Community Experience Distilled

Building Android Games with Cocos2d-x

Learn to create engaging and spectacular games for Android using Cocos2d-x

Foreword by Carlos Piñan, CTO, Vitrum Games

Raydelto Hernandez

[PACKT] PUBLISHING open source* community experience distilled

Building Android Games with Cocos2d-x

Learn to create engaging and spectacular games for Android using Cocos2d-x

Raydelto Hernandez

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Building Android Games with Cocos2d-x

First published: March 2015

Production reference: 1240315

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

ISBN 978-1-78528-383-3

www.packtpub.com

Cover image by Ricardo Diaz (diazric@gmail.com)

Credits

Author
Raydelto Hernandez

Reviewers
Alejandro Duarte
Emanuele Feronato
Akihiro Matsuura
Luma

Commissioning Editor
Veena Pagare

Acquisition Editor
Sam Wood

Content Development Editor
Samantha Gonsalves

Technical Editor
Manan Patel

Copy Editor
Deepa Nambiar

Project Coordinator
Sanchita Mandal

Proofreaders
Maria Gould
Julie Jackson
Joyce Littlejohn

Indexer
Priya Sane

Production Coordinator
Komal Ramchandani

Cover Work
Komal Ramchandani

Foreword

I really like this book and the work Raydelto did writing it. This book teaches you from scratch how to develop games using Cocos2d-x and how the JNI works. I know this book will be useful for anyone who is new in this framework or has experience in it. I read each chapter and I can say that Raydelto wrote each chapter well, so I'm sure this book contains his passion and each reader will enjoy and learn because it is an excellent book.

Carlos Piñan
CTO, Vitrum Games

About the Author

Raydelto Hernandez is a software engineer and university professor living in Santo Domingo, Dominican Republic. He has developed games for BlackBerry 10, Android, and iOS. He currently teaches topics related to game development at INTEC University.

At the age of 22, he started teaching at Las Americas Institute of Technology, where he earned the Professor of the Year award many times. Then he the became director of the software development center. He has also taught at UNAPEC University, UNIBE University, and INTEC University, where he is currently a research professor.

He is a BlackBerry Certified Builder, Zend Certified Engineer, Certified Java programmer, CompTIA A+ professional, and Microsoft Technology Associate.

Raydelto has posted on YouTube dozens of video tutorials on different programming environments, which are also available on his website at `www.raydelto.org`. Here, you can also find information about his open source projects.

Raydelto has founded two technology companies, Soluciones GBH (in 2005) and AST Technology (in 2008), which develop software for all platforms.

About the Reviewers

Alejandro Duarte has been coding since 1994, when he was 13 years old, using languages such as Basic, C, C++, Assembler, C#, PHP, Java, Groovy, and Lua. He focuses mainly on Java technologies for enterprise applications, but always keeps an eye on the video game development industry and technologies. He has collaborated with several open source and closed source projects based on Cocos2d-x, Processing, Irrlicht Engine, and the Allegro library.

Alejandro has worked for companies in Colombia and the United Kingdom as a software developer and consultant. He is the author of *Vaadin 7 UI Design by Example: Beginner's Guide*, and a technical reviewer for *Cocos2d-x by Example Beginner's Guide*. He maintains several open source projects hosted on GitHub (`github.com/alejandro-du`).

You can contact him at `alejandro.d.a@gmail.com` or through his personal blog (`www.alejandrodu.com`). If you are feeling social, you can follow him on Twitter at `@alejandro_du`.

Emanuele Feronato has been studying programming languages since the early 1980s, with a particular interest in game development. He taught online programming for the European Social Fund (ESF), and then founded a web development company in Italy.

As a game developer, he has developed Flash games sponsored by the biggest game portals, and his games have been played more than 90 million times. He is currently porting most of them on mobile platforms as well as developing HTML5 games that have been featured in the most prominent mobile web markets, such as Amazon.

Emanuele has worked as a technical reviewer for Packt Publishing and published the books *Flash Game Development by Example*, *Box2D for Flash Games*, and *Learning Cocos2d-JS Game Development*.

His blog, `www.emanueleferonato.com`, is one of the most visited blogs about indie game development.

First, I would like to thank Packt Publishing for giving me the opportunity to review this book, especially Richard Harvey, Azharuddin Sheikh, and Samantha Golsalves for helping me to improve this book's quality.

The biggest thank you obviously goes to my blog readers and my Facebook fans for appreciating my work and giving me the will to write more and more. Also, thank you for playing my games, and I hope you will enjoy playing them as much as I enjoyed developing them.

Finally, a special thank you goes to my wife, Kirenia, who patiently waited for me late at nights while I was reviewing the book.

Akihiro Matsuura has 3 years of experience as a Cocos2d-x developer. He founded his own company called Syuhari, 5 years ago. He also has more than 20 years of experience as a programmer. He has written three technical books in Japanese.

I wish to thank the author and Packt Publishing for giving me the opportunity to review this book.

Luma has several years of experience on iOS and Android, and he focuses on game development for mobile platforms. He is the creator of WiEngine, Cocos2dx-better, and Cocos2dx-classical. His GitHub page can be found at `https://github.com/stubma`.

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 **v**

Chapter 1: Setting Up Your Development Environment **1**

Cocos2d-x overview **2**
Setting up Java **2**
Setting up the Android SDK **3**
Downloading the Android SDK 3
Downloading Eclipse 4
Setting up the Eclipse ADT plugin 5
Setting up the Android Native Development Kit **5**
Setting up Apache Ant **6**
Setting up Python **6**
Setting up Cocos2d-x **7**
Creating your first project 8
Setting up the Eclipse IDE 9
Template code walk-through **10**
Java classes 10
The Android application configuration 11
C++ classes 11
Scenes 12
Summary **13**

Chapter 2: Graphics **15**

Creating scenes **15**
Understanding Layers 16
Using the director 16
Pausing the game **17**
Organizing our resources files 17
Creating our pause scene header file 18

Creating the pause scene implementation file 18
Transitions 20
Understanding nodes 21
Understanding sprites 21
Creating sprites 21
Positioning sprites 21
Setting anchor points 22
Understanding the Cocos2d-x coordinate system 22
Adding sprites to the scene 24
Positioning sprites outside the visible area 25
Positioning the player sprite 26
Understanding actions 26
Moving sprites 26
Creating sequences 27
Animating sprites 28
Improving performance with sprite sheets 29
Game menus 29
Handling multiple screen resolutions 30
Putting everything together 31
Summary 34
Chapter 3: Understanding Game Physics 35
Setting up the physics world 36
Collision detection 38
Handling gravity 40
Handling physics properties 41
Applying velocity 41
Linear damping 42
Applying force 42
Applying impulse 42
Applying torque 44
Putting everything together 44
Summary 47
Chapter 4: User Input 49
Understanding the Event Dispatcher mechanism 49
Handling the touch event 50
Handling multi-touch events 53
Handling accelerometer events 55
Keeping the screen alive 56
Handling the Android back key pressed event 57
Putting everything together 58
Summary 63

Chapter 5: Handling Text and Fonts **65**
Creating TrueType font labels **65**
Creating our GameOverScene 66
Calling our GameOverScene when a player loses 67
Customizing the GameOverScene 67
Adding label effects 67
Creating system fonts **69**
Creating bitmap font labels **70**
Adding more bombs to our game 70
Putting everything together **73**
Summary **79**
Chapter 6: Audio **81**
Playing background music and sound effects **81**
Modifying audio properties **83**
Handling audio when leaving the game 83
The new audio engine **84**
New features included in the new audio engine 85
Adding a mute button to our game **86**
Putting everything together **89**
Summary **95**
Chapter 7: Creating Particle Systems **97**
Creating collections of Cocos2d-x objects **97**
Exploding bombs 99
Adding particle systems to our game **100**
Configuring particle systems **102**
Creating customized particle systems **104**
Putting everything together **107**
Summary **112**
Chapter 8: Adding Native Java Code **113**
Understanding the Cocos2d-x structure for the Android platform **113**
Understanding the JNI capabilities **114**
Adding Java code to the Cocos2d-x game **116**
Adding ads to the game by inserting Java code **119**
Configuring the environment **120**
Modifying the Android manifest **123**
Adding the Java code 123
Putting everything together **125**
Summary **134**
Index **135**

Preface

Cocos2d-x is the most used open source game framework. It has official support from Microsoft for their mobile and desktop platform, and its small core runs faster than other frameworks, allowing it to run on low-end Android devices with outstanding performance. It is currently maintained by an active open source development community led by the author of the original Cocos2d for iPhone and Chukong Technologies.

This introductory book will guide you through the steps for creating a simple two-dimensional game for Android from scratch. During this journey, you will learn the fundamentals of the Cocos2d-x C++ multiplatform game framework and how to handle sprites, add physics to your games, play sounds, display text, generate realistic explosions using particle systems, and add native Android functionalities using Java Native Interface (JNI).

What this book covers

Chapter 1, *Setting Up Your Development Environment*, guides you step by step for configuring Cocos2d-x and all its prerequisites.

Chapter 2, *Graphics*, covers handling backgrounds, sprites, animating them, and boosting their performance using sprite sheets.

Chapter 3, *Understanding Game Physics*, demonstrates the basics of the new Cocos2d-x physics engine based on Chipmunk, which was introduced in Cocos2d-x version 3.0. We will create physics-based bodies, add gravity to them, and detect collisions.

Chapter 4, *User Input*, is where we add interaction to our game, allowing it to interact with the user with the help of touch listeners and the accelerometer.

Chapter 5, *Handling Text and Fonts*, proves that handling text is crucial for game development. Irrespective of the complexity of your game, the odds are that you will display information, sometimes with foreign character sets. This chapter shows you how to use simple true-type fonts and more stylized bitmap fonts to make your game look more professional.

Chapter 6, *Audio*, shows that part of the emotion of playing a game comes from the music and sound effects. Within this chapter, you learn how to add background music and sound effects to your game using the CocosDenshion audio engine, which has been present since the original Cocos2d iPhone game engine. This chapter also covers how to play media using the new audio engine and highlights the major differences between them.

Chapter 7, *Creating Particle Systems*, illustrates the creation of realistic explosions, fire, snow, rain using the built-in particle systems engine. This chapter shows you how to create your own particle system when you require a customized effect, using the most popular tools.

Chapter 8, *Adding Native Java Code*, helps you when you need to add native code for creating and invoking Android-specific behavior from within your Cocos2d-x game activity. We do this using the Java Native Interface (JNI) mechanism, available on the Android platform.

What you need for this book

In order to follow this book's narrative and be able to reproduce all the steps, you will need a PC with Windows 7 or higher, any Linux distribution or a Mac running the operating system, and OS X 10.10 Yosemite. Most of the tools that we'll use throughout the book are free to download. We've explained how to download and install them.

Who this book is for

This book was written for people with little or no experience in game programming, and with notions of the C++ programming language, who are willing to create their first Android game in a very comprehensive way.

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 folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows: "In order to add accelerometer support to our game, we are first going to add the following method declaration to our `HelloWorldScene.h` header file."

A block of code is set as follows:

```
void HelloWorld::movePlayerByTouch(Touch* touch, Event* event)
{
  Vec2 touchLocation = touch->getLocation();
  if(_sprPlayer->getBoundingBox().containsPoint(touchLocation)){
    movePlayerIfPossible(touchLocation.x);
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Size screenSize = glview->getFrameSize();
Size designSize(768, 1280);
std::vector<std::string> searchPaths;
searchPaths.push_back("sounds");
```

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

```
cocos new MyGame -p com.your_company.mygame -l cpp -d NEW_PROJECTS_DIR
```

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: "Clicking on the **Next** button moves you to the next screen."

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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 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.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Setting Up Your Development Environment

In this chapter, we will explain how to download and set up all the required tools to get you started with setting up an environment for building games for the Android platform. Although there are huge similarities between the Mac OS and the Windows development environment, we will cover all the details regarding the installation in both of these operating systems.

The following topics will be covered in this chapter:

- Cocos2d-x overview
- Setting up Java
- Setting up the Android SDK
- Setting up the Android Native Development Kit (NDK)
- Setting up Apache Ant
- Setting up Python
- Setting up Cocos2d-x
- Setting up the Eclipse IDE
- Template code walk-through

Cocos2d-x overview

Cocos2d-x is a C++ cross-platform port of the popular iOS gaming framework Cocos2d. It was first released in November 2010, and bought in 2011 by Chukong Technologies, a Beijing-based mobile gaming company. Nevertheless, it is still maintained by an active community of more than 400,000 developers worldwide, including Ricardo Quesada, the creator of the original Cocos2d iPhone engine.

This framework encapsulates all the game details, such as sound, music, physics, user inputs, sprites, scenes, and transitions, so the developer will only have to focus on the game logic rather than re-inventing the wheel.

Setting up Java

The Android platform technology stack is based on the Java technology; that is why the first item to be downloaded will be the Java Development Kit (JDK). Although Java JDK 8 is the latest version at the time of writing this book, it is not officially supported by all Android versions, so we will download JDK 6, all the template Java codes generated by Cocos2d-x can be successfully compiled with this version.

Java Runtime Environment (JRE) is not enough for building the Android applications, since it only contains the files required for running the Java applications, but it does not contain the tools required for building the Java applications.

You can download the JDK 6 from Oracle at `http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase6-419409.html` regardless of your development environment.

If Windows is your current environment, then after installing JDK you are required to add the path of the binaries folder to the PATH environment variable. This path will look like this: `C:\Program Files\Java\jdk1.6.0_45\bin`.

Open a new system console and type `javac -version`, if Java compiler's version number is displayed, then you have successfully installed JDK in your system.

JDK 7 is required for building the applications for Android 5.0 and higher. You should download this version if you are targeting the latest Android versions. But, if you want your game to be compatible with the Android versions that are older than 4.4, then you should pick JDK 6.

Setting up the Android SDK

The Android SDK contains all the required command line tools for building an Android application. It has versions for Windows, Mac, and GNU/Linux operating systems.

Android Studio is now the only officially supported IDE; nevertheless, Cocos2d-x 3.4 provides only out-of-the-box support for Eclipse, which was the former official IDE for Android development. It is no longer available for downloading, since it is not in active development any more, but you may download Eclipse manually and install the **Android Development Tools** (**ADT**) by following the steps below.

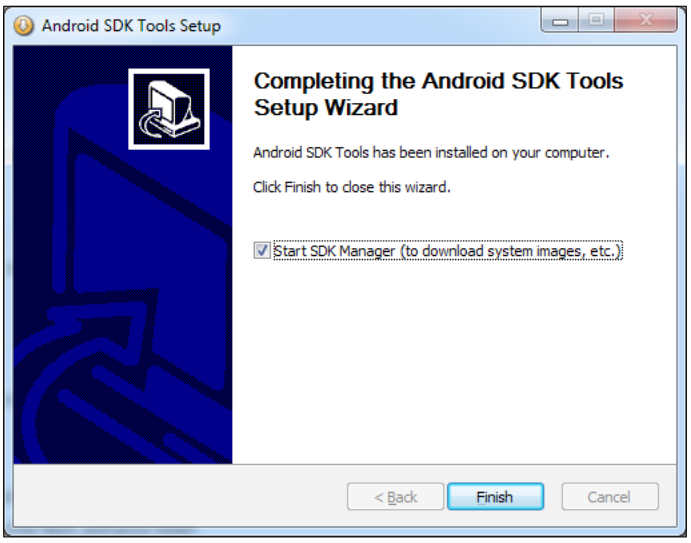
Downloading the Android SDK

You can download Android SDK from the link: `http://developer.android.com/sdk`. At the bottom of the page, under **Other Download Options**, you will find the option for downloading the SDK tools. Choose the version that matches your operating system.

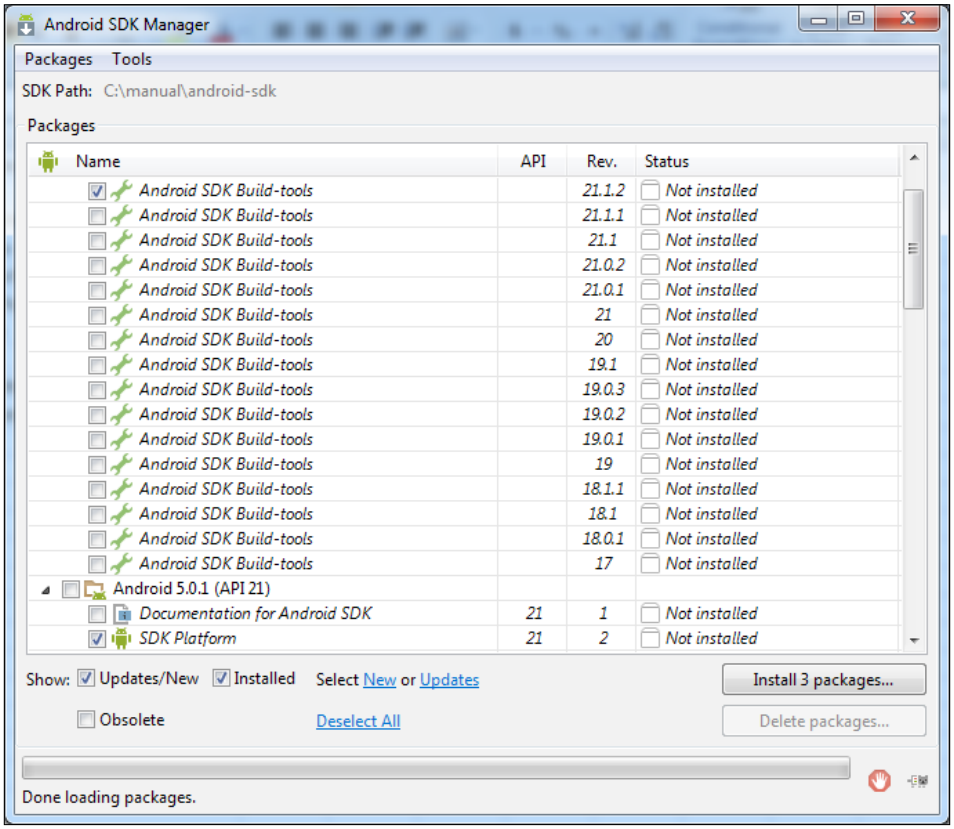
At the time of writing this book, the latest version of SDK was 24.0.2.

Run the Android SDK Installer and install the Android SDK on your computer.

When the Android SDK finishes installing, it is not yet ready to build the Android apps. So, at the final screen of the installation wizard, mark the checkbox for **Start SDK Manager** so you can download the required components for building your games, as shown in the following screenshot:



Once the Android SDK Manager starts, select **Android SDK Platform-tools** and **Android SDK Build-tools** from the `Tools` folders. Then select **SDK Platform** from your desired API level, as shown in the following screenshot:



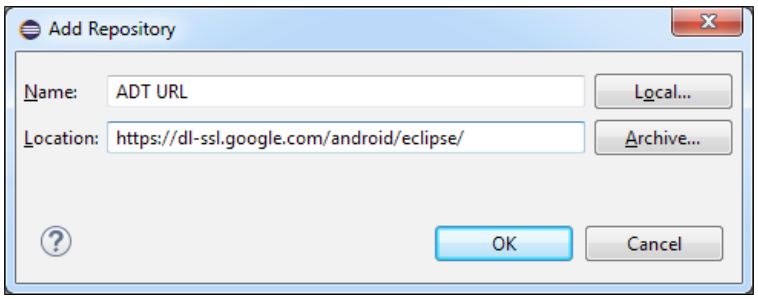
Downloading Eclipse

Download the latest version of the Eclipse IDE for Java Developers from `http://www.eclipse.org/downloads`. It will suggest the download versions compatible with your current operating system, select the version that better suits your operating system platform which will either be 32-bit or 64-bit.

At the time of writing this book, Eclipse Luna (4.4.1) was the latest version.

Setting up the Eclipse ADT plugin

Open Eclipse and navigate to **Help** | **Install new Software** and add the Eclipse ADT download location, which is `https://dl-ssl.google.com/android/eclipse/`, as shown in the following screenshot:



Click on **OK**, then select the **Developer Tools** checkbox, and click on **Next** in order to finish the ADT installation wizard.

Setting up the Android Native Development Kit

We have already downloaded the Android SDK that allows you to create Android applications using the Java Technology; nevertheless, the Cocos2d-x framework is written in C++, so you will need the Android Native Development Kit (NDK) in order to build the C++ code for the Android platform.

Android's official documentation clearly states that you should use this native kit for specific circumstances, but you should not use it just because you are familiar with the C++ language, or because you want your application to perform faster. The manufacturer makes this suggestion because the Android core API is only available for Java.

Download the latest NDK revision. At the time this book was written, it was 10d. This version of NDK will allow you to build for all the Android platforms, including the latest.

You can download the latest version of the Android NDK for all the platforms from the following link:

`https://developer.android.com/tools/sdk/ndk`

After downloading it, run the executable file. It will decompress the Android NDK directory on the current path; you need to be aware of this path since you will need it later.

Setting up Apache Ant

Apache Ant is a build management tool widely used for automating the Java projects build process. It has been introduced in Cocos2d-x 3.0 for building the framework for the Android platform. It has made the Android build process simpler and enhanced the cross-platform build. Back in Cocos2d-x 2.x, building the Android apps within the Windows operating system required simulating the UNIX environment by using Cygwin. This required minor hacks for successfully building the code, many of them still remain undocumented on the official Cocos2d-x site.

This tool can be downloaded from the link: `https://www.apache.org/dist/ant/binaries/`

At the time of writing this book, version 1.9.4 was the latest. This tool is a cross-platform tool so a single download will work on any operating system that provides support for the Java technology.

In order to install this tool, just unzip the file. Remember the path since you will need it during the Cocos2d-x setup process.

Setting up Python

All the Cocos2d-x configuration files are written in Python. If you are using Mac OS or any Linux distribution, it will already be installed on your OS. So, you can skip this section.

If you are using Windows, you need to download Python 2 from the following link: `https://www.python.org/ftp/python/2.7.8/python-2.7.8.msi`.

Take in to consideration that Python, as Cocos2d-x, keeps simultaneous support for versions 2 and 3. Cocos2d-x only supports Python 2. At the time of writing this book the latest version of the 2 branch was 2.7.8.

After the installer finishes with the setup, you should manually add the Python installation path to the PATH environment variable. The default installation path is `C:\Python27`.

Open a new system console and type `python`, if the Python console is shown, as seen in the following screenshot, then it means that Python has been installed correctly:

For setting an environment variable on Windows, click on the **Start** button and type: `edit the system environment` variables, click on it and hit the **Environment Variables** button, and then the environment variables configuration dialog will be displayed.

Setting up Cocos2d-x

Now that you have all the Cocos2d-x pre-requisites for building your first game for the Android platform, you are going to download the Cocos2d-x 3.4 framework and set it up with the help of the following steps:

1. You can download the source code from `http://www.cocos2d-x.org/download`. Be aware that this page also has the link for downloading the Cocos2d-x branch 2, which is not covered in this book, and the manufacturer has officially announced that the new features will only be available in branch 3.
2. After downloading the compressed Cocos2d-x source code, uncompress it to your desired location.
3. In order to configure Cocos2d-x, open your system terminal and point to the path where you have uncompressed it, and type `setup.py`. It will require you to specify `ANDROID_NDK_PATH`, here you will specify the root directory of the NDK that you have previously uncompressed in the previous sections. Secondly, it will require you to specify `ANDROID_SDK_ROOT`, here you will specify the directory path from where you have chosen to install the Android SDK during the installation process. Then, it will require you to set `ANT_ROOT`, where you will specify the root directory of your ant installation. Finally, close the terminal, and open a new one so that the changes can take effect.

Creating your first project

Now, Cocos2d-x is set up, and it is ready for creating your first project. You can do so by typing the following command:

```
 cocos new MyGame -p com.your_company.mygame -l cpp -d
NEW_PROJECTS_DIR
```

This script has created an Android template code for your game that will run in all the Android devices containing the Android API 9 or higher, that is Android 2.3 (Gingerbread) and later.

Take in to consideration that the package name should contain exactly two dots, as the example shows, if it has less or more, then the project creation script will not work. The `-l cpp` parameter means that the new project is going to use C++ as the programming language, which is the only one that is covered in this book.

Cocos2d-x 3.x, as opposed to branch 2.x, allows you to create your project outside of the framework directory structure. Therefore, you can create your project at any location, and not just inside the `projects` directory, as it was in the previous versions.

It will take a while, since it will copy all the framework files to your new project's path. After it finishes, plug your Android device to your computer, then you can easily run the template `HelloWorld` code by typing the following command within your new project's path:

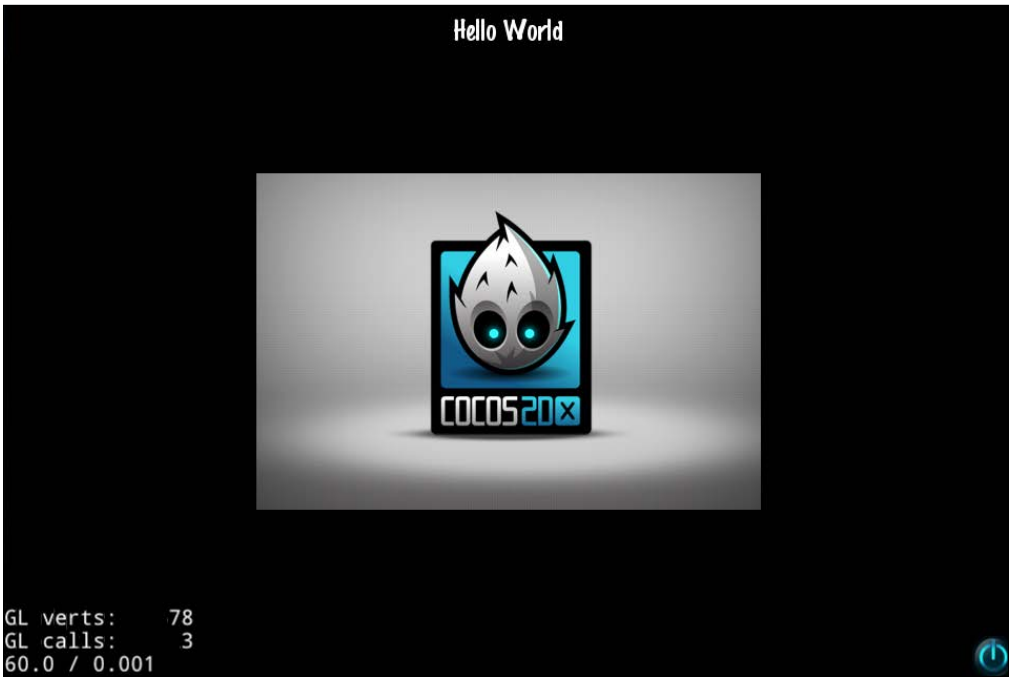
```
cocos run -p android
```

Alternatively, you could run the following command regardless of your current path on the terminal:

```
cocos run -p android /path/to/project
```

For building and running Cocos2d-x 3.4 for Windows, you will need Microsoft Visual Studio 2012 or 2013.

Now, you should be able to see the Cocos2d-x logo and a text that says **Hello World**, as we can see in the following image:



Setting up the Eclipse IDE

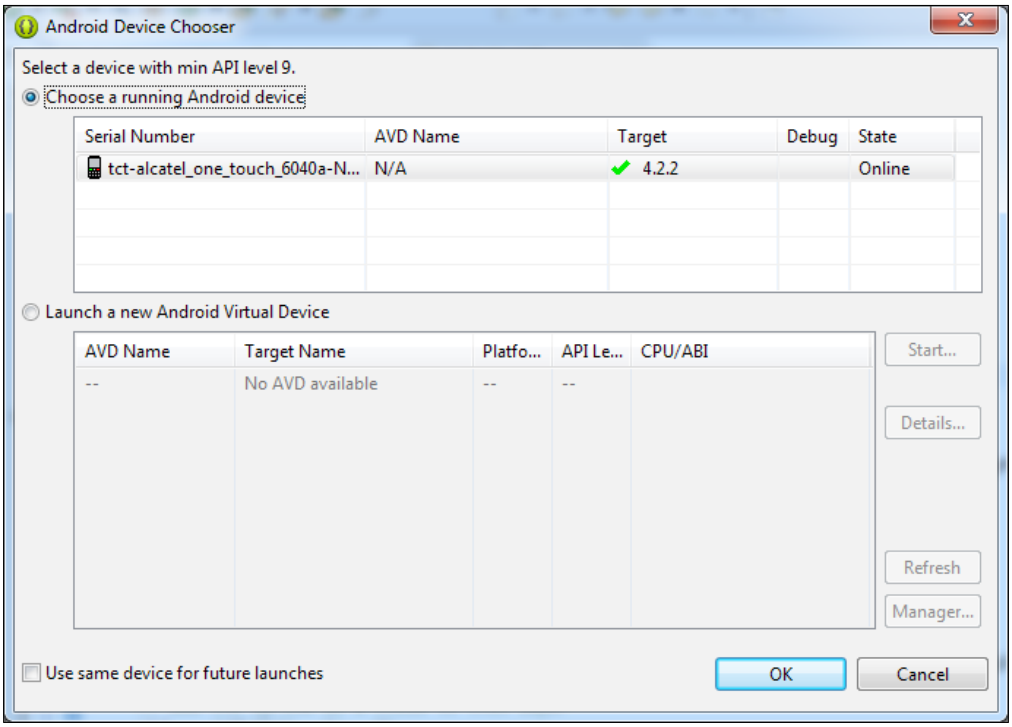
Cocos2d-x branch 3 has improved the Android building process significantly.

Back in branch 2, it was necessary to manually configure many environment variables within IDE, import many core projects, and handle dependencies. Even after completing all the steps, the Cygwin Windows UNIX port integration with Eclipse was never polished to work flawlessly, so minor hacks were required.

Building Cocos2d-x 3.4 within Eclipse is as simple as importing the project and clicking on the **Run** button. In order to achieve this, within the ADT, navigate to **File** | **Import** | **General** | **Existing Projects into Workspace**, select the path, where Cocos2d-x has created the new project from the previous sections. Then click on **Finish**.

Cocos2d-x Android template project is created using the API Level 10 as target platform. If you don't have this version installed on your system, you should change it by right-clicking on the project from the package explorer, click on **Properties**, and select your preferred installed Android API version from the **Project Build Target** box.

Now, right-click on the project name in the package explorer, click on **run as,** and finally, click on **Android Application**. The following pop up will be displayed, and it will require you to specify the Android device on which you want to launch the Cocos2d-x game:



After picking your Android device, you will see the HelloWorld game scene as it was shown when we ran the **Run** command in the previous section.

Template code walk-through

In this section, we will explain the main parts of the Cocos2d-x template code generated in the previous sections by the project creation script.

Java classes

We now have one Java class in our project named `AppActivity` that has no members and extends the `Cocos2dxActivity` class from the core library. We can also see that 22 Java classes from the core library have been referenced in the project. This code is aimed to make our C++ code work, and we don't have to modify it at all.

Android application configuration

The generated `AndroidManifest.xml`, which is the android configuration file, requires the permission `android.permission.INTERNET`, which allows your Android application to use the Internet connection on your device; nevertheless, this is not needed by our simple game code since there is no Internet interaction. So, you may delete this line from the `AndroidManifest.xml` file if you wish. Your game will be shown in landscape by default, but if you wish to create a game that runs in portrait mode, then you should change the `android:screenOrientation` value from `landscape` to `portrait`.

In order to change the Android application name, you may modify the `app_name` value located on the `strings.xml` file; it will affect the launcher icon's text and the application identifier within the Android OS.

When you are creating your own games, you will have to create your own classes, and those will often be more than the two classes that were created by the script. Every time you create a new class, you need to add its name to the `LOCAL_SRC_FILES` property of the `Android.mk` make file located inside the `jni` folder of your new project directory structure. So, when your `cpp` code is built by the C++ build tools, it knows which files it should compile.

C++ classes

Two C++ classes have been created: `AppDelegate` and `HelloWorldScene`. The first one is in charge of launching the Cocos2d-x framework and passing the control to the developer. The framework loading process happens within this class. If the Cocos2d-x core framework is successfully launched on the target device, it will run the `applicationDidFinishLaunching` method, which is the first game-specific function to be run.

The code is very straightforward and it is documented such that you will be able to grasp its logic easily. Our first minor change to the code will be to hide the debug information that shows by default on this sample game. You could simply guess that in order to achieve this you should only send `false` as a parameter for the `setDisplayStats` method call in the `director` singleton instance, as we can see in the following code listing:

```
bool AppDelegate::applicationDidFinishLaunching() {
    // initialize director
    auto director = Director::getInstance();
    auto glview = director->getOpenGLView();
    if(!glview) {
```

```
            glview = GLViewImpl::create("My Game");
            director->setOpenGLView(glview);
        }
        // turn on display FPS
        director->setDisplayStats(false);
        // set FPS. the default value is 1.0/60 if you don't call this
        director->setAnimationInterval(1.0 / 60);
        // create a scene. it's an autorelease object
        auto scene = HelloWorld::createScene();
        // run
        director->runWithScene(scene);
        return true;
    }
```

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.

Scenes

As we will cover in the chapters later in this book, Cocos2d-x handles the scene concept just like movies; movies are composed by scenes, so are the Cocos2d-x games. We can visualize the different screens, such as loading, main menu, world selection, gameplay levels, ending credits, and so on, as the different scenes. Each scene has a class that defines its behavior. The template code has only one scene named `HelloWorld` scene that is initialized and launched from within the `AppDelegate` class. The scene flow is managed by the game director as we have seen in the previous code. The `Director` class has all the basic features to drive the game, just like a director does during a movie. There is a single shared instance of the director class that is used within the whole application scope.

`HelloWorldScene` contains the layer that represents all the visible areas that show up when we run our HelloWorld application, that is, the hello world label, the Cocos2d-x logo, and the menu showing the exit option.

Within the `init` method, we do the instantiation of the visual elements, and then we add it to the scene using the `addChild` method inherited from the `Node` core class.

Summary

In this chapter, we have introduced the Cocos2d-x 3.4 gaming framework, and explained how to download and install it. We have also explained all of its pre-requisites. We have configured our work environment, launched our first Android application into an actual device, and gone through a quick code overview of the main aspects of the template code generated by the script.

In the next chapter, we will cover how to create and manipulate all of our game graphics, such as the main character, enemies, obstacles, backgrounds, and so on.

2
Graphics

In this chapter, we will cover how to create and handle all your game graphics. We will create the scenes, the transitions between those using the game director, create sprites, locate them in the desired position, move them around using actions, and bring our characters to life using animation.

The following topics will be covered in this chapter:

- Creating scenes
- Understanding nodes
- Understanding sprites
- Understanding actions
- Animating sprites
- Adding the game menus
- Handling multiple screen resolutions

Creating scenes

The scene concept is very important within the Cocos2d-x game engine, since all the displayed screens in our game are considered scenes. Creating an analogy between Cocos2d-x and the Android native Java development, we can say that a Cocos2d-x scene is equivalent to what Android calls activity.

In the previous chapter we introduced the `AppDelegate` class, and we explained that it has the responsibility of loading the framework on the device and then executing the game-specific code. This class contains the `ApplicationDidFinishLaunching` method, which is the entry point of our code. In this method, we instantiate the scene that is going to first be displayed in our game, and then request the `director` to load it, as we can see in the following code listing:

```
bool AppDelegate::applicationDidFinishLaunching() {
    auto director = Director::getInstance();
  // OpenGL initialization done by cocos project creation script
    auto glview = director->getOpenGLView();
    auto scene = HelloWorld::createScene();
    director->runWithScene(scene);
    return true;
}
```

All the C++ code is run in a single Android activity; nevertheless, we can add native activities to our game.

Understanding Layers

Scene, in itself, is not an object container, that is why it should contain at least one instance of the `Layer` class so that we can add objects to it. This layer creation process has been encapsulated in the framework of macro `CREATE_FUNC`. You just have to invoke the macro and pass the name of the class as a parameter, and it will generate the Layer creation code.

Layer manipulation had multiple uses related to event handling in the previous versions of the framework; nevertheless, the event dispatcher engine was completely rewritten in version 3.0. The only reason why the Layer concept still exists in Cocos2d-x 3.4 is compatibility. The framework creators officially announced that they may remove the Layer concept in further versions.

Using the director

Scenes are controlled by the Cocos2d-x director, which is a class that handles our game flow. It applies the singleton design pattern, which ensures that there is only one instance of the class. It controls the kind of scene that should be presented through a scene stack, similar to how Android handles scenes.

This means that the last scene pushed to the stack is the one that is going to be presented to the user. When the scene is removed, the user will be able to see the scene that was previously visible.

When we are using the single director instance more than once in a single function, we can store its reference on a local variable as follows:

```
auto director = Director::getInstance();
```

We can also store it on a class attribute so that it is accessible from all over the class. This will allow us to type less, and it will also represent a performance improvement, so that we are not making several calls to the `getInstance` static method each time we want to access the singleton instance.

Director instance can also provide us with useful information, such as screen dimensions, and debug information, which is enabled in our Cocos project by default.

Pausing the game

Let us start and create our game. The first feature that we are going to add is the functionality for pausing and resuming our game. Let's start building – we'll start by setting up the screen that will appear when we pause the game.

We will achieve this by adding a new pause scene to the scene stack. When this screen is removed from the stack, the HelloWorld scene will show up because it was the displayed screen before the pause scene was pushed into the scene stack. The following code listing shows how we can easily pause our game:

Organizing our resources files

When we created our Cocos2d-x project, some resources such as images and fonts have been added by default to the `Resources` folder of our project. We are going to organize them, so that it is easier to handle them. For that matter, we are going to create an `Image` folder in the `Resources` directory. In this new folder, we are going to put all our images. Later on in this chapter, we will explain how we are going to organize the different versions of each image according to the Android device screen resolution.

Within the resources bundled with this chapter, we have provided you with the images that you will need in order to build the code for this chapter.

Creating our pause scene header file

First, let us create our pause scene header file. We have created it using the `HelloWorld.h` header file as a reference:

```
#ifndef __Pause_SCENE_H__
#define __Pause_SCENE_H__

#include "cocos2d.h"

class Pause : public cocos2d::Layer
{
public:
    static cocos2d::Scene* createScene();
    virtual bool init();
  void exitPause(cocos2d::Ref* pSender);
    CREATE_FUNC(Pause);
private:
  cocos2d::Director *_director;
  cocos2d::Size _visibleSize;
};

#endif // __Pause_SCENE_H__
```

You can avoid typing `cocos2d` each time you refer to the Cocos2d-x class contained in the `cocos2d` namespace by typing `using namespace cocos2d`; nevertheless, using it in a header file is considered a bad practice, because the code may fail to compile it when there are any repeated field names within all the included namespaces.

Creating the pause scene implementation file

Now, let us create our pause scene implementation file. Analogous to what we did in the previous section, we will create this file based on the `HelloWorld.cpp` file created by the project creation script.

Within the following code, you will find the menu creation code bundled in the Cocos2d-x template project. We will explain how to create the game menus in a further section of this chapter, you will also learn font creation, and this will be explained in detail in *Chapter 5*, *Handling Texts and Fonts*.

```
#include "PauseScene.h"

USING_NS_CC;

Scene* Pause::createScene()
{
    auto scene = Scene::create();
    auto layer = Pause::create();
    scene->addChild(layer);
    return scene;
}

bool Pause::init()
{
    if ( !Layer::init() )
    {
      return false;
    }
  _director = Director::getInstance();
  _visibleSize = _director->getVisibleSize();
  Vec2 origin = _director->getVisibleOrigin();
  auto pauseItem = MenuItemImage::create("play.png",
  "play_pressed.png", CC_CALLBACK_1(Pause::exitPause, this));
  pauseItem->setPosition(Vec2(origin.x + _visibleSize.width -
  pauseItem->getContentSize().width / 2,
  origin.y + pauseItem->getContentSize().height / 2));
  auto menu = Menu::create(pauseItem, nullptr);
  menu->setPosition(Vec2::ZERO);
  this->addChild(menu, 1);
  auto label = Label::createWithTTF("PAUSE", "fonts/Marker
  Felt.ttf", 96);
  label->setPosition(origin.x + _visibleSize.width/2,
  origin.y + _visibleSize.height /2);
  this->addChild(label, 1);
  return true;
}

void Pause::exitPause(cocos2d::Ref* pSender){
  /*Pop the pause scene from the Scene stack.
  This will remove current scene.*/
  Director::getInstance()->popScene();
}
```

In the generated `HelloWorldScene.h` scene, we are now adding the following line of code after the definition of the `menuCloseCallback` method:

```
void pauseCallback(cocos2d::Ref* pSender);
```

Now, let us create the implementation for the `pauseCallBack` method in the `HelloWorldScene.cpp` implementation file:

```
void HelloWorld::pauseCallback(cocos2d::Ref* pSender){
  _director->pushScene(Pause::createScene());
}
```

Finally, modify the `closeItem` creation by making it call the `pauseCallBack` method rather than the `menuCloseCallBack` method, so this line will look like this:

```
    auto closeItem = MenuItemImage::create("pause.png",
"pause_pressed.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
this));
```

Now we have created a simple pause scene, and this is pushed to the scene stack when the close button is pressed and it is closed when the blue button is pressed from within the pause scene.

We will now add the `PauseScene.cpp` file to the android makefile named `Android.mk` located on the `jni` folder of your eclipse project on the `LOCAL_SRC_FILES` section above `HelloWorldScene.cpp`.

Transitions

The director is also responsible for playing the transitions while swapping scenes, Cocos2d-x 3.4 currently offers more than 35 different scene transition effects, such as fades, flips, page turn, split, and zoom among others.

Transition is a subclass of the `Scene` class, which means that you can pass a transition instance to any method that receives a scene object, such as `runWithScene`, `replaceScene`, or `pushScene` methods from the `director` class.

Let us use a simple transition effect, when passing from the gameplay scene to the pause scene. We will simply do this by creating a new instance of the `TransitionFlipX` class and passing it to the director's `pushScene` method:

```
void HelloWorld::pauseCallback(cocos2d::Ref* pSender){
  _director->pushScene(TransitionFlipX::create(1.0,
  Pause::createScene()));
}
```

Understanding nodes

Node represents all the visible objects on the screen, it is, in fact, the superclass of all the scene elements, including the scene itself. It is the base framework class, and it has the basic methods that allow you to handle graphics characteristics, such as position and depth.

Understanding sprites

In our game, the sprites represent the images of our scenes, just like the background, the enemies, and our player.

Later in *Chapter 4*, *User Input*, we will add event listeners to the scenes, so that it can interact with the user.

Creating sprites

It is very easy to instantiate the Cocos2d-x core classes. We have seen that the `scene` class has a `create` method; similarly, the `sprite` class has a static method with the same name, as we can see in the following code snippet:

```
auto sprBomb = Sprite::create("bomb.png");
```

Cocos2d-x currently supports PNG, JPG, and TIF image formats for sprites; nevertheless, it is highly recommended that we use the PNG images, because of its transparency capabilities, which are not present in either the JPG or the TIF format, and also because of the image quality that is provided by this format in a fair file size. That is why you will see that all the Cocos2d-x-generated templates and samples use this image format.

Positioning sprites

Once we have created our own sprite, we can easily position it on the screen by using the `setPosition` method, but before doing it, we will explain the anchor point concept.

Setting anchor points

All sprites have a reference point called the **anchor point**. When we position a sprite using the `setPosition` method, what the framework actually does is that it sets the specified two-dimensional position to the anchor point so that it affects the whole image. By default, the anchor point is set to the sprite's center, as we can see in the following image:

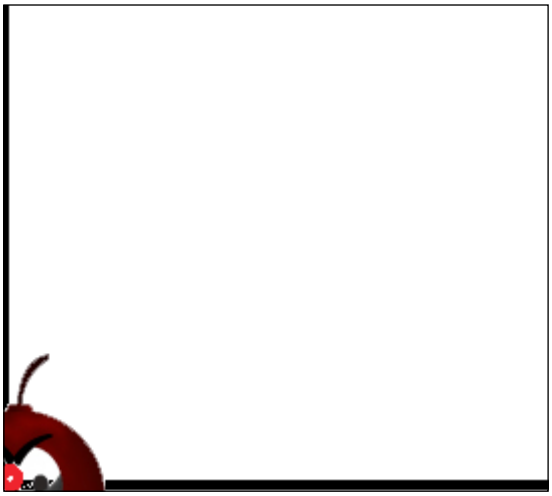
Understanding the Cocos2d-x coordinate system

The Cocos2d-x coordinate system, unlike most of the computerized graphic engines, has its origin (0,0) at the lower left part of the screen, as we can see in the following image:

So if we want to position our sprite at the origin point (0,0), we can achieve it by calling the `setPosition` method contained in the sprite class. It is overloaded, so it can receive either two floats indicating x and y position, a `Point` class instance, or a `Vec2` instance. Although the `Vec2` instances are used in the generated code, the official Cocos2d-x documentation states that passing the floating numbers is up to 10 times faster.

```
sprBomb -> setPosition(0,0);
```

After executing this code, we can see that only the upper right zone of the sprite is visible, which represents only 25 percent of its size, as we have shown in the following image:

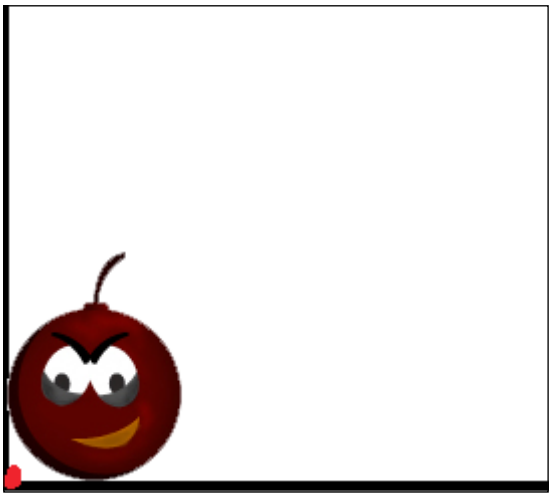


So, in case you would like the sprite to be shown at the origin, you have many options, such as positioning the sprite at the point that corresponds to half of the sprite's height and half of its width, which could be determined by using the sprite method, `getContentSize`, which returns a size object that has the sprite height and width properties. Another option, which might be easier than this, is to reset the sprite anchor point to (0,0) ; so that when the sprite is positioned at the origin of the screen, it is completely visible and located at the lower leftmost region of the screen. The `setAnchorPoint` method receives a `Vec2` instance as the parameter. In the following code listing, we are passing a `Vec2` instance pointing to the origin (0,0):

```
sprBomb -> setAnchorPoint(Vec2(0,0));
sprBomb -> setPosition(0,0);
```

The `Vec2` class has a constructor that receives no parameters, it creates a `Vec2` object with an initial value of 0,0.

When we execute our code, we get the following result:



The reason why the default anchor point location is at the center of the sprite is because it makes it easier to be positioned at the center of the screen.

Adding sprites to the scene

After our sprite object is created and positioned, we need to add it to our scene by using the `addChild` method, which contains two parameters: the pointer to the node that you want to add to the scene and an integer that represents its position in *z* axis. The node with the highest *z* value will be displayed above those with lower values:

```
this->addChild(sprBomb,1);
```

Now let us add the background image to our `HelloWorld` scene: we are going to do it by following the same steps we used in order to position the bomb at the lower left area of the screen in the `init` method:

```
auto bg = Sprite::create("background.png");
bg->setAnchorPoint(Vec2());
bg->setPosition(0,0);
this->addChild(bg, -1);
```

We have now added our background in a *z* position of -1, so any node with a position of 0 or higher will be displayed on top of the background, as we can see in the following image:

Positioning sprites outside the visible area

Now we have a bomb, which doesn't move, at the bottom of the screen. We will now position it at the top center zone of the screen, outside the visible region, so that when we get this sprite to move, it will look like it is raining bombs.

As we previously mentioned, let us position our bomb in the visible area, and then, in the next section we will make it move toward the ground using the actions functionality:

```
sprBomb->setPosition(_visibleSize.width / 2, _visibleSize.height +
sprBomb->getContentSize().height/2);
```

We have removed the `setAnchorPoint` sentence; so, now, the bombs have a default anchor point, and we have modified the `setPosition` statement, so that now, we are positioning it just in the visible area.

Positioning the player sprite

Now, let us create and position our player sprite.

```
auto player = Sprite::create("player.png");
player->setPosition(_visibleSize.width / 2, _visibleSize.height*
0.23);
this->addChild(player, 0);
```

In the previous code we created a player sprite. We used the default anchor point that points directly to the center of the image and centered it horizontally by positioning it to half of the screen width and 23 percent of the screen height, since the background image provided in this book for this chapter was drawn within those proportions. We have added it with a z value of 0, so this means that it will be displayed in the background.

Now let us take care of the bomb, let us position it in the visible area, and then, in the next section, we will make it move it toward the ground using the actions functionality:

```
sprBomb->setPosition(_visibleSize.width / 2, _visibleSize.height +
sprBomb->getContentSize().height/2);
```

We have removed the `setAnchorPoint` sentence; so now, the bombs have the default anchor point, and we have modified the `setPosition` statement, so now, we are positioning it in the visible area.

Throughout this chapter we have used many images, which, as we have previously mentioned are stored in the `Resources` folder of our Cocos2d-x project. You could create sub folders to organize your files.

Understanding actions

We can easily tell our sprites to perform concrete actions, such as jump, move, skew, and so on. It requires a few lines to get our sprites to execute the desired action.

Moving sprites

We can make our sprite move to a specific area of the screen by creating a `MoveTo` action and then making the sprite execute the action.

In the following code listing, we are making the bomb fall to the bottom of the screen by simply writing the following code lines:

```
auto moveTo = MoveTo::create(2, Vec2(sprBomb-
>getPositionX(), 0 - sprBomb->getContentSize().height/2));
sprBomb->runAction(moveTo);
```

We have created a `moveTo` node that will move the bomb sprite to the current horizontal position, but it will also move it to the bottom of the screen until it is not visible. In order to achieve this, we made it move to the y position of the negative half of the height of the sprite. Since the anchor point is set to the center point of the sprite, moving it to the negative half of its height will be enough to make it move outside the screen's visible area.

As you can see, it is crashing with our player sprite, but the bomb just continues downward as it still doesn't detect the collisions. In the next chapter, we will add collision handling to our game.

Cocos2d-x 3.4 has its own physics engine, which includes an easy mechanism to detect the collisions among sprites.

If we want to move our sprite to a position that is relative to its current location, we can use the `MoveBy` class, which receives the parameters of how much we want the sprite to move horizontally and vertically:

```
auto moveBy = MoveBy::create(2, Vec2(0, 250));
sprBomb->runAction(moveBy);
```

You can make the sprite move in the opposite direction by using the `reverse` method.

Creating sequences

Sometimes we have a predefined sequence of actions that we want to execute in several parts of our code, this can be handled by sequences. As the name suggests, it consists of a list of actions that are executed in a predefined order, which can be reversed if necessary.

It is very common to use sequences each time we use actions, so in the sequence we add the `moveTo` node, and then a function call that executes a method after the movement has finished, so that it will allow us to delete the sprite from the memory, reposition it, or do any other common tasks that are required in video games.

In the following code, we are creating a sequence where we first demand the bombs to move to the ground, and we request to execute the `moveFinished` method:

```
//actions
auto moveFinished =
CallFuncN::create(CC_CALLBACK_1(HelloWorld::moveFinished,
this));
auto moveTo = MoveTo::create(2, Vec2(sprBomb-
>getPositionX(), 0 - sprBomb->getContentSize().height/2));
auto sequence = Sequence::create(moveTo, moveFinished,
nullptr);
sprBomb->runAction(sequence);
```

Note that we pass a `nullptr` parameter at the end of the sequence, so Cocos2d-x will stop executing the items in the sequence, when it sees this value; if you don't specify it, then this will cause your game to crash.

Since version 3.0, Cocos2d-x recommends using the `nullptr` keyword for referring to the null pointers rather than utilizing the conventional NULL macro, which will still work, but is not considered as best practice within C++.

Animating sprites

In order to give a more professional aspect to our game, we could animate our sprites so that it does not constantly show a still image but rather displays animated characters, enemies, and obstacles. Cocos2d-x provides an easy mechanism to add these kinds of animations to our sprites, as we can appreciate in the following code listing:

```
//Animations
Vector<SpriteFrame*> frames;
Size playerSize = sprPlayer->getContentSize();
frames.pushBack(SpriteFrame::create("player.png", Rect(0, 0,
playerSize.width, playerSize.height)));
frames.pushBack(SpriteFrame::create("player2.png", Rect(0, 0,
playerSize.width, playerSize.height)));
auto animation = Animation::createWithSpriteFrames(frames,0.2f);
auto animate = Animate::create(animation);
sprPlayer->runAction(RepeatForever::create(animate));
```

Improving performance with sprite sheets

Although we can create sprite animations based on the images located in several files, as we have done in our previous code, it would be very inefficient to load a large number of files. That is why we prefer loading a single file that contains several images. In order to achieve this, a plain text file with the `plist` extension indicates the exact location of each image within the file, Cocos2d-x is able to read this plain text file and extract all the images from a single sprite sheet file. There are many tools that allow you to create your own sprite sheet, the most popular one is the texture packer, which you can download from `https://www.codeandweb.com/texturepacker` and try for free on Windows or Mac OS.

In this chapter, the resources we are including are: a `plist` file named `bunny.plist` and the `bunny_ss.png` sprite sheet created with texture packer. You can load any frame of this sheet with the following code:

```
SpriteFrameCache* cache = SpriteFrameCache::getInstance();
cache->addSpriteFramesWithFile("bunny.plist");
auto sprBunny =
Sprite::createWithSpriteFrameName("player3.png");
sprBunny -> setAnchorPoint(Vec2());
```

Game menus

It is very common to have menus in some part of our games, such as the main screen and the configuration screen. This framework provides us with a simple way to add menus to our games.

The following code listing shows the menu creation process:

```
auto closeItem = MenuItemImage::create(
  "pause.png", "CloseSelected.png",
  CC_CALLBACK_1(HelloWorld::pause_pressed, this));
closeItem->setPosition(Vec2(_visibleSize.width -
  closeItem->getContentSize().width/2 , closeItem->
  getContentSize().height/2));
auto menu = Menu::create(closeItem, nullptr);
menu->setPosition(Vec2::ZERO);
this->addChild(menu, 1);
```

As we can see from the previous listing, we have first created a menu item by instantiating the `MenuItemImage` class and passing three parameters to the `create` method: the first parameter indicates what image should be displayed for the menu item, the second is the image that should be displayed when the image is selected, and the third parameter specifies the method that should be invoked when the menu item is selected.

Cocos2d-x branch 3 now allows the programmer to use the lambda expressions to handle the menu items.

Handling multiple screen resolutions

When creating games, you should decide which screen resolutions you are planning to support, and then create all the images with a size that will neither look pixelated on high resolution screens nor affect performance while loading on the low capability devices. All those versions of images should have the same name but they should be stored in different directories in the `Resources` folder.

In this example, we have three directories: the first with the images in high resolution, the second with the images in mid resolution, and the third with the images in low resolution.

After having all our images in the adequate sizes so as to suit all the resolution needs, we have to write the code that picks the right image set based on the device's screen resolution. As we have mentioned before, the `AppDelegate` class contains `applicationDidFinishLaunching`, which is launched immediately after the Cocos2d-x framework is loaded on the device. In this method we are going to write our multiple screen resolution code, as we can see in the following code listing:

```
bool AppDelegate::applicationDidFinishLaunching() {
  auto director = Director::getInstance();
  // OpenGL initialization done by cocos project creation script
  auto glview = director->getOpenGLView();
  Size screenSize = glview->getFrameSize();
  Size designSize = CCSizeMake(768, 1280);
  std::vector<std::string> searchPaths;

if (screenSize.height > 800){
  //High Resolution
  searchPaths.push_back("images/high");
  director->setContentScaleFactor(1280.0f / designSize.height);
}
else if (screenSize.height > 600){
```

```
    //Mid resolution
    searchPaths.push_back("images/mid");
    director->setContentScaleFactor(800.0f / designSize.height);
  }
  else{
    //Low resolution
    searchPaths.push_back("images/low");
    director->setContentScaleFactor(320.0f / designSize.height);
  }
    FileUtils::getInstance()->setSearchPaths(searchPaths);
    glview->setDesignResolutionSize(designSize.width,
    designSize.height, ResolutionPolicy::NO_BORDER );
    auto scene = HelloWorld::createScene();
    director->runWithScene(scene);
    return true;
  }
```

Modify the `AndroidManifest.xml` file by setting the value of `android:screenOrientation` to `portrait`.

Putting everything together

Here is the complete code of the `HelloWorldScene.cpp` implementation file, where we have created and positioned our background, our animated player, and our moving bomb:

```
#include "HelloWorldScene.h"
#include "PauseScene.h"

USING_NS_CC;

Scene* HelloWorld::createScene()
{
  // 'scene' is an autorelease object
  auto scene = Scene::create();

  // 'layer' is an autorelease object
  auto layer = HelloWorld::create();

  // add layer as a child to scene
  scene->addChild(layer);

  // return the scene
  return scene;
}
```

Next in the `init` function, we are going to instantiate and initialize our sprites:

```
bool HelloWorld::init()
{
  if ( !Layer::init() )
  {
    return false;
  }
  _director = Director::getInstance();
  _visibleSize = _director->getVisibleSize();
  auto origin = _director->getVisibleOrigin();
  auto closeItem = MenuItemImage::create("pause.png",
  "pause_pressed.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
  this));

  closeItem->setPosition(Vec2(_visibleSize.width - closeItem-
  >getContentSize().width/2 ,
  closeItem->getContentSize().height/2));

  auto menu = Menu::create(closeItem, nullptr);
  menu->setPosition(Vec2::ZERO);
  this->addChild(menu, 1);
  auto sprBomb = Sprite::create("bomb.png");
  sprBomb->setPosition(_visibleSize.width / 2, _visibleSize.height
  + sprBomb->getContentSize().height/2);
  this->addChild(sprBomb,1);
  auto bg = Sprite::create("background.png");
  bg->setAnchorPoint(Vec2::Zero);
  bg->setPosition(0,0);
  this->addChild(bg, -1);
  auto sprPlayer = Sprite::create("player.png");
  sprPlayer->setPosition(_visibleSize.width / 2,
  _visibleSize.height * 0.23);
  this->addChild(sprPlayer, 0);
```

Next, we will add animations using the following code:

```
  Vector<SpriteFrame*> frames;
  Size playerSize = sprPlayer->getContentSize();
  frames.pushBack(SpriteFrame::create("player.png", Rect(0, 0,
  playerSize.width, playerSize.height)));
  frames.pushBack(SpriteFrame::create("player2.png", Rect(0, 0,
  playerSize.width, playerSize.height)));
  auto animation = Animation::createWithSpriteFrames(frames,0.2f);
  auto animate = Animate::create(animation);
  sprPlayer->runAction(RepeatForever::create(animate));
```

Here we are going to create the sequence that will move our bomb from the top to the bottom of the screen. After the movement is complete, we will specify to call the `moveFinished` method. We just use this for printing a log message for testing purposes:

```
  //actions
  auto moveFinished =
  CallFuncN::create(CC_CALLBACK_1(HelloWorld::moveFinished,
  this));
  auto moveTo = MoveTo::create(2, Vec2(sprBomb->getPositionX(),
  0 - sprBomb->getContentSize().height/2));
  auto sequence = Sequence::create(moveTo, moveFinished, nullptr);
  sprBomb->runAction(sequence);
  return true;
}

void HelloWorld::moveFinished(Node* sender){
  CCLOG("Move finished");
}

void HelloWorld::pauseCallback(cocos2d::Ref* pSender){
  _director->pushScene(TransitionFlipX::create(1.0,
  Pause::createScene()));
}
```

The following image shows us how our game looks after putting together all the codes done during this chapter:

Summary

In this chapter, we have seen how to create our game scenes, and how to add sprites and menus to it. We have also learned how to animate sprites easily and move them around the screen.

In the next chapter, we will learn how to move our sprites in a more realistic way by using the built-in physics engine; with it, we will easily configure the motion and add collision detection to our game.

3
Understanding Game Physics

In this chapter, we will cover how to add physics to our games by using the built-in engine provided by Cocos2d-x based on the popular Chipmunk framework. We are going to explain the following topics:

- Setting up the physics world
- Detecting collisions
- Handling gravity
- Handling physics properties

For more information about the Chipmunk physics engine, you may visit `https://chipmunk-physics.net`.

Physics engines encapsulate all the complexity related to giving our scene realistic motions, such as adding gravity to an object so it is attracted to the bottom of the screen, or detecting collision between bodies and so on.

While working with physics, we should keep in mind that we are dealing with a physics world in our scene, and that all the physics elements participating in the world are known as physics bodies. These bodies have properties such as mass, position, and rotation. These may be changed to customize the bodies. One physics body may be attached to another by a joint definition.

Take in to consideration that, from a physics perspective, the physics body is not aware of the sprites and other objects outside of the physics world, but we will see in this chapter how to link the sprites to the physics bodies.

One of the most common characteristics of video games is collision detection; we often need to know when the objects collide with each other. This can be easily done by defining the shapes that would represent the collision area on each body, and then specifying a collision listener, as we will show later in this chapter.

Finally, we will be covering the Box2D physics engine, which is a completely separate physics engine and has nothing to do with Chipmunk. Box2D is written in C++, and Chipmunk is written in C.

Setting up the physics world

In order to enable physics in our game, we need to add the following lines to our `HelloWorldScene.h` header file:

```
cocos2d::Sprite* _sprBomb;
  void initPhysics();
  bool onCollision(cocos2d::PhysicsContact& contact);
  void setPhysicsBody(cocos2d::Sprite* sprite);
```

Here we have created an instance variable for the `_sprBomb` variable, so that it can be accessed from within all the instance methods. In this particular case, we would like to have access to the bomb instance in the `onCollision` method that will be called each time the collisions between the physics bodies are detected, so that we can make the bomb disappear by simply setting its visible attribute to false.

Now let us move to our `HelloWorld.cpp` implementation file, and make a few changes in order to set up our physics world.

First, let us modify our `createScene` method, so that it will now look like this:

```
Scene* HelloWorld::createScene()
{
  auto scene = Scene::createWithPhysics();
  scene->getPhysicsWorld()->setGravity(Vect(0,0));
  auto layer = HelloWorld::create();
  //enable debug draw
  scene->getPhysicsWorld()->setDebugDrawMask
  (PhysicsWorld::DEBUGDRAW_ALL);
  scene->addChild(layer);
  return scene;
}
```

In the previous versions of the Cocos2d-x branch 3, you needed to specify the physics world that the current scene layer would use. But, it is not necessary in version 3.4 and the `setPhysicsWorld` method has been removed from the `Layer` class.

Here we can see that we are now creating the scene instance using the `createWithPhysics` static method contained in the `Scene` class, rather than using the simple create method.

The second step that we are going to perform here is to set the gravity to (0,0), so that the physics world's gravity will not attract our sprites to the bottom of the screen.

Then, we will enable the physic's engine debug draw, which will allow us to see all the physics bodies. This option will help us during the development phase, we will use the COCOS2D_DEBUG macro so that it only displays the debug draw when running in the debug mode as follows:

```
#if COCOS2D_DEBUG
  scene->getPhysicsWorld()->setDebugDrawMask
(PhysicsWorld::DEBUGDRAW_ALL);
#endif
```

In the following screenshot, we can see the red circles that surround the bomb and the player sprite. This represents the physics body attached to each player sprite:

Now let us implement our `setPhysicsBody` method, which receives, as a parameter, a sprite object pointer referring to the sprite to which I will add a physics body. This method will create a circle that will represent the physics body, and therefore, the collision area. The circle will be created with a radius that is half of the width of the sprite, so that it covers as much of the sprite's area as possible.

```
void HelloWorld::setPhysicsBody(cocos2d::Sprite* sprite){
  auto body = PhysicsBody::createCircle
  (sprite->getContentSize().width/2);
  body->setContactTestBitmask(true);
  body->setDynamic(true);
  sprite -> setPhysicsBody(body);
}
```

Circles are commonly used for detecting the collisions since they require less CPU effort for detecting the collisions in each frame; nevertheless, their precision can be unacceptable in some cases.

Now let us add the physics bodies to our player and to our bomb sprite, within our `init` method. In order to do this, we will call our instance method setPhysicsBody after the initialization of each of those sprites.

Collision detection

First, let us implement our `onCollision` instance method. This is going to be called each time a collision between two physics bodies is detected. As we can see in the following code, when the bomb physics body collides with our player, it makes the bomb invisible:

```
bool HelloWorld::onCollision(PhysicsContact& contact){
  _sprBomb->setVisible(false);
  return false;
}
```

This is a good place for putting some logs during the development process in order to find out when the collisions are being detected. In Cocos2d-x 3.4 you can print the log messages using the `CCLOG` macro. This could be turned on by defining the macro `COCOS2D_DEBUG` as follows: `#define COCOS2D_DEBUG 1`.

As we can see, this method returns a Boolean value. It indicates whether these two bodies can collide again. In this particular case, we will return false, indicating that as soon as these two physics bodies collide they should not continue to collide. If we return the true indication, then these two objects will continue to collide, and this will cause our player sprite to move, thus giving our game an undesired visual effect.

Now, let us enable our game to detect when our bomb collides with our player. In order to do this we will create an `EventListenerPhysicsContact` instance, we are going to set it so that when two physics bodies start colliding, it should call our `onCollision` instance method. Then, we will add our event listener to the event dispatcher. We are going to create these three simple steps inside our `initPhysics` instance method. So, our code will look like this:

```
void HelloWorld::initPhysics()
{
  auto contactListener = EventListenerPhysicsContact::create();
  contactListener->onContactBegin =
  CC_CALLBACK_1(HelloWorld::onCollision,this);
  getEventDispatcher() ->addEventListenerWithSceneGraphPriority
  (contactListener,this);
}
```

Our `init` method code will look like this:

```
bool HelloWorld::init() {
  if( !Layer::init() ){
    return false;
  }
  _director = Director::getInstance();
  _visibleSize = _director->getVisibleSize();
  auto origin = _director->getVisibleOrigin();
  auto closeItem = MenuItemImage::create("pause.png",
  "pause_pressed.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
  this));
  closeItem->setPosition(Vec2
  (_visibleSize .width - closeItem->getContentSize().width/2,
  closeItem->getContentSize().height/2));

  auto menu = Menu::create(closeItem, nullptr);
  menu->setPosition(Vec2::ZERO);
```

```
    this->addChild(menu, 1);
    _sprBomb = Sprite::create("bomb.png");
    _sprBomb->setPosition(_visibleSize .width/2,
    _visibleSize .height + _sprBomb->getContentSize().height/2);
    this->addChild(_sprBomb,1);
    auto bg = Sprite::create("background.png");
    bg->setAnchorPoint(Vec2());
    bg->setPosition(0,0);
    this->addChild(bg, -1);
    auto sprPlayer = Sprite::create("player.png");
    sprPlayer->setPosition(_visibleSize .width / 2,
    _visibleSize .height * 0.23);
    setPhysicsBody(sprPlayer);
    this->addChild(sprPlayer, 0);
    //Animations
    Vector<SpriteFrame*> frames;
    Size playerSize = sprPlayer->getContentSize();
    frames.pushBack(SpriteFrame::create("player.png",
    Rect(0, 0, playerSize.width, playerSize.height)));
    frames.pushBack(SpriteFrame::create("player2.png",
    Rect(0, 0, playerSize.width, playerSize.height)));
    auto animation = Animation::createWithSpriteFrames(frames,0.2f);
    auto animate = Animate::create(animation);
    sprPlayer->runAction(RepeatForever::create(animate));
    setPhysicsBody(_sprBomb);
    initPhysics();
    return true;
}
```

Handling gravity

Now that we have successfully used the built-in physics engine to detect the collision, let us play a little bit with gravity. Go to the `createScene` method and modify the parameters that we have sent to the constructor. In our game, we have used the `(0,0)` value, since we did not want our world to have any gravity force that moves our bodies on either the *x* or the *y* axis.

Now, give it a try, change the values to positive or negative. When we use a negative value on the *x* axis, it will attract the body toward the left, and when we use the negative value on the *y* axis, it will attract the body toward the bottom.

Changing these values and understanding the physics added to our game might give you some ideas for your next games.

Handling physics properties

Now that we have created our scene, which corresponds to the physics world, we now have the ability to change the physics properties, such as the velocity, linear damping, force, impulse, and torque for each body.

Applying velocity

In the previous chapter, we managed to move our bomb from the top of the screen to the bottom using a `MoveTo` action. Now that we are using the built-in physics engine, we can achieve the same effect just by setting a velocity to the bomb. This can be done by simply calling the `setVelocity` method of the bomb sprite's physics body. Velocity is a vector quantity; therefore, the method mentioned receives a `Vect` instance as the parameter. The value of *x* will represent its horizontal component; on this axis a positive value means that the body will move to the right and a negative value means that the body will move to the left. The *y* value affects the vertical movement. Positive values move the body toward the top of the screen and negative values move the body to the bottom of the screen.

We have added the following line to the `init` method of the `HelloWorld.cpp` implementation file just before the return statement:

```
_sprBomb->getPhysicsBody()->setVelocity(Vect(0,-100));
```

Remember to remove the line of code that requests the bomb sprite to execute the `MoveTo` action so that you can confirm that the bomb is now moving because of its velocity parameter.

Now let us go to the `onCollision` method, there we are going to set the bomb's velocity to zero when it collides with our player sprite.

```
_sprBomb -> getPhysicsBody()->setVelocity(Vect());
```

Similar to the `Vec2` class, the null constructor will initialize all the vector values to zero.

Linear damping

We can decrease the velocity of our physics body to create a friction effect. One way to achieve this is by calling the `linearDamping` method and specifying the rate of change of the body's velocity. The value should be a float between `0.0` and `1.0`.

You can test the linear damping by setting the value of the bomb's physics body to `0.1f`, and observe how the velocity of the bomb is decreased.

```
_sprBomb->getPhysicsBody()->setLinearDamping(0.1f);
```

Remember to document or remove this line after testing the linear damping so that the game does not behave in an unexpected way.

Applying force

We can apply an immediate force to a body by simply calling the `applyForce` method of the physics body to which we want to apply it. Analogous to the methods explained in the previous sections, it receives a vector as a parameter, which means that the force has vertical and horizontal components.

We can test this method by applying a force to the bomb that will make it move to the right as soon as it collides with our player sprite in our `onCollision` method.

```
_sprBomb->getPhysicsBody()->applyForce(Vect(1000,0));
```

Applying impulse

In the previous section we have added an immediate force to our physics body, now we can add a continuous force by applying an impulse to it by calling the `applyImpulse` method.

Add the following line of code, after applying the immediate force to the physics body in the `onCollision` method:

```
_sprBomb->getPhysicsBody()->applyImpulse(Vect(10000,0));
```

Run the game now, and you will see how the bomb moves to the right.

Remove the lines of code that add force and impulse to our bomb within the `onCollision` method.

Applying torque

Finally, let us make our bomb rotate after it collides with our player sprite. We can do that by applying a torque force to the bomb's physics body by using the `applyTorque` method, which receives a float number; if positive, it will make the physics body rotate counter-clockwise.

Let us add an arbitrary positive torque to the `onCollision` method just before the return statement:

```
    auto body = _sprBomb -> getPhysicsBody();
body->applyTorque(100000);
```

Now add a negative value to the `applyTorque` method, and you will see how the physics body rotates clockwise.

Putting everything together

After all of the modifications, our `onCollision` method looks like this:

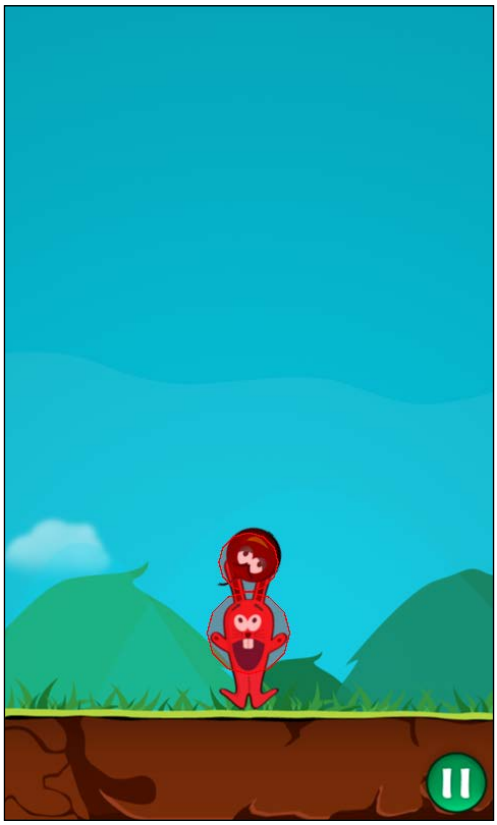
```
bool HelloWorld::onCollision(PhysicsContact& contact){
  auto body = _sprBomb -> getPhysicsBody();
  body->setVelocity(Vect());
  body->applyTorque(100900.5f);
  return false;
}
```

Our `init` method now looks like this:

```
bool HelloWorld::init()
{
  if( !Layer::init() ){
    return false;
  }
  _director = Director::getInstance();
  _visibleSize = _director->getVisibleSize();
  auto origin = _director->getVisibleOrigin();
  auto closeItem = MenuItemImage::create("CloseNormal.png",
  "CloseSelected.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
  this));
```

```
  closeItem->setPosition(Vec2
(_visibleSize .width - closeItem->getContentSize().width/2,
closeItem->getContentSize().height/2));

  auto menu = Menu::create(closeItem, nullptr);
  menu->setPosition(Vec2::ZERO);
  this->addChild(menu, 1);
  _sprBomb = Sprite::create("bomb.png");
  _sprBomb->setPosition(_visibleSize .width/2,
  _visibleSize .height + _sprBomb->getContentSize().height/2);
  this->addChild(_sprBomb,1);
  auto bg = Sprite::create("background.png");
  bg->setAnchorPoint(Vec2());
  bg->setPosition(0,0);
  this->addChild(bg, -1);
  auto sprPlayer = Sprite::create("player.png");
  sprPlayer->setPosition(_visibleSize .width/2,
  _visibleSize .height * 0.23);
  setPhysicsBody(sprPlayer);

  this->addChild(sprPlayer, 0);
  //Animations
  Vector<SpriteFrame*> frames;
  Size playerSize = sprPlayer->getContentSize();
  frames.pushBack(SpriteFrame::create("player.png",
  Rect(0, 0, playerSize.width, playerSize.height)));
  frames.pushBack(SpriteFrame::create("player2.png",
  Rect(0, 0, playerSize.width, playerSize.height)));
  auto animation = Animation::createWithSpriteFrames(frames,0.2f);
  auto animate = Animate::create(animation);
  sprPlayer->runAction(RepeatForever::create(animate));

  setPhysicsBody(_sprBomb);
  initPhysics();
  _sprBomb->getPhysicsBody()->setVelocity(Vect(0,-100));
  return true;
}
```

The following screenshot shows how our game looks after the modifications:



Box2D physics engine

So far, we have used the built-in physics engine provided by the framework, which is based on the chipmunk C physics library; nevertheless, Cocos2d-x also provides the integration in its API with the Box2D physics engine.

In order to create a Box2D world, we instantiate the `b2World` class and then we pass to its constructor a `b2Vec` object that represents the world's gravity. The world instance has an instance method for creating `b2Bodies`. The sprite class has a method named `setB2Body`, which allows us to associate a Box2D physics body to any given sprite. This is smoother than how it was in the branch 2 of the framework; more code was required to tie a `b2Body` with a sprite.

Although the Box2D integration is easy to use, I strongly recommend using the built-in physics engine as the Box2D integration is no longer under active development.

Summary

We have added physics to our game by creating a physics world and physics bodies that represent the bomb and our player sprite, and we have used the collision detection mechanism provided by the built-in physics engine in a very few steps. We have also showed how to change the gravity parameters so that the physic bodies move according to the force of gravity. We have easily changed the physics properties of the bodies, such as velocity, friction, force, impulse, and torque with a single line of code for each of those. So far, our player ignores our user events. In the next chapter, we will cover how to add user interaction to our game.

4
User Input

So far, we have added the graphics that move around the screen and collide with each other, but it is not fun yet, since the player cannot control our main character and it would not be a game unless our user can interact with it. In this chapter we will add user interaction to our game. The following topics will be covered within this chapter:

- Understanding the Event Dispatcher mechanism
- Handling the touch event
- Handling the accelerometer events
- Keeping the screen alive
- Handling the Android back key pressed event

Understanding the Event Dispatcher mechanism

Event handling is now different from how it was in the previous version (version 2) of Cocos2d-x. From version 3.0 onwards, we now have a unified event dispatching mechanism, called Event Dispatcher, that handles all sorts of user input events, which may happen during the game.

We can handle many kinds of user input events, such as touches, keyboard key press, acceleration, and mouse motion. In the following sections we will cover how to handle the user input events that relate to mobile games, such as touch, and accelerometer.

There are many classes that allow us to listen for each of the previously mentioned events; once we have instantiated any of these classes, we need to add them to the event dispatcher so, when a user event is triggered, it will call the method defined by its corresponding listener.

You can access the event dispatcher either by the `_eventDispatcher` instance attribute inherited from the `Node` class, or by calling the `getEventDispatcher` static method located in the Cocos2d-x API `Director` class.

Cocos2d-x Event Dispatcher mechanism uses the observer design pattern, which is the pattern used for handling the user input events on the Android native applications.

Handling the touch event

The most common way to create interaction between games and users is through touch events. Handling the touch event is very straightforward in Cocos2d-x.

In this section, we will allow the user to move our player sprite by touching it and moving it to the desired position.

The first thing that we will do is create the `initTouch`, `movePlayerByTouch`, and `movePlayerIfPossible` methods in the `HelloWorldScene.h` class header file, as we can see in the following code listing:

```
void initTouch();
void movePlayerByTouch(cocos2d::Touch* touch,
cocos2d::Event* event);
void movePlayerIfPossible(float newX);
```

Now let us add the initialization code to the `initTouch` method in the implementation file `HelloWorldScene.cpp`. In this simple game, we are going to use a single touch that is going to be used to move around our bunny character, no multi-touch handling is required.

In order to handle the single touch, we will create a new instance of the `EventListenerTouchOneByOne` class, then we are going to specify what our game should do when the touch event begins, when the touch is moving, and when the touch event ends. In the following code listing, after instantiating the `EventListenerTouchOneByOne` class, we will specify the methods that should be called when the events `onTouchBegan`, `onTouchMoved`, and `onTouchEnded` are triggered. For the purpose of the current game, we are only going to be using the `onTouchMoved` event. For this we are going to create a callback to our method, `movePlayerByTouch`, for the other two methods we are going to create empty structures through the lambda functions. You can learn more about the C++ lambda functions from the link `http://en.cppreference.com/w/cpp/language/lambda`.

```
void HelloWorld::initTouch() {
  auto listener = EventListenerTouchOneByOne::create();
  listener->onTouchBegan = [](Touch* touch, Event* event){
    return true;
  }
  listener->onTouchMoved = CC_CALLBACK_2
  (HelloWorld::movePlayerByTouch,this);
  listener->onTouchEnded = [=](Touch* touch, Event* event){};
  _eventDispatcher->addEventListenerWith
  SceneGraphPriority(listener, this);
}
```

By convention, all the C++ member variables are named with an underscore prefix.

Now that we have encapsulated all the touch listener initialization codes into one method, let us call it our `init` method by adding the following line at the end of the method:

```
initTouch();
```

We will now create the `movePlayerIfPossible` method. This will move the player sprite only if the new requested position on the horizontal axis is not out of the screen limits, as we can see in the illustration. This method will be used to move our player sprite with the touch input event, and it is also going to be used in the next section where we are going to move our player sprite by using the accelerometer.

```
void HelloWorld::movePlayerIfPossible(float newX){
  float sprHalfWidth = _sprPlayer->getBoundingBox().size.width/2;
  if(newX >= sprHalfWidth && newX < visibleSize.width -
  sprHalfWidth){
    _sprPlayer->setPositionX(newX);
  }
}
```

In this method, we have used the "Tell, Don't Ask" design principle by making the validation in the method that verifies whether the player goes out of the screen in the method. This has saved us from repeating the logic of validating if the player sprite is out of the screen in the touch and accelerometer event handling methods.

Finally, we will now create the `movePlayerByTouch` method, which will be called by the event dispatcher as soon as a touch event is triggered. In this method, we are going to evaluate the location on the screen, and whether the place where the user has touched the screen intersects the sprite's bounding rectangle:

```
void HelloWorld::movePlayerByTouch(Touch* touch, Event* event){
  auto touchLocation = touch->getLocation();
  if(_sprPlayer->getBoundingBox().containsPoint(touchLocation)){
    movePlayerIfPossible(touchLocation.x);
  }
}
```

Handling multi-touch events

In the previous sections, we enabled the touch event required for this game, which is a single touch; nevertheless, Cocos2d-x also handles the multi-touch capability, which we will cover in this section.

Although our game does not require the multi-touch feature, we will create a test code so that we can move our player sprite and our bomb simultaneously. In order to do this, we are going to add the methods, `initMultiTouch` and `moveByMultitouch`, at the end of our `HelloWorldScene.h` header file as follows:

```
void initMultiTouch();
void moveByMultiTouch(const std::vector<cocos2d::Touch*>& touches,
cocos2d::Event* event);
```

Now, let us add its implementation to the `HelloWorldScene.cpp` implementation file. We will begin by the `initMultiTouch` initialization method:

```
void HelloWorld::initMultiTouch() {
  auto listener = EventListenerTouchAllAtOnce::create();
  listener->onTouchesBegan = [](const std::vector<Touch*>&
  touches, Event* event){};
  listener->onTouchesMoved = CC_CALLBACK_2
  (HelloWorld::moveByMultiTouch,this);
```

```
    listener->onTouchesEnded = [](const std::vector<Touch*>&
    touches, Event* event){};
    _eventDispatcher->addEventListenerWithScene
    GraphPriority(listener, this);
}
```

Here we can find similarities with the previous single-touch initialization method, but there are many differences, the most notorious one is that we are now instantiating the `EventListenerTouchAllAtOnce` class, instead of instantiating the `EventListenerTouchOneByOne` class as we had previously done. Although its event properties are named like its single-touch version, you may notice that they are now written in the plural, so that it is now referring to the touches instead of the touch, for example, `onTouchesBegan`. Now, it will also expect a different set of parameters, since we will now work with multiple touches, the event methods will now receive a `std::vector` parameter, where it will receive the collection of the touches that have occurred simultaneously.

As shown in the previous code, we are going to call our `moveByMultiTouch` method every time our player moves the touches around, so we are now showing the implementation code for this method:

```
void HelloWorld::moveByMultiTouch(const std::vector<Touch*>&
touches, Event* event){
  for(Touch* touch: touches){
    Vec2 touchLocation = touch->getLocation();
    if(_sprPlayer->getBoundingBox().containsPoint(touchLocation)){
      movePlayerIfPossible(touchLocation.x);
    }else if(_sprBomb->getBoundingBox().containsPoint
    (touchLocation)){
      _sprBomb->setPosition(touchLocation);
    }
  }
}
```

As you can see from the previous code we are now handling multiple touches, and in the `moveByMultiTouch` method we are iterating through all the touches, and for each one, we are verifying if it is touching either our bomb or our bunny player sprite, and if it does so, then it will move the touched sprite to the touched location.

Finally, let us call the `initMultiTouch` initialization method at the end of our `init` method as follows:

```
initMultiTouch();
```

As previously mentioned, the objective of this section is to show you how simple it is to handle the multi-touch events; nevertheless, since we are not going to use it in our game, you may remove the call to the `initMultiTouch` method that we have just added to our `init` method, as soon as you finish testing the multi-touch feature.

Handling accelerometer events

Another common way of interacting between the game and the player is the accelerometer, which allows us to have hours of fun by moving our phone around in order to move our character and achieve the game's objective.

In order to add the accelerometer support to our game, we are first going to add the following method declaration to our `HelloWorldScene.h` header file:

```
void movePlayerByAccelerometer(cocos2d::Acceleration*
acceleration, cocos2d::Event* event);
void initAccelerometer();
```

Now, let us create the code to our `HelloWorld.cpp` implementation file that corresponds to our accelerometer initialization. The first thing that we are going to do is enable the accelerometer sensor on the device, by calling the static method `setAccelerometerEnabled` located on the `Device` class, then we are going to create the event listener that will listen to the accelerometer's events and finally, we will add it to the event dispatcher, as shown in the following code:

```
void HelloWorld::initAccelerometer(){
  Device::setAccelerometerEnabled(true);
  auto listener = EventListenerAcceleration::create
  (CC_CALLBACK_2(HelloWorld::movePlayerByAccelerometer, this));
  _eventDispatcher->addEventListenerWithScene
  GraphPriority(listener, this);
}
```

The most common way to add an event listener to the dispatcher is through the `addEventListenerWithSceneGraphPriority` method, which will assign the `z` order of the node sent as the second parameter as its priority. This is useful when we have many listeners that are triggered at the same time and we want to specify which code should run first.

At this point, we have initialized the accelerometer, and we have created in the previous section, the `movePlayerIfPossible` method that will move the player sprite and make sure that it will not go out of the screen limits. Now we are going to create the implementation code for the `movePlayerByAccelerometer` method, which is going to be called as soon as an accelerometer event is triggered. Since the acceleration value that we get is very low, we multiply it by ten so that our player sprite moves faster.

```
void HelloWorld::movePlayerByAccelerometer(cocos2d::Acceleration*
acceleration, cocos2d::Event* event){
  int accelerationMult = 10;
  movePlayerIfPossible(_sprPlayer->getPositionX() +
  (acceleration->x * accelerationMult));
}
```

Finally, let us call our accelerometer initialization code at the end of our `init` method from the `HelloWorldScene.cpp` implementation file as follows:

```
initAccelerometer();
```

Keeping the screen alive

In the previous section we added the accelerometer interaction to our game, which means that our player is controlling the main character by moving the phone rather than by touching the screen, this will cause many android devices to turn off the screen after a period of inactivity (not touching the screen). Of course, no one would like that our Android device's screen suddenly goes black; in order to prevent this from happening, we will call the `setKeepScreenOnJni` method, introduced in the previous version 3.3 of the framework. Prior to this version, this annoying situation was considered a framework flaw, and now it has finally been fixed.

We first need to include the helper in our `HelloWorldScene.cpp` header file as follows:

```
#include "../cocos2d/cocos/platform/android/jni/
Java_org_cocos2dx_lib_Cocos2dxHelper.h"
```

Then, we are going to add the following line at the end of our `init` method in the `HelloWorldScene.cpp` implementation file:

```
setKeepScreenOnJni(true);
```

Handling the Android back key pressed event

A common mistake that I have seen in many games developed by using Cocos2d-x is that the game does nothing when the back button is pressed. Android users are used to pressing the back button whenever they want to go back to the previous activity. If the application does nothing when the back button is pressed, then it would confuse the user because it would not be an expected behavior, less experienced users may even have a hard time trying to exit the game.

We can easily trigger a custom code any time the user presses the back button by adding an `EventListenerKeyboard` method to our event dispatcher.

First we are going to add the `initBackButtonListener` and `onKeyPressed` method declarations in our `HelloWorldScene.h` header file, as shown in the following code listing:

```
void initBackButtonListener();
void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode,
cocos2d::Event* event);
```

Now, let us add the implementation code for our `initBackButtonListener` in our `HelloWorldScene.cpp` implementation file. We are first going to instantiate the `EventListenerKeyboard` class, then we are required to specify the methods that are going to be called on the `onKeyPressed` event and the `onKeyReleased` event or we will get a runtime error. We are going to create an empty method implementation and assign it to the `onKeyPressed` property through the C++11 lambda expressions, and then we are going to add a callback to our `onKeyPressed` method for the listener's `onkeyRelease` property. Then, as we had done previously, we will add this listener to the event dispatcher mechanism:

```
void HelloWorld::initBackButtonListener(){
  auto listener = EventListenerKeyboard::create();
  listener->onKeyPressed = [=](EventKeyboard::KeyCode keyCode,
  Event* event){};
  listener->onKeyReleased =
  CC_CALLBACK_2(HelloWorld::onKeyPressed, this);
  _eventDispatcher->addEventListenerWithScene
  GraphPriority(listener, this);
}
```

We are now going to implement the code for our `onKeyPressed` method. This will tell the `Director` to end the game, if the key pressed is the back button key:

```
void HelloWorld::onKeyPressed(EventKeyboard::KeyCode keyCode,
Event* event){
  if(keyCode == EventKeyboard::KeyCode::KEY_BACK){
    Director::getInstance()->end();
  }
}
```

Finally, we will call the `initBackButtonListener` method at the end of the `init` method as follows:

```
initBackButtonListener();
```

Be aware that you should add the `EventListenerKeyboard` listener to the event dispatcher in each scene where you want to catch the back button pressed event.

Putting everything together

After all the codes that we have added during this chapter, now, our `HelloWorldScene.h` header file will look like this:

```
#ifndef __HELLOWORLD_SCENE_H__
#define __HELLOWORLD_SCENE_H__
#include "cocos2d.h"
class HelloWorld : public cocos2d::Layer
{
public:
  static cocos2d::Scene* createScene();
  virtual bool init();
  void pauseCallback(cocos2d::Ref* pSender);
  CREATE_FUNC(HelloWorld);
private:
  cocos2d::Director *_director;
  cocos2d::Size visibleSize;
  cocos2d::Sprite* _sprBomb;
  cocos2d::Sprite* _sprPlayer;
  void initPhysics();
  bool onCollision(cocos2d::PhysicsContact& contact);
  void setPhysicsBody(cocos2d::Sprite* sprite);
  void initTouch();
```

```
  void movePlayerByTouch(cocos2d::Touch* touch,
cocos2d::Event* event);
  void movePlayerIfPossible(float newX);
  void movePlayerByAccelerometer
(cocos2d::Acceleration* acceleration, cocos2d::Event* event);
  void initAccelerometer();
  void initBackButtonListener();
  void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode,
cocos2d::Event* event);
};

#endif // __HELLOWORLD_SCENE_H__
```

The final `HelloWorldScene.cpp` implementation file will look like this:

```
#include "HelloWorldScene.h"
#include "PauseScene.h"
#include "../cocos2d/cocos/platform/android/jni/
Java_org_cocos2dx_lib_Cocos2dxHelper.h"

USING_NS_CC;
Scene* HelloWorld::createScene()
{
  auto scene = Scene::createWithPhysics();
  scene->getPhysicsWorld()->setGravity(Vect(0,0));
  auto layer = HelloWorld::create();
  //enable debug draw
  //scene->getPhysicsWorld()->setDebugDrawMask
  (PhysicsWorld::DEBUGDRAW_ALL);
  scene->addChild(layer);
  return scene;
}
```

Next, we will try to move the player, so that it does not go outside the screen:

```
void HelloWorld::movePlayerIfPossible(float newX){
  float sprHalfWidth = _sprPlayer->getBoundingBox().size.width/2;
  if(newX >= sprHalfWidth && newX < visibleSize.width -
  sprHalfWidth){
    _sprPlayer->setPositionX(newX);
  }
}
void HelloWorld::movePlayerByTouch(Touch* touch, Event* event)
{
  Vec2 touchLocation = touch->getLocation();
```

```
    if(_sprPlayer->getBoundingBox().containsPoint(touchLocation)){
      movePlayerIfPossible(touchLocation.x);
    }
  }
```

As you can see in the following two methods, `initTouch` and `initAccelerometer`, we have created functions for each initializing task. This will allow us to simplify our code, and make it easier to read:

```
void HelloWorld::initTouch()
{
  auto listener = EventListenerTouchOneByOne::create();
  listener->onTouchBegan = [](Touch* touch, Event* event){return
  true;};
  listener->onTouchMoved =
  CC_CALLBACK_2(HelloWorld::movePlayerByTouch,this);
  listener->onTouchEnded = [=](Touch* touch, Event* event){};
  _eventDispatcher->addEventListenerWithScene
  GraphPriority(listener, this);
}
void HelloWorld::initAccelerometer()
{
  Device::setAccelerometerEnabled(true);
  auto listener = EventListenerAcceleration::create
  (CC_CALLBACK_2(HelloWorld::movePlayerByAccelerometer, this));
  _eventDispatcher->addEventListenerWithScene
  GraphPriority(listener, this);
}

void HelloWorld::movePlayerByAccelerometer(cocos2d::Acceleration*
acceleration, cocos2d::Event* event)
{
  movePlayerIfPossible(_sprPlayer->getPositionX() +
  (acceleration->x * 10));
}
```

Now, we will initialize physics. To do this, we will call the initPhysics() method from within init():

```
bool HelloWorld::init()
{
  if( !Layer::init() ){
    return false;
  }
```

```
_director = Director::getInstance();
visibleSize = _director->getVisibleSize();
auto origin = _director->getVisibleOrigin();
auto closeItem = MenuItemImage::create("pause.png",
"pause_pressed.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
this));
closeItem->setPosition(Vec2(visibleSize.width -
closeItem->getContentSize().width/2,
closeItem->getContentSize().height/2));

auto menu = Menu::create(closeItem, nullptr);
menu->setPosition(Vec2::ZERO);
this->addChild(menu, 1);
_sprBomb = Sprite::create("bomb.png");
_sprBomb->setPosition(visibleSize.width/2,
visibleSize.height + _sprBomb->getContentSize().height/2);
this->addChild(_sprBomb,1);
auto bg = Sprite::create("background.png");
bg->setAnchorPoint(Vec2());
bg->setPosition(0,0);
this->addChild(bg, -1);
_sprPlayer = Sprite::create("player.png");
_sprPlayer->setPosition(visibleSize.width/2,
visibleSize.height * 0.23);
setPhysicsBody(_sprPlayer);

this->addChild(_sprPlayer, 0);
//Animations
Vector<SpriteFrame*> frames;
Size playerSize = _sprPlayer->getContentSize();
frames.pushBack(SpriteFrame::create("player.png",
Rect(0, 0, playerSize.width, playerSize.height)));
frames.pushBack(SpriteFrame::create("player2.png",
Rect(0, 0, playerSize.width, playerSize.height)));
auto animation = Animation::createWithSpriteFrames(frames,0.2f);
auto animate = Animate::create(animation);
_sprPlayer->runAction(RepeatForever::create(animate));

setPhysicsBody(_sprBomb);
initPhysics();
_sprBomb->getPhysicsBody()->setVelocity(Vect(0,-100));
initTouch();
initAccelerometer();
setKeepScreenOnJni(true);
initBackButtonListener();
```

```
    return true;
}

void HelloWorld::pauseCallback(cocos2d::Ref* pSender){
    _director->pushScene(TransitionFlipX::create(1.0,
Pause::createScene()));
}
void HelloWorld::initBackButtonListener(){
    auto listener = EventListenerKeyboard::create();
    listener->onKeyPressed = [=](EventKeyboard::KeyCode keyCode,
    Event* event){};
    listener->onKeyReleased =
    CC_CALLBACK_2(HelloWorld::onKeyPressed, this);
    _eventDispatcher->addEventListenerWithScene
    GraphPriority(listener, this);
}

void HelloWorld::onKeyPressed(EventKeyboard::KeyCode keyCode,
Event* event){
    if(keyCode == EventKeyboard::KeyCode::KEY_BACK){
      Director::getInstance()->end();
    }
}
```

As we can see in the following image, we finally got our bunny to move around the screen and leave its initial central position.

Please note that in the previous code listing we have omitted the segments of the code that are not relevant to this chapter. If you want to see how the complete code listing looks at this point, then you can find it in the resource materials bundled with this book.

Summary

In this chapter, we allowed the user to control our game by using two different input mechanisms, that is, touching the screen and moving the phone around in order to use the accelerometer sensor; we have also made our game stop whenever the back button is pressed.

In the next chapter, we will cover the different ways of adding text to our games.

5
Handling Text and Fonts

It is very common in our games to add text for displaying information to the player. This could be done by using either the TrueType fonts or bitmap fonts, which will allow us more flexibility, and is, indeed, the most used type of font in professional games because it allows us to give a custom look to our games. The following topics will be covered in this chapter:

- Creating TrueType font labels
- Adding label effects
- Creating system fonts
- Creating bitmap font labels

Creating TrueType font labels

Adding text by using the TrueType font is really straightforward. Open the `PauseScene.cpp` implementation file that we have created in *Chapter 2*, *Graphics*. In the `init` method, you will see that we have created an instance of the `Label` class by calling the static method, `createWithTTF`. This method receives three parameters, the first one is the string that we want to draw, the second one is another string representing the font file that you want to use, including the path within the `Resources` folder where it is located, and the third one represents the font size.

The `Label` class was introduced in the Cocos2d-x Version 3.x. It combines the TrueType font and the bitmap font handling in a single class. However, although deprecated, the former label handling classes are still available in API for compatibility reasons.

Now, let us change the third parameter value from 24 to 96 in the `createWithTTF` method to make the font bigger:

```
auto label = Label::createWithTTF("PAUSE", "fonts/Marker Felt.ttf",
96);
```

The Marker Felt font is included in the template Cocos2d-x project generated by the `cocos new` command.

Creating our GameOverScene

It is now time to create a game over scene, this will be displayed as soon as a bomb crashes into our `bunny` sprite.

We are going to do this by copying our `PauseScene.cpp` and `PauseScene.h` files in our `Classes` directory and renaming them as `GameOverScene.cpp` and `GameOverScene.h` respectively.

Remember that each time you add a new source file to your Cocos2d-x folder you need to add the class to the `Android.mk` file contained in the `jni` folder, so that this new source file is compiled during the next build.

Now in the `GameOverScene.h` and `GameOverScene.cpp` files, perform a find and replace on both of the files and replace the word `Pause` with the word `GameOver`.

Finally, substitute the first lines of the code for the `GameOverScene.cpp` implementation file for these:

```
#include "GameOverScene.h"
#include "HelloWorldScene.h"
```

Within the body of the `exitPause` method of the `GameOverScene.cpp` implementation file, we are going to substitute the only line that this method has with this one:

```
Director::getInstance()->replaceScene(TransitionFlipX::
create(1.0, HelloWorld::createScene()));;
```

Calling our GameOverScene when a player loses

We have created our game over scene; now let us show it as soon as the bomb collides with our `player` sprite. In order to achieve this, we will add the following line of code to the `onCollision` method contained in the `HelloWorld` class.

```
_director->replaceScene(TransitionFlipX::create(1.0,
GameOver::createScene()));
```

Now, include the game over scene header file to our `gameplay` class by adding the following line to the beginning of the `HelloWorldScene.h` header file:

```
#include "GameOverScene.h"
```

Customizing the GameOverScene

Now, we don't want the black background color, so we are going to add the same background that we were using in our gameplay exactly as we had done in *Chapter 2*, *Graphics*:

```
auto bg = Sprite::create("background.png");
bg->setAnchorPoint(Vec2());
bg->setPosition(0,0);
this->addChild(bg, -1);
```

Now we are going to change the TrueType font label, which we have copied from `PauseScene`, it will now read as `Game Over`. In the next section, we will add some effects to this label.

```
auto label = Label::createWithTTF("Game Over", "fonts/Marker Felt.
ttf", 96);
```

Adding label effects

Now we are going to add the effects that are only available for the TrueType fonts.

Let us enable an outline for our font. The `enableOutline` method from the `Label` class receives two parameters, a `Color4B` instance and an integer, which represents the outline size—the greater the number the thicker the outline:

```
label->enableOutline(Color4B(255, 0, 0, 100),6);
```

Now, let us add some glow effect to our font:

```
label->enableGlow(Color4B(255, 0, 0, 255));
```

Finally, let us add the shadow effect, which is currently supported by all of the three label types, to our label.

```
label->enableShadow();
```

As you will notice from the following screenshot, the effects overlap each other, so decide which one looks better to you:

The `Color4B` constructor method receives four parameters. The first three are the **red, green and blue** (**RGB**) components, and the fourth one is the `alpha` component. This will allow us to add some transparency effect and it can range from 0 to 255. The label instances don't support the customized effects, such as giving different colors to each word of the text, using different fonts for a single text, or embedding an image in a label.

If you are interested in adding any of these font effects to your game, then you could use the `CCRichLabelTTF` class created by Luma Stubma. This is available at `https://github.com/stubma/cocos2dx-better`.

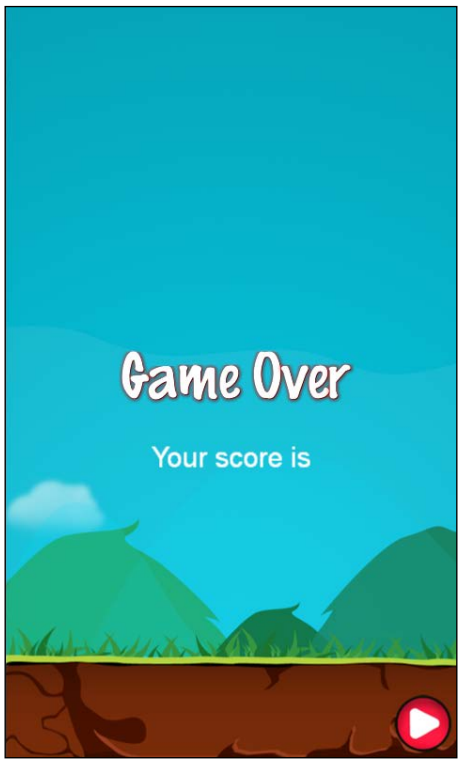
Creating system fonts

You can create the labels that use the fonts of the host operating system; therefore, you would not be required to provide a font file. It is recommended that you use this kind of label only for testing purposes, since it would decrease the framework's flexibility because the selected font may not be available on the user's Android operating system version.

Just for the sake of testing, below our current text, we will add the following label in the `init` method of the `GameOverScene.cpp` implementation file:

```
auto label2 = Label::createWithSystemFont("Your score is", "Arial",
48);
label2->setPosition(origin.x + visibleSize.width/2,origin.y +
visibleSize.height /2.5);
this->addChild(label2, 1);
```

This code yields the following result:



Creating bitmap font labels

So far we have seen how easy it is to create labels by using TrueType and system fonts, now we will perform some extra steps in order to achieve a more professional style for our labels. As mentioned before, bitmap fonts are the most common kinds of labels used in professional games.

As its name suggests, bitmap fonts are generated by the images that represent each character, which will allow us to draw any font that we want, but it will have all the disadvantages of bitmap, such as the risk of our label getting pixelated, the lack of flexibility when handling the different sizes, and extra space on the disk and RAM for handling these kinds of fonts.

There are several applications that you can use for creating bitmap fonts. The most common one is **Glyph Designer**, which you can acquire from the URL `https://71squared.com`. This application was originally released for the Mac OS, but in early 2015 **Glyph Designer X** was released for Windows as well. You can also create your own bitmap fonts by using **Littera**, the free online application. It is available at `http://kvazars.com/littera`. For the purposes of this book, we have included the code for a bitmap font in our chapters. We are going to use this bitmap font code for displaying the total score of the player in the game over scene.

Adding more bombs to our game

Taking in to consideration that now we have a game over scene, let us make this game a little more difficult by adding more bombs to it. We are going to use the Cocos2d-x scheduler mechanism, which will allow us to call a method during every given period of time. We are going to add the `addBombs` method to the `HelloWorldScene` class and within the `init` method of the aforementioned class we are going to schedule it so that it gets called every eight seconds:

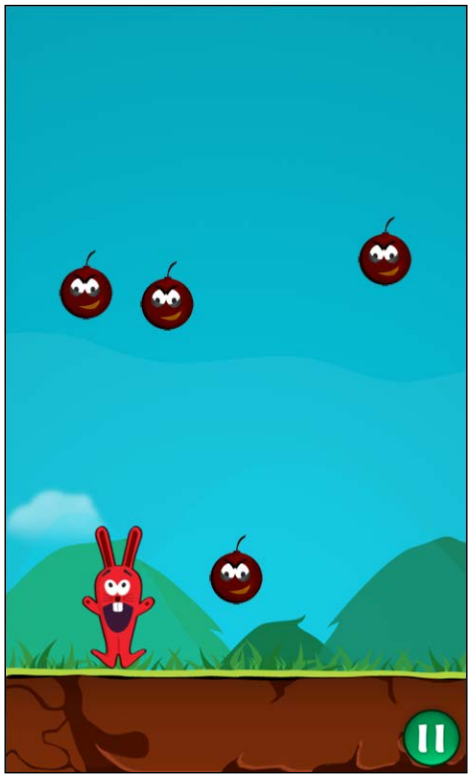
```
schedule(CC_SCHEDULE_SELECTOR(HelloWorld::addBombs), 8.0f);
```

We are going to add three bombs that are in a random position to our scene, and it will happen each time the `addBombs` method is called:

```
void HelloWorld::addBombs(float dt)
{
   Sprite* bomb = nullptr;
   for(int i = 0 ; i < 3 ; i++)
   {
         bomb = Sprite::create("bomb.png");
         bomb->setPosition(CCRANDOM_0_1() * visibleSize.width,
         visibleSize.height + bomb->getContentSize().height/2);
```

```
            this->addChild(bomb,1);
            setPhysicsBody(bomb);
            bomb->getPhysicsBody()->setVelocity(Vect(0, ( (CCRANDOM_0_1()
            + 0.2f) * -250) ));
        }
    }
```

This code yields the following result:



By using the `CC_SCHEDULE_SELECTOR` macro, we are creating a custom selector, which in this case is called a **custom time interval**. The selected function should receive the `float` parameter that represents the amount of time that has elapsed between the last call and the current call, so that you can calculate a uniform game pace independently of the hardware processing speed. If you don't pass the second `float` parameter to the schedule function, then it will execute the selected function in each frame.

In the scene we are going to add another method to the scheduler, which is going to be called every three seconds, and it is going to add 10 points to the player's score. So the longer the player can avoid being hit by a bomb, the greater his or her score will be.

Now we have more than two physics bodies, this means that we have to modify our `onCollision` method so that it only changes to `gameOverScene` if the `player` sprite is involved in a collision. For this, we are going to add the following lines of code at the beginning of the method:

```
auto playerShape = _sprPlayer->getPhysicsBody()->getFirstShape();
if(playerShape != contact.getShapeA() && playerShape != contact.
getShapeB())
   {
      return false;
   }
```

If the method does not return, it means that the player sprite was indeed involved in a collision. So, we are going to write the score of our player that is stored in the member variable `_score` by using the Cocos2d-x built-in storage mechanism:

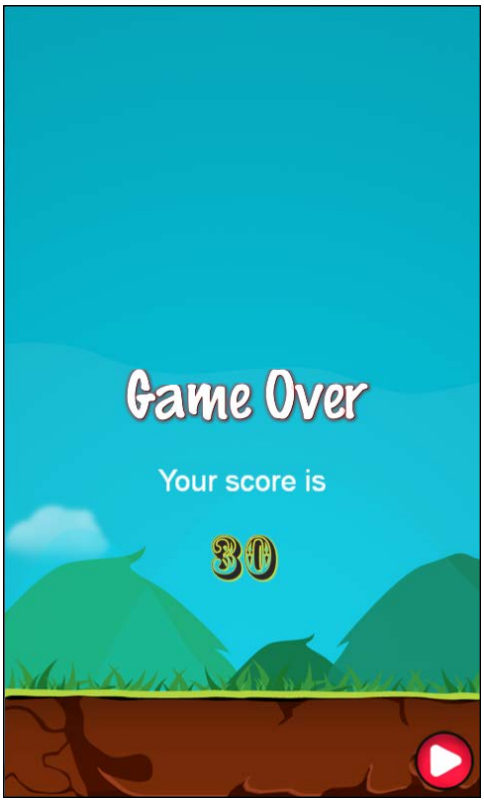
```
UserDefault::getInstance()->setIntegerForKey("score",_score);
```

The `UserDefault` class grants us access to the Cocos2d-x data storage mechanism. It can store `bool`, `int`, `float`, `double`, and `string` values. The data stored by using this class may be persisted by calling the `flush` method, which stores the data in an XML file.

We can create our bitmap fonts in a similar way to how we created our TrueType font and system font. We will add the following lines of code in the `init` method of the `GameOverScene.cpp` implementation file:

```
char scoreText[32];
int score = UserDefault::getInstance()->getIntegerForKey("score",0);
sprintf(scoreText, "%d", score);
auto label3 = Label::createWithBMFont("font.fnt", scoreText);
label3->setPosition(origin.x + visibleSize.width/2,origin.y +
visibleSize.height /3.5);
this->addChild(label3, 1);
```

The preceding code will produce the following result:



Putting everything together

After all our modifications, this is how our `HelloWorldScene.h` header file will look:

```
#ifndef __HELLOWORLD_SCENE_H__
#define __HELLOWORLD_SCENE_H__

#include "cocos2d.h"

#include "PauseScene.h"
#include "GameOverScene.h"
```

The inclusion of `GameOverScene.h` is the only change that we have made to this header file during this chapter:

```
class HelloWorld : public cocos2d::Layer
{
public:
    static cocos2d::Scene* createScene();
    virtual bool init();
    void pauseCallback(cocos2d::Ref* pSender);
    CREATE_FUNC(HelloWorld);
private:
   cocos2d::Director *_director;
   cocos2d::Size visibleSize;
   cocos2d::Sprite* _sprBomb;
   cocos2d::Sprite* _sprPlayer;
   int _score;
   void initPhysics();
   bool onCollision(cocos2d::PhysicsContact& contact);
   void setPhysicsBody(cocos2d::Sprite* sprite);
   void initTouch();
   void movePlayerByTouch(cocos2d::Touch* touch, cocos2d::Event*
   event);
   void movePlayerIfPossible(float newX);
   void movePlayerByAccelerometer(cocos2d::Acceleration*
   acceleration, cocos2d::Event* event);
   void initAccelerometer();
   void initBackButtonListener();
   void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode,
   cocos2d::Event* event);
   void updateScore(float dt);
   void addBombs(float dt);
};


#endif // __HELLOWORLD_SCENE_H__
```

Now, our `HelloWorldScene.cpp` implementation file looks like this:

```
#include "HelloWorldScene.h"
#include "../cocos2d/cocos/platform/android/jni/Java_org_cocos2dx_lib_
Cocos2dxHelper.h"

USING_NS_CC;

Scene* HelloWorld::createScene()
```

```
{
   auto scene = Scene::createWithPhysics();
   scene->getPhysicsWorld()->setGravity(Vect(0,0));
   auto layer = HelloWorld::create();
   //enable debug draw
   //scene->getPhysicsWorld()->setDebugDrawMask(PhysicsWorld::DEBUGDR
   AW_ALL);
   scene->addChild(layer);
   return scene;
}
```

We will now add the code for events and physics:

```
void HelloWorld::updateScore(float dt)
{
   _score += 10;
}

void HelloWorld::addBombs(float dt)
{
   Sprite* bomb = nullptr;
   for(int i = 0 ; i < 3 ; i++)
   {
      bomb = Sprite::create("bomb.png");
      bomb->setPosition(CCRANDOM_0_1() * visibleSize.width,
      visibleSize.height + bomb->getContentSize().height/2);
      this->addChild(bomb,1);
      setPhysicsBody(bomb);
      bomb->getPhysicsBody()->setVelocity(Vect(0,
      ( (CCRANDOM_0_1() + 0.2f) * -250) ));
   }
}

}

bool HelloWorld::init()
{
    if ( !Layer::init() )
    {
        return false;
    }
   _score = 0;
   _director = Director::getInstance();
   visibleSize = _director->getVisibleSize();
```

```
auto origin = _director->getVisibleOrigin();
auto closeItem = MenuItemImage::create("pause.png",
"pause_pressed.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
this));

closeItem->setPosition(Vec2(visibleSize.width - closeItem-
>getContentSize().width/2,
closeItem->getContentSize().height/2));

auto menu = Menu::create(closeItem, nullptr);
menu->setPosition(Vec2::ZERO);
this->addChild(menu, 1);
_sprBomb = Sprite::create("bomb.png");
_sprBomb->setPosition(visibleSize.width / 2,
visibleSize.height + _sprBomb->getContentSize().height/2);
this->addChild(_sprBomb,1);
auto bg = Sprite::create("background.png");
bg->setAnchorPoint(Vec2());
bg->setPosition(0,0);
this->addChild(bg, -1);
_sprPlayer = Sprite::create("player.png");
_sprPlayer->setPosition(visibleSize.width / 2,
visibleSize.height * 0.23);
setPhysicsBody(_sprPlayer);
this->addChild(_sprPlayer, 0);
//Animations
Vector<SpriteFrame*> frames;
Size playerSize = _sprPlayer->getContentSize();
frames.pushBack(SpriteFrame::create("player.png",
Rect(0, 0, playerSize.width, playerSize.height)));
frames.pushBack(SpriteFrame::create("player2.png",
Rect(0, 0, playerSize.width, playerSize.height)));
auto animation =
Animation::createWithSpriteFrames(frames,0.2f);
auto animate = Animate::create(animation);
_sprPlayer->runAction(RepeatForever::create(animate));

setPhysicsBody(_sprBomb);
initPhysics();
_sprBomb->getPhysicsBody()->setVelocity(Vect(0,-100));
initTouch();
```

```
    initAccelerometer();
    setKeepScreenOnJni(true);
    initBackButtonListener();
    schedule(CC_SCHEDULE_SELECTOR (HelloWorld::updateScore), 3.0f);
    schedule(CC_SCHEDULE_SELECTOR (HelloWorld::addBombs), 8.0f);
    return true;
}


void HelloWorld::pauseCallback(cocos2d::Ref* pSender){
    _director->pushScene(TransitionFlipX::
    create(1.0, Pause::createScene()));
}
```

Our `GameOverScene.h` header file now looks like this:

```
#ifndef __GameOver_SCENE_H__
#define __GameOver_SCENE_H__

#include "cocos2d.h"
#include "HelloWorldScene.h"

class GameOver : public cocos2d::Layer
{
public:
    static cocos2d::Scene* createScene();
    virtual bool init();
    void exitPause(cocos2d::Ref* pSender);
    CREATE_FUNC(GameOver);
private:
    cocos2d::Sprite* sprLogo;
    cocos2d::Director *director;
    cocos2d::Size visibleSize;
};

#endif // __Pause_SCENE_H__
```

And finally, our `GameOverScene.cpp` implementation file will look like this:

```
#include "GameOverScene.h"

USING_NS_CC;

Scene* GameOver::createScene()
{
```

```
    auto scene = Scene::create();
    auto layer = GameOver::create();
    scene->addChild(layer);
    return scene;
}

bool GameOver::init()
{
    if ( !Layer::init() )
    {
        return false;
    }
   director = Director::getInstance();
   visibleSize = director->getVisibleSize();
   Vec2 origin = director->getVisibleOrigin();
   auto pauseItem = MenuItemImage::create("play.png",
   "play_pressed.png", CC_CALLBACK_1(GameOver::exitPause, this));
   pauseItem->setPosition(Vec2(origin.x + visibleSize.width -
   pauseItem->getContentSize().width / 2, origin.y + pauseItem-
   >getContentSize().height / 2));
   auto menu = Menu::create(pauseItem, NULL);
   menu->setPosition(Vec2::ZERO);
   this->addChild(menu, 1);
   auto bg = Sprite::create("background.png");
   bg->setAnchorPoint(Vec2());
   bg->setPosition(0,0);
   this->addChild(bg, -1);
```

In the following lines of code, we have created the three types of fonts that we have covered during this chapter:

```
auto label = Label::createWithTTF("Game Over", "fonts/Marker
Felt.ttf", 96);
label->enableOutline(Color4B(255, 0, 0, 100),6);
label->enableGlow(Color4B(255, 0, 0, 255));
label->enableShadow();
label->setPosition(origin.x + visibleSize.width/2,
origin.y + visibleSize.height /2);
this->addChild(label, 1);
auto label2 = Label::createWithSystemFont("Your score is",
"Arial", 48);
```

```
    label2->setPosition(origin.x + visibleSize.width/2,origin.y
    + visibleSize.height/2.5);
    this->addChild(label2, 1);
    char scoreText[32];
    int score = UserDefault::getInstance()-
    >getIntegerForKey("score",0);
    sprintf(scoreText, "%d", score);
    auto label3 = Label::createWithBMFont("font.fnt", scoreText);
    label3->setPosition(origin.x + visibleSize.width/2,origin.y
    + visibleSize.height /3.5);
    this->addChild(label3, 1);
    return true;
}

void GameOver::exitPause(cocos2d::Ref* pSender){
    Director::getInstance()- >replaceScene(TransitionFlipX::
    create(1.0, HelloWorld::createScene()));
}
```

Summary

In this chapter, we have seen how to add text to our games by using the TrueType fonts, system fonts and bitmap fonts, as well as how to add effects to those. Label creation is very straightforward; you just need to call its creation static method, and after you have added those to the scene, you can position those on the screen, just like you positioned the sprites on the screen.

In the next chapter, we will cover the new audio engine that was written from scratch on the Version 3 for substituting the traditional `CocosDenshion` audio engine that was bundled with the engine since its predecessor `cocos2d` for iPhone.

6
Audio

The Cocos2d-x framework comes with an audio engine called `CocosDenshion`, which has been inherited from Cocos2d for iPhone. This engine encapsulates all the complexity of playing sound effects and background music. Now, Cocos2d-x has another audio engine that was built from scratch with the aim of providing more flexibility than the `CocosDenshion` library. Be aware that there is no plan to eliminate the `CocosDenshion` audio engine from the Cocos2d-x framework, it is now common in Cocos2d-x to have redundant components so that programmers can pick whatever fits their needs better.

The following topics will be covered within this chapter:

- Playing background music and sound effects
- Modifying the audio properties
- Handling audio when leaving the game
- The new audio engine

Playing background music and sound effects

In order to add the background music to our game by using the `CocosDenshion` audio engine, the first step is to add the following file inclusion to our `HelloWorldScene.cpp` implementation file:

```
#include "SimpleAudioEngine.h"
```

In this header file, within the private member segment, we will also add the declaration of our new `initAudio` method, which we are going to use for starting our background music and for preloading the audio effect that is going to be played each time a bomb crashes into our `player` sprite:

```
void initAudio();
```

Now in the `HelloWorld.cpp` implementation file, we are going to use the `CocosDenshion` namespace so that we don't have to implicitly refer to this namespace each time that we access the audio engine singleton instance:

```
using namespace CocosDenshion;
```

Now in the same implementation file, we are going to write the body of the `initAudio` method, which as we have previously mentioned will start playing a looped music background. We are providing this along with the source code of this chapter, and we are going to preload the audio effect that is going to be played each time our player loses. The second parameter of the `playBackgroundMusic` method is a Boolean, which determines whether or not we would like our background music to repeat forever.

```
void HelloWorld::initAudio()
{
   SimpleAudioEngine::getInstance()->playBackgroundMusic("music.mp3",
   true);
   SimpleAudioEngine::getInstance()->preloadEffect("uh.wav");
}
```

Let us create a folder called `sounds` in our `Resources` directory, so that we can add all our sound files there in an organized way. After doing this, we will need to add the following line to our `AppDelegate.cpp` implementation file after the instantiation of `searchPaths std::vector` for adding the `sounds` directory to the search path so that the audio engine can find our files:

```
searchPaths.push_back("sounds");
```

We encourage you to organize your `Resources` folder, create a folder for sounds with subfolders for audio and music, so that we don't have everything in the root directory.

Let us go to the `onCollision` method that is called each time two physics bodies collide. If the player sprite's physics body is involved in a crash, then we are going to stop the background music and play the `uh.wav` sound effect before changing to the game over scene by adding the following lines of code:

```
SimpleAudioEngine::getInstance()->stopBackgroundMusic();
SimpleAudioEngine::getInstance()->playEffect("uh.wav");
```

Finally, we are going to add a call for our `initAudio` method at the end of the `init` method in the `HelloWorld.cpp` implementation file:

```
initAudio();
```

Modifying audio properties

You can easily modify the background music and the basic audio properties of the sound effect by calling the `setBackgroundMusicVolume` method and the `setEffectsVolume` method. Both receive a `float` value as a parameter, where `0.0` means mute and `1.0` means maximum volume, as the following code listing shows:

```
SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(0.5f);
SimpleAudioEngine::getInstance()->setEffectsVolume(1.0f);
```

Handling audio when leaving the game

When the game activity is no longer active, the background music and the sound effects will not stop automatically, they should be stopped manually by removing the following comment block from the `applicationDidEnterBackgound` method in the `AppDelegate` class:

```
// if you use SimpleAudioEngine, it must be pause
 SimpleAudioEngine::getInstance()->pauseBackgroundMusic();
```

In order to make this new line of code work, we need to add the same line that we have added to the `HelloWorld.cpp` implementation file in order to use the `CocosDenshion` namespace:

```
using namespace CocosDenshion;
```

Your game will stop all the current sounds when the user switches to another application. We need to resume the music as soon as the user returns to our game. We can do that in the same way as we did before, but now, we will remove the following comment block from the `applicationWillEnterForeground` method in the `AppDelegate` class:

```
 // if you use SimpleAudioEngine, it must resume here
SimpleAudioEngine::getInstance()->resumeBackgroundMusic();
```

The new audio engine

Cocos2d-x 3.4 in its experimental phase had a new audio engine built from scratch in order to add more functionalities and flexibility. The new audio engine of Cocos2d-x is now available for the Android, iOS, the Mac OS, and the win-32 platforms. It is able to reproduce up to 24 simultaneous sounds on the Android platform; this number may change depending on the platform.

If you run the tests bundled with the Cocos2d-x framework, then you can test both of the audio engines. At runtime, they may sound without any apparent difference but they are internally very different.

Unlike the `CocosDenshion` engine, there is no distinction between the sound effect and the background music in this new engine. Therefore, the framework has only one `setVolume` method in contrast to `CocosDenshion`, which has two methods—`setBackgroundMusicVolume` and the `setEffectsVolume`. Later on in this section, we will show you how to adjust the volume of each played audio regardless of whether or not it is a sound effect of the background music.

Let us add a new method declaration to our `HelloWorldScene.h` header file named `initAudioNewEngine`, which as its name suggests will initialize the audio capabilities of our game, but it will now use the new audio engine for the same.

We need to include the new engine header in our `HelloWorldScene.h` file as follows:

```
#include "audio/include/AudioEngine.h"
```

Let us include the following line of code in our `HelloWorld.cpp` implementation file, so that we can invoke the `AudioEngine` class directly without referring to its namespace each time we want to use it:

```
using namespace cocos2d::experimental;
```

Now, let us write our code for the `initAudioNewEngine` method in our implementation file as follows:

```
void HelloWorld::initAudioNewEngine()
{
   if(AudioEngine::lazyInit())
   {
      auto musicId = AudioEngine::play2d("music.mp3");
      AudioEngine::setVolume(musicId, 0.25f);
      CCLOG("Audio initialized successfully");

   }else
```

```
        {
            log("Error while initializing new audio engine");
        }
    }
```

In contrast to `CocosDenshion`, which uses a singleton instance, the new audio engine has all of its methods declared static.

As we can see from the previous code listing, we are calling the `lazyInit` method before calling the `play2d` method. Despite the fact that `play2d` calls the `lazyInit` method internally, we want to know as soon as possible whether or not our Android system is going to be able to reproduce the audio and take actions. Be aware that you also need to find out whether something is wrong with the audio initialization when the `play2d` method returns the `AudioEngine::INVALID_AUDIO_ID` value.

Each time that we play any given sound by calling the `play2d` method, it will return a unique incremental `audioID` zero-based index, which we will store, so that we can refer to that specific audio instance each time that we want to perform a particular action on it, such as change volume, move to a specific position, stop, pause, or resume.

One disadvantage of the new audio engine is that it still has a limited amount of supported audio formats. It does not currently support the `.wav` files. So in order to play the `uh.wav` sound, we will convert it to mp3 and then we will play it in the `onCollision` method by invoking `play2d` as follows:

```
AudioEngine::stopAll();
AudioEngine::play2d("uh.mp3");
```

We have included the new `uh.mp3` audio file in the code resources archive provided with this chapter.

For our game, we are going to implement both; the traditional `CocosDenshion` engine, which is the most mature audio engine that provides us with the basic features that we need, such as playing the sound effects and the background music; and the same audio functionalities in the new engine.

New features included in the new audio engine

The `play2d` method is overloaded so that we can specify whether we want to loop the sound, the initial volume, and the audio profile that we would like to apply to it. The `AudioProfile` class is a part of the Cocos2d-x framework, it has only three properties: `name`, which cannot be empty; `maxInstances`, which will define how many sounds will be played simultaneously; and `minDelay`, which is a `double` data type that will specify what the minimum delay between the sounds is going to be.

Another feature that the new audio engine has is the ability to play an audio from a custom position by calling the `setCurrentTime` method and passing the `audioID` method and the custom position in seconds, which are represented by a `float`.

In the new audio engine, you can specify that you would like to invoke a function when a given audio instance has completed playing. This can be done by calling the `setFinishCallback` method.

Each time an audio is played, it is cached so that it is not necessary to read it again from the filesystem. If we would like to free some resources, then we could call the `uncacheAll` method to remove all the buffer that is used internally by the audio engine for playing back the audio, or you could remove any specific audio from the cache by calling the `uncache` method and specifying the path of the file of the filesystem that you want to remove.

The purpose of this section was to make you aware of another audio engine that is in an experimental phase, and if there is any audio feature that you would like to add to your game, which `CocosDenshion` does not have, then you should check the other audio engine to see whether it has what you are looking for.

The new audio engine can play up to 32 sounds simultaneously on the Mac OS, iOS, and the win-32 platforms, but it can only play up to 24 simultaneous sounds on Android.

Adding a mute button to our game

Before ending this chapter, we are going to add a mute button to our game so that we can simply set our sound effects and our background music volume to zero in a single touch.

In order to achieve this, we are going to add two methods to the `HelloWorld` class; one for initializing the buttons and the other for actually muting all the sounds.

In order to achieve this, we are going to add the following lines to our `HelloWorldScene.h` header file under the private section:

```
int _musicId;
cocos2d::MenuItemImage* _muteItem;
cocos2d::MenuItemImage* _unmuteItem;
```

```
void initMuteButton();
void muteCallback(cocos2d::Ref* pSender);
```

Now, we are going to add the `initMuteButton` implementation code to the `HelloWorldScene.cpp` file as follows:

```
void HelloWorld::initMuteButton()
{
   _muteItem = MenuItemImage::create("mute.png", "mute.png",
   CC_CALLBACK_1(HelloWorld::muteCallback, this));

   _muteItem->setPosition(Vec2(_visibleSize.width - _muteItem-
   >getContentSize().width/2 ,
   _visibleSize.height - _muteItem->getContentSize().
   height / 2));
   _unmuteItem = MenuItemImage::create("unmute.png", "unmute.png",
   CC_CALLBACK_1(HelloWorld::muteCallback, this));

   _unmuteItem->setPosition(Vec2(_visibleSize.width - _unmuteItem-
   >getContentSize().width/2 ,
   _visibleSize.height - _unmuteItem->getContentSize().height /2));
   _unmuteItem -> setVisible(false);

   auto menu = Menu::create(_muteItem, _unmuteItem , nullptr);
   menu->setPosition(Vec2::ZERO);
   this->addChild(menu, 1);
}
```

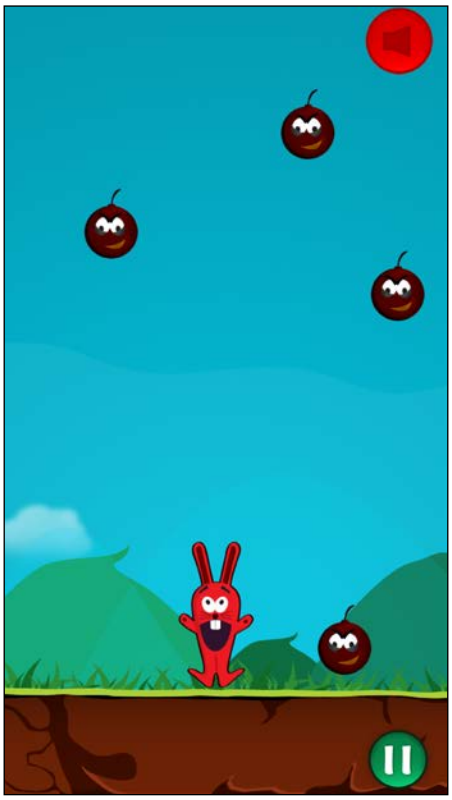
As you can see we have just created a new menu, where we have added two buttons, one for muting our game and the other invisible for unmuting it. We have stored each of these in the member variables, so that we can access them by the `muteCallback` method, which we have declared in the following code listing:

```
void HelloWorld::muteCallback(cocos2d::Ref* pSender)
{
   if(_muteItem -> isVisible())
   {

      //CocosDenshion
      //SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(0);
      AudioEngine::setVolume(_musicId, 0);
   }else
```

```
        {
            //SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(1);
            AudioEngine::setVolume(_musicId, 1);
        }

        _muteItem->setVisible(!_muteItem->isVisible());
        _unmuteItem->setVisible(!_muteItem->isVisible());
    }
```

Here, we are basically just asking whether or not the `_muteItem` menu item is visible. If it is visible, then set the volume to zero by using `CocosDenshion`, the new audio engine, else set the volume to its maximum value, that is, one. In either case, change the actual visibility value in both mute and unmute menu items.

We can see the final result in the following screenshot:



Putting everything together

This is how our `applicationDidFinishLaunching` method from our `AppDelegate.cpp` implementation file looks after we have added the line for including the `sounds` folder in the `resources` path:

```
bool AppDelegate::applicationDidFinishLaunching() {
    auto director = Director::getInstance();
    // OpenGL initialization done by cocos project creation script
    auto glview = director->getOpenGLView();
    if(!glview) {
    glview = GLViewImpl::create("Happy Bunny");
    glview->setFrameSize(480, 800);
    director->setOpenGLView(glview);
   }

   Size screenSize = glview->getFrameSize();
   Size designSize(768, 1280);
   std::vector<std::string> searchPaths;
   searchPaths.push_back("sounds");

   if (screenSize.height > 800){
      //High Resolution
      searchPaths.push_back("images/high");
      director->setContentScaleFactor(1280.0f / designSize.height);
   }
   else if (screenSize.height > 600){
      //Mid resolution
      searchPaths.push_back("images/mid");
      director->setContentScaleFactor(800.0f / designSize.height);
   }
   else{
      //Low resolution
      searchPaths.push_back("images/low");
      director->setContentScaleFactor(320.0f / designSize.height);
   }
   FileUtils::getInstance()->setSearchPaths(searchPaths);
   glview->setDesignResolutionSize(designSize.width, designSize.
   height, ResolutionPolicy::EXACT_FIT);
   auto scene = HelloWorld::createScene();
   director->runWithScene(scene);
   return true;
}
```

The following code listing shows how our `HelloWorldScene.h` header file looks after the changes we have made during this chapter:

```
#ifndef __HELLOWORLD_SCENE_H__
#define __HELLOWORLD_SCENE_H__

#include "cocos2d.h"
#include "PauseScene.h"
#include "GameOverScene.h"

class HelloWorld : public cocos2d::Layer
{
public:
    static cocos2d::Scene* createScene();
    virtual bool init();
    CREATE_FUNC(HelloWorld);
private:
   cocos2d::Director *_director;
   cocos2d::Size _visibleSize;
   cocos2d::Sprite* _sprBomb;
   cocos2d::Sprite* _sprPlayer;
   cocos2d::MenuItemImage* _muteItem;
   cocos2d::MenuItemImage* _unmuteItem;
   int _score;
   int _musicId;
   void initPhysics();
   void pauseCallback(cocos2d::Ref* pSender);
   void muteCallback(cocos2d::Ref* pSender);
   bool onCollision(cocos2d::PhysicsContact& contact);
   void setPhysicsBody(cocos2d::Sprite* sprite);
   void initTouch();
   void movePlayerByTouch(cocos2d::Touch* touch, cocos2d::Event*
   event);
   void movePlayerIfPossible(float newX);
   void movePlayerByAccelerometer(cocos2d::Acceleration* acceleration,
   cocos2d::Event* event);
   void initAccelerometer();
   void initBackButtonListener();
   void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode,
   cocos2d::Event* event);
   void updateScore(float dt);
   void addBombs(float dt);
   void initAudio();
   void initAudioNewEngine();
```

```
    void initMuteButton();
};

#endif // __HELLOWORLD_SCENE_H__
```

And finally, after adding the audio management code, this is how our `HelloWorldScene.cpp` implementation file looks:

```
#include "HelloWorldScene.h"#include "SimpleAudioEngine.h"
#include "audio/include/AudioEngine.h"
#include "../cocos2d/cocos/platform/android/jni/Java_org_cocos2dx_lib_
Cocos2dxHelper.h"

USING_NS_CC;
using namespace CocosDenshion;
using namespace cocos2d::experimental;

Scene* HelloWorld::createScene()
{
  //no changes here
}

// physics code …

// event handling code …
```

In the following method, we will initialize the audio by using the new audio engine. Notice that we will store the ID that corresponds to the background music of the audio instance in the `_musicId` integer member variable:

```
void HelloWorld::initAudioNewEngine()
{
   if(AudioEngine::lazyInit())
   {
      _musicId = AudioEngine::play2d("music.mp3");
      AudioEngine::setVolume(_musicId, 1);
      AudioEngine::setLoop(_musicId,true);
      CCLOG("Audio initialized successfully");
   }else
   {
      CCLOG("Error while initializing new audio engine");
   }
}
```

Here, we are performing the same initialization job that we did in the previous method, but now we are doing so by using the `CocosDenshion` audio engine:

```
void HelloWorld::initAudio()
{
   SimpleAudioEngine::getInstance()->playBackgroundMusic("music.
   mp3",true);
   SimpleAudioEngine::getInstance()->preloadEffect("uh.wav");
   SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(1.0f);
}
```

In the following method, we are creating a simple menu for presenting the options for muting and unmuting the game. Here we are storing the mute and the unmute sprites in their corresponding member variables, so that we can later access them within the `muteCallback` method for manipulating their `visibility` property:

```
void HelloWorld::initMuteButton()
{
   _sprMute = Sprite::create("mute.png");
   _sprUnmute = Sprite::create("unmute.png");
   _muteItem = MenuItemImage::create("mute.png", "mute.png",
   CC_CALLBACK_1(HelloWorld::muteCallback, this));
   _muteItem->setPosition(Vec2(_visibleSize.width - _muteItem-
   >getContentSize().width/2 ,
   _visibleSize.height - _muteItem->getContentSize().height / 2));
   _unmuteItem = MenuItemImage::create("unmute.png", "unmute.png",
   CC_CALLBACK_1(HelloWorld::muteCallback, this));
   _unmuteItem->setPosition(Vec2(_visibleSize.width - _unmuteItem-
   >getContentSize().width/2 ,
   _visibleSize.height - _unmuteItem->getContentSize(). height /2));
   _unmuteItem -> setVisible(false);
   auto menu = Menu::create(_muteItem, _unmuteItem , nullptr);
    menu->setPosition(Vec2::ZERO);
    this->addChild(menu, 1);

}
```

The following method will be called each time the mute or the unmute menu item is pressed, in this method we just set the volume to zero and show the mute or unmute button depending on the option that was touched:

```
void HelloWorld::muteCallback(cocos2d::Ref* pSender)
{
   if(_muteItem -> isVisible())
```

```
    {
        //CocosDenshion
        //SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(0);
        AudioEngine::setVolume(_musicId, 0);


    }else
    {
        //SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(1);
        AudioEngine::setVolume(_musicId, 1);
    }

    _muteItem->setVisible(!_muteItem->isVisible());
    _unmuteItem->setVisible(!_muteItem->isVisible());
}
```

The single modification that we made to the `init` method was to add the call to the `initMuteButton();` method at the end of it:

```
bool HelloWorld::init()
{
    if ( !Layer::init() )
    {
        return false;
    }
    _score = 0;
    _director = Director::getInstance();
    _visibleSize = _director->getVisibleSize();
    auto origin = _director->getVisibleOrigin();
    auto closeItem = MenuItemImage::create("pause.png",
    "pause_pressed.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
    this));
    closeItem->setPosition(Vec2(_visibleSize.width - closeItem-
    >getContentSize().width/2, closeItem->getContentSize().height/2));
    auto menu = Menu::create(closeItem, nullptr);
    menu->setPosition(Vec2::ZERO);
    this->addChild(menu, 1);
    _sprBomb = Sprite::create("bomb.png");
    _sprBomb->setPosition(_visibleSize.width / 2, _visibleSize.height +
    _sprBomb->getContentSize().height/2);
    this->addChild(_sprBomb,1);
    auto bg = Sprite::create("background.png");
```

```
    bg->setAnchorPoint(Vec2());
    bg->setPosition(0,0);
    this->addChild(bg, -1);
    _sprPlayer = Sprite::create("player.png");
    _sprPlayer->setPosition(_visibleSize.width / 2, _visibleSize.height
    * 0.23);
    setPhysicsBody(_sprPlayer);
    this->addChild(_sprPlayer, 0);
    //Animations
    Vector<SpriteFrame*> frames;
    Size playerSize = _sprPlayer->getContentSize();
    frames.pushBack(SpriteFrame::create("player.png", Rect(0, 0,
    playerSize.width, playerSize.height)));
    frames.pushBack(SpriteFrame::create("player2.png", Rect(0, 0,
    playerSize.width, playerSize.height)));
    auto animation = Animation::createWithSpriteFrames(frames,0.2f);
    auto animate = Animate::create(animation);
    _sprPlayer->runAction(RepeatForever::create(animate));
    setPhysicsBody(_sprBomb);
    initPhysics();
    _sprBomb->getPhysicsBody()->setVelocity(Vect(0,-100));
    initTouch();
    initAccelerometer();
    #if (CC_TARGET_PLATFORM == CC_PLATFORM_ANDROID)
    setKeepScreenOnJni(true);
    #endif
    initBackButtonListener();
    schedule(CC_SCHEDULE_SELECTOR(HelloWorld::updateScore), 3.0f);
    schedule(CC_SCHEDULE_SELECTOR(HelloWorld::addBombs), 8.0f);
    initAudioNewEngine();
    initMuteButton();
    return true;
}
```

As you can see, despite the fact that we have used the new audio engine for playing sounds, we have shown all the code that is required for using the traditional `CocosDenshion` audio engine. To enable the `CocosDenshion` implementation, you would only need to call the `initAudio` method at the bottom of the `init` method of the `HelloWorld.cpp` file, instead of calling the `initAudioNewEngine` method, and finally, you will need to remove the comment slashes for the `CocosDenshion` implementation code in the `onCollision` method and comment the new audio engine playback code.

Summary

In this chapter, we have added the background music and the sound effects to our game in a very simple way by using the two audio engines bundled with the Cocos2d-x framework.

In the next chapter, we will cover how to add particle systems to our games in order to simulate more realistic explosions each time a bomb hits our `player` sprite.

7
Creating Particle Systems

You can easily simulate fire, smoke, explosions, snow, and rain by using the built-in particle system bundled with the Cocos2d-x framework. This chapter will teach you how to create the effects mentioned here and will also teach you to customize them.

The following topics will be covered in this chapter:

- Creating collections of the Cocos2d-x objects
- Adding particle systems to our game
- Configuring the particle systems
- Creating customized particle systems

Creating collections of the Cocos2d-x objects

We are going to add a particle system to our game to simulate the explosions each time the player touches a bomb. In the interest of doing this, we are going to use the `Vector` class located in the Cocos2d-x framework to create a collection of all the bomb objects created in our game, so that when the player touches the screen, we are going to traverse this collection to verify if the player has touched any of the bombs.

If the player touches any bomb, we are going to:

- Show an explosion at the location where the bomb sprite was
- Make the bomb invisible
- Remove the bomb from the screen by using the inherited `removeChild` method, and finally
- Remove the bomb object from the collection, so that the next time we traverse the vector, it is disregarded

For this matter, we are going to add the bomb collection to our `HelloWorldScene.h` definition file as follows:

```
cocos2d::Vector<cocos2d::Sprite*> _bombs;
```

Be aware that we are specifying that we want to use the `Vector` class bundled with the `cocos2d` namespace, so that the compiler can clearly know that we are referring to the framework's built-in collection class and not the `Vector` class contained in the `std` namespace. Although it is possible to use the `Vector` class in the `std` namespace, the one located in the framework is optimized so as to be used in the Cocos2d-x object collection.

The `Vector` class introduced in the Cocos2d-x 3.0 represents the object collections by using the C++ standards, as opposed to the deprecated `CCArray` class that modeled the Cocos2d-x object collections, by using the objective C container classes. This new class takes care of the reference counting mechanism used in Cocos2d-x for memory management, it also adds the functionalities that are non-existent in `std::vector`, such as the `random`, `contains`, and `equals` methods.

You should only use the `std::vector` instances if you need to pass them as a parameter to a function of a Cocos2d-x API class that is expecting the data type such as the `setSearchPaths` method in the `FileUtils` class.

Now let us go to the `init` method located in our `HelloWorldScene.cpp` implementation file, and just next to the declaration of the `_sprBomb` variable that holds a reference to the first bomb sprite, we are going to add this reference to our new `_bombs` collection as follows:

```
_bombs.pushBack(_sprBomb);
```

Now, let us go to the `addBombs` method that we have created in our previous chapters for adding more bombs to our game. In this method, we are going to add each bomb spawned in our game scene to the `_bombs` collection as follows:

```
void HelloWorld::addBombs(float dt)
{
Sprite* bomb = nullptr;
for(int i = 0; i < 3; i++){
  bomb = Sprite::create("bomb.png");
  bomb->setPosition(CCRANDOM_0_1() * visibleSize.width,
visibleSize.height + bomb->getContentSize().height/2);
  this->addChild(bomb,1);
  setPhysicsBody(bomb);
```

```
  bomb->getPhysicsBody()->setVelocity(Vect(0,
( (CCRANDOM_0_1() + 0.2f) * -250) ));
  _bombs.pushBack(bomb);
}
}
```

Exploding bombs

We want our bombs to explode when we touch them. To achieve this, we are going to create our `explodeBombs` method. In the `HelloWorldScene.h` header file, we are going to write the declaration as follows:

```
bool explodeBombs(cocos2d::Touch* touch, cocos2d::Event* event);
```

Now we are going to write the method body in the `HelloWorldScene.cpp` implementation file; as mentioned earlier, each time the player touches the screen we can verify the touched location and compare it to the location of each bomb. If any intersection is found, then the touched bombs are going to disappear. As yet, we are not going to add any particle systems, we are going do that in later sections:

```
bool HelloWorld::explodeBombs(cocos2d::Touch* touch,
cocos2d::Event* event)
{
Vec2 touchLocation = touch->getLocation();
cocos2d::Vector<cocos2d::Sprite*> toErase;
for(auto bomb : _bombs){
  if(bomb->getBoundingBox().containsPoint(touchLocation)){
    bomb->setVisible(false);
    this->removeChild(bomb);
    toErase.pushBack(bomb);
  }
}

for(auto bomb : toErase){
  _bombs.eraseObject(bomb);
}
return true;
}
```

Please note that we have created another vector for the purpose of adding all the bombs that were touched by the user, and then they were removed from the `_bombs` collection in the other loop. The reason we did this instead of removing the objects directly from the first loop is that it would have given a runtime error. This is because we cannot perform concurrent modifications in a single collection, that is, we cannot remove an item from a collection while we are iterating over it. If we did that, then we would get a runtime error.

The `Vector` class was introduced in the Cocos2d-x 3.0. It has replaced the `CCArray` class previously used in Cocos2d-x 2.x. We can iterate over the `Vector` instances by using C++11 for each feature; therefore, the `CCARRAY_FOREACH` macro that we have used in Cocos2d-x 2.x for iterating over the Cocos2d-x objects is no longer needed.

Now, we are going to add a callback to our touch listener on the `onTouchBegan` property by making the following changes in our `initTouch` method located in our `HelloWorldScene.cpp` implementation file:

```
void HelloWorld::initTouch()
{
  auto listener = EventListenerTouchOneByOne::create();
  listener->onTouchBegan =
CC_CALLBACK_2(HelloWorld::explodeBombs,this);
  listener->onTouchMoved =
CC_CALLBACK_2(HelloWorld::movePlayerByTouch,this);
  listener->onTouchEnded = [=](Touch* touch, Event* event){};
  _eventDispatcher->addEventListenerWithScene
GraphPriority(listener, this);
}
```

And that is it, now the bombs will disappear whenever you touch them. In the next section, we will add an explosion effect to enhance the appearance of our game.

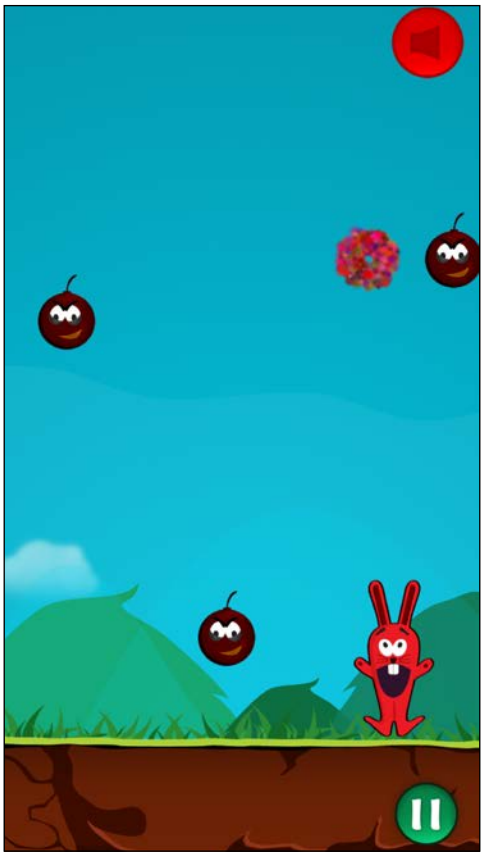
Adding particle systems to our game

Cocos2d-x has built-in classes that allow you to render the most common visual effects such as explosions, fire, fireworks, smoke, and rain, among others, by displaying a large number of small graphic objects referred to as particles.

It is very easy to implement. Let us add a default explosion effect, by simply adding the following lines to our `explodeBombs` method:

```
bool HelloWorld::explodeBombs(cocos2d::Touch* touch,
cocos2d::Event* event){
  Vec2 touchLocation = touch->getLocation();
  cocos2d::Vector<cocos2d::Sprite*> toErase;
  for(auto bomb : _bombs){
    if(bomb->getBoundingBox().containsPoint(touchLocation)){
      auto explosion = ParticleExplosion::create();
      explosion->setPosition(bomb->getPosition());
      this->addChild(explosion);
      bomb->setVisible(false);
```

```
        this->removeChild(bomb);
        toErase.pushBack(bomb);
      }
    }
    for(auto bomb : toErase){
      _bombs.eraseObject(bomb);
    }
    return true;
}
```



You can try the other particle systems embedded in the engine by changing the name of the particle class in the first highlighted line of the preceding code listing with the following class names: `ParticleFireworks`, `ParticleFire`, `ParticleRain`, `ParticleSnow`, `ParticleSmoke`, `ParticleSpiral`, `ParticleMeteor`, and `ParticleGalaxy`.

Configuring the particle systems

In the previous section, we created a realistic explosion by simply adding three lines of code. We can customize many parameters of a particle system. For instance, we could modify how much we would like our particle system to expand by modifying the life property.

We could also modify how big or how small we would want our particle system to be at the beginning and the size that we want it to have at the end, by setting the values to the `startSize` property and the `endSize` property. For example, if we would like to simulate the turbine of a rocket, then we could configure the emitter to start with a small size and to end with a big size.

We can adjust the angle to which the particle is going to move by modifying the angle property. You can assign random angles to your particle systems, so that it will look more realistic.

The particle systems can have two modes, radius and gravity. The most common particle systems use the gravity mode, on which we can parameterize the gravity, speed, radial, and tangential acceleration. This means that the particles created by the emitter are attracted by a force called gravity, and we can customize theirs horizontal and vertical components. The radial mode has the radial movement and rotation, so this mode of the particle systems will swirl in a spiral.

The total amount of particles can also be changed through the `totalParticles` property. The greater the amount of particles, the thicker the particle system will appear, but be aware that the amount of particles rendered also impacts the running performance. To give you an idea, the default explosion particle system has 700 total particles and the smoke effect has 200 total particles.

You can modify the properties mentioned in this section by calling the method, set<property name> in the emitter instance. For instance, if you want to modify the total particles of the system, then call the `setTotalParticles` method.

In the following code listing, we will modify the total amount of particles, the speed, and the lifespan of the particle system:

```
bool HelloWorld::explodeBombs(cocos2d::Touch* touch,
cocos2d::Event* event){
  Vec2 touchLocation = touch->getLocation();
  cocos2d::Vector<cocos2d::Sprite*> toErase;
  for(auto bomb : _bombs){
    if(bomb->getBoundingBox().containsPoint(touchLocation)){
```

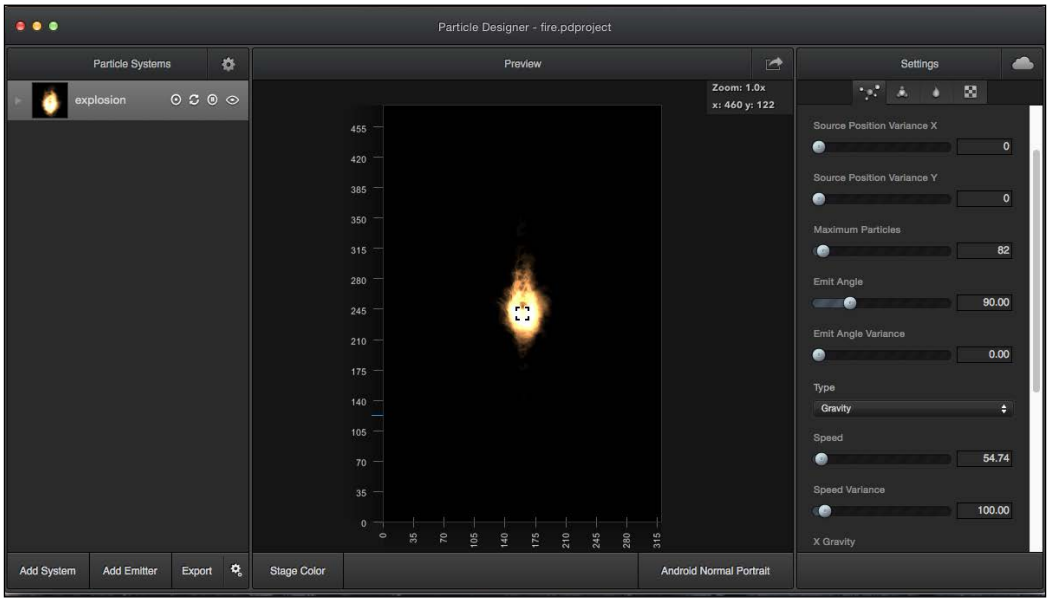
```
            auto explosion = ParticleExplosion::create();
            explosion->setDuration(0.25f);
            AudioEngine::play2d("bomb.mp3");
            explosion->setPosition(bomb->getPosition());
            this->addChild(explosion);
            explosion->setTotalParticles(800);
            explosion->setSpeed(3.5f);
            explosion->setLife(300.0f);
            bomb->setVisible(false);
            this->removeChild(bomb);
            toErase.pushBack(bomb);
        }
    }

    for(auto bomb : toErase){
      _bombs.eraseObject(bomb);
    }
    return true;
}
```

Creating customized particle systems

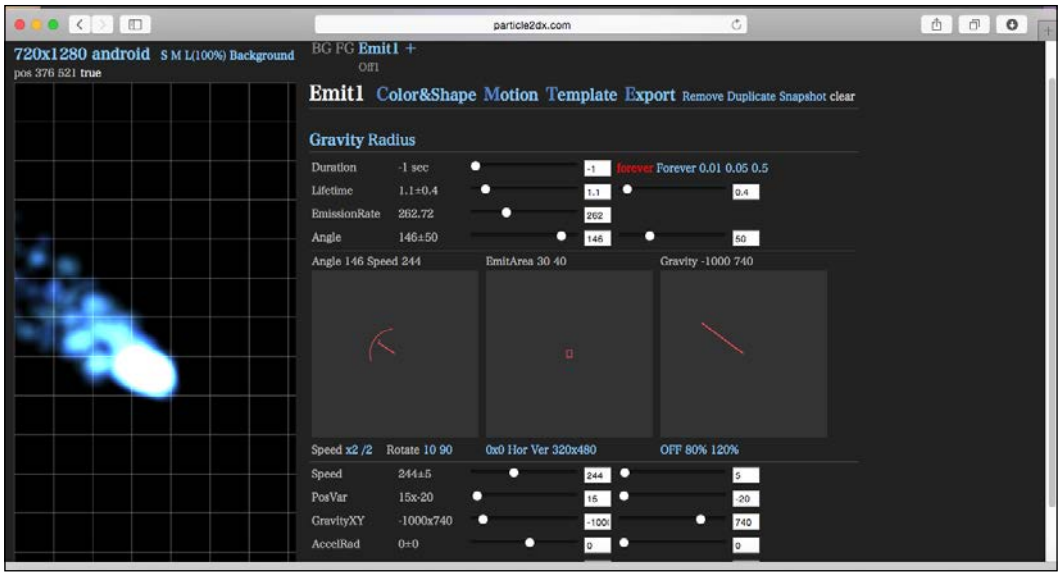
So far, we have tried all the particle systems that are bundled with the Cocos2d-x framework, but during our journey as games developers, there will be many situations where we will have to create our own particle systems.

There are tools which allow us to create particle systems and adjust their properties in a very graphical way. This allows us to create the **What You See Is What You Get** (**WYSIWYG**) type of particle system.

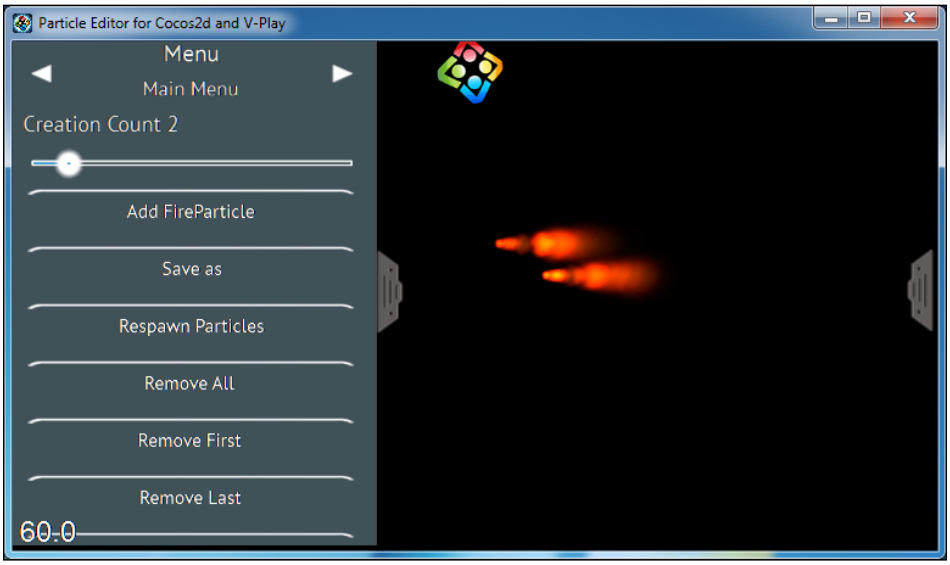
The most common application for creating the particle systems, which is actually mentioned several times within the official Cocos2d-x documentation, is called Particle Designer. It is currently available only for the Mac OS, and you need to buy the license in order to export the particle system to a plist file. You can download it and try it for free from the following link: `https://71squared.com/particledesigner`. The Particle Designer is shown in the following screenshot:



You can also create your particle systems in a graphical way by using the following web app hosted at `http://www.particle2dx.com/` for free:



You can also use the V-Play particle editor, which can be downloaded and used for free on the Windows, Android, iOS, and Mac platforms. These tools are available and can be downloaded from `http://games.v-play.net/particleeditor`.



By using any of the previously mentioned tools, you can adjust the particle system properties, such as max particles, duration, lifetime, emission rate, and angle, and then save it in a plist file.

We have created our own particle system and exported it to a plist file. We are including it in the code archive bundled with the source code of this chapter. We have placed this plist file in a newly created folder, which is in the `Resources` directory named `particles`.

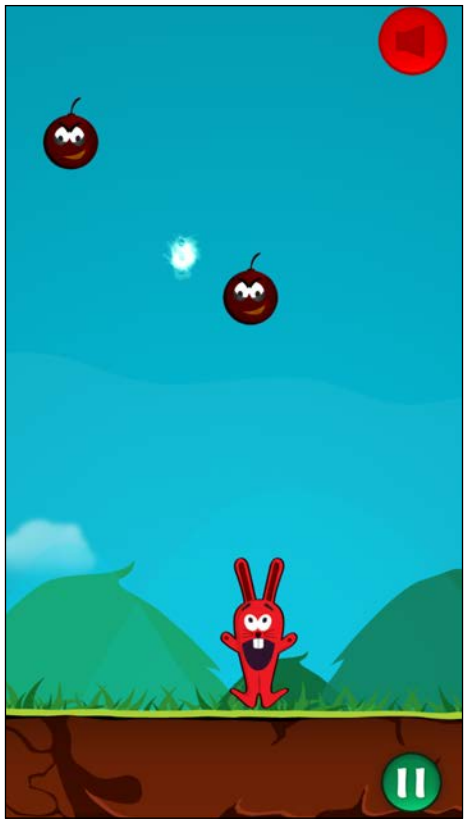
Since our plist file is not in the root of the `Resources` folder, we need to add the `particles` directory to the search path within the `applicationDidFinishLaunching` method of the `AppDelegate` class by adding the following line of code just after adding the `sounds` directory to the `searchPaths`:

```
searchPaths.push_back("particles");
```

The following code listing shows how to display our custom particle system by using the `ParticleSystemQuad` class and passing as a parameter to its `create` static method the name of the plist file generated by using the tool:

```
bool HelloWorld::explodeBombs(cocos2d::Touch* touch,
cocos2d::Event* event){
  Vec2 touchLocation = touch->getLocation();
  cocos2d::Vector<cocos2d::Sprite*> toErase;
  for(auto bomb : _bombs){
    if(bomb->getBoundingBox().containsPoint(touchLocation)){
      AudioEngine::play2d("bomb.mp3");
      auto explosion = ParticleSystemQuad::create("explosion.plist");
      explosion->setPosition(bomb->getPosition());
      this->addChild(explosion);
      bomb->setVisible(false);
      this->removeChild(bomb);
      toErase.pushBack(bomb);
    }
  }
  for(auto bomb : toErase){
    _bombs.eraseObject(bomb);
  }
  return true;
}
```

As you can see we have also added a line of code for playing an audio each time a bomb reaches our player sprite, so that it adds a more realistic effect. We have added this MP3 file to the code provided in this chapter.



Putting everything together

During this chapter, we have added the particle systems to our game for adding realistic explosions each time our player touches a bomb. To achieve this goal, we have modified the `HelloWorldScene.h` header file and the `HelloWorldScene.cpp` implementation file.

This is how our `HelloWorldScene.h` header file looks, after the modifications made during this chapter:

```
#ifndef __HELLOWORLD_SCENE_H__
#define __HELLOWORLD_SCENE_H__
#include "cocos2d.h"
#include "PauseScene.h"
```

```
#include "GameOverScene.h"

class HelloWorld : public cocos2d::Layer{
public:
  static cocos2d::Scene* createScene();
  virtual bool init();
  CREATE_FUNC(HelloWorld);
private:
  cocos2d::Director *_director;
  cocos2d::Size _visibleSize;
  cocos2d::Sprite* _sprBomb;
  cocos2d::Sprite* _sprPlayer;
  cocos2d::Vector<cocos2d::Sprite*> _bombs;
  cocos2d::MenuItemImage* _muteItem;
  cocos2d::MenuItemImage* _unmuteItem;
  int _score;
  int _musicId;
  void initPhysics();
  void pauseCallback(cocos2d::Ref* pSender);
  void muteCallback(cocos2d::Ref* pSender);
  bool onCollision(cocos2d::PhysicsContact& contact);
  void setPhysicsBody(cocos2d::Sprite* sprite);
  void initTouch();
  void movePlayerByTouch(cocos2d::Touch* touch,
  cocos2d::Event* event);
  void movePlayerIfPossible(float newX);
  bool explodeBombs(cocos2d::Touch* touch, cocos2d::Event* event);
  void movePlayerByAccelerometer
  (cocos2d::Acceleration* acceleration, cocos2d::Event* event);
  void initAccelerometer();
  void initBackButtonListener();
  void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode,
  cocos2d::Event* event);
  void updateScore(float dt);
  void addBombs(float dt);
  void initAudio();
  void initAudioNewEngine();
  void initMuteButton();
};

#endif // __HELLOWORLD_SCENE_H__
```

Finally, the following code listing shows how our `HelloWorldScene.cpp` implementation file looks after the modifications that we have made during this chapter:

```
#include "HelloWorldScene.h"
#include "SimpleAudioEngine.h"
#include "audio/include/AudioEngine.h"
#include "../cocos2d/cocos/platform/android/jni/
Java_org_cocos2dx_lib_Cocos2dxHelper.h"

USING_NS_CC;
using namespace CocosDenshion;
using namespace cocos2d::experimental;
//Create scene code …
//User input event handling code
```

Within the following method, we first verify whether or not the user has touched a bomb, if the user has, then an explosion particle system will be rendered at the location where the bomb was at the moment of the touch.

```
bool HelloWorld::explodeBombs(cocos2d::Touch* touch,
cocos2d::Event* event){
  Vec2 touchLocation = touch->getLocation();
  cocos2d::Vector<cocos2d::Sprite*> toErase;
  for(auto bomb : _bombs){
    if(bomb->getBoundingBox().containsPoint(touchLocation)){
      AudioEngine::play2d("bomb.mp3");
      auto explosion = ParticleSystemQuad::create
      ("explosion.plist");
      explosion->setPosition(bomb->getPosition());
      this->addChild(explosion);
      bomb->setVisible(false);
      this->removeChild(bomb);
      toErase.pushBack(bomb);
    }
  }
  for(auto bomb : toErase){
    _bombs.eraseObject(bomb);
  }
  return true;
}
```

In the following method, we have added an event listener that will be triggered each time the user touches the screen in order to verify whether a bomb has been touched:

```
void HelloWorld::initTouch(){
  auto listener = EventListenerTouchOneByOne::create();
  listener->onTouchBegan =
  CC_CALLBACK_2(HelloWorld::explodeBombs,this);
  listener->onTouchMoved =
  CC_CALLBACK_2(HelloWorld::movePlayerByTouch,this);
  listener->onTouchEnded = [=](Touch* touch, Event* event){};
  _eventDispatcher->addEventListenerWithScene
  GraphPriority(listener, this);
}
```

In the following method, we are adding the newly spawned bombs to our new `cocos2d:Vector` collection by using its `pushBack` method:

```
void HelloWorld::addBombs(float dt)
{
  Sprite* bomb = nullptr;
  for(int i = 0; i < 3; i++){
    bomb = Sprite::create("bomb.png");
    bomb->setPosition(CCRANDOM_0_1() * visibleSize.width,
    visibleSize.height + bomb->getContentSize().height/2);
    this->addChild(bomb,1);
    setPhysicsBody(bomb);
    bomb->getPhysicsBody()->setVelocity
    (Vect(0, ( (CCRANDOM_0_1() + 0.2f) * -250) ));
    _bombs.pushBack(bomb);
  }
}
```

Now we are going to see how our `init` method looks after the modifications done in this chapter. Note that we have added the first bomb, which was created during the initialization phase, to our new `cocos2d:Vector _bombs` collection.

```
bool HelloWorld::init()
{
  if ( !Layer::init() ){
    return false;
  }
  _score = 0;
  _director = Director::getInstance();
  _visibleSize = _director->getVisibleSize();
  auto origin = _director->getVisibleOrigin();
```

```
auto closeItem = MenuItemImage::create("pause.png",
"pause_pressed.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
this));
closeItem->setPosition(Vec2
(_visibleSize.width - closeItem->getContentSize().width/2 ,
closeItem->getContentSize().height/2));
auto menu = Menu::create(closeItem, nullptr);
menu->setPosition(Vec2::ZERO);
this->addChild(menu, 1);
_sprBomb = Sprite::create("bomb.png");
_sprBomb->setPosition(_visibleSize.width/2,
_visibleSize.height + _sprBomb->getContentSize().height/2);
this->addChild(_sprBomb,1);
auto bg = Sprite::create("background.png");
bg->setAnchorPoint(Vec2());
bg->setPosition(0,0);
this->addChild(bg, -1);
_sprPlayer = Sprite::create("player.png");
_sprPlayer->setPosition(_visibleSize.width/2,
_visibleSize.height * 0.23);
setPhysicsBody(_sprPlayer);

this->addChild(_sprPlayer, 0);
//Animations
Vector<SpriteFrame*> frames;
Size playerSize = _sprPlayer->getContentSize();
frames.pushBack(SpriteFrame::create("player.png",
Rect(0, 0, playerSize.width, playerSize.height)));
frames.pushBack(SpriteFrame::create("player2.png",
Rect(0, 0, playerSize.width, playerSize.height)));
auto animation = Animation::createWithSpriteFrames(frames,0.2f);
auto animate = Animate::create(animation);
_sprPlayer->runAction(RepeatForever::create(animate));
setPhysicsBody(_sprBomb);
initPhysics();
_sprBomb->getPhysicsBody()->setVelocity(Vect(0,-100));
initTouch();
initAccelerometer();
#if (CC_TARGET_PLATFORM == CC_PLATFORM_ANDROID)
setKeepScreenOnJni(true);
#endif
initBackButtonListener();
schedule(CC_SCHEDULE_SELECTOR(HelloWorld::updateScore), 3.0f);
```

```
    schedule(CC_SCHEDULE_SELECTOR(HelloWorld::addBombs), 8.0f);
    initAudioNewEngine();
    initMuteButton();
    _bombs.pushBack(_sprBomb);
    return true;
}
```

Summary

In this chapter, we have seen how to use the particle systems to simulate realistic fire, explosions, rain, and snow in our game, how to customize them, and how to create them from scratch. We have also seen how to use the `Vector` class bundled with the Cocos2d-x API for creating collections of the Cocos2d-x objects.

In next chapter, we will show you how to add the Android native code to our game by using Java Native Interface (JNI).

8
Adding Native Java Code

Until now, we have been creating our game by using only the programming language in which the Cocos2d-x game framework has been written (C++); nevertheless, the Android API written by Google is only available in the Java layer of the application. In this chapter, you will learn how to communicate our native C++ code with the higher end Java core by using the **Java Native Interface** (**JNI**) capabilities.

The following topics will be covered in this chapter:

- Understanding the Cocos2d-x structure for the Android platform
- Understanding the JNI capabilities
- Adding the Java code to the Cocos2d-x game
- Adding ads to the game by inserting the Java code

Understanding the Cocos2d-x structure for the Android platform

In *Chapter 1*, *Setting Up Your Development Environment*, when we were installing all the required components for building the Cocos2d-x framework, we told you to download and install Android **Native Development Kit** (**NDK**), which allows us to build the Android applications by using the C++ language instead of using the mainstream Java technology core for which the Android API has been written.

When an Android application is launched, it looks in its `AndroidManisfest.xml` file for an activity definition that has the intent filter `android.intent.action.MAIN`, and then it will run the Java class. The following listing shows the segment of the `AndroidManifest.xml` file generated by the Cocos new script, where the activity to be launched when the Android application starts is specified:

```
<activity
   android:name="org.cocos2dx.cpp.AppActivity"
   android:configChanges="orientation"
   android:label="@string/app_name"
   android:screenOrientation="portrait"
   android:theme="@android:style/Theme.NoTitleBar.Fullscreen" >
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

The Cocos2d-x project creation script has created a Java class named `AppActivity` and it is located in the `src` folder of the `proj.android` directory under the `org.cocos2dx.cpp` Java package name. This class has no body and extends from the `Cocos2dxActivity` class, as we can appreciate in the following code listing:

```
package org.cocos2dx.cpp;

import org.cocos2dx.lib.Cocos2dxActivity;

public class AppActivity extends Cocos2dxActivity {
}
```

The `Cocos2dxActivity` class loads the native C++ framework core in its `onCreate` method.

Understanding the JNI capabilities

JNI offers a bridge between the C++ code and the Java code. The Cocos2d-x framework provides us with a JNI helper, which makes it easier to integrate the C++ code and the Java code.

The `JniHelper` C++ class has a method called `getStaticMethodInfo`. This receives as parameters, a `JniMethodInfo` object to store all the data that is required to invoke the corresponding Java code, the class name where the static method is located, the name of the method, and its signature.

In order to find out the method signature for JNI, you could use the `javap` command: so, for instance, if we would like to know what the signatures of the methods contained in the `AppActivity` class are , then we just need to open a console window , go to your `proj.android\bin\classes` directory, and type the following command:

```
SET CLASSPATH=.
javap -s org.cocos2dx.cpp.AppActivity
```

In this particular case you will receive as a response, the signature of the `null` constructor created automatically for the class as follows:

```
Compiled from "AppActivity.java"
public class org.cocos2dx.cpp.AppActivity extends org.cocos2dx.lib.
Cocos2dxActivity {
  public org.cocos2dx.cpp.AppActivity();
    Signature: ()V
}
```

Then, through the property `env` attached to the `JniMethodInfo` instance, we can invoke the Java method with a set of methods contained by this object, all starting with `Call`.... In the code that we are going to write in the next section, we will use the `CallStaticVoid` method in order to call a static method that does not return any value, as its name suggests. Please note that if you want to pass a Java string as a parameter, then you will need to call the `NewStringUTF` method of the `env` property, pass `const char*`, and it will return a `jstring` instance, which you can use to pass to a Java method that receives a string, as we show in the following code listing:

```
JniMethodInfo method;
JniHelper::getStaticMethodInfo(method, CLASS_NAME,"showToast","(Ljava/
lang/String;)V");
jstring stringMessage = method.env->NewStringUTF(message);
method.env->CallStaticVoidMethod(method.classID,
method.methodID, stringMessage);
```

Finally, if you have created instances of `jstring` or any other Java abstraction class within your C++ code, then make sure that you delete those after passing the value to the Java core, so that we don't have unnecessary references in the memory. This can be achieved by calling the `DeleteLocalRef` method located in the `env` property of the `JniMethodInfo` instance and passing the Java abstraction reference that you want to remove:

```
method.env->DeleteLocalRef(stringMessage);
```

The concepts exposed in this section will be applied to the code listings in the next section.

Adding Java code to the Cocos2d-x game

Now we are going to create a simple integration between these two technologies, which will allow our Cocos2d-x C++ game to show a toast message using the Android Java API.

A toast in Android is a pop-up message that is displayed for a specified amount of time without the option of being hidden before the predefined time. The screenshot at the end of this section shows what a toast message looks like.

Cocos2d-x runs within a Java activity, so in order to show the native Android toast message, we will create a Java class that will have a static method called `showToast`. This will receive a string, and then it will show it in a toast. In order to have access to the Cocos2d-x game activity, we will add a static attribute of type `Activity` to that class and will initialize it in the overridden `onCreate` method. Then, we will create a public static method that will allow us to access this instance from anywhere within our Java code. After these modifications our `AppActivity` Java class code will look like this:

```
package org.cocos2dx.cpp;

import org.cocos2dx.lib.Cocos2dxActivity;
import android.app.Activity;
import android.os.Bundle;

public class AppActivity extends Cocos2dxActivity {
   private static Activity instance;
    @Override
    protected void onCreate(final Bundle savedInstanceState) {
       instance = this;
        super.onCreate(savedInstanceState);

    }
    public static Activity getInstance(){
       return instance;
    }
}
```

Now, let us create the referred `JniFacade` Java class inside our `com.packtpub.jni` package, which in its body will only have one static void method that receives a string as a parameter, and then shows a toast on the UI thread with the received message as follows:

```
package com.packtpub.jni;

import org.cocos2dx.cpp.AppActivity;
import android.app.Activity;
import android.widget.Toast;

public class JniFacade {
   private static Activity activity = AppActivity.getInstance();

   public static void showToast(final String message) {
      activity.runOnUiThread(new Runnable() {
         @Override
         public void run() {
            Toast.makeText(activity.getBaseContext(), message, Toast.
            LENGTH_SHORT).show();
         }
      });
   }
}
```

Now that we have our code in the Java side, let us add the `JniBridge` C++ class to our `classes` folder.

In the `JniBridge.h` header file we will write the following:

```
#ifndef __JNI_BRIDGE_H__
#define __JNI_BRIDGE_H__
#include "cocos2d.h"

class JniBridge
{
public:
   static void showToast(const char* message);
};

#endif
```

Now let us create the implementation file, `JniBridge.cpp`, where we are going to invoke our static Java method called `showToast`, which receives a string as a parameter:

```
#include "JniBridge.h"
#define CLASS_NAME "com/packtpub/jni/JniFacade"
#define METHOD_NAME "showToast"
#define PARAM_CODE "(Ljava/lang/String;)V"


USING_NS_CC;

void JniBridge::showToast(const char* message)
{
   JniMethodInfo method;
   JniHelper::getStaticMethodInfo(method, CLASS_NAME, METHOD_NAME,
PARAM_CODE);
   jstring stringMessage = method.env->NewStringUTF(message);
    method.env->CallStaticVoidMethod(method.classID, method.methodID,
stringMessage);
    method.env->DeleteLocalRef(stringMessage);
}
```

As we can see, here we are using the `JniMethodInfo` structure and the `JniHelper` class bundled with the Cocos2d-x framework in order to invoke the `showToast` method and send it the c string in our C++ code, which was converted into a Java string.

Now let us include our `JniBridge.h` header file in our `HelloWorldScene.cpp` implementation file, so that we can have access to our bridge to the Java code from within our main scene class:

```
#include "JniBridge.h"
```

Now at the end of the `init` method located in the `HelloWorld.cpp` implementation file, we are going to call the `showToast` static method in order to use the Android Java API for displaying a native toast message, displaying the text sent from our C++ code as follows:

```
JniBridge::showToast("Hello Java");
```

This yields the following result:

As we can appreciate from the previous screenshot, we have achieved our goal of displaying a native Java toast message from our C++ game logic code.

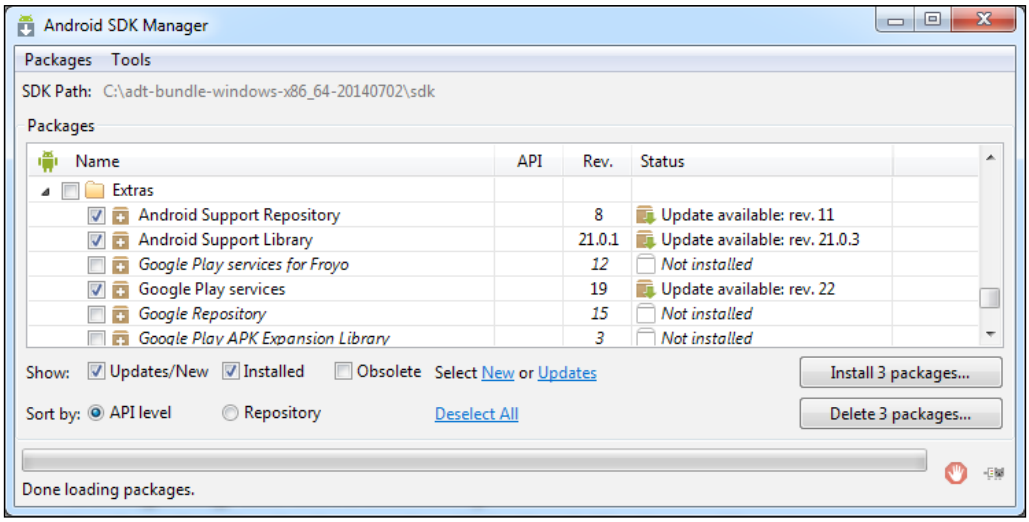
Adding ads to the game by inserting Java code

In the previous section, we have created an interaction between our C++ game logic code and the Java layer of our Android app by using JNI. In this section, we are going to modify our Android-specific code in order to show the Google **AdMob** banners in our Android game.

AdMob is a Google platform that allows you to monetize your app by displaying ads, it also has analytics tools, and tools for in-app purchases.

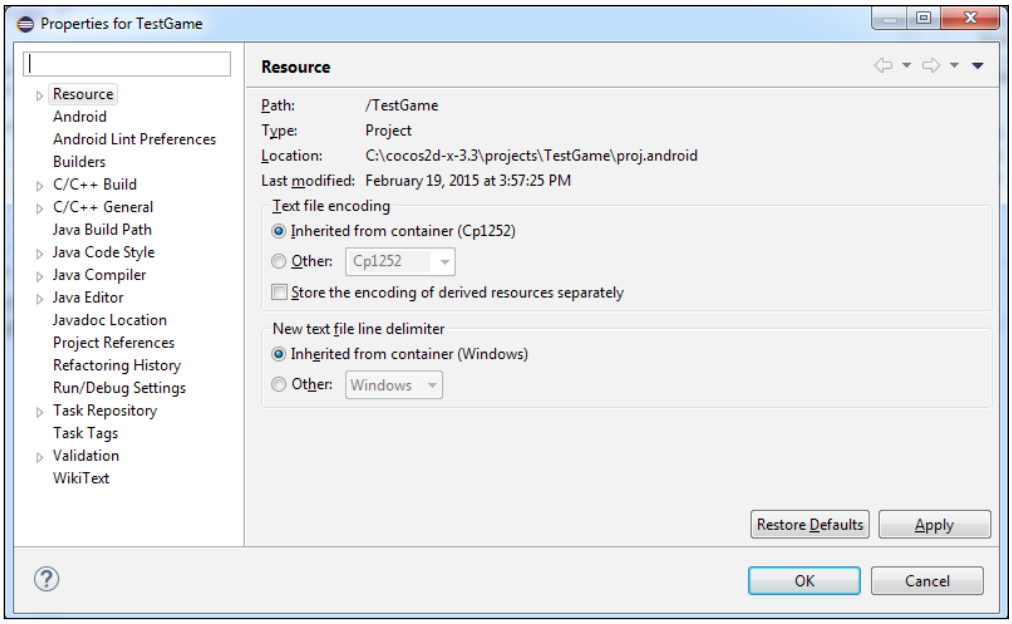
Configuring the environment

In order to show the Google AdMob banners, we need to add the `Google Play Services` library to our project. In order to do this, we first need to download it and its dependency, the Android Support Library, by using the Android SDK Manager:

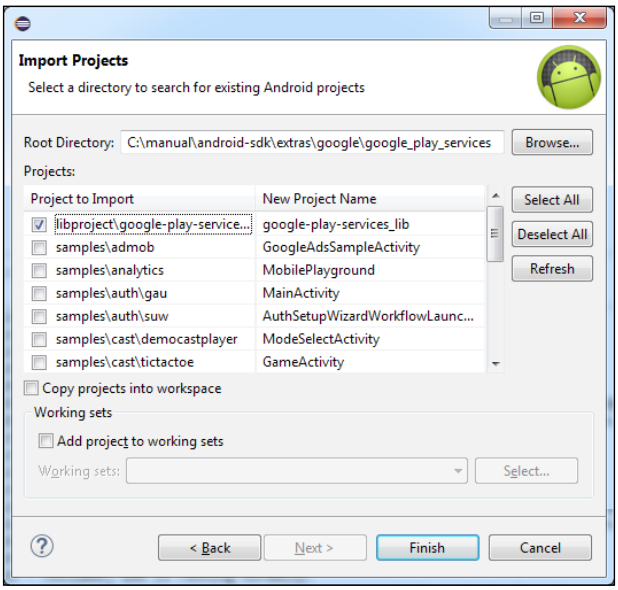


After you have successfully downloaded the **Google Play Services** and its dependencies, you will need to add the Android.support.v4 to your project, since it is required by the Google Play Services library. For this, we are going to copy the `android-support-v4.jar` file located on the following path: `<ADT PATH>\sdk\extras\android\support\v4` to the `libs` folder contained in our Android project, then we are going to add it to our build path by right-clicking on our project in the Eclipse's package explorer , then click on **Build Path**, and then click on **Configure Build Path**. The **Java Build Path** configuration window will show up, click on the **Add JARS...** button and add the `android-support-v4.jar` file in the `libs` folder.

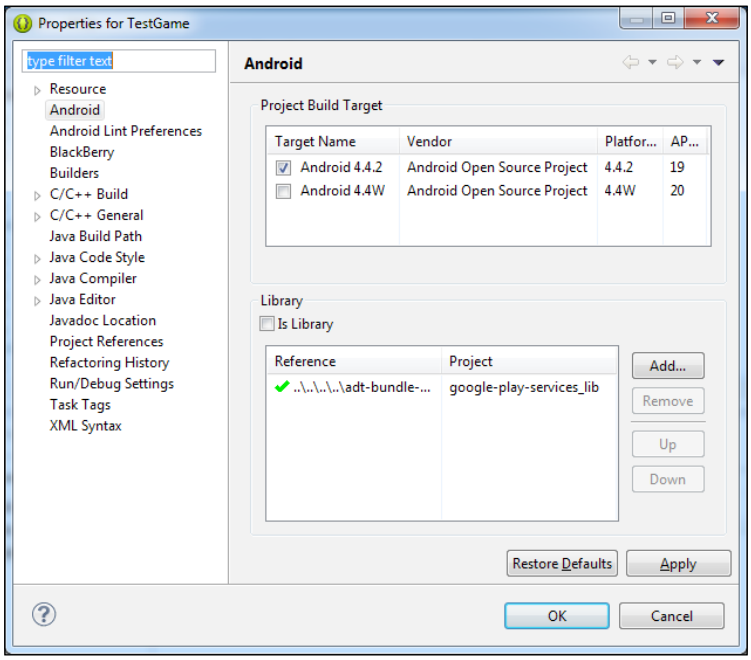
Now, we are going to copy the Google Play Services code that we have just downloaded. This is now located on the `<ADT PATH>\sdk\extras\google\google_play_services` to our workspace path. You can find out what the path to your workspace is by right-clicking on your Eclipse Java project, then clicking on the **properties**, and finally, selecting the **Resource** option on the left; there you will see the **Location** information as shown in the following screenshot:



We have set up the dependency, so let us now add the Google Play Services library, by navigating to **File** | **import** | **Android** | **Existing Android Code Into Workspace** | **Browse ...**. Then, browse to the location where you have copied the Google Play Services from the previous step. Deselect all projects but `google-play-services_lib` and click on **Finish**:



Now that we have the `google-play-services_lib` project in our workspace, let us configure it as a library for our Cocos2d-x game project. For this, we are going to right-click our project on the package explorer again, click on the **properties**, click on the **Android** section in the left pane, and then at the lower bottom section of the screen, we are going to click the **Add...** button in order to add the `google-play-services_lib` library to our Eclipse project, as shown in the following screenshot:



Now we are all set and ready to move to our next section, where we are going to use the libraries that we have just added for displaying the Google AdMob ads.

Now that our AdMob banner is going to be shown at the top of the screen, we are now going to move our mute button to the bottom, so that it is not covered by the banner. We are going to achieve this by changing the position of the mute and unmute buttons. Instead of assigning the height of the screen minus half of the mute sprite height as its vertical position, we will now set its *y* component as the screen height minus the double of the mute button's height, as we show in the following line of code in the `initMuteButton` method:

```
    _muteItem->setPosition(Vec2(_visibleSize.width -
_muteItem->getContentSize().width/2 ,_visibleSize.height -
_muteItem->getContentSize().height * 2));
```

Modifying the Android manifest

In this section, we are going to modify our Android manifest in order to insert it in the required configuration for using the Google Play Services library.

We just need to add two snippets, one of which will be just next to the opening application tag that will indicate the version of the Google Play Services being used, as we can appreciate in the following code listing:

```
<meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
```

The second snippet that we are going to add is the `AdActivity` declaration, which will be added just next to the declaration of our game activity, so that our game is aware of this built-in activity located in the Google Play Services library:

```
<activity
    android:name="com.google.android.gms.ads.AdActivity"
    android:configChanges="keyboard|keyboardHidden
    |orientation| screenLayout|uiMode|screenSize|
    smallestScreenSize" />
```

Adding the Java code

Now that we have the libraries configured and the Android manifest modified, the ads library is ready for use. We are going to add an ad initialization method to our `AppActivity` class, and we are going to call it after we call the implementation of its superclass.

For the sake of the following example, we are going to use a sample AdMob ID, which you may substitute with your own ID. You can find more information about how to create your own AdMob ID at `http://www.google.com/admob`.

```
private void initAdMob() {
   final String ADMOB_ID = "ca-app-
   pub-7870675803288590/4907722461";
   final AdView adView;
   final FrameLayout adViewLayout;

   FrameLayout.LayoutParams adParams = new FrameLayout.
   LayoutParams(
         FrameLayout.LayoutParams.MATCH_PARENT,
         FrameLayout.LayoutParams.WRAP_CONTENT);
```

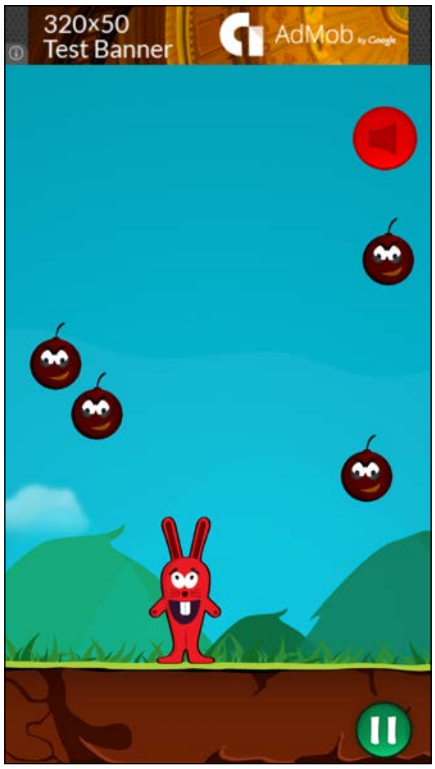
```
        adParams.gravity = Gravity.TOP | Gravity.CENTER_HORIZONTAL;

        AdRequest adRequest = new AdRequest.Builder().
              addTestDevice(AdRequest.DEVICE_ID_EMULATOR).
              addTestDevice("E8B4B73DC4CAD78DFCB44AF69E7B9EC4").build();

        adView = new AdView(this);
        adView.setAdSize(AdSize.SMART_BANNER);
        adView.setAdUnitId(ADMOB_ID);
        adView.setLayoutParams(adParams);
        adView.loadAd(adRequest);
        adViewLayout = new FrameLayout(this);
        adViewLayout.setLayoutParams(adParams);
        adView.setAdListener(new AdListener() {
              @Override
              public void onAdLoaded() {
                adViewLayout.addView(adView);
              }
        });
        this.addContentView(adViewLayout, adParams);
    }
```

In contrast to the previous section we are not using JNI, since we are not interacting at all with the C++ code; instead, we are modifying the Android activity that was created by the `cocos` command in order to add more graphic elements to view a side of the OpenGL E's view that was defined in the template.

We simply created a frame layout programmatically and added an `adView` instance to it; finally, we added this frame layout as a content view to the game activity, then we specified its desired position by using the gravity layout parameters, and this is how we were finally able to display the Google ad at the top of the screen. Please note that you could modify the location of the ad, that is, where you want it to be displayed, by simply modifying the layout parameters.

Be aware that we have added `adView` to our frame layout after the ad was successfully loaded. By using `AdListener`, if you add the `adView` instance before the ad finishes launching, then it will not be displayed.

This is what our Google AdMob looks like after tying everything together:



Putting everything together

We have accomplished our goal of embedding the core Java code in our Cocos2d-x game. Now we are going to show all the parts of our game that have been modified throughout this chapter together.

Here, we show the header file of the C++ JNI bridge (`JniBridge.h`) that we have created from scratch:

```
#ifndef __JNI_BRIDGE_H__
#define __JNI_BRIDGE_H__
```

```
#include "cocos2d.h"

class JniBridge
{
public:
   static void showToast(const char* message);
};

#endif
```

Now that we have defined the header of our `JniBridge`, let us write the implementation file (`JniBridge.cpp`):

```
#include "JniBridge.h"
#include "platform/android/jni/JniHelper.h"
#define CLASS_NAME "com/packtpub/jni/JniFacade"
#define METHOD_NAME "showToast"
#define PARAM_CODE "(Ljava/lang/String;)V"

USING_NS_CC;

void JniBridge::showToast(const char* message)
{
   JniMethodInfo method;
   JniHelper::getStaticMethodInfo(method, CLASS_NAME, METHOD_
   NAME,PARAM_CODE);
   jstring stringMessage = method.env->NewStringUTF(message);
   method.env->CallStaticVoidMethod(method.classID, method.methodID,
   stringMessage);
   method.env->DeleteLocalRef(stringMessage);
}
```

Now let us see what our gameplay class header (`HelloWorldScene.h`) looks like after including our `JniBridge`:

```
#ifndef __HELLOWORLD_SCENE_H__
#define __HELLOWORLD_SCENE_H__

#include "cocos2d.h"
#include "PauseScene.h"
#include "GameOverScene.h"
```

```
#include "JniBridge.h"

class HelloWorld : public cocos2d::Layer
{
public:
    static cocos2d::Scene* createScene();
    virtual bool init();
    void pauseCallback(cocos2d::Ref* pSender);
    CREATE_FUNC(HelloWorld);
private:
   cocos2d::Director *_director;
   cocos2d::Size visibleSize;
   cocos2d::Sprite* _sprBomb;
   cocos2d::Sprite* _sprPlayer;
   cocos2d::Vector<cocos2d::Sprite*> _bombs;
   cocos2d::MenuItemImage* _muteItem;
   cocos2d::MenuItemImage* _unmuteItem;
   int _score;
   int _musicId;
   void initPhysics();
   bool onCollision(cocos2d::PhysicsContact& contact);
   void setPhysicsBody(cocos2d::Sprite* sprite);
   void initTouch();
   void movePlayerByTouch(cocos2d::Touch* touch,
   cocos2d::Event* event);
   bool explodeBombs(cocos2d::Touch* touch,
   cocos2d::Event* event);
   void movePlayerIfPossible(float newX);
   void movePlayerByAccelerometer(cocos2d::Acceleration*
   acceleration, cocos2d::Event* event);
   void initAccelerometer();
   void initBackButtonListener();
   void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode,
   cocos2d::Event* event);
   void updateScore(float dt);
   void addBombs(float dt);
   void initAudio();
   void initAudioNewEngine();
   void initMuteButton();
};

#endif // __HELLOWORLD_SCENE_H__
```

Now we will show you what the `HelloWorldScene.cpp` method looks like at the end of this, the final chapter of the book:

```
#include "HelloWorldScene.h"
USING_NS_CC;
using namespace CocosDenshion;
using namespace cocos2d::experimental;

// User input handling code …
void HelloWorld::initMuteButton()
{
   _muteItem = MenuItemImage::create("mute.png", "mute.png",
   CC_CALLBACK_1(HelloWorld::muteCallback, this));

   _muteItem->setPosition(Vec2(_visibleSize.width -
   _muteItem->getContentSize().width/2 ,
   _visibleSize.height -
   _muteItem->getContentSize().height * 2));
```

We have changed the position of the mute button in the code, so that it is not covered by the ad:

```
    _unmuteItem = MenuItemImage::create("unmute.png", "unmute.png",
    CC_CALLBACK_1(HelloWorld::muteCallback, this));

   _unmuteItem->setPosition(Vec2(_visibleSize.width -
   _unmuteItem->getContentSize().width/2 ,
   _visibleSize.height -
   _unmuteItem->getContentSize().height *2));
   _unmuteItem -> setVisible(false);

   auto menu = Menu::create(_muteItem, _unmuteItem , nullptr);
   menu->setPosition(Vec2::ZERO);
   this->addChild(menu, 2);
}
```

```
// on "init" you need to initialize your instance
bool HelloWorld::init()
{
  if ( !Layer::init() )
  {
     return false;
  }
  _score = 0;
  _director = Director::getInstance();
  visibleSize = _director->getVisibleSize();
  auto origin = _director->getVisibleOrigin();
  auto closeItem = MenuItemImage::create("CloseNormal.png",
  "CloseSelected.png", CC_CALLBACK_1(HelloWorld::pauseCallback,
  this));
  closeItem->setPosition(Vec2(visibleSize.width -
  closeItem->getContentSize().width/2 ,
  closeItem->getContentSize().height/2));

  auto menu = Menu::create(closeItem, nullptr);
  menu->setPosition(Vec2::ZERO);
  this->addChild(menu, 1);
  _sprBomb = Sprite::create("bomb.png");
  _sprBomb->setPosition(visibleSize.width / 2,
  visibleSize.height + _sprBomb->getContentSize().height/2);
  this->addChild(_sprBomb,1);
  auto bg = Sprite::create("background.png");
  bg->setAnchorPoint(Vec2());
  bg->setPosition(0,0);
  this->addChild(bg, -1);
  _sprPlayer = Sprite::create("player.png");
  _sprPlayer->setPosition(visibleSize.width / 2, visibleSize.height *
  0.23);
  setPhysicsBody(_sprPlayer);
  this->addChild(_sprPlayer, 0);
  //Animations
  Vector<SpriteFrame*> frames;
  Size playerSize = _sprPlayer->getContentSize();
  frames.pushBack(SpriteFrame::create("player.png",
  Rect(0, 0, playerSize.width, playerSize.height)));
```

```
    frames.pushBack(SpriteFrame::create("player2.png",
    Rect(0, 0, playerSize.width, playerSize.height)));
    auto animation = Animation::createWithSpriteFrames(frames,0.2f);
    auto animate = Animate::create(animation);
    _sprPlayer->runAction(RepeatForever::create(animate));
    setPhysicsBody(_sprBomb);
    initPhysics();
    _sprBomb->getPhysicsBody()->setVelocity(Vect(0,-100));
    initTouch();
    initAccelerometer();
    #if (CC_TARGET_PLATFORM == CC_PLATFORM_ANDROID)
    setKeepScreenOnJni(true);
    #endif
    initBackButtonListener();
    schedule(schedule_selector(HelloWorld::updateScore), 3.0f);
    schedule(schedule_selector(HelloWorld::addBombs), 8.0f);
    initAudioNewEngine();
    initMuteButton();
    _bombs.pushBack(_sprBomb);
    JniBridge::showToast("Hello Java");
    return true;
}
```

This is what our `AppActivity.java` class looks like after all our modifications:

```
package org.cocos2dx.cpp;

import org.cocos2dx.lib.Cocos2dxActivity;
import android.app.Activity;
import android.os.Bundle;
import android.view.Gravity;
import android.widget.FrameLayout;
import com.google.android.gms.ads.AdListener;
import com.google.android.gms.ads.AdRequest;
import com.google.android.gms.ads.AdSize;
import com.google.android.gms.ads.AdView;

public class AppActivity extends Cocos2dxActivity {
   private static Activity instance;

   private void initAdMob() {
```

```
        final String ADMOB_ID = "ca-app-
        pub-7870675803288590/4907722461";
        final AdView adView;
        final FrameLayout adViewLayout;

        FrameLayout.LayoutParams adParams = new FrameLayout.
        LayoutParams(FrameLayout.LayoutParams.MATCH_PARENT,
        FrameLayout.LayoutParams.WRAP_CONTENT);
        adParams.gravity = Gravity.TOP | Gravity.CENTER_HORIZONTAL;
        AdRequest adRequest = new AdRequest.Builder().
        addTestDevice(AdRequest.DEVICE_ID_EMULATOR).
        addTestDevice("E8B4B73DC4CAD78DFCB44AF69E7B9EC4").build();

        adView = new AdView(this);
        adView.setAdSize(AdSize.SMART_BANNER);
        adView.setAdUnitId(ADMOB_ID);
        adView.setLayoutParams(adParams);
        adView.loadAd(adRequest);
        adViewLayout = new FrameLayout(this);
        adViewLayout.setLayoutParams(adParams);
        adView.setAdListener(new AdListener() {
              @Override
              public void onAdLoaded() {
              adViewLayout.addView(adView);
              }
        });
        this.addContentView(adViewLayout, adParams);
    }

    @Override
    protected void onCreate(final Bundle savedInstanceState) {
        instance = this;
        super.onCreate(savedInstanceState);
        initAdMob();
    }

    public static Activity getInstance() {
        return instance;
    }
}
```

This is what our `JniFacade.java` class file looks like at the end of this chapter: package `com.packtpub.jni`:

```
import org.cocos2dx.cpp.AppActivity;
import android.app.Activity;
import android.widget.Toast;

public class JniFacade {
   private static Activity activity = AppActivity.getInstance();

   public static void showToast(final String message) {
      activity.runOnUiThread(new Runnable() {
         @Override
         public void run() {
            Toast.makeText(activity.getBaseContext(), message, Toast.
            LENGTH_SHORT).show();
         }
      }
   }
}
```

This is what our `Android.mk` file located at `proj.android\jni` looks like after we add our `JniBridge.cpp` file in this chapter:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

$(call import-add-path,$(LOCAL_PATH)/../../cocos2d)
$(call import-add-path,$(LOCAL_PATH)/../../cocos2d/external)
$(call import-add-path,$(LOCAL_PATH)/../../cocos2d/cocos)

LOCAL_MODULE := cocos2dcpp_shared

LOCAL_MODULE_FILENAME := libcocos2dcpp

LOCAL_SRC_FILES := hellocpp/main.cpp \
          ../../Classes/JniBridge.cpp \
          ../../Classes/AppDelegate.cpp \
          ../../Classes/PauseScene.cpp \
          ../../Classes/GameOverScene.cpp \
```

```
                    ../../Classes/HelloWorldScene.cpp

LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes

LOCAL_STATIC_LIBRARIES := cocos2dx_static

include $(BUILD_SHARED_LIBRARY)

$(call import-module,.)
```

Finally, this is what our `AndroidManifest.xml` file looks like at the end of this book:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packt.happybunny"
    android:installLocation="auto"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="9" />

    <uses-feature android:glEsVersion="0x00020000" />

    <application
        android:icon="@drawable/icon"
        android:label="@string/app_name" >
        <meta-data
           android:name="com.google.android.gms.version"
           android:value="@integer/google_play_services_version" />

        <!-- Tell Cocos2dxActivity the name of our .so -->
        <meta-data
           android:name="android.app.lib_name"
           android:value="cocos2dcpp" />

        <activity
           android:name="org.cocos2dx.cpp.AppActivity"
           android:configChanges="orientation"
           android:label="@string/app_name"
           android:screenOrientation="portrait"
           android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="com.google.android.gms.ads.AdActivity"
            android:configChanges="keyboard|keyboardHidden
            |orientation| screenLayout|uiMode|screenSize
            |smallestScreenSize" />
    </application>

    <supports-screens
        android:anyDensity="true"
        android:largeScreens="true"
        android:normalScreens="true"
        android:smallScreens="true"
        android:xlargeScreens="true" />

    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

Summary

In this chapter, we have seen how to add the interaction between our C++ game logic code and the Android core Java layer by using JNI, we have also displayed the Google AdMob banner in our game by directly modifying the Java `Activity` class code that was created during the invocation of the `cocos` command.

Index

A

accelerometer events
handling 55
actions
defining 26
sequences, creating 27
sprites, moving 26
AdMob ID
URL 123
ads
adding, to game by inserting Java code 119
Android application configuration 11
Android back key pressed event
handling 57
Android Development Tools (ADT) 3
Android manifest
Java code, adding 123, 124
modifying 123
Android Native Development Kit
about 113
setting up 5, 6
Android SDK
downloading 3, 4
Eclipse ADT plugin, setting up 5
Eclipse, downloading 4
setting up 3
Apache Ant
setting up 6
URL 6
applicationDidFinishLaunching method 89-94
audio engine
about 84, 85
features, including 85, 86
mute button, adding to game 86-88
audio properties
audio, handling 83
modifying 83

B

background music
playing 81, 82
bitmap font labels
bombs, adding to game 70-73
creating 70

C

Chipmunk physics engine
URL 35
C++ lambda functions
URL 51
Cocos2d-x
about 2
Eclipse IDE, setting up 9, 10
first project, creating 8
Java code, adding to game 116-119
setting up 7
structure, defining for Android platform 113, 114
Cocos2d-x objects
bombs, exploding 99, 100
collections, creating 97, 98
collision detection 38, 39
customized particle systems
creating 104-106
custom time interval 71

E

Eclipse
  downloading 4
  URL 4
Eclipse ADT plugin
  setting up 5
  URL 5
Eclipse IDE
  setting up 9, 10
environment
  configuring 120-122
Event Dispatcher mechanism
  defining 49, 50

G

game
  pause scene header file, creating 18
  pause scene implementation file, creating 18-20
  pausing 17
  resources files, organizing 17
game menus
  about 29, 30
  implementation file 31-33
  multiple screen resolutions, handling 30
Glyph Designer
  about 70
  URL 70
Glyph Designer X 70
gravity
  handling 40, 41
  physics properties, handling 41

H

HelloWorldScene.h header file
  about 73-78
  displaying 58-60

J

Java
  setting up 2
Java code
  adding, to Cocos2d-x game 116-119
  embedding, in Cocos2d-x game 125-133
  inserting, for adding ads 119
Java Native Interface (JNI) 113
JDK 6
  URL 2
JNI capabilities
  defining 114, 115

L

Littera
  about 70
  URL 70

M

multi-touch events
  handling 53, 54

N

nodes
  defining 21

O

onCollision method 44-46

P

Particle Designer
  about 104
  URL 104
particle systems
  adding, to game 100, 101, 107-110
  configuring 102
  URL 104
pause scene header file
  creating 18
pause scene implementation file
  creating 18-20
  transitions 20
physics properties, gravity
  force, applying 42
  handling 41
  impulse, applying 42, 43
  linear damping 42
  torque, applying 44
  velocity, applying 41

physics world
setting up 36-38
Python
setting up 6, 7

R

red, green and blue (RGB) 68

S

scenes
Cocos2d-x director, using 16, 17
creating 15
Layer class, defining 16
screen touch
maintaining 56
sound effects
playing 81, 82
source code, Cocos2d-x
URL 7
sprites
adding, to scene 24, 25
anchor points, setting 22
animating 28
Cocos2d-x coordinate system, defining 22-24
creating 21
defining 21
performance, improving with sprite sheets 29
player sprite, positioning 26
positioning 21
positioning, outside visible area 25, 26
system fonts
creating 69

T

template code
about 10
Android application configuration 11
C++ classes 11
Java classes 10
scenes 12
touch event
handling 50-53
multi-touch events, handling 53, 54
TrueType font labels
creating 65, 66
GameOverScene, calling when player loses 67
GameOverScene, creating 66
game over scene, customizing 67
label effects, adding 67, 68

W

What You See Is What You Get (WYSIWYG) 104

Thank you for buying
Building Android Games with Cocos2d-x

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.

[PACKT] open source
community experience distilled
PUBLISHING

Learning Cocos2d-x Game Development

ISBN: 978-1-78398-826-6 Paperback: 266 pages

Learn cross-platform game development with Cocos2d-x

1. Create a Windows Store account and upload your game for distribution.
2. Develop a game using Cocos2d-x by going through each stage of game development process step by step.

Cocos2d-x Game Development Essentials

ISBN: 978-1-78398-786-3 Paperback: 136 pages

Create iOS and Android games from scratch using Cocos2d-x

1. Create and run Cocos2d-x projects on iOS and Android platforms.
2. Find practical solutions to many real-world game development problems.
3. Learn the essentials of Cocos2d-x by writing code and following step-by-step instructions.

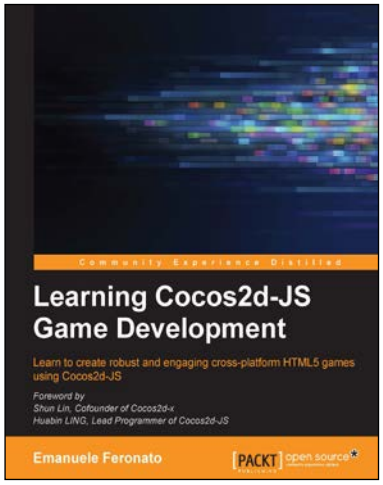
[PACKT] open source
community experience distilled
PUBLISHING

Cocos2d Game Development Blueprints

ISBN: 978-1-78398-788-7 Paperback: 440 pages

Design, develop, and create your own successful iOS games using the Cocos2d game engine

1. Enhance your game development skills by building different types of iOS games in Cocos2d.
2. Create games of many different genres using the powerful features of the Cocos2d framework.
3. A step-by-step guide with techniques to discover the potential of Cocos2d Game Engine with interesting projects.



Learning Cocos2d-JS Game Development

ISBN: 978-1-78439-007-5 Paperback: 188 pages

Learn to create robust and engaging cross-platform HTML5 games using Cocos2d-JS

1. Create HTML5 games running both on desktop and mobile devices, played with both mouse and touch controls.
2. Add advanced features such as realistic physics, particle effects, scrolling, tweaking, sound effects, background music, and more to your games.
3. Build exciting cross-platform games and build a memory game, an endless runner and a physics-driven game.

Please check **www.PacktPub.com** for information on our titles