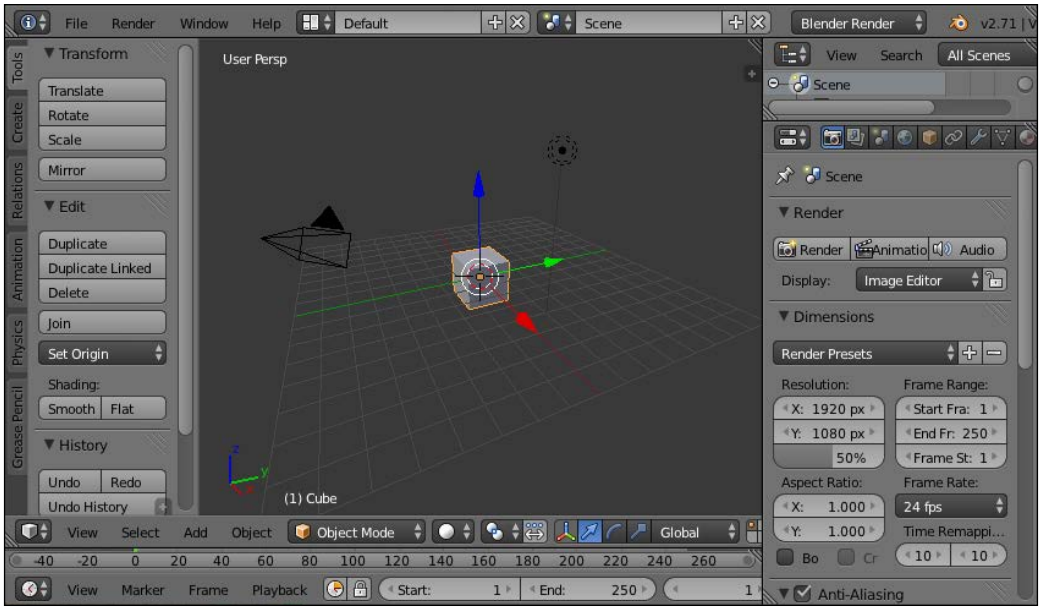
I would like you to try to convert our game to 3D after learning the concepts in this chapter and combining that with the game logic from previous chapters. The source files for this chapter and the 3D files can be found in the `chapter7.zip` file provided along with the book.

Introducing the third dimension

Before we start exploring 3D features of LibGDX, let's learn some basics of 3D and how 3D content can be created. The difference between 2D and 3D is the addition of a new axis—the *z* axis. We are familiar with this newcomer after using the `Vector3` class. The *z* axis adds depth to the scene in addition to width and height, and it can be considered perpendicular to the screen.

Creating 3D content

This may not be something a programmer will be doing, but it is always good to know how to create basic 3D content. There are many applications out there that can help us create 3D content, but one name that stands apart is **Blender**. Blender is an incredible tool to create, texture, rig, animate, and composite 3D content. As a matter of fact, it even has its own game engine that runs on Python. The icing on the cake is that this incredibly powerful application is **free**. The UI of Blender is shown in the following screenshot:



You can download Blender from `www.blender.org`.

Although Blender can seem very intimidating to start with, download it and try to learn more about it. While creating your content in Blender, remember the following points:

- Do not create any more vertices than absolutely necessary (low poly modeling)
- Triangulate your mesh
- Create your models in the right size
- Recalculate normals outside once you finish modeling
- Apply textures to the faces using UV unwrapping or multiface texturing
- Use **Power of Two** (**POT**) textures (for example, 64 x 64, 128 x 512, and so on)
- The animation key frame interpolation should be linear
- Bake all animations into vertex animations
- Export only the required models in the FBX format

We can also export in the `.obj` format, but the `.obj` loader in LibGDX is experimental. Depending on the modeling tool you use, there can be many issues, including flipped axes, flipped normals, missing textures, missing animations, and so on. You will need to work closely with the 3D artist to come up with a schema that works perfectly.

The PerspectiveCamera class

In order to perceive the depth, we need to use a different kind of camera—the `PerspectiveCamera` class. A perspective camera is set up as follows:

```
    camera = new PerspectiveCamera(70, Gdx.graphics.getWidth(), Gdx.
  graphics.getHeight());
    camera.position.set(0f, 4f, 3f);
    camera.lookAt(0, 0, 0);
    camera.near = 0.1f;
    camera.far = 300f;
    camera.update();
```

The `PerspectiveCamera` class takes in the field of view and size of the area as parameters and has a convenient `lookAt` method to point towards the correct direction. Only items within the `near` and `far` attributes will be rendered by the camera.

Converting 3D files to G3DB

LibGDX supports a `.g3db` file format, which is an optimized binary file format that works with LibGDX efficiently. We can convert the `.obj` or `.fbx` formats to `.g3db` using the `fbx-conv` application that is available at `https://github.com/libgdx/fbx-conv`. The binary can be downloaded directly from `http://libgdx.badlogicgames.com/fbx-conv/`. We need to run the application via the command line as follows:

- For Windows, use the following command:

  `fbx-conv-win32.exe [options] <input> [<output>]`

- For Linux, use the following command:

  `fbx-conv-lin64 [options] <input> [<output>]`

- For Mac, use the following command:

  `fbx-conv-mac [options] <input> [<output>]`

I have provided the `planeanim1.fbx` file that you can try to convert. It has the animated plane 3D mesh:

```
fbx-conv-mac -f planeanim1.fbx
```

LibGDX also supports a JSON-based format, `.g3dj`. This format can be used for debugging purposes, as we can open it up using any text editor to explore the details:

```
fbx-conv-mac -f -o G3DJ planeanim1.fbx
```

Playing with primitives

3D content is created or loaded into LibGDX as a `Model` class instance. The `Model` class has the mesh, texture, and animation data in the form of nodes arranged in a hierarchy. A `Model` class is rendered by creating a `ModelInstance` class. Let's create a new project or a new package within our existing project to play with 3D. I am using `com.csharks.thrustcopter.thirdDimension.Sample3D` for 3D experiments. We can create many 3D primitives in LibGDX, for example, a box, sphere, cone, cylinder, capsule, and arrow. To create a `Model` class in code, we will need the help of the `ModelBuilder` class. The following code creates a box of size (1,1,1):

```
ModelBuilder modelBuilder = new ModelBuilder();
Model model = modelBuilder.createBox(1f, 1f, 1f,
  new Material(ColorAttribute.createDiffuse(Color.GREEN)),
  Usage.Position | Usage.Normal);
ModelInstance instance=new ModelInstance(model);
```

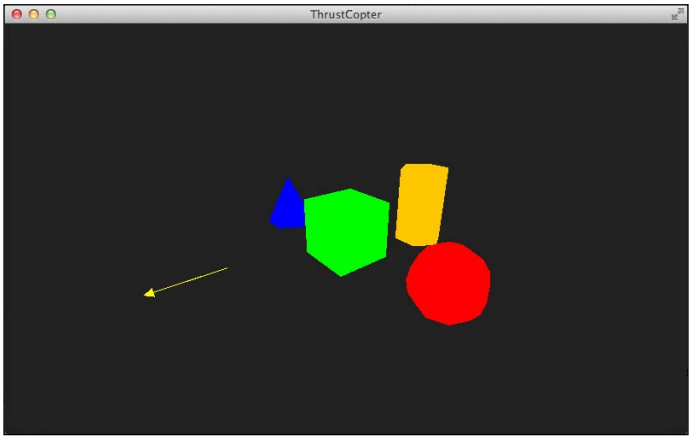
We assign a `Material` instance with a diffuse `ColorAttribute` instance of green. The `Material` class describes how the model should be rendered and provides attributes to the shader. The `ModelBuilder` class has other convenient methods such as `createCylinder`, `createSphere`, `createArrow`, `createCone`, and `createCapsule` that are evident. The `Usage.Position` attribute defines the position of our primitive and the `Usage.Normal` attribute defines normals so that lights can be effectively applied. Remember that all models should be disposed after use.

Rendering the ModelInstance classes

Remember how we used `SpriteBatch` to efficiently render our 2D content? We will use its 3D cousin `ModelBatch` to render the 3D `ModelInstance` classes. Their usage is very similar:

```
modelBatch = new ModelBatch(new DefaultShaderProvider());
...
public void render(float delta) {
  Gdx.gl.glViewport(0, 0, Gdx.graphics.getWidth(),
    Gdx.graphics.getHeight());
  Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT |
    GL20.GL_DEPTH_BUFFER_BIT);
  Gdx.gl.glClearColor(0.13f, 0.13f, 0.13f, 1);
  modelBatch.begin(camera);
  modelBatch.render(instance);
  modelBatch.end();
  super.render(delta);
}
```

When the class is rendered, you will find that there is no shading applied and everything looks plain, as shown in the following screenshot:



We need to add an environment consisting of the `DirectionalLight` and `AmbientLight` classes so that the proper shading is applied and the content looks more 3D than 2D:

```
environment = new Environment();
environment.set(new ColorAttribute(ColorAttribute.AmbientLight,
0.4f, 0.4f, 0.4f, 1f));
environment.add(new DirectionalLight().set(0.8f, 0.8f, 0.8f, -
0.8f, 0.3f, -1f));
```

In the `render` method, insert the following line of code:

```
modelBatch.render(instance, environment);
```

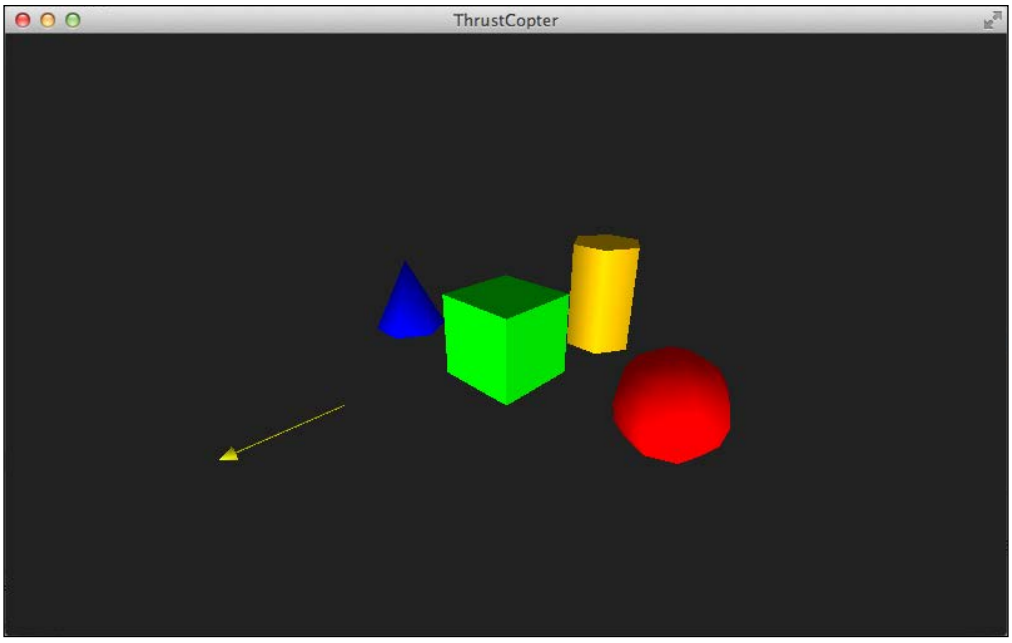
The `DirectionalLight` class takes a color and direction as parameters. In order to explore the 3D world, we can take the help of `CameraInputController` that enables us to touch and drag objects in the 3D world:

```
cameraController = new CameraInputController(camera);
Gdx.input.setInputProcessor(cameraController);
```

In the `render` method, insert the following line of code:

```
cameraController.update();
```

Check out the `Sample3D` source file to see the code in action:



Loading 3D models

We can use `AssetManager` to load the `.g3db` or `.g3dj` files using the following code:

```
game.manager.load("planeanim1.g3db", Model.class);
game.manager.finishLoading();
Model model = game.manager.get("planeanim1.g3db", Model.class);
ModelInstance plane = new ModelInstance(model);
plane.transform.setToTranslation(0, 0, -1f);
```

To position the plane, we use the `setToTranslation` method of the `ModelInstance` transform.

Playing animations

Our plane's 3D model has an animation where the front propeller rotates. Each animation is identified by a name or ID. When in doubt, we can open up the `.g3dj` file to check the name in the animation section at the bottom of the file. For the plane, the animation name is `Scene`, which is something Blender applied automatically. We will use `AnimationController` to play animations:

```
controller = new AnimationController(plane);
controller.setAnimation("Scene",-1);
```

In the `render` method, insert the following line of code:

```
controller.update(delta);
```

Providing a value of `-1` in the `setAnimation` method gets the animation to loop infinitely, which is the case with our propeller animation. When we need to do something after an animation is played, we will use an `AnimationListener` event as follows:

```
controller.setAnimation("Scene",2, new AnimationListener() {
  @Override
  public void onLoop(AnimationDesc animation) {
  }
  @Override
  public void onEnd(AnimationDesc animation) {
  }
});
```

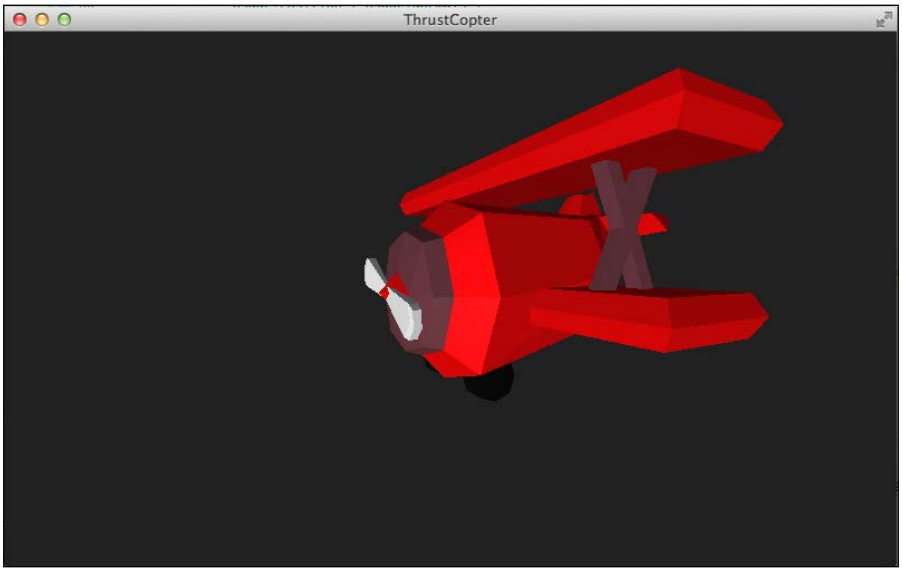
We can play animations one after the other using queuing:

```
controller.queue(id, loopCount, speed, listener, transitionTime);
```

Animations can be blended together over a time for a smooth transition using the `animate` method:

```
controller.animate(id, transitionTime)
```

To move, scale, or rotate a `ModelInstance` class with code, we just need to set its transform matrix values in the `render` method. We can use the `setToTranslation`, `setToRotation`, `setToScaling`, `rotate`, `scale`, and `translate` methods for this purpose. Check out the `Sample3D` source file to see the code in action and the screenshot can be seen as follows:



Interacting with 3D objects

Let's discuss how to select 3D objects present in a scene. We will add 10 planes to the scene, store them in an array named `instances`, and select any of them by tapping. The selected plane will be set to rotate until another selection is made. We will add an `InputAdapter` class's instance named `myAdapter` and wire it to accept inputs via an `InputMultiplexer` interface, as we also need our `cameraController` instance to receive inputs. The code is as follows:

```
InputAdapter myAdapter=new InputAdapter(){
  @Override
  public boolean touchDown (int screenX, int screenY, int pointer, int
button) {
    selecting = getObject(screenX, screenY);
    return selecting >= 0;
```

```
    }
    @Override
    public boolean touchDragged (int screenX, int screenY, int pointer)
{
      return selecting >= 0;
    }
    @Override
    public boolean touchUp (int screenX, int screenY, int pointer, int
button)   {
      if (selecting >= 0) {
        if (selecting == getObject(screenX, screenY))
          setSelected(selecting);
        selecting = -1;
        return true;
      }
      return false;
    }
};
Gdx.input.setInputProcessor(new InputMultiplexer(myAdapter,
cameraController));
```

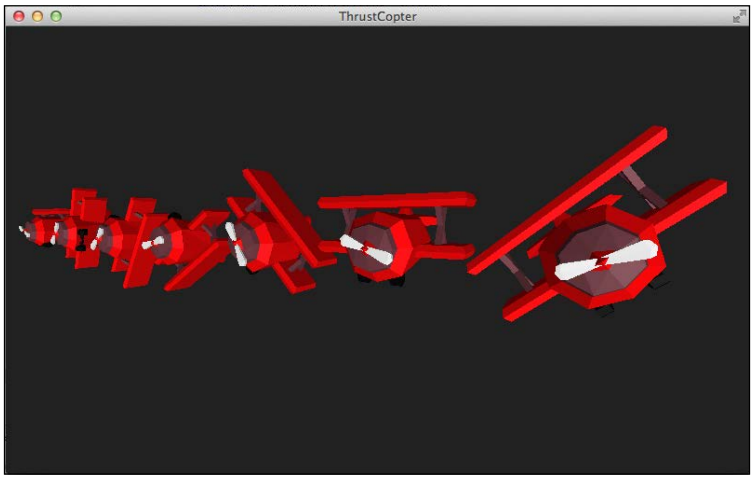
The `getObject` function is where the action happens and we use ray picking to select the 3D content that intersects with the ray. We also choose the object that is closest to the camera as follows, as many objects may fall in the line of sight of the ray:

```
public int getObject (int screenX, int screenY) {
  Ray ray = camera.getPickRay(screenX, screenY);
  int result = -1;
  float distance = -1;
  for (int i = 0; i < instances.size; ++i) {
    final ModelInstance instance = instances.get(i);
    instance.transform.getTranslation(position);
    position.add(center);
    float dist2 = ray.origin.dst2(position);
    if (distance >= 0f && dist2 > distance) continue;
    if (Intersector.intersectRaySphere(ray, position, radius, null)) {
      result = i;
      distance = dist2;
    }
  }
  return result;
}
```

The `Intersector` helper class also has methods to look for an intersection with a bounding box (`intersectRayBoundsFast`), but we are looking for a sphere around our plane. The radius of the sphere is calculated to contain the maximum width of the bounding box even when our plane is rotated. The bounding box, the bounding box center, and the radius of the bounding sphere are calculated as follows:

```
plane.calculateBoundingBox(bounds);
center.set(bounds.getCenter());
dimensions.set(bounds.getDimensions());
radius = dimensions.len() / 2f;
```

Check out the `Interaction3D` source file for this experiment, as shown in the following screenshot:



Most of the 3D experiments in LibGDX and their explanations are done by Xoppa. Do check out the blog at `http://blog.xoppa.com/`.

3D frustum culling in LibGDX

In a complex 3D game, there can be many objects that need to be added to the scene. 3D operations are very resource-hungry and unless managed properly, this can affect the performance of the game. Using frustum culling, we can easily limit the number of items being rendered by the camera. We only need to render those objects that fall within the field of view of the camera. By adding an `isVisible` function, we can check each `ModelInstance` class before rendering. The code is as follows:

```
private boolean isVisible(ModelInstance instance) {
  instance.transform.getTranslation(position);
```

```
    return camera.frustum.pointInFrustum(position);
}
```

This checks whether the center of the 3D object falls within the camera frustum, but this can be inaccurate for objects with volume. Alternatively, we can check against the bounding box of the 3D object as shown in the following code:

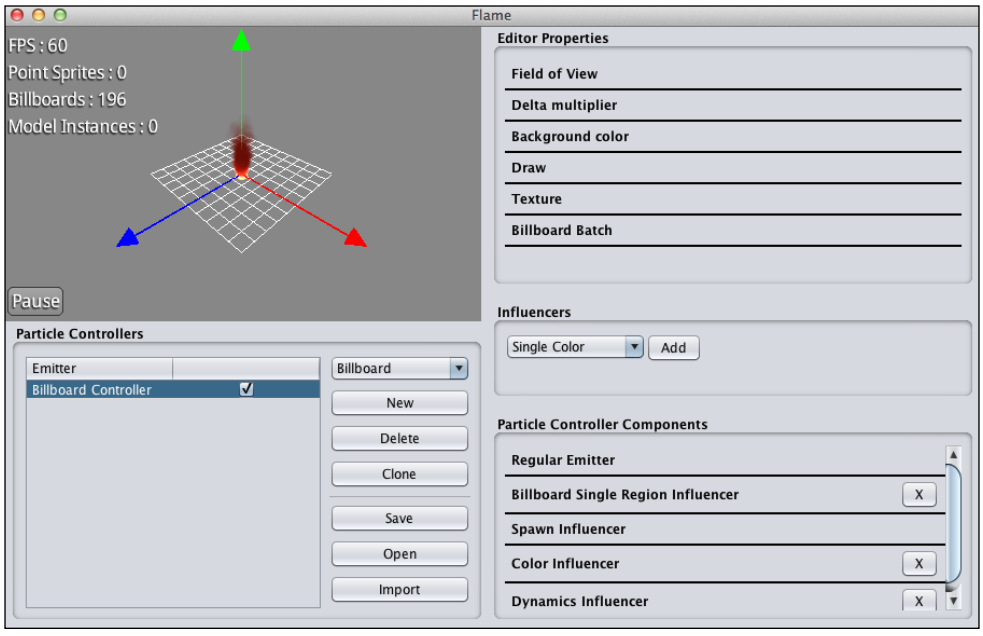
```
position.add(center);
return camera.frustum.boundsInFrustum(position, dimensions);
```

Such a calculation can again return wrong values if we rotate the object, as the bounding boxes are not rotated. In such cases, it is safe to look for the bounding sphere. The code is as follows:

```
return camera.frustum.sphereInFrustum(position, radius);
```

3D particles with Flame

Remember the 2D particle editor? 2D particles are not suited for 3D, as they fail when dealing with perspective and depth. Hence, we have their 3D big brother along with a GUI editor named **Flame**. Flame can be run from the `gdx-tools` project. Explaining the editor is out of scope of this book, but I am sure that from your experience with the LibGDX particle editor you will definitely figure it out. Make sure you are using LibGDX Version 1.3.0 or later for this feature, as shown in the following screenshot:



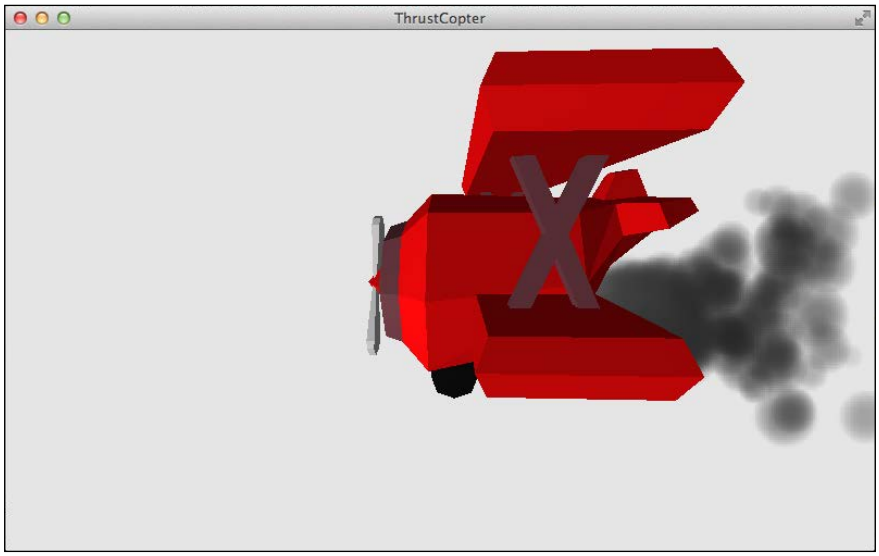
I created the trailing smoke particle effect and saved it in a file named `Smoke3D`. We need to copy the trailing smoke particle effect and the `pre_particle.png` file to our `assets` folder before adding the smoke to our plane. Make sure that you import the right package, as many names are the same for the 2D and 3D particle classes. The code is as follows:

```
particleSystem = ParticleSystem.get();
PointSpriteParticleBatch pointSpriteBatch = new
PointSpriteParticleBatch();
pointSpriteBatch.setCamera(camera);
particleSystem.add(pointSpriteBatch);

ParticleEffectLoader.ParticleEffectLoadParameter loadParam = new
ParticleEffectLoader.ParticleEffectLoadParameter(particleSystem.
getBatches());
ParticleEffectLoader loader = new ParticleEffectLoader(new
InternalFileHandleResolver());
game.manager.setLoader(ParticleEffect.class, loader);
game.manager.load("Smoke3D", ParticleEffect.class, loadParam);
game.manager.finishLoading();

ParticleEffect originalEffect = game.manager.get("Smoke3D");
effect = originalEffect.copy();
effect.init();
effect.start();
particleSystem.add(effect);
```

Creating the 3D particle effect requires a lot of work. We are creating separate batches to handle these particles as we need to get maximum performance. The effect needs to be copied and cannot be used directly. To render the effect, use the following code:

```
particleSystem.update();
particleSystem.begin();
particleSystem.draw();
particleSystem.end();
modelBatch.render(particleSystem);
```

Check out the source file for this experiment, `SmokingPlane` and the screenshot can be seen as follows:



Using bullet physics

The `Bullet` class is a 3D collision detection and physics library that is similar to Box2D (`http://bulletphysics.org/mediawiki-1.5.8/index.php/Main_Page`). LibGDX has a wrapper for `Bullet`, but it has a steep learning curve due to the JavaCPP bridge. Please check out the official `Bullet` manual at `https://github.com/erwincoumans/bullet2/blob/master/Bullet_User_Manual.pdf?raw=true`. The `Bullet` package is an extension and needs to be added manually. Follow the same process that you followed for Box2D with the following details, and refresh the Gradle dependencies.

Core dependency:

```
compile "com.badlogicgames.gdx:gdx-bullet:$gdxVersion"
```

Desktop dependency:

```
compile "com.badlogicgames.gdx:gdx-bullet-platform:
$gdxVersion:natives-desktop"
```

Android dependency:

```
compile "com.badlogicgames.gdx:gdx-bullet:$gdxVersion"
natives "com.badlogicgames.gdx:gdx-bullet-platform:
$gdxVersion:natives-armeabi"
natives "com.badlogicgames.gdx:gdx-bullet-platform:
$gdxVersion:natives-armeabi-v7a"
natives "com.badlogicgames.gdx:gdx-bullet-platform:
$gdxVersion:natives-x86"
```

iOS dependency:

```
compile "com.badlogicgames.gdx:gdx-bullet:$gdxVersion"
natives "com.badlogicgames.gdx:gdx-bullet-platform:
$gdxVersion:natives-ios"
```

There is no HTML dependency as `Bullet` is not yet compatible with the HTML build. I will try to introduce the core functionality of `Bullet`, which is within the scope of our book. There are LibGDX books out there that cover `Bullet` in much more detail, but that is for expert developers.

Check out Xoppa's blog on adding bullet physics for more details at `http://blog.xoppa.com/using-the-libgdx-3d-physics-bullet-wrapper-part1/`.

Creating the bullet world

Most of the steps required to set up rigid body collisions in `Bullet` is similar to the steps in Box2D, but it has a few additional steps as we are dealing with an additional dimension—3D physics calculations are much more complex. We need to start `Bullet` with `Bullet.init` before we initialize any of the related variables. Creating the bullet world requires four variables. We need a collision configuration to configure the bullet collision detection, a collision dispatcher to handle near phase collision detection, a broad phase detection helper, and a constraint solver that deals with connected bodies, as shown in the following code:

```
Bullet.init();
collisionConfiguration = new btDefaultCollisionConfiguration();
dispatcher = new btCollisionDispatcher(collisionConfiguration);
broadphase = new btDbvtBroadphase();
solver = new btSequentialImpulseConstraintSolver();
world = new btDiscreteDynamicsWorld(dispatcher, broadphase,
solver, collisionConfiguration);
world.setGravity(new Vector3(0, -9.81f, 1f));
```

In the preceding code, we are only using the default values for simplicity, but remember to dispose them.

Adding rigid bodies

The `Bullet` class has rigid body dynamics as well as soft body dynamics. As in Box2D, we have three kinds of rigid bodies—static, dynamic, and kinematic. We create a static or kinematic rigid body by providing a mass equal to zero. Dynamic rigid bodies require a `MotionState` class that optimizes physics calculations using call callbacks when a change happens. All rigid bodies require a collision shape relevant to the shape of the object. There are many types of collision shapes, such as box, sphere, cone, cylinder, capsule, and so on. The following code creates a primitive 3D box and adds a box-shaped rigid body to the `Bullet` world:

```
Model box = modelBuilder.createBox(1f, 1f, 1f, new
Material(ColorAttribute.createDiffuse(Color.BLUE)), Usage.Position |
Usage.Normal);
ModelInstance boxInstance=new ModelInstance(box);
btDefaultMotionState motionState = new btDefaultMotionState
(boxInstance.transform);
motionState.setWorldTransform(boxInstance.transform.trn(0, 0, 0));
btCollisionShape boxshape = new btBoxShape(new Vector3(0.5f, 0.5f,
0.5f));
btRigidBody boxbody = new btRigidBody(1, motionState, boxshape);
world.addRigidBody(boxbody);
```

We have assigned a mass of 1 kg to our box. We need to step the physics simulation as we did in the case of Box2D. Hence, we call the following line of code in the `render` method:

```
world.stepSimulation(Gdx.graphics.getDeltaTime(), 5);
```

Once the physics is calculated, we need to synchronize the `motionState` instance with the `ModelInstance` transform using the following code:

```
motionState.getWorldTransform(boxInstance.transform);
```

To simulate more accurate collisions, we may need to create collision shapes that are very similar to our 3D mesh. Check out the source file to see how the `convexHullShape` instance is used for our plane in the twin stacks experiment file named `BulletSample`. We apply an impulse to the plane to make it fly using the following code:

```
planebody.applyCentralImpulse(new Vector3(0,0,-65));
```

Please do remember to dispose everything you create, since `Bullet` is C code, which doesn't get help from a garbage collector.

Collision detection

To detect collisions, we need to extend `ContactListener` and override the relevant function. We can store and compare the `userData` values to detect which object has collided, as we did in Box2D:

```
public static class MyContactListener extends ContactListener {
  @Override
  public void onContactStarted (btCollisionObject colObj0,
btCollisionObject   colObj1) {
    if(colObj0.userData=="plane" || colObj1.userData=="plane"){
      Gdx.app.log("ContactCallback", "Plane Collides" );
    }
  }
}
contactListener = new MyContactListener();
```

Adding shadows

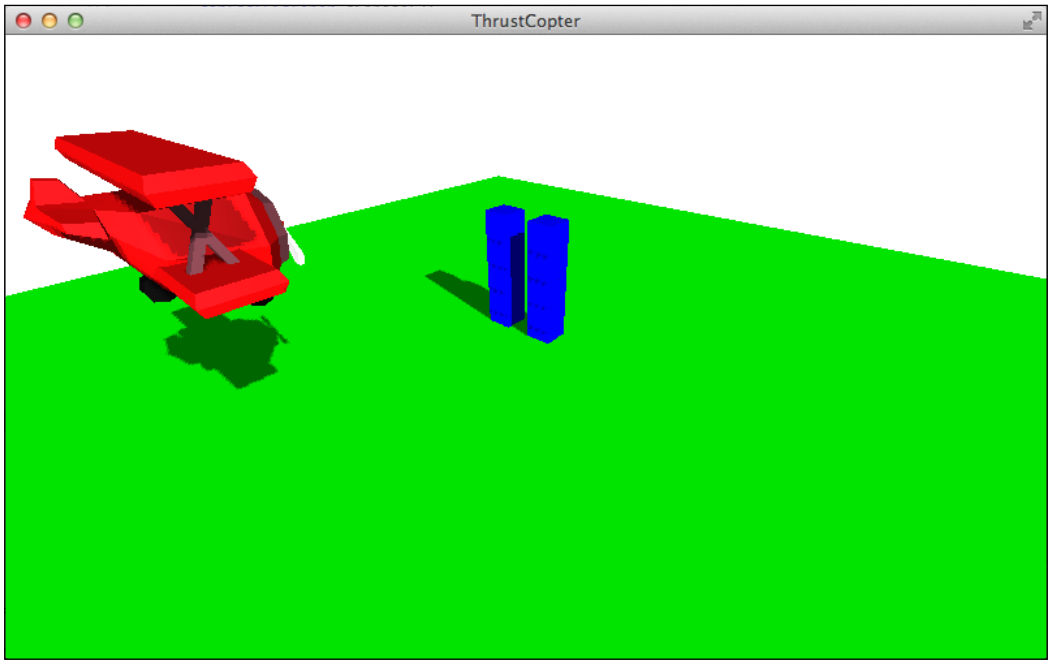
Shadows play a very important role in 3D simulations, as they add depth and realism. The implementation of shadows in LibGDX is experimental and may still be evolving, but it is very easy to add. We need a `DirectionalShadowLight` class and an additional `ModelBatch` class to add shadows as follows:

```
shadowLight = new DirectionalShadowLight(1024, 1024, 60, 60, 1f, 300);
shadowLight.set(0.8f, 0.8f, 0.8f, -1f, -.8f, -.2f);
environment.add(shadowLight);
environment.shadowMap = shadowLight;
shadowBatch = new ModelBatch(new DepthShaderProvider());
```

In the `render` method, we need to draw the shadow before we draw the model instances using the following code:

```
shadowLight.begin(Vector3.Zero, camera.direction);
shadowBatch.begin(shadowLight.getCamera());
shadowBatch.render(instances);
shadowBatch.end();
shadowLight.end();
```

Check out the previous shared code where our plane crashes with two sets of stacked boxes. This is shown in the following screenshot:



We have just touched the tip of the iceberg with `Bullet`, but it is so powerful that even now many Hollywood movies use it for special effects.

Summary

This surely was one of the toughest chapters in this book, but it is most rewarding once you start using what you learned to implement the 3D functionalities. You learned how to create 3D primitives and load 3D models. We used the `fbx-conv` application to convert 3D models to the LibGDX-friendly `.g3db` format. You also learned about `ModelBatch`, `PerspectiveCamera`, `Environment`, and `ModelInstance` classes. Then, you learned about the 3D particle effects editor, Flame. Finally, you learned about bullet physics and created a simple rigid body dynamics simulation.

In the next chapter, you will learn to load TMX tile maps and implement data persistence.

8
Saving Our Data

Our game requires handling of data between the different play sessions. For example, we will need to save the scores and calculate a high-score list based on all the games played. We will also need to compare a new score with the old ones and save the high scores in the high-score list for future display. Similarly, we will also need to save the value of the sound volume and check whether we have set our game's volume to mute. In this chapter, we will explore the different ways in which we persist data, covering the following topics in the process:

- Exploring `Preferences` in LibGDX
- Using `Preferences` to save and load the sound status and sound volume
- Learning to access files in the internal, local, and external storage
- Saving local high scores
- Creating a new scene to display the leaderboard
- Creating tile map-based levels using the TMX tile map parser
- Using the Tiled tool to create a tile-based level map

We will not make any changes to the game logic in this chapter. Also, we will not use the Box2D-based game logic to proceed further. The source files for this chapter can be found in the `chapter8.zip` file provided along with the book.

Persisting game preferences

The easiest way to save and load persisting data is to use `Preferences`. This can be used when the data to be saved is small, for example, a few different variables, due to size limitations for this mode. On desktops, `Preferences` are written onto an XML file saved in the home directory with the name specified while creating the `Preferences` file. On Android, it is saved as `SharedPreferences`, while on iOS an `NSMutableDictionary` class is written to file. We can have multiple `Preferences` files saved under different names.

The `Preferences` class is the only way to have persisting data while running on the browser, as the browser cannot get file access due to security reasons.

In order to access a saved `Preferences` value we can use the following code like it is used in `ThrustCopter.java` class:

```
protected Preferences getPrefs()
{
if(preferences==null){
  preferences = Gdx.app.getPreferences("ThrustCopter");
}
if(preferences==null){
  System.out.println("null preferences");
}
return preferences;
}
```

We can save `boolean`, `int`, `float`, `long`, `string`, and `map` via `preferences`.

Saving and loading sound preferences

We need to persist two of the variables related to handling sound in our game—`soundEnabled` and `soundVolume`. We will make use of some new functions in the `ThrustCopter` class to do this:

```
public void saveSoundStatus(){
  getPrefs().putBoolean( "soundstatus", soundEnabled );
}
public boolean loadSoundStatus( ){
  return getPrefs().getBoolean("soundstatus",true);
}
public void saveSoundVolume(){
  getPrefs().putFloat( "soundvolume", soundVolume );
}
public float loadSoundVolume( ){
  return getPrefs().getFloat("soundvolume",1.0f);
}
public void flushPref(){
   getPrefs().flush();
}
public void saveAll(){
  System.out.println("saving preferences");
  saveSoundVolume();
  saveSoundStatus();
  flushPref();
}
```

The Boolean value `soundEnabled` is saved using the `putBoolean` method and loaded using the `getBoolean` method. Similarly, the float equivalent of these methods are used for a float value `soundVolume`. Once the values for `preferences` are set via the corresponding `put` method, we need to call the `flush` method to save it to the device memory. In the `create` method of the `ThrustCopter` class, we use the following code to get the saved values of these variables:

```
soundEnabled=loadSoundStatus();
soundVolume=loadSoundVolume();
```

The `get` methods do have the provision to provide default values if there are no `preferences` saved. For `soundVolume`, we provide a default value of `1.0f` and `soundEnabled` is `TRUE` by default. When we access the sound options from the menu, we will be altering these values. Hence, we need to save the altered values while exiting the sound options display by using the following code:

```
game.saveAll();
```

This way, the next time we run our game, we will get the same sound status and sound volume that we had set earlier. Generally, `Preferences` are enough for our game's purpose, but we will need other alternatives for complex games.

Implementing a local leaderboard

A leaderboard shows a list of finite number of high scores of our game. A local leaderboard is limited to the device on which the game runs; hence, it lists only the scores achieved on that device. For such a system, we need to save the finite number of scores and compare any new score with them to decide whether to store the new score as part of the leaderboard. We may as well use `Preferences` for this, but I want to demonstrate how we can read and write files in LibGDX.

Filesystems and access permissions

LibGDX is cross-platform and has to deal with all kinds of filesystems across the different devices. On desktops, there are usually no restrictions on reading or writing files. For the web platform, file access is extremely restricted. The iOS platform behaves almost the same as the desktop platform, with access to internal, external, absolute and local file types. For Android, files can be saved within the APK file that has read-only permission. These files are exclusive to the application and other applications cannot access them. An Android application also has access to the internal storage that is exclusive to that particular application. Internal storage is a portion of the internal memory that is dedicated to the application where we have read and write access.

An Android application also has access to external storage that is not exclusive to the application. We have read and write permissions for external files, but these files can be accessed by other applications as well. Also, the external storage access requires explicit permissions that are set in the `AndroidManifest` file. It is not safe to save data in the external storage, as the user can always remove the external storage memory card, thereby denying access to those files.

Therefore, it is always better to check for the availability of the storage type via code before we actually read or write data to them. The following code checks the availability of the external storage:

```
Gdx.files.isExternalStorageAvailable();
```

Note that file access is prone to throw runtime exceptions as the file may not be present, or SD card may be removed, and so on.

We can check for the existence of the file using the following code:

```
Gdx.files.external("mydata.txt").exists();
```

Reading and writing files

We can read any file type if we know the access path by using a `FileHandle` class. The following code reads the `mydata.txt` file placed in the `assets` folder of our Android project. Such a file is packaged along with the APK and is therefore internal. Refer to the following code:

```
FileHandle file = Gdx.files.internal("mydata.txt");
String text = file.readString();
```

We cannot write to the internal storage, but we can write to external, local, and absolute storage. Writing a string to a file is easy, and it is shown in the following code:

```
FileHandle file = Gdx.files.local("mydata.txt");
file.writeString("some new data", false);
```

While writing the code, we can specify whether to append to an existing file or to completely overwrite the content. Usually, we need to save some sensitive data into files, which means we do not want anyone to access these files and alter the contents. In order to secure files, we can try encrypting files to make them harder to be read. The easiest way to encode is to use the `Base64Coder` class. Refer to the following code:

```
Base64Coder.encodeString( textcontent );
```

Use the following code to decode the content after reading from the file:

```
Base64Coder.decodeString( filestring );
```

Please be aware that encoding may not essentially make the files safe and secure, as any encoding is decodable. An encoded text will seem gibberish to anyone who accesses it. However, be aware that this is a standard encoding method; hence, we have a standard decoding method that can be employed to get correct data.

The leaderboard

Let's implement a local leaderboard for our Thrust Copter game. We will save 10 high scores into a `.json` file. For this purpose, we will create a new class called `SaveManager` that can be used to save any data on any of our future game projects as well. Data is logically stored within a `Save` class in an `ObjectMap` as key-object pairs so that any class can be saved as follows:

```
public static class Save{
  public ObjectMap<String, Object> data = new ObjectMap<String,
Object>();
}
```

We save the data as a local `.json` file in the `bin` folder with the name `scores.json` using the following code:

```
private FileHandle file = Gdx.files.local("bin/scores.json");
```

The `.json` file is loaded and saved via the following methods, and we use `Base64Coder` to encode and decode the file:

```
private Save getSave(){
  Save save = new Save();
  if(file.exists()){
  Json json = new Json();
  if(encoded)save = json.fromJson(Save.class, Base64Coder.
decodeString(file.readString()));
  else save = json.fromJson(Save.class,file.readString());
  }
  return save;
}

public void saveToJson(){
```

```
    Json json = new Json();
    json.setOutputType(OutputType.json);
    if(encoded) file.writeString(Base64Coder.encodeString(json.
prettyPrint(save)), false);
    else file.writeString(json.prettyPrint(save), false);
  }
```

In order to access individual data from the `ObjectMap` class, we have specific methods:

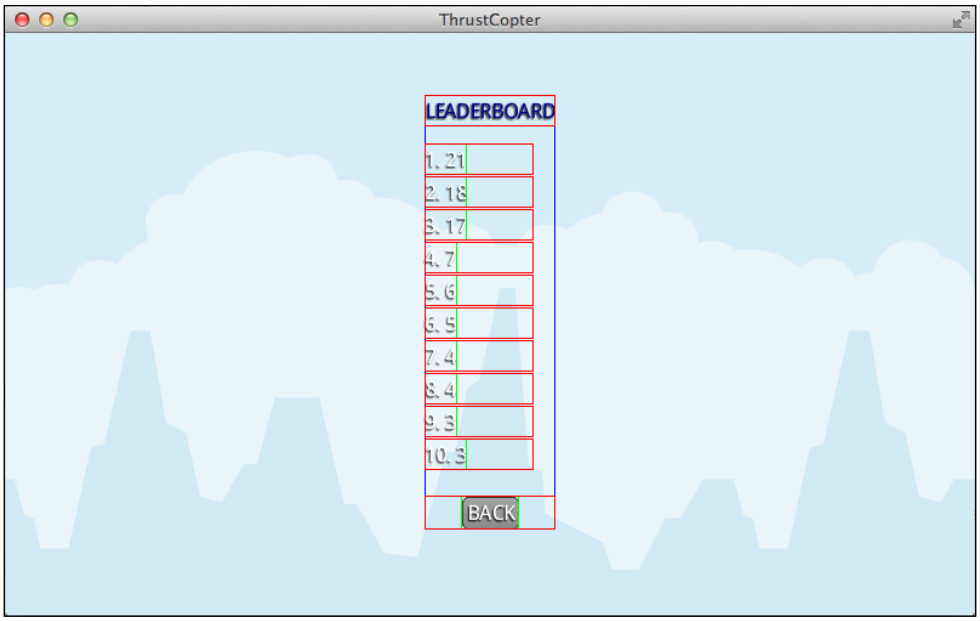
```
public <T> T loadDataValue(String key, Class type){
  if(save.data.containsKey(key))return (T) save.data.get(key);
  else return null;
}
public void saveDataValue(String key, Object object){
  save.data.put(key, object);
  saveToJson();
}
```

`Generics` enables us to read any data type. When specific data is read, we need to check it for a `null` value, which is a value that is returned when the data is not found. When the game runs for the first time, we need to store the value for all the 10 scores. This is done by the `prepareLocalScores` function in the `ThrustCopter` class. We save the scores using the keys `Score1`, `Score2`, and so on up to `Score10`. We check to see whether one of the score values is `null`, which indicates that we are running the game for the first time. In such a case, we store `0` as value for all 10 scores as follows:

```
private void prepareLocalScores() {
  saveManager=new SaveManager(true);
  if(saveManager.loadDataValue("Score1", int.class)==null){
    for(int i=1;i<=10;i++){
      saveManager.saveDataValue("Score"+i, 0);
    }
  }
}
```

Saving and displaying scores

We have a new `LeaderboardScene` class to display the leaderboard when it is accessed via the **LEADERBOARD** button in the menu. This class uses Scene2D to create labels to list all the stored score values, as shown in the following screenshot:



There is nothing special in the `LeaderboardScene` class other than the listing of all the stored score values. The code is as follows:

```
Label score;
for(int i=1;i<=10;i++){
  score=new Label(i+". "+ game.saveManager.loadDataValue("Score"+i,
int.class), skin);
  table.add(score).padBottom(2).align(Align.left);
  table.row();
}
```

To save a score, we will call the `checkAndStoreScore` function from the `endGame` function within the `ThrustCopterScene` class. The `checkAndStoreScore` method checks whether the current score is higher than the lowest score saved in the leaderboard, and then it inserts the score in the correct position. The code is as follows:

```
private void checkAndStoreScore() {
  int finalScore=(int)(starCount+score);
  int lowestScore=game.saveManager.loadDataValue("Score10", int.
class);
  if(finalScore>lowestScore){
    int[] scores = new int[10];
    for(int i=1;i<=10;i++){
```

```
        scores[i-1]=game.saveManager.loadDataValue("Score"+i,
int.class);
      }
      scores[9]=finalScore;
      for(int i=9;i>0;i--){
        if(scores[i]>scores[i-1]){
          finalScore=scores[i-1];
          scores[i-1]=scores[i];
          scores[i]=finalScore;
        }else{
          break;
        }
      }
      Gdx.app.log("info", "saving new score");
      for(int i=1;i<=10;i++){
        game.saveManager.saveDataValue("Score"+i, scores[i-1]);
      }
    }
}
```

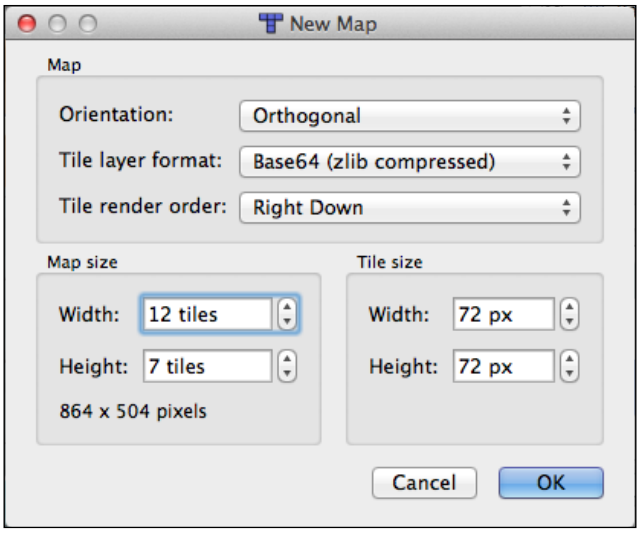
You can find the `scores.json` file in the `bin` folder of your desktop project.

Tile-based level design

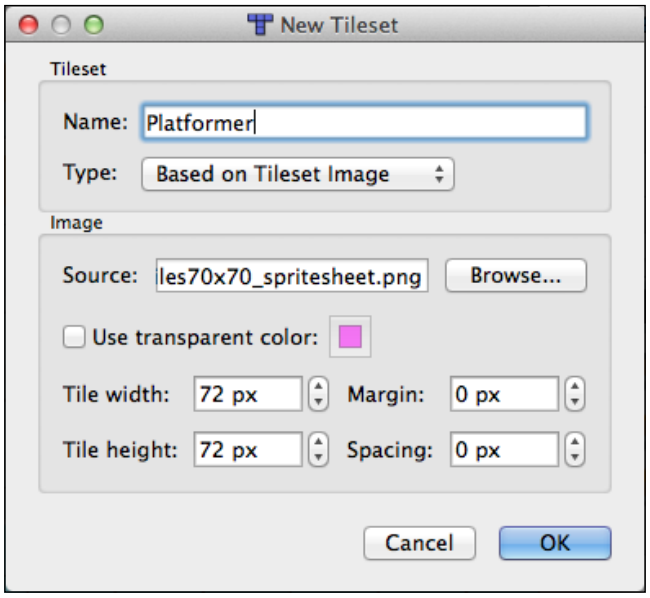
Let's learn something that is not related to our game. Some kinds of games can be easily implemented using a tile-based approach, for example, 2D platformers, top-down scrollers, side scrollers, hexagonal tile games, and isometric games. In these games, the levels are often created using grids of graphic blocks called tiles. Let's learn how to create such a level and load it into LibGDX.

Using Tiled

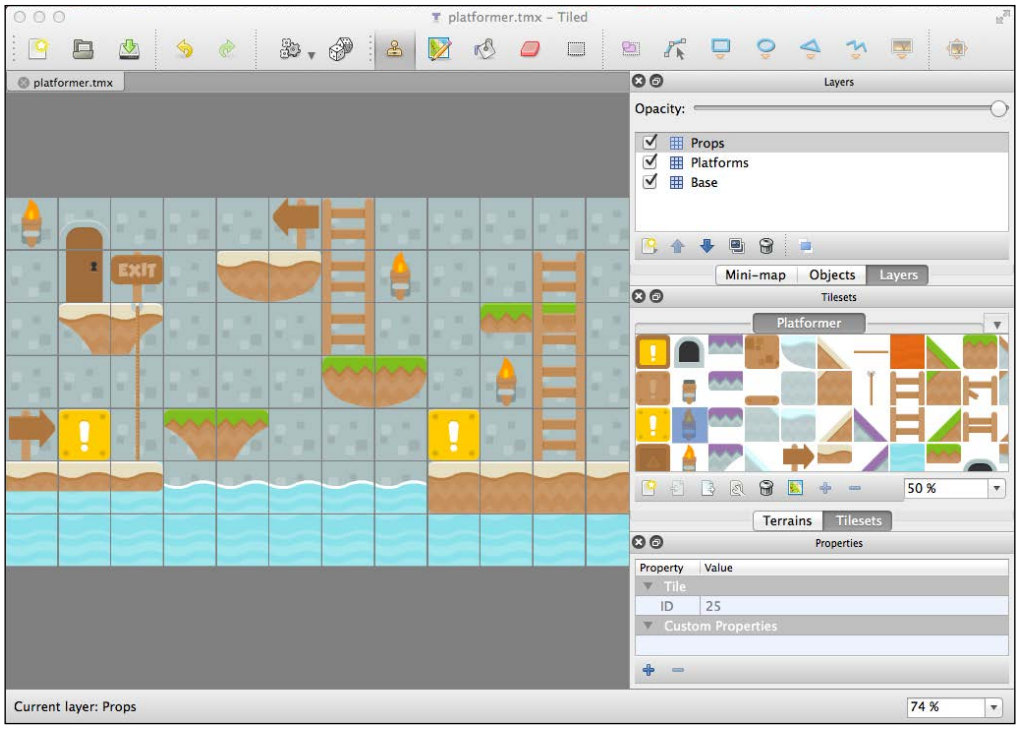
Tiled is a free application that can be used to create tile-based levels, and it can be downloaded from `www.mapeditor.org`. Tiled allows us to design levels easily using just a few mouse clicks. First, we need to create a new map by specifying the tile height, tile width, rows, and columns that is shown in the following screenshot:



We will use the tilesheet named `tiles70x70_spritesheet.png`, which is provided in the source and has tiles of 72 x 72 pixels. In order to fill our screen, which is of the size 800 x 480 pixels, we will need 12 columns and 7 rows of tiles. Once we have the blank map, we need to add a new Tileset, as shown in the following screenshot:



We provide a name to the Tileset, provide the height and width details of the tiles, and select the tilesheet. Then, we can see each individual tiles within the Tiled window. We can select individual tiles and draw on the map to place tiles, as shown in the following screenshot:



We can add multiple tile layers to make everything look right. For example, we will need to set some **Base** tiles to fill the whole map as the base layer. Then, we can have a **Platforms** layer where we can place all our platforms. Then, we can have a **Props** layer where we can place the props and specific items. Layers can be also be used for collision detection, Box2D, or other special info. Once you have designed your level, it can be saved as a `.tmx` file. The preceding screenshot shows how the `platformer.tmx` level is created.

Loading TMX levels

To display the Tiled TMX levels, we need to copy the `.tmx` file along with the tilesheet `.png` file into our `assets` folder. Then, we can use our asset manager to load it as follows:

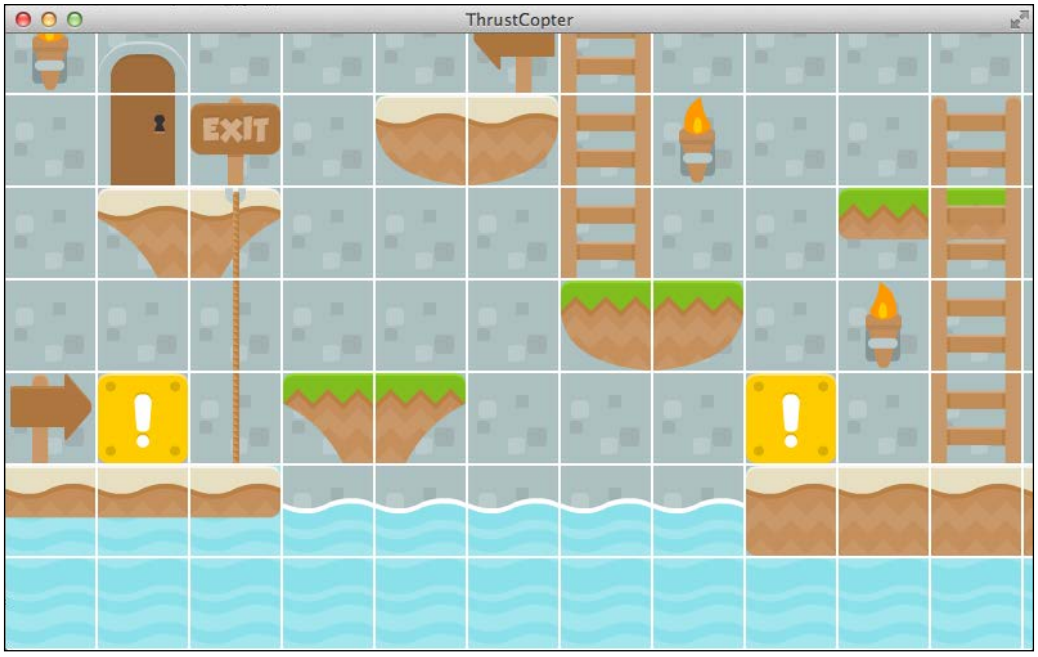
```
private TiledMap map;
private TiledMapRenderer renderer;
```

```
...
game.manager.setLoader(TiledMap.class, new TmxMapLoader(new
InternalFileHandleResolver()));
game.manager.load("platformer.tmx", TiledMap.class);
game.manager.finishLoading();
map = game.manager.get("platformer.tmx");
renderer=new OrthogonalTiledMapRenderer(map);
```

Then in the `render` method add the following code:

```
game.camera.update();
renderer.setView(game.camera);
renderer.render();
```

Please have a look at the following screenshot:



There are some gaps in between the tiles in this case, which are due to the imperfect packing of tiles in the tilesheet `.png` as I have used a free tilesheet that is available at `http://kenney.nl/`. If you have a perfectly packed tilesheet, there won't be any noticeable gaps. Once we have the `TiledMap` class, we can access individual layers and individual cells using following code:

```
TiledMapTileLayer layer =
(TiledMapTileLayer)map.getLayers().get(0);
```

```
Cell cell= layer.getCell(3, 5);
```

If the `.tmx` file is not encoded, then we can open it up in a normal text editor such as Notepad.

Summary

In this chapter, you learned about persisting data and the usage of Tiled maps. You learned how to create TMX tile maps using the Tiled tool. We loaded a TMX map and successfully rendered it using `OrthogonalTiledMapRenderer`. You learned how to use `Preferences` to save simple variables and used it to store our sound status. We implemented a local leaderboard scene using local file access. We also created a `SaveManager` class to load and save encrypted scores list into a `.json` file. Encryption and decryption was done using the `Base64Coder` class.

In the next chapter, we will finish our game by adding the final touches and making it ready for publishing.

9
Finishing Our Android Game

It's time to wrap up our primary project, our Android game. There are some very important features that any game should have such as ways to monetize, global leaderboards, game analytics, global achievements, and social integration. We will explore different ways to accomplish these features and finalize our game for release. In this chapter, we will explore the following topics:

- Interfacing platform-specific code
- AdMob (Google Mobile Ads) integration
- Google Analytics integration
- Leaderboard and achievements using Google Play services
- Flurry analytics, Swarm social integration, and other services
- Finalizing the Android project with icons and other details

We will not make any changes to the game logic in this chapter and the source files for this chapter can be found in the `chapter9.zip` file provided along with the book. Note that most of these services require unique IDs or secret keys that are specific to your account.

Using Google's offerings

For all the additional features we mentioned previously, we have many third-party offerings, and the most important ones will be introduced later in this chapter. Let's first learn how to use all those features that Google has to offer. Google provides AdMob (Google Mobile Ads) to display ads to monetize our game. Google Analytics can be used to track basic app data. Google Play services can be used to implement and track global leaderboards and achievements. Before we start implementing all of these, we need to ensure the following points:

- Use the SDK manager to update to the latest Android SDK tools

- Download and install Google Play services via the SDK manager

Interfacing platform-specific code

This chapter deals with an Android project, and much of what we will do will be specific to that platform. We need a way to detect the currently running platform to decide whether to invoke these features or not. Hence, we add a new public Boolean variable, `isAndroid`, in the `ThrustCopter` class, which is `false` by default. We can detect `ApplicationType` using the following code in the `create` method:

```
switch (Gdx.app.getType()) {
case Android:
  isAndroid=true;
break;
case Desktop:
break;
case WebGL:
break;
case iOS:
break;
default:
}
```

Now, we can check whether the game is running on an Android device using the following code:

```
if(game.isAndroid){
...
}
```

From the core project, we need to call the Android main class to invoke Android-specific code. We enable this using a new interface created in the core project: `IActivityRequestHandler`. Then, we make sure our `AndroidLauncher` main class implements this interface as follows:

```
public class AndroidLauncher extends AndroidApplication implements
IActivityRequestHandler{
...
initialize(new ThrustCopter(this), config);
```

Note that we are passing `this` as a parameter to `ThrustCopter`, which provides a reference to the implemented interface. As this is Android-specific, other platforms' `start` classes can pass `null` as an argument, as we will only use this parameter on the Android platform. In the `ThrustCopter` class, we save the reference with the name `handler`, as shown in the following code:

```
public ThrustCopter(IActivityRequestHandler IARH) {
    handler=IARH;
...
```

Visit `https://github.com/libgdx/libgdx/wiki/Interfacing-with-platform-specific-code` for more information.

Implementing Google Analytics tracking

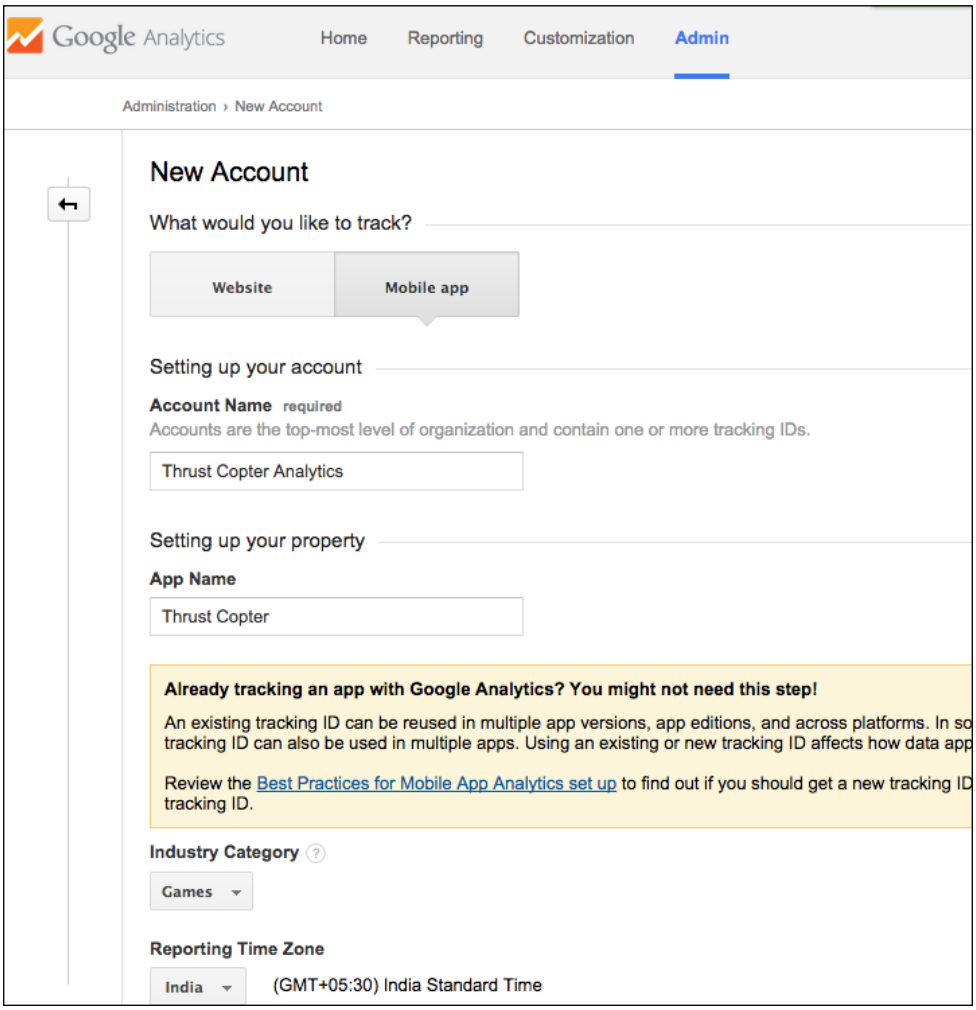
The default implementation of Google Analytics will automatically provide the following information about your app: the number of users and sessions, session duration, operating systems, device models, and geography. To start off, we need to create a Google Analytics property and app view. Go ahead and start using Google Analytics by accessing it at `https://www.google.com/analytics/web/?hl=en`. Create a new account and select **Mobile app** and fill in the details. Once all details are entered, click on **Get Tracking ID** to generate a new tracking ID. The tracking ID will be unique for each account.

The Google Analytics version may change in future, which means the way they are integrated may also change. Check out the Google developers portal for details at `https://developers.google.com/analytics/devguides/collection/android/v4/`.

The `AndroidManifest` file needs to have the following permissions and the `minSdkVersion` instance should be set to `9`, as follows:

```
<uses-sdk android:minSdkVersion="9" android:targetSdkVersion="23" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name=
"android.permission.ACCESS_NETWORK_STATE" />
```

Have a look at the following Google Analytics registration form:



Copy the library project at `<android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib/` to the location where you maintain your Android app projects. Import the library project into your Eclipse workspace. Click on **File**, select **Import**, select **Android**, click on **Existing Android Code Into Workspace**, and browse to the copy of the library project to import it. This step is important for all the Google-related services that we are about to integrate. We need to refer to this library project from our `Thrust Copter-android` project. Right-click on the `Thrust Copter-android` project and select **Properties**. Select the **Android** section, which will display a blank `Library` section to the right. Click on **Add...** to select our library project and add it as a reference.

Adding tracker configuration files

We can provide configuration files to create Analytics trackers. Usually, we need only one tracker, which is usually called the global tracker, to report the basic analytics data. We add the `global_tracker.xml` file to the `res/xml` folder in the Android project. Copy this file from the source provided. Update the `ga_trackingId` section with the new tracking ID you got on creating your application entry on the Google Analytics site. The `screenName` section consists of the different scenes that will be tracked. We added the `MenuScene` and `ThrustCopterScene` classes to the `screenName` section. This needs to be changed for each game as follows:

```
<screenName name="com.csharks.thrustcopter.ThrustCopterScene">Thrust
Copter Game</screenName>
    <screenName name="com.csharks.thrustcopter.MenuScene">Thrust
Copter Menu</screenName>
```

Once the tracker XML file is in place, add the following element to the Android Manifest application part:

```
<meta-data
    android:name=
"com.google.android.gms.analytics.globalConfigResource"
    android:resource="@xml/global_tracker" />
```

We need to access the tracker and report activity start, stop, and scene changes. This can be done using the following code in the `AndroidLauncher` class:

```
Tracker globalTracker;
```

Then, add the following code within the `onCreate` method:

```
GoogleAnalytics analytics = GoogleAnalytics.getInstance(this);
globalTracker=analytics.newTracker(R.xml.global_tracker);
```

Now, we will move on to reporting. We added a new function in the `IActivityRequestHandler` interface called `setTrackerScreenName(String path)`, which needs to be implemented as well:

```
@Override
protected void onStart(){
  super.onStart();
  GoogleAnalytics.getInstance(this).reportActivityStart(this);
}
@Override
public void onStop() {
  super.onStop();
```

```
    GoogleAnalytics.getInstance(this).reportActivityStop(this);
  }
  @Override
  public void setTrackerScreenName(String path) {
    globalTracker.setScreenName(path);
    globalTracker.send(new HitBuilders.AppViewBuilder().build());
  }
```

We also need to report screen names as well when we switch scenes. We do this within the constructors of `MenuScene` and `ThrustCopterScene` as follows:

```
if(game.isAndroid){
  game.handler.setTrackerScreenName("com.csharks.thrustcopter.
MenuScene");
}
```

It's time to test whether everything is working. Connect your Android device and run our Android project on it. We can see that the analytics reporting is showing up in logcat. Once we have significant data, we can access the Google Analytics Web interface to analyze how the game is being played by the masses.

Adding Google Mobile Ads

Legacy AdMob is being renamed Google Mobile Ads, which is now linked with Google AdSense. First, we need to set up AdMob to serve ads by visiting `https://www.google.com/ads/admob/index.html`. Click on the **Monetize** section and use the **Add your app manually** option to set up a new banner ad. This will allot a new AdMob ad unit ID. The Ads API is also part of the Google Play services platform that we have already integrated into our Android project. We have already added as follows the necessary permissions to `AndroidManifest`, but we need to add the following as well:

```
<!--This meta-data tag is required to use Google Play Services.-->
<meta-data android:name="com.google.android.gms.version"
android:value="@integer/google_play_services_version" />
 <!--Include the AdActivity configChanges and theme. -->
<activity android:name="com.google.android.gms.ads.AdActivity"
android:configChanges="keyboard|keyboardHidden|orientation|screen
Layout|uiMode|screenSize|smallestScreenSize"
android:theme="@android:style/Theme.Translucent" />
```

AdMob needs its own view, whereas LibGDX creates its own view when initializing. A typical way of coexisting will be our `Game` view in fullscreen with the `Ad` view overlaid. We will use `RelativeLayout` to arrange both views. We need to replace the `initialize` method with the `initializeForView` method, which lacks some functionality; we need to specify those manually. The `onCreate` method of the `AndroidLauncher` class has the following new code:

```
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
getWindow().clearFlags(WindowManager.LayoutParams.FLAG_FORCE_NOT_
FULLSCREEN);

RelativeLayout layout = new RelativeLayout(this);
View gameView = initializeForView(new ThrustCopter(this), config);
layout.addView(gameView);

// Add the AdMob view
RelativeLayout.LayoutParams adParams =
        new RelativeLayout.LayoutParams(RelativeLayout.LayoutParams.
WRAP_CONTENT,
            RelativeLayout.LayoutParams.WRAP_CONTENT);
adParams.addRule(RelativeLayout.ALIGN_PARENT_TOP);
adParams.addRule(RelativeLayout.CENTER_HORIZONTAL);
adView = new AdView(this);
adView.setAdSize(AdSize.BANNER);
adView.setAdUnitId(AD_UNIT_ID);
startAdvertising();
layout.addView(adView, adParams);
setContentView(layout);
```

The `startAdvertising` function is as follows:

```
private void startAdvertising() {
  AdRequest adRequest = new AdRequest.Builder().build();
  adView.loadAd(adRequest);
}
```

The `IActivityRequestHandler` class has a new method, `showAds(boolean show)`, that toggles the visibility of the `adView` instance. The method is implemented as follows:

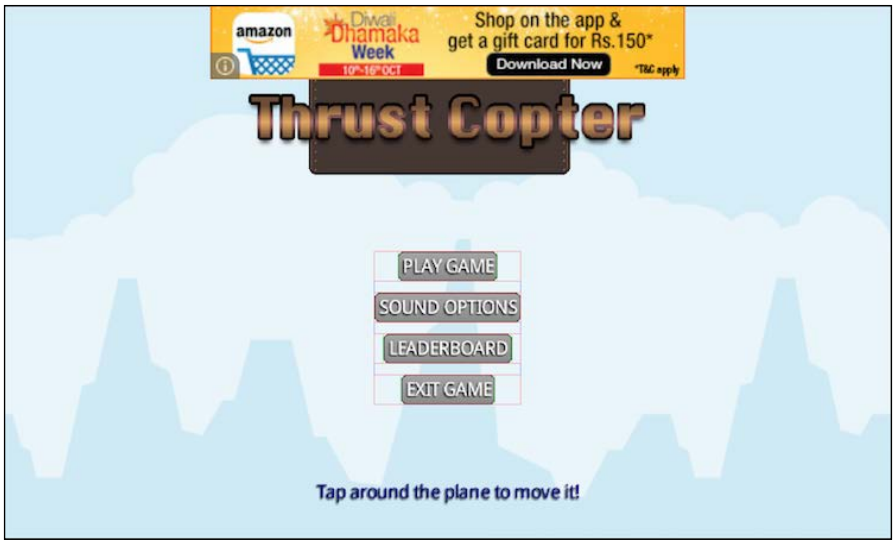
```
@Override
public void showAds(boolean show) {
  handler.sendEmptyMessage(show ? SHOW_ADS : HIDE_ADS);
}
```

Here, `handler`, which is used to access `adView` from the thread that created it, is initialized as follows:

```
private final int SHOW_ADS = 1;
private final int HIDE_ADS = 0;

protected Handler handler = new Handler()
    {
        @Override
        public void handleMessage(Message msg) {
            switch(msg.what) {
                case SHOW_ADS:
                {
                    adView.setVisibility(View.VISIBLE);
                    break;
                }
                case HIDE_ADS:
                {
                    adView.setVisibility(View.GONE);
                    break;
                }
            }
        }
    };
```

For more information, visit `https://github.com/libgdx/libgdx/wiki/Admob-in-libgdx`.

Alternatively, `runOnUiThread` can be used instead of the `handleMessage` method. This is explained later in the *Leaderboard and achievements using Google Play services* section. Now, we can show ads in the menu and hide it when we switch to the game, as shown in the following screenshot:



Leaderboards and achievements using Google Play services

It's time to add a global leaderboard and achievements using Google Play services. This will be done only for the Android project and we will keep the local leaderboard system in place for other platforms. To integrate these, we need access to Google Developer Console, which is only available to those who have a Google Developer account. Please go ahead and sign up for one if you do not have one yet. You will need it to publish your Android games to Google Play Store.

Enroll for a Google Developer account at `http://developer.android.com/distribute/googleplay/start.html`. This has a $25 fee.

You can access the Developer console at `https://play.google.com/apps/publish/`. This is where you publish a new app or add Google Play services to your app. We need to create one leaderboard and five achievements for our game. Google insists that we add five achievements in order to allow publishing the app. The use of the Developer console to create leaderboards and achievements is well detailed in the step-by-step guide at `https://developers.google.com/games/services/console/enabling`. Once you create your app entry with a leaderboard and achievements, you will get an app ID, leaderboard ID, and achievements IDs.

Please be aware that every external service integration step may change when a new version is released. Here, integration of LibGDX Google Play services is based on the tutorial found at `http://fortheloss.org/tutorial-set-up-google-services-with-libgdx/`.

You will need to provide a package name while linking your app to create a client ID. Note that our Android package will have a `.android` part as well. In our case, the package name is `com.csharks.thrustcopter.android`. A client ID is to be generated for `debug.keystore` as well as your release signing certificate. We already have a `debug.keystore` file and will be creating a final signing certificate in the next chapter. Do not forget to link it as well before publishing the game.

Linking BaseGameUtils

We have already wired up the `google-play-services` library project and updated the `Manifest` file. In addition, we need some helper classes from the sample project `BaseGameUtils`, which needs to be downloaded from `https://developers.google.com/games/services/downloads/#samples`. Once the files are extracted, a folder named `android-basic-samples-master` will be created with the `BasicSamples` and `Scripts` folders. Google now supports Android Studio by default, and hence the projects won't work with Eclipse directly. We need to create Eclipse-compatible projects. Copy the `make_eclipse_compat` script file to the folder containing the `BasicSamples` folder and run it to create a new folder named `eclipse_compat` that has all the projects.

Import the `libraries/BaseGameUtils` project to Eclipse by going to **File** | **Import** | **Android** | **Existing Android Code Into Workspace**. Copy the `libraries/BaseGameUtils` project to your workspace while importing it, as you did with the `google-play-services` library. Assign this project to a library project by going to **Properties** | **Android** and selecting the **isLibrary** checkbox. We also need to link the `google-play-services` library project to this project by going to **Properties** | **Java Build Path** | **Projects** | **Add** and selecting that project. Our Android project needs to link to this project as a library by navigating to **Properties** | **Android** and clicking on **Add** in the **Library** section to add `BaseGameUtils`.

You may need to clean the `BasegameUtils` file to make errors go away. Create an `ids.xml` file in the `res/values` folder within the Android project where we will keep all our game-related IDs:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_id">your app id</string>
    <string name="leaderboard_id">your leaderboard id</string>
    <string name="achievement10_id">your achievement id 1</string>
    <string name="achievement20_id">your achievement id 2</string>
    <string name="achievement30_id">your achievement id 3</string>
    <string name="achievement40_id">your achievement id 4</string>
    <string name="achievement50_id">your achievement id 5</string>
</resources>
```

In the case of Thrust Copter, the achievements are collecting 10, 20, 30, 40, and 50 stars—hence the name. The Android `Manifest` file needs to know the app ID, for which we add the following code inside the `application` tag:

```
<meta-data android:name="com.google.android.gms.games.APP_ID"
android:value="@string/app_id"/>
<meta-data android:name="com.google.android.gms.version"
android:value="@integer/google_play_services_version"/>
```

Wiring with code

We need to add new functions to our `IActivityRequestHandler` class and implement them in the `AndroidLauncher` class to access the leaderboard and achievements. The most important methods are as follows:

```
public void login();
public void logOut();
 public boolean isSignedIn();
 public void submitScore(int score);
 public void unlockAchievement(int stars);
 public void showScores();
 public void showAchievements();
```

The method names are self-explanatory. We will create an instance of the `GameHelper` class from `BaseGameUtils` to facilitate communication with Google Play services. The `GameHelper` instance needs to know about the life cycle of the app and when the app responds to an `Intent` class. The `AndroidLauncher` class needs to implement the `GameHelperListener` interface so that it can respond to events. Refer to the following code:

```
public class AndroidLauncher extends AndroidApplication implements
IActivityRequestHandler, GameHelperListener{
```

Then, add the following line of code in the `onCreate` method:

```
gameHelper=new GameHelper(this, GameHelper.CLIENT_GAMES);
gameHelper.enableDebugLog(true, "GPS");
gameHelper.setup(this);
```

The following code shows how we can implement the `IActivityRequestHandler` functions using `gameHelper`:

```
  @Override
  protected void onStart(){
    ...
    gameHelper.onStart(this);
  }
  @Override
  public void onStop() {
    ...
    gameHelper.onStop();
  }
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
Intent data)
  {
    super.onActivityResult(requestCode, resultCode, data);
    gameHelper.onActivityResult(requestCode, resultCode, data);
  }
  @Override
  public void login() {
    try
    {
      runOnUiThread(new Runnable()
      {
        //@Override
        public void run()
        {
          gameHelper.beginUserInitiatedSignIn();
        }
      });
    }
    catch (Exception e)
    {
      Gdx.app.log("ThrustCopter", "Log in failed: " + e.getMessage()
+ ".");
    }
```

```
    }
    @Override
    public void logOut() {
      try
      {
        runOnUiThread(new Runnable()
        {
          //@Override
          public void run()
          {
            gameHelper.signOut();
          }
        });
      }
      catch (Exception e)
      {
        Gdx.app.log("ThrustCopter", "Log out failed: " +
e.getMessage() + ".");
      }
    }
    @Override
    public boolean isSignedIn() {
      return gameHelper.isSignedIn();
    }
    @Override
    public void submitScore(int score) {
      if (isSignedIn())
      {
        Games.Leaderboards.submitScore(gameHelper.getApiClient(),
getString(R.string.leaderboard_id), score);
      }
    }
    public void unlockAchievements(String achievementID) {
      if (isSignedIn()){
        Games.Achievements.unlock(gameHelper.getApiClient(),
achievementID);
      }
    }
    @Override
    public void showScores() {
      if (isSignedIn() == true)  startActivityForResult(Games.
Leaderboards.getLeaderboardIntent (gameHelper.getApiClient(),
getString(R.string.leaderboard_id)), 9002);
    }
    @Override
```

```
  public void showAchievements() {
    if (isSignedIn()) { startActivityForResult (Games.Achievements.
getAchievementsIntent (gameHelper.getApiClient()),9002);
        }
  }
  @Override
  public void onSignInFailed() {
    Gdx.app.log("ThrustCopter", "Sign in fail");
  }
  @Override
  public void onSignInSucceeded() {
    Gdx.app.log("ThrustCopter", "SignedIn");
  }
```

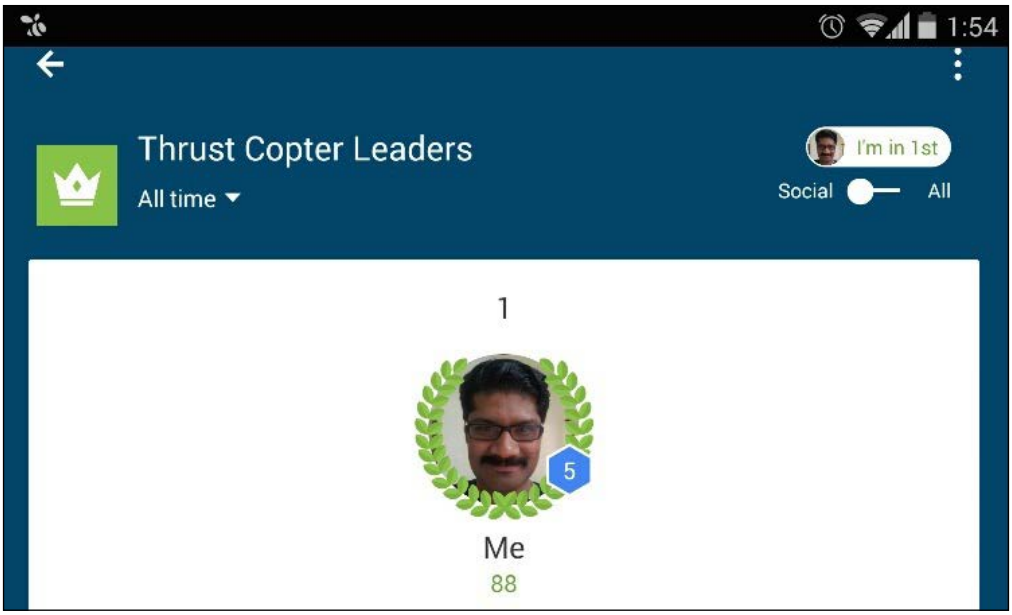
The `MenuScene` leaderboard button code is updated to show the global leaderboard on an Android device as follows:

```
if(game.isAndroid){
      game.handler.showScores();
}else{
      game.setScreen(new LeaderboardScene(game));
}
```

Similarly, the `checkAndStoreScore` function in `ThrustCopterScene` is changed to report the score on an Android device:

```
private void checkAndStoreScore() {
    int finalScore=(int)(starCount+score);
    if(game.isAndroid){
      game.handler.submitScore(finalScore);
    }else{
    ...
    }
}
```

Explore the code further to find how achievements are being reported. When you test the game on your Android device with an active Internet connection, we can see that Google Play automatically logs us in. By default, your Developer account will be added as tester; however, if we need others to test our game, then we need to explicitly add them in the Developer console, as shown in the following screenshot:



To integrate the same features on an iOS device, we need RoboVM bindings. Please visit `https://github.com/BlueRiverInteractive/robovm-ios-bindings` for more details.

Famous third-party alternatives

Although Google offers all these features that we have integrated, there is a high probability that you need to look for other alternatives as well, which may be better in some way or another. Let me share some of those alternatives with you.

Flurry for analytics

Visit `http://www.flurry.com/` to explore what Flurry has to offer. Flurry analytics is almost an industry standard when it comes to games. The integration for Android is explained at `http://support.flurry.com/index.php?title=Analytics/GettingStarted/Android`. Along with this, Flurry also offers ads. Basic tracking involves adding just a few lines of code after including the JAR files:

```
FlurryAgent.onStartSession(this, "YOUR_API_KEY");
```

Add the following code as well:

```
FlurryAgent.onEndSession(this);
```

Ads from InMobi

A famous alternative to AdMob is InMobi Ads. You will need to create an account and Ad by signing in at `http://www.inmobi.com/`. Once this is done, you can integrate the Ad as per the instructions given in the integration docs, but this may be a complicated affair. An alternative is to use Ad Mediation through AdMob. Check out the link `https://developers.google.com/mobile-ads-sdk/docs/admob/android/mediation-networks` to enable AdMob to serve InMobi Ads.

Swarm – the all-in-one package

Swarm is a third-party solution offering leaderboards, achievements, monetization, social features, and cloud storage. Visit `http://swarmconnect.com/` to check out all that Swarm has to offer. Integrating Swarm may not be very easy, but considering the features it offers, it is worth the effort. Social sharing is part of Swarm, which in itself makes it a great solution. Swarm offers a LibGDX-specific tutorial to integrate its solutions at `http://swarmconnect.com/admin/docs/libgdx`.

Wiring up social sharing manually is not an easy process, and I advise you to use readily available solutions for Facebook and Twitter sharing. An easy alternative to share content to social media is by using `ACTION_SEND`. Visit `http://developer.android.com/training/sharing/send.html` for more information.

Creating icons

The icons to be used for our game are in the `res` folder in the Android project's folder. The `drawable` folders contain dpi-specific icons to be used depending on the device. We can manually create each of these icons and replace the `ic_launcher.png` file or can use some tools to create all the versions of icons from a single larger icon image. One such tool is available at `http://romannurik.github.io/AndroidAssetStudio/icons-launcher.html`. Once the icons are in place, we are all set to publish our game. You are welcome to go ahead and add a pause screen to your game, from where a player can choose to resume the game, mute the sound, or go back to the menu.

Summary

In this chapter, you learned how to handle platform-specific code. Although we implemented everything for the Android project only, we can use RoboVM bindings to do the same for iOS. You learned how to use Google Play services to integrate AdMob, Analytics, a leaderboard, and achievements. Finally, we created icons, and our Android project is now ready to be published.

In the next chapter, we will publish our game to different platforms.

10
Time to Publish

This chapter is about publishing our game to different platforms and stores. LibGDX and Eclipse handle many aspects related to this like magic, but there are steps involved that we need to be aware of. An interesting fact is that LibGDX also allows us to deploy a game to OUYA or as an applet on the Web. This final chapter will cover the following topics:

- Publishing to Google Play
- Publishing to Apple App Store
- Publishing to the Web
- Publishing to a desktop
- New developments and needful resources

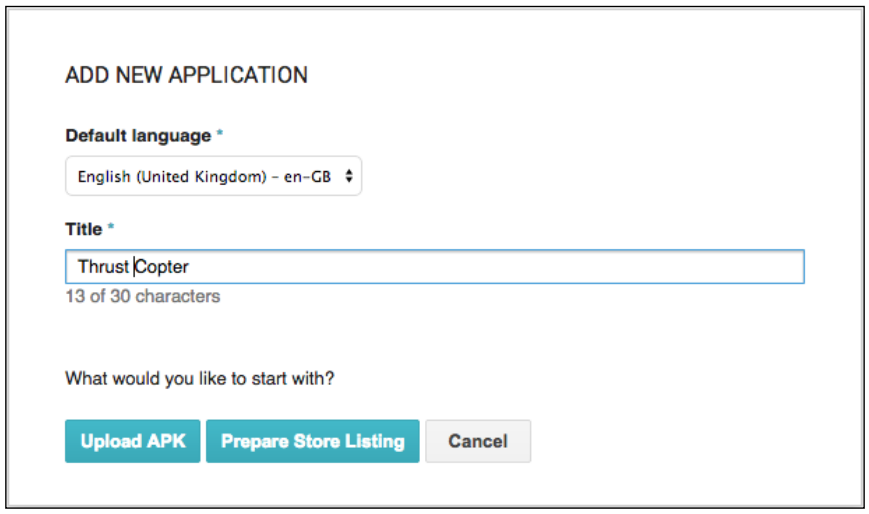
There is no source code for this chapter.

Publishing the Android version

In order to publish the Android version of the game to Google Play Store, we need an Android developer account. You can enroll for an account using `http://developer.android.com/distribute/googleplay/start.html` and access the developer console at `https://play.google.com/apps/publish/`. This is where you publish a new app or add Google Play services to your app. We need to create an entry for our game in Google Play Console to later upload our release version of the `.apk` file.

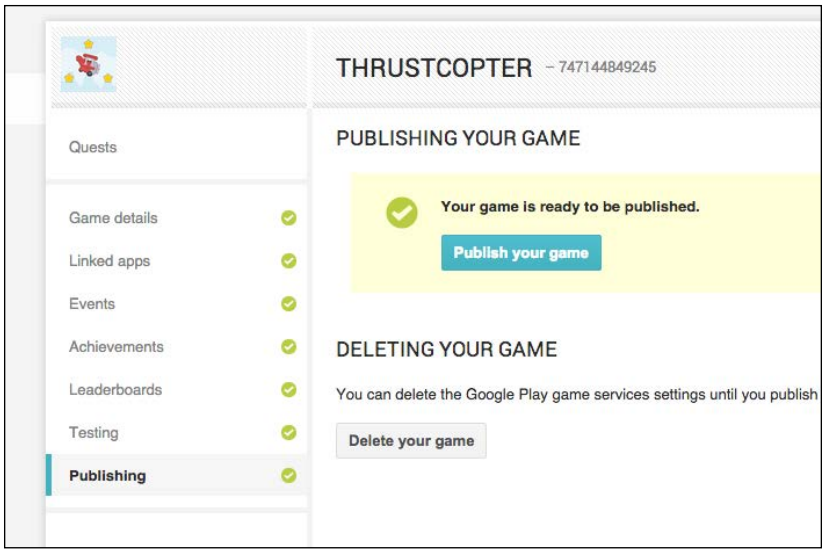
Preparing the store listing

In the publishing portal, select **Add New Application** to start the process. This will show a pop up where you can either upload the `.apk` file or prepare the store listing. We need to prepare the store listing first, so select this option and proceed providing the necessary details, as shown in the following screenshot:



It is wise to check the launch checklist at `http://developer.android.com/distribute/tools/launch-checklist.html`.

We need to provide both short and long descriptions of the game and multiple graphic assets such as screenshots, icons, promo images, feature graphics, video links, and so on. We also need to select the application type, which in this case is **Games**. At the **Pricing & Distribution** section, we can set our game as either a paid or free game. It is here that we select the countries where the game will be shown. The main section of **Game services** is where we added our Google Play services entry. Make sure all the graphic assets and details needed in the **Game services** entry for the game are also filled out. You will need to add icons to your achievements and ensure they are all there, as shown in the following screenshot:



Preparing to release the APK

Next up, we need to upload the Android binary file to the **APK** section for which we first need to create our release build.

Check out `http://developer.android.com/tools/publishing/preparing.html` to learn how to prepare your app for release.

A few things to remember before you release your app are as follows:

- Remove all debugging code
- Remove all unwanted classes
- Remove unused assets
- Publish Google Play services
- Create a release build

We need to publish our Google Play services entry before we actually publish the game. But the release-signing key needs to be linked. If this is done by following the previous chapter, then you can publish the Google Play services entry. If you have not created a signing certificate for release, please wait until you create one. Create a Client ID in the **Link App** section of the Google Play services entry using this release keystore.

To create a release build for our Android game, we need to right-click on the Android project and select **Export**, then navigate to **Android | Export Android Application**. Your project will be selected; if not, select the relevant project for exporting and click on **Next**. If you have an existing keystore, then select **Use Existing Keystore**; if not, select **Create New Keystore**.

To create a new keystore, please follow the documentation given at `http://developer.android.com/tools/publishing/app-signing.html`.

Using an existing keystore, you will need to provide the location of the keystore you want to sign with and the password. In the next screen, you will need to select the alias and enter its password as well. Next up, you can select a destination for your APK and click on **Finish**. Now that we have our final APK, we can upload it to the **APK** section of the developer console. Click on the **Upload your first apk to Production** button to do this. After you upload the APK, it is just a matter of publishing it to the Play Store by changing its status on the top-right dropdown. It may take a few hours for the game to appear in the Google Play Store though. I have shared the final `.apk` file for Thrust Copter.

The Google Play link for our Thrust Copter game is `https://play.google.com/store/apps/details?id=com.csharks.thrustcopter`.

Publishing the desktop version

A desktop version is one that can work on a PC, Mac, or Linux. We can do this easily by creating a runnable JAR using Eclipse. Right-click on the desktop project, navigate to `Export | Java | Runnable Jar`, and click on **Next**. In the next screen, we need to select the launch configuration used to launch our desktop game from the dropdown and select a location to create the output JAR file. Clicking on **Finish** will create a `.jar` file that can be run on any desktop with Java installed.

Alternatively, we can make the desktop version run natively without Java dependency by packing JRE along with it, using tools such as JarWrapper. For more information, refer to `https://github.com/stbachmann/JarWrapper` and also `http://launch4j.sourceforge.net/`.

I have shared the runnable `.jar` file for Thrust Copter.

Publishing the Web version

Publishing the HTML version is straightforward. Right-click on the HTML project and navigate to **Google | GWT Compile**. This will launch the compile options window. Keep everything default and select **Compile**. A pop up will ask for the `war` directory and you can point to it within the HTML project folder. The compilation will take time, and we can track the progress in the console window. Once it is finished, you can find the final files within the `war` folder.

When publishing to the Web, we can also publish the Web version as an applet. Check out this tutorial at `http://javaeggs.com/2014/publishing-your-libgdx-game-as-an-applet/`.

Before you compile the HTML project, make sure the HTML build works from within Eclipse as explained in *Chapter 1*, *Wiring Up*. For Thrust Copter, you may get an error that classes are not found. You may need to add these class folders to the following `ThrustCopter.gwt.xml` file found in the `src` directory of the core project:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit
trunk//EN" "http://google-web-toolkit.googlecode.com/svn/trunk/distro-
source/core/src/gwt-module.dtd">
<module>
  <source path="com/csharks/thrustcopter" />
  <source path="com/csharks" />
  <source path="com/matsemann/libgdxloadingscreen" />
  <source path="com/matsemann/libgdxloadingscreen/screen" />
</module>
```

Once we have the Web build, we can use it to create an app that is deployable on the Windows 8.1 platform. Follow this link, `http://www.badlogicgames.com/forum/viewtopic.php?f=17&t=14766`.

Publishing the iOS version

Deployment on Apple App Store should not be considered as an easy task. The steps involved can be very intimidating for a first-timer. But that has not stopped millions of developers out there.

A very detailed tutorial on App Store publishing can be found at `http://www.raywenderlich.com/8003/how-to-submit-your-app-to-apple-from-no-account-to-app-store-part-1` and `http://www.raywenderlich.com/8045/how-to-submit-your-app-to-apple-from-no-account-to-app-store-part-2`.

We need to meet the following requirements for publishing the iOS version:

- iOS developer account (costs $99 per year)
- Mac with the latest Xcode IDE (from Mac App Store)
- Development and distribution certificates installed on the Mac
- New App ID and iTunes Connect entry for the new app
- Development and distribution provisioning profiles for the app
- Testing devices added to the developer portal and provisioning profiles
- The final release IPA that uses the distribution provisioning profile signed with a distribution certificate
- A lot of patience

Important websites you will need to use are `developer.apple.com` and `itunesconnect.apple.com`. We need the app entry on the iTunes Connect website with the **Ready to Upload Binary** status in order to upload our `.ipa` file.

First steps at the developer portal

Once you have acquired the developer account, create and install both the development and distribution certificates on your Mac. The next step is to create a new App ID by providing the correct bundle ID, which in this case is `com.csharks.thrustcopter`. Once we have the App ID, we need to create the development and distribution provisioning profiles for it. The development provisioning profile lets us test our app on selected devices. These devices need to be added to the developer portal and selected to be part of the provisioning profile while we're creating them. While creating provisioning profiles, you will be asked to select the App ID, certificate, and devices, and you'll be asked to provide a name. We can use Xcode to sync these profiles with Mac or manually download and add them.

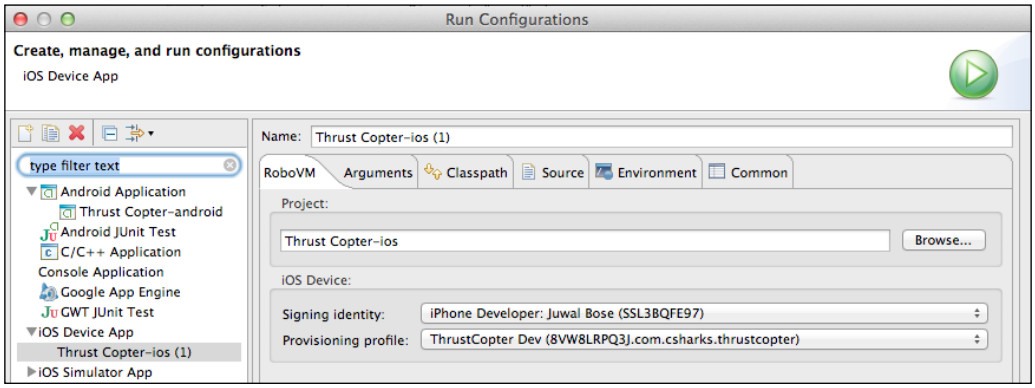
Preparing the iOS project

An iOS app requires multiple launch images and a collection of icons to support the different devices. Check out the list of these devices at `https://developer.apple.com/library/IOs/documentation/UserExperience/Conceptual/MobileHIG/IconMatrix.html`. These images need to be replaced in the data folder in the iOS project folder. You need to take care that you follow the naming conventions exactly. We can set the build and the app version in the `robovm.properties` file. The `info.plist.xml` file determines most of the properties and configurations, the important ones of which are detailed as follows:.

- `UIDeviceFamily`: The array that follows this key enables iPhone and iPad publishing. Currently, it has two values, meaning we are publishing a universal app that supports both devices.
- `UISupportedInterfaceOrientations`: The array after this key determines the supported screen orientations. You can add or remove orientations as needed.
- `CFBundleIconFiles`: You can add all icons in the array after this key.

Testing the build on a device

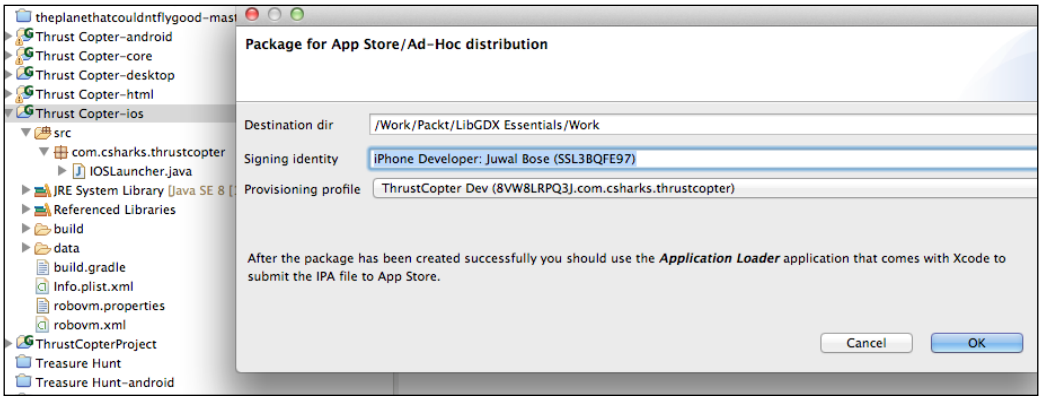
You may have already tested your game project on an iOS simulator by right-clicking on the project and navigating to **Run As | iOS Simulator App**. But before we take the final build, we should test the project on a real iOS device. For this, we need to connect the device via a USB cable. Now make sure the run configuration is correct by selecting the iOS project and Eclipse tab by navigating to **Run | RunConfigurations**. Set the correct signing identity and provisioning profile, as shown in the following screenshot:



Now right-click on the iOS project and go to **Run As | iOS Device App**. Note that it's important that you set the correct app in the `robovm.properties` file. Remember that LibGDX has added `.iOSLauncher` to the end of the App ID. Also, you may need to update the `UISupportedInterfaceOrientations` entry in the `info.plist.xml` file to have only the needed orientations.

Creating the IPA

To create an IPA for testing, we need to use the development certificate and development provisioning profile. Right-click on the iOS project and go to **RoboVM Tools | Package for App Store/Ad-Hoc distribution**. This will launch a pop up asking for a destination folder, the signing-certificate dropdown, and the provisioning profiles dropdown. Select the correct details and hit **OK** to start the compilation, as shown in the following screenshot:



We need to do the same with the distribution certificate and distribution provisioning profile in order to create an IPA to be uploaded to the App Store. The Application Loader app on Mac can be used to do this. At the time of this writing, the RoboVM plugin version is 0.0.14 and goes with RoboVM 1.0.0 alpha4 and LibGDX 1.4.1. Ensure that you update to the latest version of all the plugins when dealing with Apple, as they tend to be not very backward compatible with their systems. Also, please actively follow the Badlogic Games blog for latest information, for example, `http://www.badlogicgames.com/wordpress/?p=3533`.

This particular version was just released and had issues that will soon be fixed. I had to duplicate clang to clang++ at `/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain /usr/bin` and update the `robovm.xml` `forceLinkClasses` properties as follows to make the compilation work:

```
<forceLinkClasses>
```

```
    <pattern>com.badlogic.gdx.scenes.scene2d.ui.*</pattern>
    <pattern>com.android.okhttp.HttpHandler</pattern>
    <pattern>com.android.okhttp.HttpsHandler</pattern>
    <pattern>com.android.org.conscrypt.**</pattern>
    <pattern>com.android.org.bouncycastle.jce.provider.
BouncyCastleProvider   </pattern>
  <pattern>com.android.org.bouncycastle.jcajce.provider.keystore.
BC$Mappings   </pattern>     <pattern>   com.android.org.
bouncycastle.jcajce.provider.keystore.bc.BcKeyStoreSpi   </pattern>
<pattern>   com.android.org.bouncycastle.jcajce.provider.keystore.
bc.BcKeyStoreSpi$Std   </pattern>    <pattern>   com.android.org.
bouncycastle.jce.provider.PKIXCertPathValidatorSpi </pattern>
      <pattern>   com.android.org.bouncycastle.crypto.digests.
AndroidDigestFactoryOpenSSL   </pattern>
  </forceLinkClasses>
```

Some useful resources and links

So, the book has come to its end, but your journey has just started. Mobile technology is a fast-evolving technology, which means you need to keep on learning and watching out for new changes and features. Do not forget to try out all the LibGDX demos and tests before you venture on. An awesome collection of LibGDX-related links can be found at the LibGDX wiki page, `https://github.com/libgdx/libgdx/wiki/External-tutorials`. Among them, one of particular importance is `libgdx-utils` at `https://bitbucket.org/dermetfan/libgdx-utils`. There is also a page that lists out the different versions of LibGDX along with the associated plugin versions: `http://libgdx.badlogicgames.com/versions.html`. Also, the link to the LibGDX forum is `http://www.badlogicgames.com/forum/`. But among all the latest developments, one that stands out is Overlap2D.

Overlap2D

Overlap2D is a Game Level and UI Editor for LibGDX games. At the time of writing, it is v0.0.6, but still very powerful and feature rich. Head over to `http://overlap2d.com/` to download your version while it is free. Using Overlap2D, we can visually design our game scene, menu UI, and much more, thus saving hours, maybe be even weeks of time.

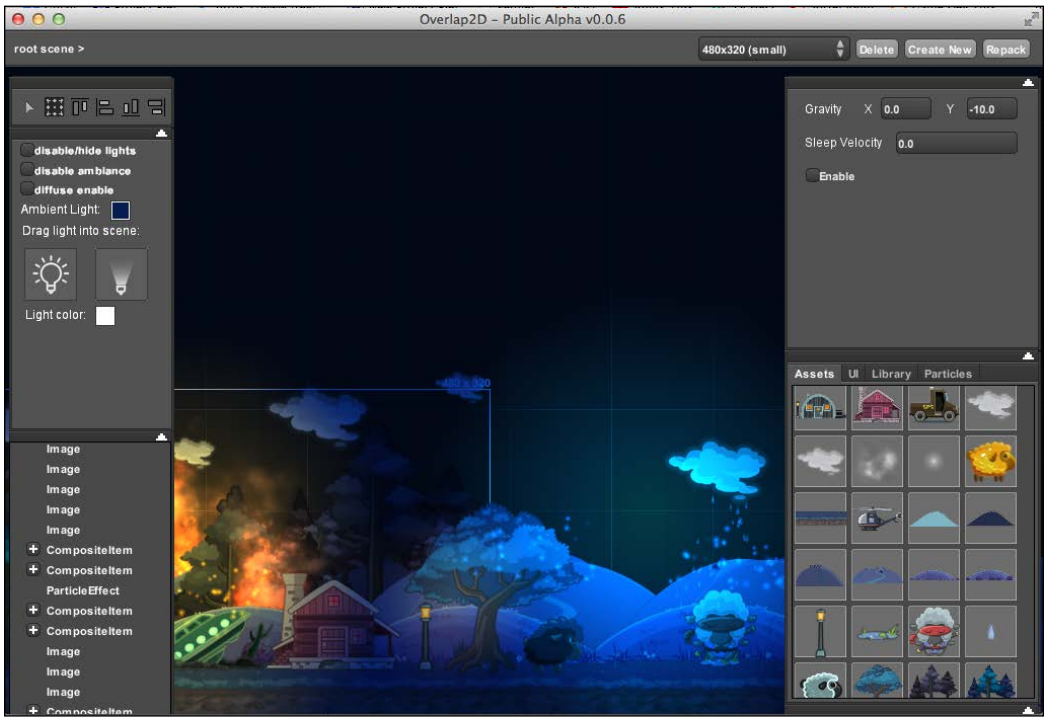
Check out the mind-blowing video of the tool in action at `https://www.youtube.com/watch?v=I0g-t0nZ-qE`.

At the time of this writing, the following features are present:

- Visual design of graphics elements

- Creation of composite elements and composite buttons
- Visual layout of Scene2D elements
- Box2D lights
- Multiple layers
- Integrated Physics Editor
- Particles effects integration
- TTF font support
- 9-patch image support

The following screenshot shows the Overlap2D UI:



Avetis Zarkharyan, the maestro behind this tool, has shared all that we can expect in the future for Overlap2D:

- Cocos2D runtime
- JavaScript runtime
- Starling runtime

- Runtime based on C++/C#
- Unity runtime
- Triggering system
- Custom-shape polygon textures
- Dynamic physics based on spine bone positions and much more

A lot of time is wasted while we create our game levels and menus using trial and error methods. Overlap2D is a life saver and makes prototyping easy and fast.

Working with Overlap2D

This part might change as the tool approaches the v1.0 release, but let me explain the basic concepts to get you started. We create a new project in Overlap2D and specify a resolution for which we have created our art assets for. This will be the largest supported resolution, and Overlap2D will be able to create assets for lower resolutions and aspect ratios. Assets are imported via **File | Import to library**, which helps us import images, particle files, spine animations, TTF fonts, and 9-patch images. Once we have the assets, we can drag-and-drop them onto the stage to design our level.

We are able to move, scale, and rotate everything that is placed on the stage. We might need to group a few items together to form a single item. This can be done in Overlap2D by selecting those items and then right-clicking on and selecting **Group into Composite**, which creates `CompositeItem`. After this, they act as part of a single item, which can be very convenient. We can also create multiple layers and place items of specific layers to add depth to the scene's level. We can also add lights to the level for setting the whole ambience of the level or lighting up specific portions of the level.

Multiple scenes can be designed in this way. Once we have everything designed, we can export it to create a JSON file with all the necessary data to recreate it within LibGDX. It will contain the following files:

- Several scenes that are `.json` files containing information on what your scene has, the coordinates, and other configurations
- Several of the `AtlasPack` classes separated for different resolutions, each containing all the game assets except for animations
- Spine animation directory for each resolution with `anim.png`, `anim.atlas`, and `anim.json`
- Sprite animation directory for each resolution with atlas packs for each
- A directory with used TTF fonts

- Particle effect files
- Finally, the `project.dt` file with a project information like scenes list and resolutions list

For a Flappy Bird tutorial to get you started with Overlap2D, refer to `http://www.gamefromscratch.com/post/2014/09/08/Guest-Tutorial-Making-Flappy-Bird-using-Overlap2D-and-LibGDX.aspx`.

We can use the `ResourceManager` class to load the assets into LibGDX. The `SceneLoader` class can be used to load a complete scene as a root actor, which is a `CompositeItem` class. We have provided names for items within the scene, and we can retrieve them using the `getItemById` method. We can add logic to a `CompositeItem` class using the `addScript` method. Such a logic class should implement the `iScript` interface that has an `init` and `act` method. The `init` method will be called when the item is initialized and placed on the stage, and the `act` method will be called in every frame. This way, we can add scripts to the entire scene or individual items, provided they are composite items.

Keep track of the development of Overlap2D and use it in your LibGDX projects. The link to the forums is `http://overlap2d.com/forums/`.

Summary

This concludes our last chapter. In this chapter, we learned how to publish our game to different platforms. We also learned about Overlap2D and how to save a lot of time using it to create our game scenes.

I hope this has been a good journey and you have learned LibGDX and grown in confidence to create your next blockbuster game. Get on with it and please do let me know regarding your game or any other queries you may have.

LevelUp!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Thank you for buying
LibGDX Game Development Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

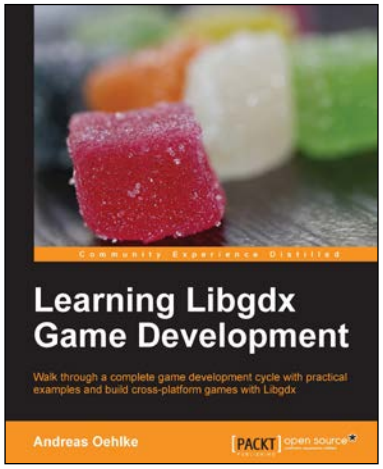
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

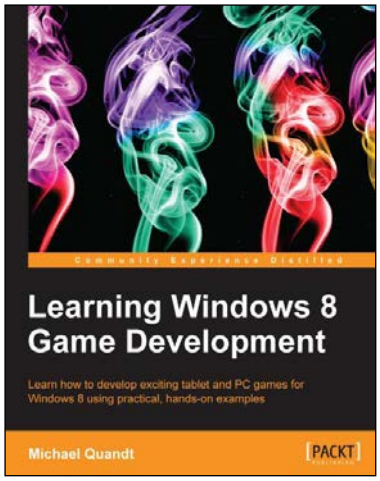


Learning Libgdx Game Development

ISBN: 978-1-78216-604-7 Paperback: 388 pages

Walk through a complete game development cycle with practical examples and build cross-platform games with Libgdx

1. Create a Libgdx multi-platform game from start to finish.
2. Learn about the key features of Libgdx that will ease and speed up your development cycles.
3. Write your game code once and run it on a multitude of platforms using Libgdx.



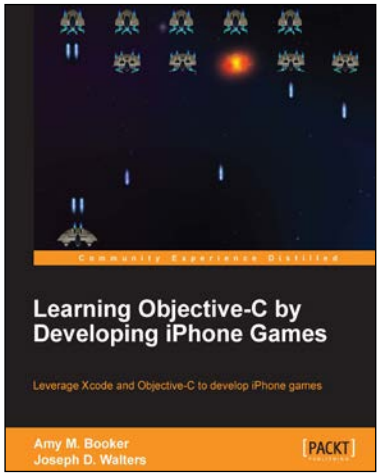
Learning Windows 8 Game Development

ISBN: 978-1-84969-744-6 Paperback: 244 pages

Learn how to develop exciting tablet and PC games for Windows 8 using practical, hands-on examples

1. Use cutting-edge technologies like DirectX to make awesome games.
2. Discover tools that will make game development easier.
3. Bring your game to the latest touch-enabled PCs and tablets.

[PACKT] open source
community experience distilled
PUBLISHING

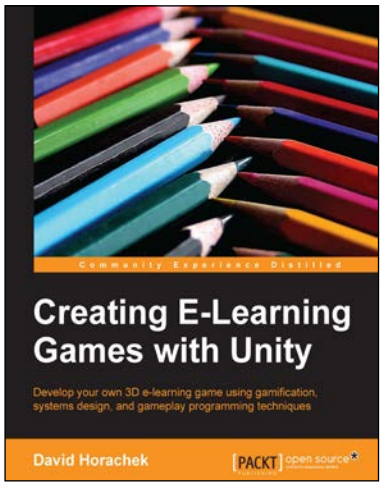


Learning Objective-C by Developing iPhone Games

ISBN: 978-1-84969-610-4 Paperback: 284 pages

Leverage Xcode and Objective-C to develop iPhone games

1. Get started with the Xcode development environment.
2. Dive deep into programming with Objective-C.
3. A practical and engaging tutorial to create vintage games such as Space Invaders and Galaga.



Creating E-Learning Games with Unity

ISBN: 978-1-84969-342-4 Paperback: 246 pages

Develop your own 3D e-learning game using gamification, systems design, and gameplay programming techniques

1. Develop a game framework for a 3D eLearning game.
2. Program dynamic interactive actors and objects to populate your game world.
3. An easy-to-follow guide along with an extensive source code to support and guide readers through the concepts in the book.