Community Experience Distilled

Cocos2d-x Game Development Essentials

Create iOS and Android games from scratch using Cocos2d-x

Frahaan Hussain Arutosh Gurung
Gareth Jones

[PACKT] PUBLISHING
open source
community experience distilled

Cocos2d-x Game Development Essentials

Create iOS and Android games from scratch using Cocos2d-x

Frahaan Hussain

Arutosh Gurung

Gareth Jones

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Cocos2d-x Game Development Essentials

First published: August 2014

Production reference: 1120814

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

ISBN 978-1-78398-786-3

www.packtpub.com

Cover image by Arutosh Gurung (admin@sonarsystems.co.uk)

Credits

Authors

Frahaan Hussain

Arutosh Gurung

Gareth Jones

Reviewers

Luma

Emanuele Feronato

Akihiro Matsuura

Taso Perdikoulias

Commissioning Editor

Ashwin Nair

Acquisition Editor

Richard Brookes-Bland

Content Development Editor

Prachi Bisht

Technical Editor

Pankaj Kadam

Copy Editors

Dipti Kapadia

Sayanee Mukherjee

Karuna Narayanan

Adithi Shetty

Stuti Srivastava

Project Coordinator

Sageer Parkar

Proofreaders

Maria Gould

Ameesha Green

Indexers

Mariammal Chettiyar

Rekha Nair

Graphics

Ronak Dhruv

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Authors

Frahaan Hussain is a young programmer who contributes towards the community in many ways, this book being the latest. He runs his own company (Sonar Systems). He has a degree in Computer Games Programming, making him fully aware of the ins and outs of game development.

Frahaan has worked for Accenture, which is the world's biggest consultancy firm, and he also runs a small personal company on the side (ThunderKeyBolt).

I would like to thank Arutosh Gurung for the artwork that went into this book, Gareth Jones for code assistance, and my father, Siddique Hussain, for his help in planning the book.

Company details:

- Company name: Sonar Systems
- Company logo:

- Official website: `http://www.sonarsystems.co.uk/`
- Facebook: `https://www.facebook.com/pages/Sonar-Systems/581403125243822`
- Twitter: `https://twitter.com/SonarSystems`
- Google+: `https://plus.google.com/+SonarsystemsCoUk/posts`
- YouTube: `https://www.youtube.com/user/sonarsystemslimited`

Arutosh Gurung is the cofounder of Sonar Systems. He is the main proofreader in the company and is an amazing artist with business acumen that backs up his creative abilities.

Gareth Jones is an amazing developer who works with Sonar Systems. He is the point of contact within the team for tips on coding.

About the Reviewers

Luma is 35 years old with 4 years of experience in Cocos2d. He is the author of two GitHub repositories, *WiEngine* and *cocos2dx-better*.

Emanuele Feronato has been studying programming languages since the early 1980s, with a particular interest in game development. He taught online programming for European Social Fund (ESF) and 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 80 million times. He now ports most of them on mobile platforms as native apps or HTML5 web apps. As a writer, he has authored the books *Flash Game Development by Example* and *Box2D for Flash Games*, both by Packt Publishing. He has also worked as a technical reviewer for Packt Publishing.

His blog `http://www.emanueleferonato.com` is one of the most visited blogs about indie programming.

I would like to thank Packt Publishing for giving me the opportunity to review this book and my little daughter, Kimora, for deleting most of my games (saved games included!) from my iPad while I was reviewing the last chapter. I love you anyway.

Akihiro Matsuura has worked as a Cocos2d-x developer for two years. He founded his own company, named Syuhari, Inc., four years ago. He has more than 20 years of experience in programming. He has written two technical books in Japanese. The first book is *iPhone SDK Recipe Book*, *Syuwa system* (`http://www.amazon.co.jp/dp/4798025798/`). The second book is *Cocos2d-x Recipe Book*, *Syuwa system*, which is also the first Cocos2d-x book in Japan (`http://www.amazon.co.jp/dp/4798038245/`).

I wish to thank the author and the publisher who gave me the opportunity to review this book.

Taso Perdikoulias has over 10 years of professional experience managing the architecture, design, and development of software solutions for major companies such as Disney/ABC Television Group, Gannett Company, Inc, The New York Times, Dow Jones, and Ganz.

Taso's mobile expertise is based on a solid foundation of gaming development. In the past four years, Taso has become a leading expert in the delivery of mobile applications for Fortune 50 corporations, delivering applications on platforms such as iOS, Android, Windows Phone, BlackBerry, and HTML5. Taso has a B.Sc. degree in Mathematical Science from McMaster University and teaches iOS game development at Humber College.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Cocos2d-x Game Development Essentials is your quick and easy guide to learning snippets of Cocos2d-x functionality for game development or how to make a game from scratch. This book uses the concept of creating a game to teach you the essentials of game development with Cocos2d-x while covering generic game design practices. This book will teach you the essentials using C++, making it ideal for the millions of existing developers out there looking to learn Cocos2d-x for a job or to start their own software house, to develop quality mobile games. Unfortunately, the number of Cocos2d-x books is extremely low with a scarcity of online resources. However, this book aims to help in the solution of providing great quality learning materials that are easy to understand and follow for beginners or experts who are looking for a refresh.

We have launched the application that we have made during the course of this book. You can refer to the following links for this application:

- YouTube: `https://www.youtube.com/user/sonarsystemslimited`
- App Store: `https://itunes.apple.com/us/artist/sonar-systems/id672545941`
- Google Play: `https://play.google.com/store/apps/developer?id=Sonar+Systems`

What this book covers

Chapter 1, *Setting Up*, shows you how to set up a new project using Cocos2d-x and preparation for the game.

Chapter 2, *Adding Scenes*, covers how to add additional scenes to create the core foundations of a game.

Chapter 3, *Adding Game Menus*, introduces the process of creating a game menu using menu items.

Chapter 4, Scene Transitions, covers how to implement screen transitions to move from one scene to the next.

Chapter 5, *The Game Sprites*, covers how to implement sprites to set the foundations of a playable game.

Chapter 6, *Implementing Actions*, covers how to make your sprites move and act with a sense of motion within the game.

Chapter 7, *Moving the Space Pod Using Touch*, covers how to interact with the spaceship sprite using touch.

Chapter 8, *Collision Detection*, shows how to implement collision detection between the player and obstacles to add the final piece of gameplay.

Chapter 9, *Adding Audio to the Game*, introduces the concept of sound files and how to implement them into the game using Cocos2d-x.

Chapter 10, *Implementing Accelerometer Support*, shows how accelerometer support can be incorporated within a game.

Chapter 11, *Problem Solving and What's Next*, discusses the unfortunate situations that are most likely to be faced while developing a game and how to overcome these issues.

What you need for this book

XCode and Eclipse will be required for development on a Mac. Eclipse will be required for development on a Windows machine.

Who this book is for

If you are a developer who is experienced in C++ and aware of the basic concepts of game development, you should feel right at home with this book. Though experience in XCode or terminal/command prompt isn't necessary, it is desirable.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Run `setup.py` using the `python ./setup.py` command."

A block of code is set as follows:

```
#include "GameScene.h"

void MainMenu::GoToGameScene(Ref *pSender)
{
    auto scene = GameScreen::createScene();

    Director::getInstance()->replaceScene(scene);
}
```

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

```
cocos new Game -p learning.sonarsystems.game -l cpp -d
/Users/sonarsystems/Desktop
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on **Finish**."

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

This chapter will show you how to set up Cocos2d-x and generate a new project. The topics covered in this chapter are as follows:

- What is Cocos2d-x?
- Setting up the project
- Housekeeping
- Implementing multiresolution support

What is Cocos2d-x?

Cocos2d-x is the most popular open source game engine in the world. Cocos2d-x incorporates two main programming languages, C++ and Lua. This book will focus on the C++ implementation. There is also a JavaScript version called Cocos2d-JS, which supports web development as well. To put it into perspective, anyone who has played a mobile game will have played one that was most likely built using Cocos2d-x. The game engine's capabilities extend beyond game development, with features for general application development. However, the biggest aspect that makes Cocos2d-x phenomenal is its cross-platform nature, allowing development for all the major mobile and desktop platforms.

Companies such as Zynga and Disney use Cocos2d-x, which shows the immense reach of the game engine; moreover, it's free. Some of the biggest and most famous games that are developed using Cocos2d-x are *Badland*, *Star Wars: Tiny Death Star*, and *Diamond Dash*.

Setting up the project

This book focuses on iOS development, and the information from this book will serve as a good foundation and reference point for all the other major platforms for game development.

Instructions to set up projects on the Android platform will be provided in this book.

Before setting up a project, there are some files that need to be downloaded in order to proceed with the project. If you want to develop a game for an iPhone, you will require a Mac and XCode with an iOS developer account to test on a physical device. However, Android development can be done on a Mac or a Windows machine, and there is no need for any special account to run the application on an Android device. The following list provides the prerequisites that need to be downloaded to set up Cocos2d-x:

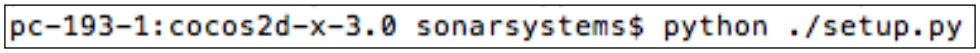
- **Cocos2d-x**: Download this from `http://www.cocos2d-x.org/download` (at the time of writing this book, v3.0 is the latest stable version that will be used throughout the book). This is the game engine used to develop the game in this book.
- **Android Developer Tools** (**ADT**): This is only needed for Android development. You can download it from `http://developer.android.com/sdk/index.html`. These tools are used to develop Android applications. The tools comprise Android SDK and Eclipse IDE.
- **Native Development Kit** (**NDK**): This is only needed for Android development and can be downloaded from `https://developer.android.com/tools/sdk/ndk/index.html`. NDK enables Android application development using programming languages such as C and C++.
- **Apache ANT**: This is only needed for Android development and can be downloaded from `http://ant.apache.org/bindownload.cgi`. This is a Java library that aids in building software.

It is recommended to save all the files in a location that is designated for development instead of leaving them in the default download directory. A development directory doesn't have to be somewhere in particular, it can be in any location you are aware of.

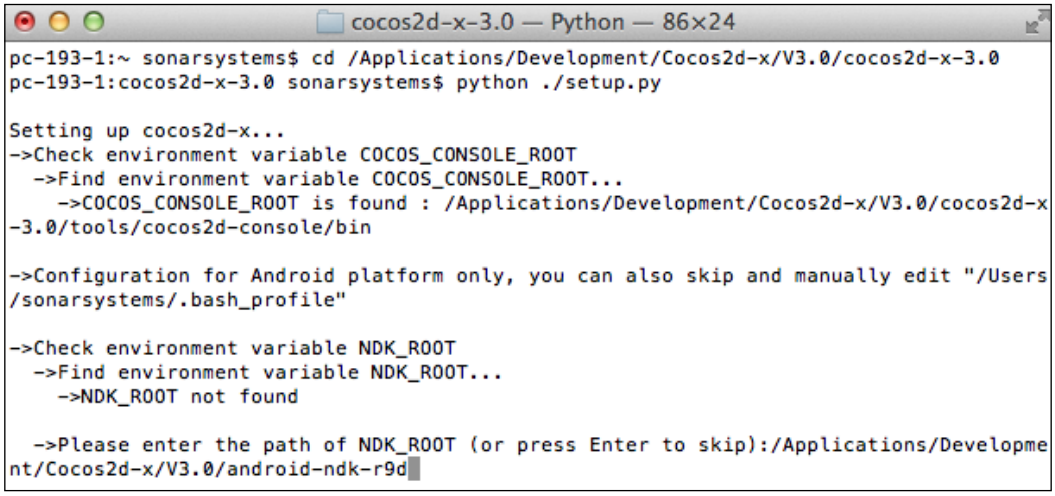
These steps will guide you through the process of setting up Cocos2d-x:

1. Extract/unpack all the downloaded files.
2. Open the terminal.
3. Change the directory in terminal to the Cocos2d-x root directory, for example, `cd /directory/location`. Have a look at the following screenshot:

4. Run `setup.py` using the `python ./setup.py` command, as shown in the following screenshot:

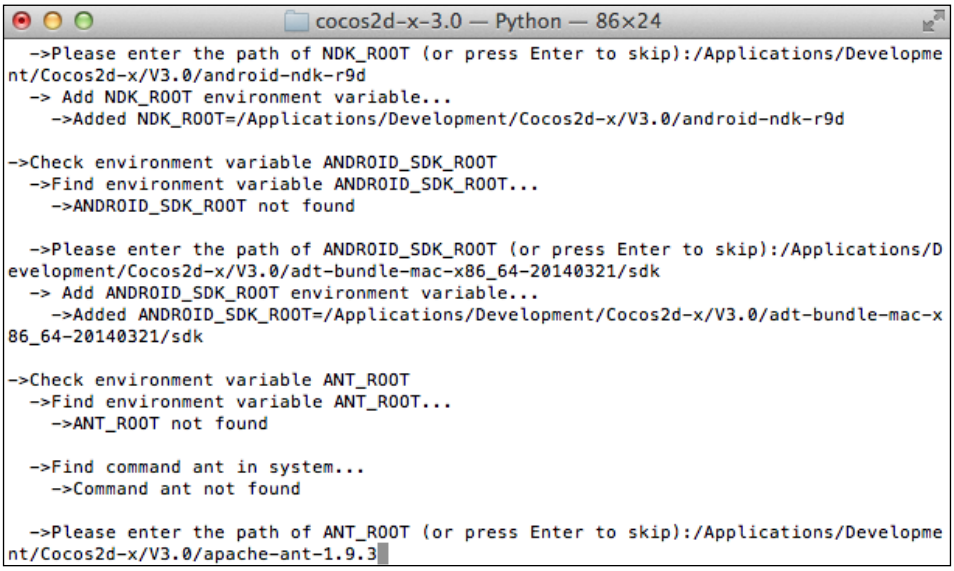


5. Terminal will ask you for `NDK_ROOT`, which is the Native Development Kit; drag-and-drop the NDK root folder onto terminal, and then press *Enter* (make sure there are no spaces at the end):



6. Now, terminal will ask you for `ANDROID_SDK_ROOT`, which is part of the ADT bundle. Drag-and-drop the SDK folder that is located in the ADT root folder onto terminal, and then press *Enter* (make sure there are no spaces at the end):

7. Next, terminal will ask you for `ANT_ROOT`. Drag-and-drop the `bin` folder that is located in the Apache root folder onto terminal, and then press *Enter* (make sure there are no spaces at the end).



8. Finally, `.bash_profile` needs to be run to add the system variables. The specific command is shown in the following terminal screenshot:

```
Please execute command: "source /Users/sonarsystems/.bash_profile" to make added syste
m variables take effect
```

The `source /Users/sonarsystems/.bash_profile` command shown in the preceding screenshot will need to be changed depending on your system directory.

You should now have successfully completed the Cocos2d-x setup process. The next steps will generate a new Cocos2d-x project to be used as a basis to create games. The following steps will guide you through the process of generating a new Cocos2d-x project:

1. Open the terminal.
2. Run the `cocos` command with the following parameters:
    - `new project name`
    - `-p package name` (the name of the application within the company/organization, which should be unique)
    - `-l programming language` (`cpp` or `lua`)
    - `-d location to generate project`

    For example, the command will look like the following:

    ```
    cocos new Game -p learning.sonarsystems.game -l cpp -d /Users/
    sonarsystems/Desktop
    ```

Congratulations, a new project has been generated. The next step is to run the project for iOS and Android.

Once the project is generated, a few folders will be included. Let's go over what each folder does:

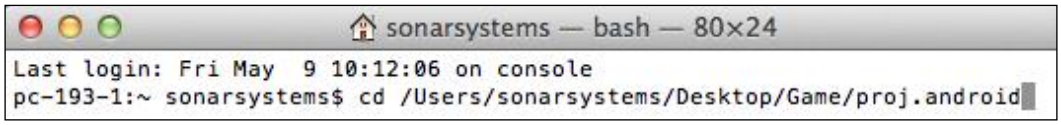
- `Classes`: This stores all the custom headers (`.h`) and implementation files (`.cpp`)
- `cocos2d`: This stores the Cocos2d library files (doesn't need modification for the purpose of this book)
- `proj.android`: This contains Android project files
- `proj.ios_mac`: This contains iOS and Mac project files
- `proj.linux`: This stores Linux project files
- `proj.win32`: This stores Windows 32 project files

- `proj.wp8-xaml`: This contains Windows Phone 8 project files
- `Resources`: This is where all the applications' external files, such as images and audio files, are stored

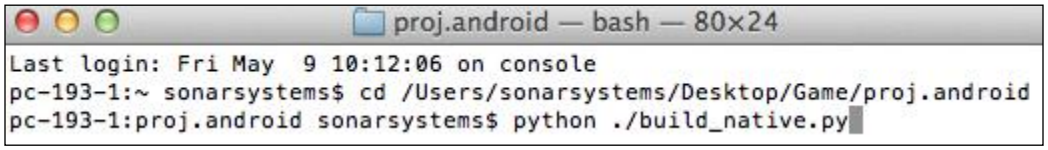
Go to the project directory and open the XCode project located in `proj.ios_mac`. Run the project, and the following screen should appear:

Now, to run the project in Eclipse for Android, use these steps:

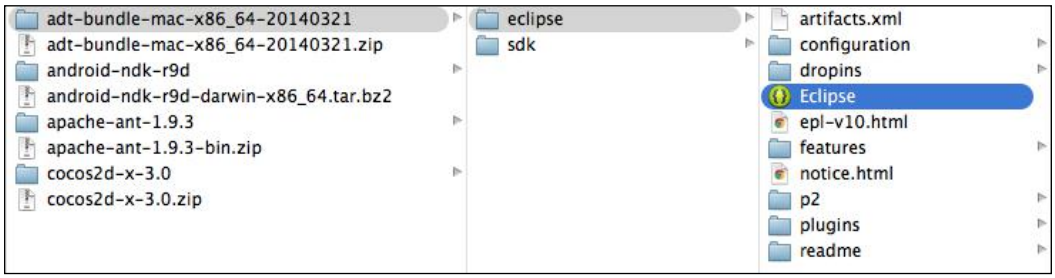
1. Open the terminal.
2. Change the directory to the `proj.android` folder in the projects root, as shown in the following screenshot:



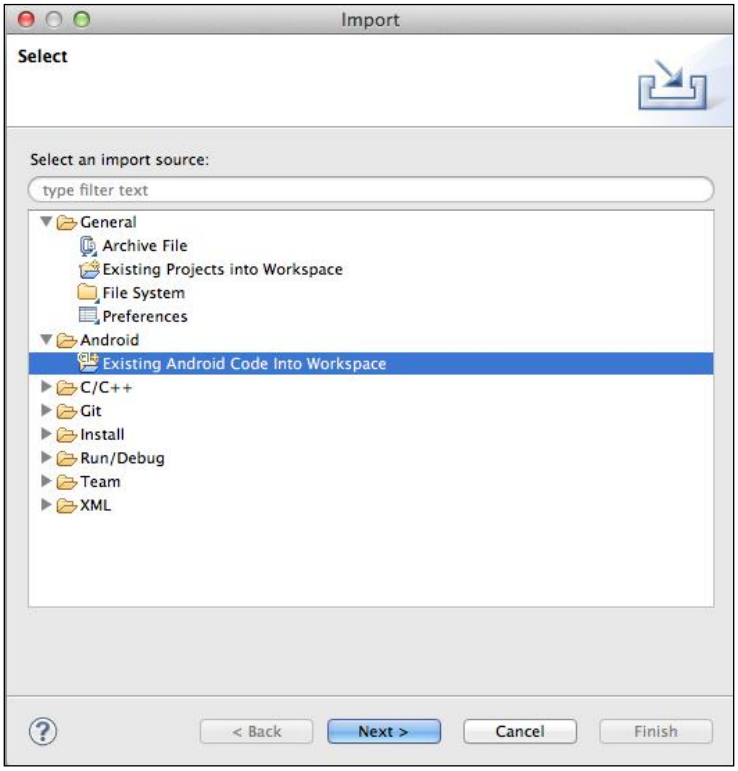
3. Run the `python ./build_native.py` command:



4. Open Eclipse from the `eclipse` folder located in ADT:

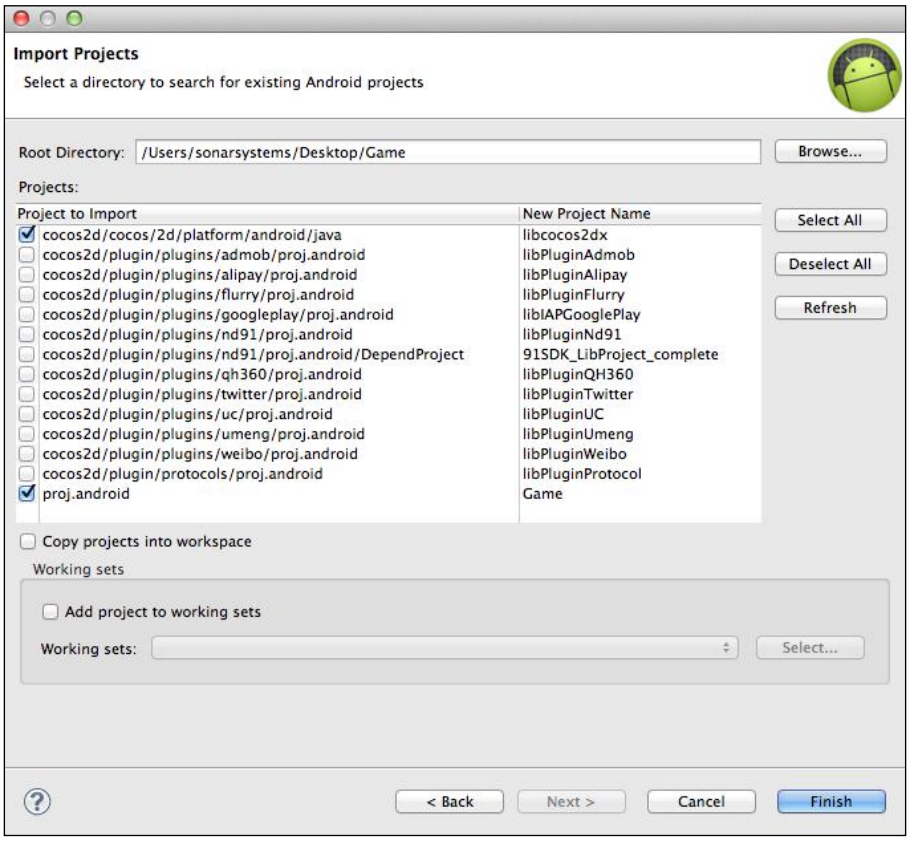


5. Go to **File** | **Import**.
6. Under **Android**, select **Existing Android Code Into Workspace**, as shown in the following screenshot:



7. Browse to the project's root directory.

8. Only select `libcocos2dx` and the project created earlier:



9. Click on **Finish**.
10. Now, run the application on an Android device.
11. Every time a change is made to the project, perform steps 1 to 3 again.

Unfortunately, Eclipse doesn't work very well with Cocos2d-x. Use an external text editor and deploy using Eclipse. Text editors such as Sublime Text 2 are good for programming. We recommend developing in XCode and then running on iOS and Android, using XCode and Eclipse respectively.

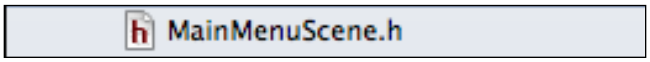
Housekeeping

A new project has been generated, but the default files are called `HelloWorldScene`; these files will be refactored to represent a real game. This chapter will cover refactoring the default files that are generated.

Refactoring HelloWorldScene.h

The following steps will guide you on how to refactor `HelloWorldScene.h`:

1. Rename `HelloWorldScene.h` to `MainMenuScene.h`:



2. Open `MainMenuScene.h`.
3. Rename the `#` commands at the top of the document, as follows:
   - From `#ifndef __HELLOWORLD_SCENE_H__` to `#ifndef __MAINMENU_SCENE_H__`
   - From `#define __HELLOWORLD_SCENE_H__` to `#define __MAINMENU_SCENE_H__`

   You will see the following screen:

```
1  #ifndef __MAINMENU_SCENE_H__
2  #define __MAINMENU_SCENE_H__
```

4. Rename the `HelloWorld` class to `MainMenu`:

```
6  class MainMenu : public cocos2d::Layer
```

5. Remove the `void menuCloseCallback(cocos2d::Ref* pSender);` function.

6. Replace `HelloWorld` in `CREATE_FUNC` with `MainMenu` (or whatever the class name is):

```
19      CREATE_FUNC(MainMenu);
```

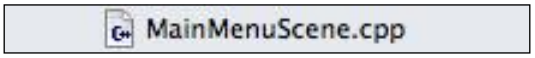
Now, `MainMenuScene.h` should look like the following:

```
#ifndef __MAINMENU_SCENE_H__
#define __MAINMENU_SCENE_H__

#include "cocos2d.h"

class MainMenu : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(MainMenu);
};
```

Refactoring HelloWorldScene.cpp

The following steps will guide you on how to refactor `HelloWorldScene.cpp`:

1. Rename `HelloWorldScene.cpp` to `MainMenuScene.cpp`:



2. Open `MainMenuScene.cpp`.
3. Include `MainMenuScene.h` instead of `HelloWorldScene.h`:

```
1   #include "MainMenuScene.h"
```

4. Before fixing the errors, remove the menu, label, and sprite from the `init()` function so that it looks like the following:

```
20  // on "init" you need to initialize your instance
21  bool HelloWorld::init()
22  {
23      //////////////////////////////
24      // 1. super init first
25      if ( !Layer::init() )
26      {
27          return false;
28      }
29
30      Size visibleSize = Director::getInstance()->getVisibleSize();
31      Point origin = Director::getInstance()->getVisibleOrigin();
32
33      return true;
34  }
```

5. Remove the entire `void HelloWorld::menuCloseCallback(Ref* pSender)` function.
6. The last step is to rename any instance of `HelloWorld` to `MainMenu`; there should be three instances:
    - `Scene* HelloWorld::createScene()` to `Scene* MainMenu::createScene()`:

```
5   Scene* MainMenu::createScene()
```

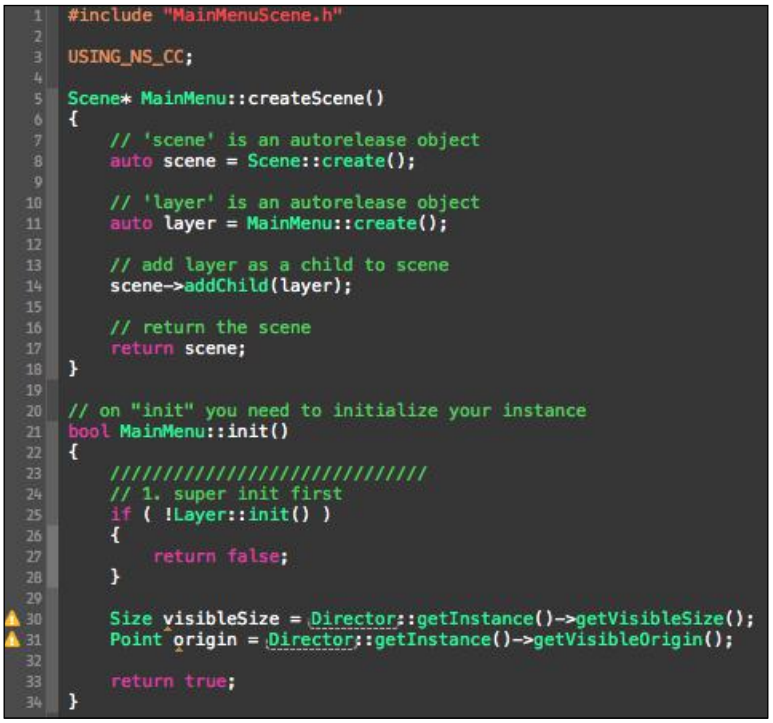
    - `auto layer = HelloWorld::create();` to `auto layer = MainMenu::create();`:

```
10      // 'layer' is an autorelease object
11      auto layer = MainMenu::create();
```

    - `bool HelloWorld::init()` to `bool MainMenu::init()`:

```
20  // on "init" you need to initialize your instance
21  bool MainMenu::init()
```

The `MainMenuScene.cpp` file should look like the following:



```
#include "MainMenuScene.h"

USING_NS_CC;

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

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

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

    // return the scene
    return scene;
}

// on "init" you need to initialize your instance
bool MainMenu::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    return true;
}
```

Refactoring AppDelegate.cpp

The following steps will guide you through refactoring `AppDelegate.cpp`:

1. Open `AppDelegate.cpp`.
2. Include `MainMenuScene.h` instead of `HelloWorldScene.h`:

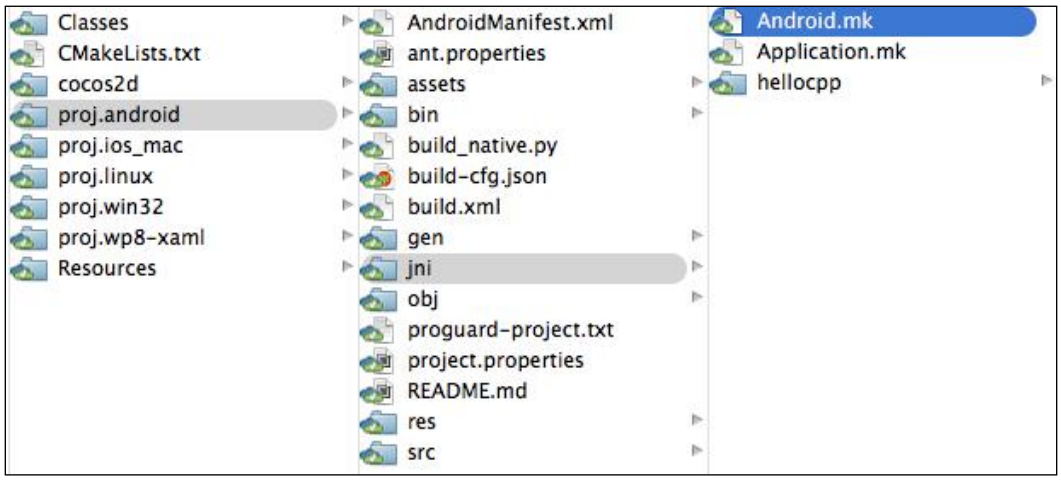
```
#include "MainMenuScene.h"
```

3. Change `auto scene = HelloWorld::createScene();` to `auto scene = MainMenu::createScene();` in the `bool AppDelegate::applicationDidFinishLaunching()` function:

```
29      // create a scene. it's an autorelease object
30      auto scene = MainMenu::createScene();
```

The project will not run for Android at the moment as the `Android.mk` file needs to be updated to include the correct files and the project needs to be rebuilt. The process of doing this will be shown in the following steps:

1. Open `Android.mk` in a text editor located in the `jni` directory at `ProjectRoot/proj.android/`:



2. Update `LOCAL_SRC_FILES` to look like the one shown in the following screenshot:

```
9   LOCAL_SRC_FILES := hellocpp/main.cpp \
10                     ../../Classes/AppDelegate.cpp \
11                     ../../Classes/MainMenuScene.cpp
```

3. This file needs to be updated every time a new `.cpp` file is added to the project. The backslash at the end of the line indicates that another file is on the line below.

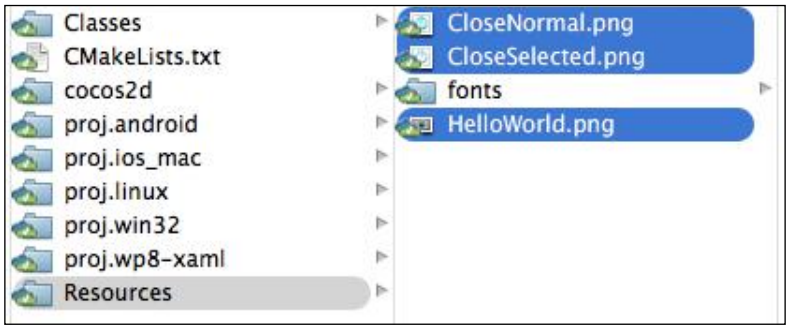
Throughout the book, new classes will be added and an assumption will be made that the `Android.mk` file has been updated.

Build the project using `./build_project.py` again, and run the project to make sure it is still working. The screen should be blank with some debug information in the bottom-left corner. If there are any errors, go back through the steps and make sure everything has been followed.

Implementing multiresolution support

This section will cover implementing multiresolution support for all iOS and Android devices.

Firstly, remove all the images from the `Resources` folder (clean your project using XCode and Eclipse):



Open `AppDelegate.cpp` and modify the `bool AppDelegate::applicatio nDidFinishLaunching()` function by adding the code after the `director->setAnimationInterval(1.0 / 60);` line, as shown in the following screenshot:

```
bool AppDelegate::applicationDidFinishLaunching() {
    // initialize director
    auto director = Director::getInstance();
    auto glview = director->getOpenGLView();
    if(!glview) {
        glview = GLView::create("My Game");
        director->setOpenGLView(glview);
    }

    // turn on display FPS
    director->setDisplayStats(true);

    // set FPS. the default value is 1.0/60 if you don't call this
    director->setAnimationInterval(1.0 / 60);

    auto fileUtils = FileUtils::getInstance();
    auto screenSize = glview->getFrameSize();
    std::vector<std::string> resDirOrders;

    // check which assets the devices requires
    if ( 2048 == screenSize.width || 2048 == screenSize.height ) // retina iPad
    {
        resDirOrders.push_back("ipadhd");
        resDirOrders.push_back("ipad");
        resDirOrders.push_back("iphonehd5");
        resDirOrders.push_back("iphonehd");
        resDirOrders.push_back("iphone");

        glview->setDesignResolutionSize(1536, 2048, ResolutionPolicy::NO_BORDER);
    }
    else if ( 1024 == screenSize.width || 1024 == screenSize.height ) // non retina iPad
    {
        resDirOrders.push_back("ipad");
        resDirOrders.push_back("iphonehd5");
        resDirOrders.push_back("iphonehd");
        resDirOrders.push_back("iphone");

        glview->setDesignResolutionSize(768, 1024, ResolutionPolicy::NO_BORDER);
    }
    else if ( 1136 == screenSize.width || 1136 == screenSize.height ) // retina iPhone (5 and 5S)
    {
        resDirOrders.push_back("iphonehd5");
        resDirOrders.push_back("iphonehd");
        resDirOrders.push_back("iphone");

        glview->setDesignResolutionSize(640, 1136, ResolutionPolicy::NO_BORDER);
    }
    else if ( 960 == screenSize.width || 960 == screenSize.height ) // retina iPhone (4 and 4S)
    {
        resDirOrders.push_back("iphonehd");
        resDirOrders.push_back("iphone");

        glview->setDesignResolutionSize(640, 960, ResolutionPolicy::NO_BORDER);
    }
    else // non retina iPhone and Android devices
    {
        if ( 1080 < screenSize.width ) // android devices that have a high resolution
        {
            resDirOrders.push_back("iphonehd");
            resDirOrders.push_back("iphone");

            glview->setDesignResolutionSize(640, 960, ResolutionPolicy::NO_BORDER);
        }
        else // non retina iPhone and Android devices with lower resolutions
        {
            resDirOrders.push_back("iphone");

            glview->setDesignResolutionSize(320, 480, ResolutionPolicy::NO_BORDER);
        }
    }

    fileUtils->setSearchPaths(resDirOrders);

    // create a scene. it's an autorelease object
    auto scene = MainMenu::createScene();

    // run
    director->runWithScene(scene);

    return true;
}
```

Let's run through the code added to `AppDelegate.cpp`:

- Line 29 is used to help handle file operations.
- Line 30 gets the device's screen size.
- Line 31 creates a vector of strings to store the image directories. There are multiple directories because the game will fall back if it cannot find the image in a higher-resolution directory.
- Lines 34 to 83 check what type of device is being used and use the appropriate directories. To support Android, the `else` statement is used to check whether the device has a screen size bigger than or equal to 1080. If it doesn't, the game uses lower-resolution assets, even for non-retina iPhones.
- Line 85 assigns the image directories to the file utility search paths so that the game searches each directory for the image.

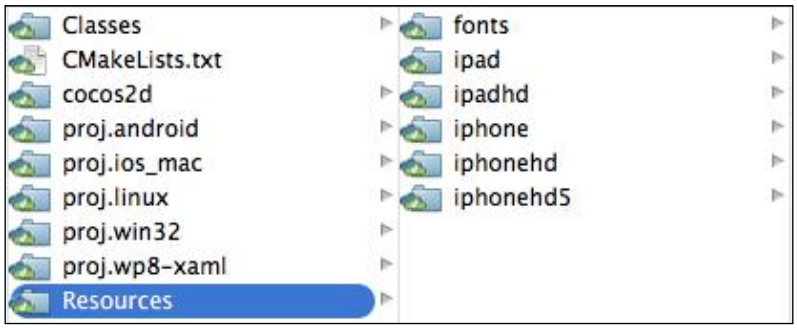
The following is a bit of extra information:

- The frame size is the physical screen size of the device in pixels.
- Setting the design resolution size sets the application's size.
- Pushing back the resolution directories allows the application to fall back to lower-resolution assets if a higher-resolution asset is missing.
- The resolution policy underlines how the application handles differences in the application's design size and the screen's physical size. We chose no border, which prevents black borders by zooming in. This might crop some of the background, but it provides the best effect. Games such as *Candy Crush Saga* use this technique.

Finally, create the following folders in the `Resources` folder:

- `iphone`
- `iphonehd`
- `iphonehd5`
- `ipad`
- `ipadhd`

This is what the `Resources` folder will look like:



Though the folders are named using iOS naming conventions, they support Android as well, using the multiresolution code that was implemented.

Downloading the example code

Download all the images required for the game from the Packt Publishing website and put them in the `Resources` folder.

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.

Summary

In this chapter, we covered how to set up Cocos2d-x and generate a new project. The next chapter will cover adding scenes to the different states the game will be in.

2
Adding Scenes

This chapter will guide you through the process of creating new scenes. Scenes are the different states that a game can be in, for example:

- The Splash scene
- The Menu scene
- The Game scene
- The Pause scene
- The Game Over scene

Almost everything that is used in Cocos2d-x is a subclass of `Node` (part of Cocos2d-x), and scenes are no exception. Nodes provide a lot of basic features such as positioning and are used in scenes as well as in other essentials such as sprites.

All the different screens are represented as scenes within Cocos2d-x. Cocos2d-x provides the functionality to switch between scenes with built-in transitions, making the game more dynamic and vibrant.

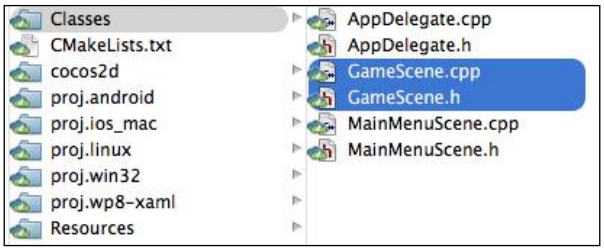
The topics that will be covered in this chapter are as follows:

- Creating new scenes
- Manipulating scenes

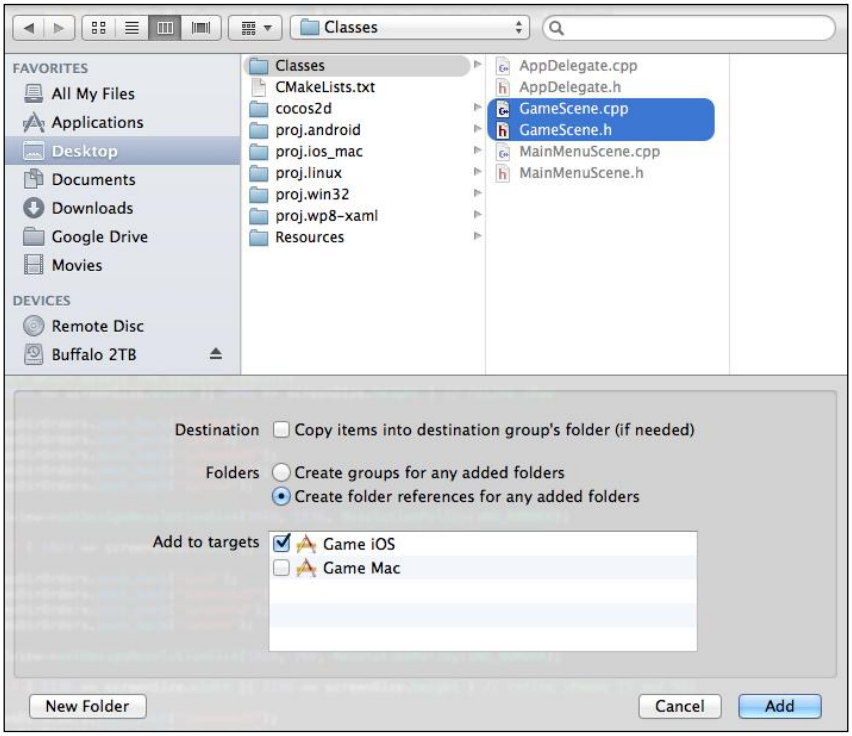
Creating new scenes

The easiest way to create new scenes is to duplicate the existing scenes provided by Cocos2d-x and modify them accordingly. They will require the same modification that was done in the *Housekeeping* section in *Chapter 1, Setting Up*. The following steps will guide you through the process:

1. Duplicate the `MainMenuScene.h` and `MainMenuScene.cpp` files and rename them as `GameScene.h` and `GameScene.cpp` respectively:



2. Then, right-click on the `Classes` folder in XCode and select **Add Files to [project name]**. Select the files to be added and click on **Add**:



Refactoring GameScene.h

The following steps will guide you through the refactoring of `GameScene.h`:

1. Open the `GameScene.h` file.
2. Rename the following `#` commands at the beginning of the document:
    - Change `#ifndef __MAINMENU_SCENE_H__` to `#ifndef __GAME_SCENE_H__`
    - Change `#define __MAINMENU_SCENE_H__` to `#define __GAME_SCENE_H__`
3. Rename the class to `GameScreen`.
4. Change `CREATE_FUNC(MainMenu)` to `CREATE_FUNC(GameScreen)`.

The `GameScene.h` file should look like the following screenshot:

```
#ifndef __GAME_SCENE_H__
#define __GAME_SCENE_H__

#include "cocos2d.h"

class GameScreen : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(GameScreen);
};

#endif // __GAME_SCENE_H__
```

Refactoring GameScene.cpp

The following steps will guide you through the refactoring of `GameScene.cpp`:

1. Open the `GameScene.cpp` file.
2. Include `GameScene.h` instead of `MainMenuScene.h`.
3. Change every instance of `MainMenu` to `GameScreen`.

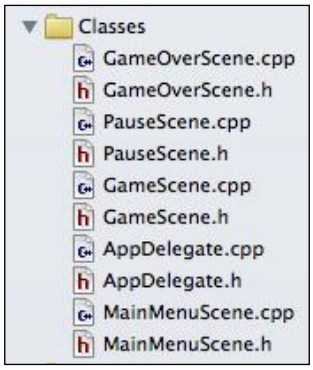
The `GameScene.cpp` file should look like the following screenshot:

```
#include "GameScene.h"

USING_NS_CC;

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

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

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

    // return the scene
    return scene;
}

// on "init" you need to initialize your instance
bool GameScreen::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    return true;
}
```

Create two new scenes using the same steps that were followed for `GameScene` with the following naming conventions:

- For the Game Over scene, use `GameOverScene.h` and `GameOverScene.cpp` as filenames and `GameOver` as the class name
- For the Pause Menu scene, use `PauseScene.h` and `PauseScene.cpp` as filenames and `PauseMenu` as the class name

All the scenes required for the game are now set up, and the XCode project's `Classes` folder should look like this:



Manipulating scenes

Without functionality, moving between scenes is not possible and they would be useless. Cocos2d-x provides great methods to move from one scene to another. Before you take a look at these methods, you need to understand how Cocos2d-x manages scenes.

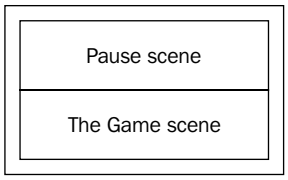
Cocos2d-x uses a stack to manage scenes, which is a **Last-In First-Out** (**LIFO**) system that runs the latest scene in the stack. The following are the main methods that are used to move between scenes:

- **Pushing a scene**: This method pushes a particular scene onto the stack while keeping the current scene but pauses its execution. A real-world example for this is when you click on a pause button, the Pause scene will be pushed onto the stack, while the Game scene still exists.
- **Popping a scene**: This method removes the top/current scene off the stack. A real-world example for this is when resuming a paused scene, the current scene (the paused scene) is removed and goes back to the Game scene.
- **Replacing a scene**: This method replaces the current scene with a new scene, essentially popping the current scene and then pushing the new scene onto the stack. A real-world example for this is when the player dies and goes to the Game Over scene.

Now that we know how Cocos2d-x manages scenes using a stack and the main methods provided, it's time to implement this functionality in the game. The following list shows the use cases for these methods:

- Replacing a scene when the player clicks on the play button from the Main Menu scene to go to the Game scene
- Pushing a scene when the player clicks on the pause button from the Game scene to go to the Pause scene
- Replacing a scene when the player dies to take them to the Game Over scene
- Popping a scene when the player clicks on the resume button from the Pause scene
- Replacing a scene when the player clicks on the retry button from the Game Over scene to go to the Game scene
- Replacing a scene when the player clicks on the main menu button from the Game Over scene to go to the Main Menu scene

There are two use cases that are not mentioned in the preceding list as they are a little more complex. These are when the player clicks on either the retry button or the main menu button from the Pause scene; in these cases, the stack will look like this:



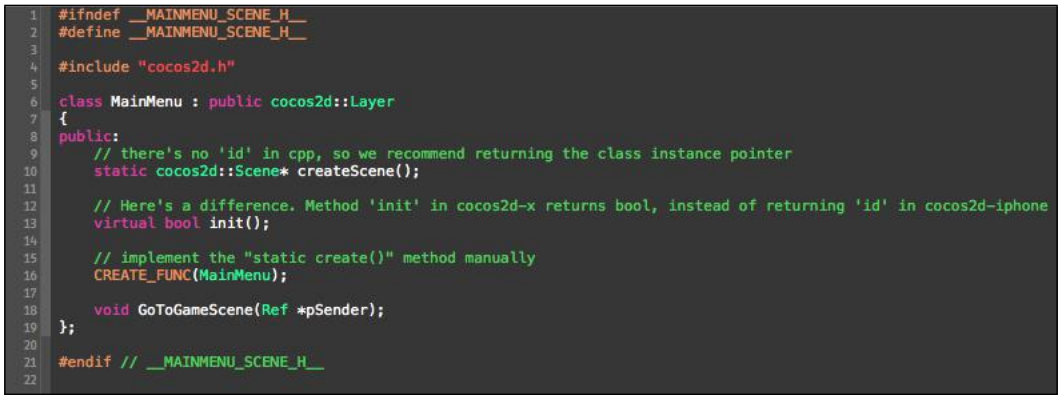
The game is on the Pause scene, but using the replace scene method would remove the Pause scene, leaving an instance of the Game scene constantly dormant. The best technique is to pop the current scene and then immediately use the replace scene method to switch from the Game scene.

Code for the Main Menu scene

Add the following code to the `MainMenuScene.h` file:

```
void GoToGameScene(Ref *pSender);
```

This is the declaration for the function that will be called when the player clicks on the play button from the Main Menu scene to replace it with the Game scene. The `MainMenuScene.h` file should look like the following screenshot:



```
#ifndef __MAINMENU_SCENE_H__
#define __MAINMENU_SCENE_H__

#include "cocos2d.h"

class MainMenu : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(MainMenu);

    void GoToGameScene(Ref *pSender);
};

#endif // __MAINMENU_SCENE_H__
```

Add the following code to the `MainMenuScene.cpp` file:

```
#include "GameScene.h"

void MainMenu::GoToGameScene(Ref *pSender)
{
    auto scene = GameScreen::createScene();

    Director::getInstance()->replaceScene(scene);
}
```

In the preceding code, the first line is used to include the Game scene so that the scene can be accessed. The `GoToGameScene` function first creates a local scene instance of `GameScene` and then replaces the current scene using the Cocos2d-x director. The `MainMenuScene.cpp` file should look like the following screenshot:

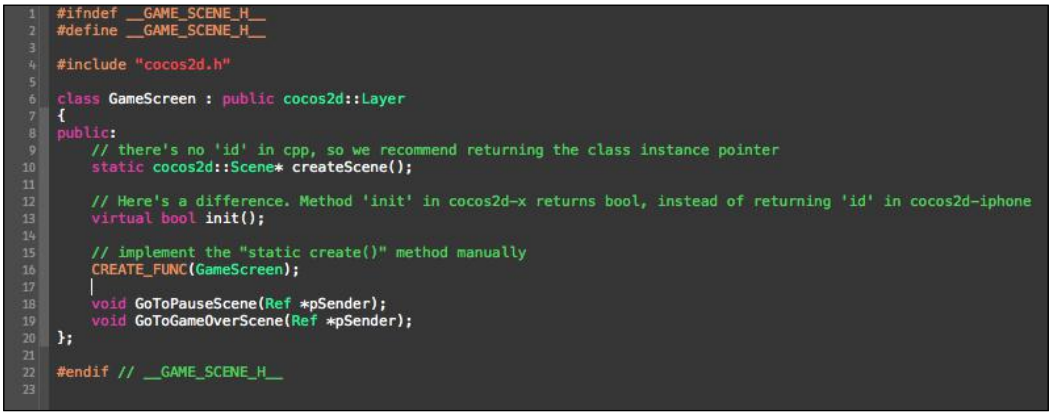
```
#include "MainMenuScene.h"
#include "GameScene.h"

USING_NS_CC;

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

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

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

    // return the scene
    return scene;
}

// on "init" you need to initialize your instance
bool MainMenu::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    return true;
}

void MainMenu::GoToGameScene(cocos2d::Ref *pSender)
{
    auto scene = GameScreen::createScene();

    Director::getInstance()->replaceScene(scene);
}
```

Code for the Game scene

Add the following code to the `GameScene.h` file:

```
void GoToPauseScene(Ref *pSender);
void GoToGameOverScene(Ref *pSender);
```

In the preceding code, the first function declaration will be called when the player clicks on the pause button in the Game scene to push the Pause scene onto the stack. The second function declaration will be called when the player dies to replace the Game scene with the Game Over scene. The `GameScene.h` file should look like the following screenshot:



```
#ifndef __GAME_SCENE_H__
#define __GAME_SCENE_H__

#include "cocos2d.h"

class GameScreen : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(GameScreen);

    void GoToPauseScene(Ref *pSender);
    void GoToGameOverScene(Ref *pSender);
};

#endif // __GAME_SCENE_H__
```

Add the following code to the `GameScene.cpp` file:

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

void GameScreen::GoToPauseScene(cocos2d::Ref *pSender)
{
    auto scene = PauseMenu::createScene();

    Director::getInstance()->pushScene(scene);
}

void GameScreen::GoToGameOverScene(cocos2d::Ref *pSender)
{
    auto scene = GameOver::createScene();

    Director::getInstance()->replaceScene(scene);
}
```

The first two lines in the preceding code snippet are used to include the Pause and Game Over scenes so that the scenes can be accessed. The `GoToPauseScene` function first creates a local scene instance of the Pause scene and then pushes it onto the stack. The `GoToGameOverScene` function first creates a local scene instance of the Game Over scene and then replaces the Game scene with it. The `GameScene.cpp` file should look like the following screenshot:

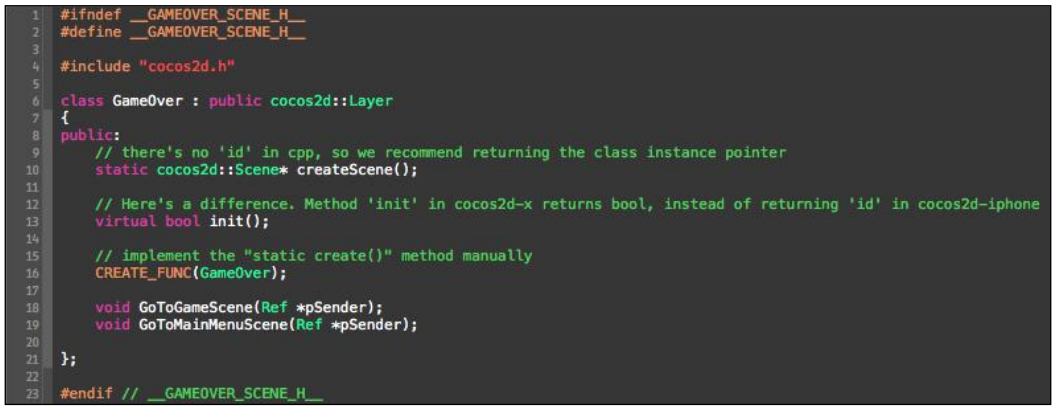
```
#include "GameScene.h"
#include "PauseScene.h"
#include "GameOverScene.h"

USING_NS_CC;

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

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

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

    // return the scene
    return scene;
}

// on "init" you need to initialize your instance
bool GameScreen::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    return true;
}

void GameScreen::GoToPauseScene(cocos2d::Ref *pSender)
{
    auto scene = PauseMenu::createScene();

    Director::getInstance()->pushScene(scene);
}

void GameScreen::GoToGameOverScene(cocos2d::Ref *pSender)
{
    auto scene = GameOver::createScene();

    Director::getInstance()->replaceScene(scene);
}
```

Code for the Game Over scene

Add the following code to the `GameOverScene.h` file:

```
void GoToGameScene(Ref *pSender);
void GoToMainMenuScene(Ref *pSender);
```

In the preceding code snippet, the first function declaration will be called when the player clicks on the retry button from the Game Over scene to replace the current scene with the Game scene. The second function declaration will be called when the player clicks on the main menu button from the Game Over scene to replace the current scene with the Main Menu scene. The `GameOverScene.h` file should look like the following screenshot:



```
#ifndef __GAMEOVER_SCENE_H__
#define __GAMEOVER_SCENE_H__

#include "cocos2d.h"

class GameOver : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(GameOver);

    void GoToGameScene(Ref *pSender);
    void GoToMainMenuScene(Ref *pSender);

};

#endif // __GAMEOVER_SCENE_H__
```

Add the following code to the `GameOverScene.cpp` file:

```
#include "GameScene.h"
#include "MainMenuScene.h"

void GameOver::GoToGameScene(cocos2d::Ref *pSender)
{
    auto scene = GameScreen::createScene();

    Director::getInstance()->replaceScene(scene);
}

void GameOver::GoToMainMenuScene(cocos2d::Ref *pSender)
{
    auto scene = MainMenu::createScene();

    Director::getInstance()->replaceScene(scene);
}
```

The first two lines in the preceding code snippet are used to include the Game and Main Menu scenes so that the scenes can be accessed. The `GoToGameScene` function first creates a local scene instance of the Game scene and then replaces the Game Over scene with it. The `GoToMainMenuScene` function first creates a local scene instance of the Main Menu scene and then replaces the Game Over scene with it. The `GameOverScene.cpp` file should look like the following screenshot:

```
#include "GameOverScene.h"
#include "GameScene.h"
#include "MainMenuScene.h"

USING_NS_CC;

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

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

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

    // return the scene
    return scene;
}

// on "init" you need to initialize your instance
bool GameOver::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    return true;
}

void GameOver::GoToGameScene(cocos2d::Ref *pSender)
{
    auto scene = GameScreen::createScene();

    Director::getInstance()->replaceScene(scene);
}

void GameOver::GoToMainMenuScene(cocos2d::Ref *pSender)
{
    auto scene = MainMenu::createScene();

    Director::getInstance()->replaceScene(scene);
}
```

Code for the Pause scene

Add the following code to the `PauseScene.h` file:

```
void Resume(Ref *pSender);
void GoToMainMenuScene(Ref *pSender);
void Retry(Ref *pSender);
```

In the preceding code snippet, the first function declaration will be called when the player clicks on the resume button from the Pause scene to pop the Pause scene off the stack. The second function declaration will be called when the player clicks on the main menu button from the Pause scene to replace the current scene with the Main Menu scene while popping all the scenes off the stack. The third function declaration will be called when the player clicks on the retry button from the Pause scene to replace the current scene with the Game scene while popping all the scenes off the stack. The `PauseScene.h` file should look like the following screenshot:

```
#ifndef __PAUSEMENU_SCENE_H__
#define __PAUSEMENU_SCENE_H__

#include "cocos2d.h"

class PauseMenu : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(PauseMenu);

    void Resume(Ref *pSender);
    void GoToMainMenuScene(Ref *pSender);
    void Retry(Ref *pSender);

};

#endif // __PAUSEMENU_SCENE_H__
```

Add the following code to the `PauseScene.cpp` file:

```
#include "GameScene.h"
#include "MainMenuScene.h"

void PauseMenu::Resume(cocos2d::Ref *pSender)
{
    Director::getInstance()->popScene();
}

void PauseMenu::GoToMainMenuScene(cocos2d::Ref *pSender)
{
    auto scene = MainMenu::createScene();

    Director::getInstance()->popScene();
    Director::getInstance()->replaceScene(scene);
```

```
}

void PauseMenu::Retry(cocos2d::Ref *pSender)
{
    auto scene = GameScreen::createScene();

    Director::getInstance()->popScene();
    Director::getInstance()->replaceScene(scene);
}
```

The first two lines in the preceding code snippet are used to include the Game and Main Menu scenes so that the scenes can be accessed. The `GoToMainMenuScene` function first creates a local scene instance of the Main Menu scene, then it pops the current scene off the stack and the Game scene is replaced with the Main Menu scene. The `Retry` function first creates a local scene instance of the Game scene, then it pops the current scene off the stack and the current Game scene is replaced with the new Game scene (restarts the game). The `PauseScene.cpp` file should look like the following screenshot:

```
#include "PauseScene.h"
#include "GameScene.h"
#include "MainMenuScene.h"

USING_NS_CC;

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

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

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

    // return the scene
    return scene;
}

// on "init" you need to initialize your instance
bool PauseMenu::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    return true;
}

void PauseMenu::Resume(cocos2d::Ref *pSender)
{
    Director::getInstance()->popScene();
}

void PauseMenu::GoToMainMenuScene(cocos2d::Ref *pSender)
{
    auto scene = MainMenu::createScene();

    Director::getInstance()->popScene();
    Director::getInstance()->replaceScene(scene);
}

void PauseMenu::Retry(cocos2d::Ref *pSender)
{
    auto scene = GameScreen::createScene();

    Director::getInstance()->popScene();
    Director::getInstance()->replaceScene(scene);
}
```

Summary

This chapter covered how to create new scenes using the default scenes provided by Cocos2d-x as templates and the techniques to switch between them. We haven't actually implemented any code to trigger the scene functions, but it will be implemented in the upcoming chapters. The next chapter will cover menu systems, which will help you to move between scenes.

3
Adding Game Menus

This chapter will guide you through the process of creating menus using the menu items provided by Cocos2d-x and implementing them within the previously created scenes to allow navigation, which was implemented in the previous chapter.

The topics that will be covered in this chapter are as follows:

- Setting up menus
- Adding menu items
- Menu alignment

Setting up the menu

So, what are menus? A menu is a technique to display buttons/navigational functionality. Menus consist of menu items, for example, a menu item image that will be used for this game. There are several items that are provided by Cocos2d-x to construct menus. Some of the commonly used items are as follows:

- The Menu Font item
- The Menu Sprite item
- The Menu Label item

For more information on any of the Cocos2d-x APIs, visit `http://www.cocos2d-x.org/wiki/Reference`.

Menus are a collection of items that are organized to form the structured buttons and can be used for a game's functionality, such as navigation. Our game will use menus for the following use cases:

- In the Main Menu scene:
  - The play button, which starts the game
- In the Game scene:
  - The pause button, which pauses the game
- In the Pause scene:
  - The resume button, which resumes the game
  - The retry button, which restarts the game
  - The main menu button, which takes you to the main menu
- In the Game Over scene:
  - The retry button, which restarts the game
  - The main menu button, which takes you to the main menu

Each scene will require its own menus along with its own menu items. All the menus will be declared and constructed within the `init()` method, but if the requirement for manipulating the menu or menu items arises, it would be best to declare them within the header so that they can be accessed outside the `init()` method.

The project has been changed from landscape mode to portrait mode. This is done using XCode project details (select the required orientation) or using `AndroidManifest.xml (android:screen Orientation=portrait)`.

Coding the menus in the Main Menu scene

The main menu requires a simple menu system for navigation between itself and the game screen. Menu item images will be used to display a game title and to display the button that the user can interact with to maneuver between scenes.

Add the following code to the `init()` method:

```
auto menuTitle =
MenuItemImage::create("MainMenuScreen/Game_Title.png",
"MainMenuScreen/Game_Title.png");

auto playItem =
MenuItemImage::create("MainMenuScreen/Play_Button.png",
"MainMenuScreen/Play_Button(Click).png",
CC_CALLBACK_1(MainMenu::GoToGameScene, this));

auto menu = Menu::create(menuTitle, playItem, NULL);
menu->alignItemsVerticallyWithPadding(visibleSize.height / 4);
this->addChild(menu);
```

Menu item images have three states: normal (not being pressed), pressed (user is tapping it), and disabled (item has been disabled).

The first statement in the preceding code snippet creates a menu title using the menu item image with the following parameters:

- The default image that has to be displayed in the title
- The image that has to be displayed when the title is tapped on (as the title cannot be tapped on, it is the same as the default image to give the illusion of a static item)

The second statement creates an image item with the following parameters:

- The default image that has to be displayed on the button
- The image that has to be displayed when the button is being tapped on
- The function that has to be called when the button is tapped on, which will take the player to the Game scene. The number `1` at the end of `CC_CALLBACK_1` specifies how many parameters the function that is being called takes

The third statement creates a menu with the `Menu` items. To add multiple items, simply separate them with a comma.

The fourth statement aligns the items of the menu vertically. There are several built-in alignment methods, including the ability to manually position the menu items, which will be covered in the next section.

The fifth statement adds the menu as a child to the scene. To add children to the scene, the `addChild()` method requires a node, which is essentially menus, sprites, labels, and so on.

The code in the `init()` method should look like the following screenshot:

```
21  // on "init" you need to initialize your instance
22  bool MainMenu::init()
23  {
24      //////////////////////////////
25      // 1. super init first
26      if ( !Layer::init() )
27      {
28          return false;
29      }
30
31      Size visibleSize = Director::getInstance()->getVisibleSize();
32      Point origin = Director::getInstance()->getVisibleOrigin();
33
34      auto menuTitle = MenuItemImage::create("MainMenuScreen/Game_Title.png",
35                                             "MainMenuScreen/Game_Title.png");
36      auto playItem = MenuItemImage::create("MainMenuScreen/Play_Button.png",
37                                            "MainMenuScreen/Play_Button(Click).png",
38                                            CC_CALLBACK_1(MainMenu::GoToGameScene, this));
39
40      auto menu = Menu::create(menuTitle, playItem, NULL);
41      menu->alignItemsVerticallyWithPadding(visibleSize.height / 4);
42      this->addChild(menu);
43
44      return true;
45  }
```

The main menu should now look like the following screenshot:

Coding the menus in the Game scene

The Game scene requires a simple menu system to pause the game. A menu item image will be used to display the button that the user can interact with in order to pause the game and load the Pause scene.

Add the following code to the `init()` method:

```
auto pauseItem =
MenuItemImage::create("GameScreen/Pause_Button.png",
"GameScreen/Pause_Button(Click).png",
CC_CALLBACK_1(GameScreen::GoToPauseScene, this));

pauseItem->setPosition(Point(pauseItem->getContentSize().width -
(pauseItem->getContentSize().width / 4) + origin.x,
visibleSize.height - pauseItem->getContentSize().height +
(pauseItem->getContentSize().width / 4) + origin.y));

auto menu = Menu::create(pauseItem, NULL);
menu->setPosition(Point::ZERO);
this->addChild(menu);
```

The first statement creates an image item with the parameters explained in the previous section.

The second statement sets the position of the menu item to the top-left corner of the screen.

Cocos2d-x starts at the bottom-left corner, so the height of the screen needs to be added to position it at the top.

The `visibleSize` variable that was included with the generated project gives the screen's visible size. The origin is where the coordinates start, which is usually (0, 0), but factoring this in will prevent any position problems on other devices, such as certain Android devices.

The third statement creates a menu with the `Menu` item. To add multiple items, simply separate them with a comma.

The fourth statement sets the menu's position to `0`, as the menu item was positioned separately.

The fifth statement adds the menu as a child to the scene. To add children to the scene, the `addChild()` method requires a node.

The code in the `init()` method should look like the following screenshot:

```
// on "init" you need to initialize your instance
bool GameScreen::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    auto pauseItem = MenuItemImage::create("GameScreen/Pause_Button.png",
                                           "GameScreen/Pause_Button(Click).png",
                                           CC_CALLBACK_1(GameScreen::GoToPauseScene, this));
    pauseItem->setPosition(Point(pauseItem->getContentSize().width - (pauseItem->getContentSize().width / 4) + origin.x,
                                 visibleSize.height - pauseItem->getContentSize().height + (pauseItem->getContentSize().width / 4) + origin.y));

    auto menu = Menu::create(pauseItem, NULL);
    menu->setPosition(Point::ZERO);
    this->addChild(menu);

    return true;
}
```

The game screen should now look like the following screenshot:

Coding the menus in the Pause scene

The Pause scene requires a menu system to resume the game, restart the game, and go back to the main menu. Menu item images will be used to display the buttons that the user can interact with in order to perform the actions mentioned.

Add the following code to the `init()` method:

```
auto resumeItem =
MenuItemImage::create("PauseScreen/Resume_Button.png",
"PauseScreen/Resume_Button(Click).png",
CC_CALLBACK_1(PauseMenu::Resume, this));

auto retryItem =
MenuItemImage::create("PauseScreen/Retry_Button.png",
"PauseScreen/Retry_Button(Click).png",
CC_CALLBACK_1(PauseMenu::Retry, this));

auto mainMenuItem =
MenuItemImage::create("PauseScreen/Menu_Button.png",
"PauseScreen/Menu_Button(Click).png",
CC_CALLBACK_1(PauseMenu::GoToMainMenuScene, this));

auto menu = Menu::create(resumeItem, retryItem, mainMenuItem,
NULL);

menu->alignItemsVerticallyWithPadding(visibleSize.height / 4);
this->addChild(menu);
```

The first three statements create the menu item images for the menu as mentioned previously.

The fourth statement creates a menu with the menu items created in the previous statements.

The fifth statement aligns the menu items vertically but with padding; they would otherwise be aligned next to each other.

The sixth statement adds the menu as a child to the scene. To add children to the scene, the `addChild()` method requires a node.

The code in the `init()` method should look like the following screenshot:

```
// on "init" you need to initialize your instance
bool PauseMenu::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    auto resumeItem = MenuItemImage::create("PauseScreen/Resume_Button.png",
                                            "PauseScreen/Resume_Button(Click).png",
                                            CC_CALLBACK_1(PauseMenu::Resume, this));
    auto retryItem = MenuItemImage::create("PauseScreen/Retry_Button.png",
                                           "PauseScreen/Retry_Button(Click).png",
                                           CC_CALLBACK_1(PauseMenu::Retry, this));
    auto mainMenuItem = MenuItemImage::create("PauseScreen/Menu_Button.png",
                                              "PauseScreen/Menu_Button(Click).png",
                                              CC_CALLBACK_1(PauseMenu::GoToMainMenuScene, this));

    auto menu = Menu::create(resumeItem, retryItem, mainMenuItem, NULL);
    menu->alignItemsVerticallyWithPadding(visibleSize.height / 4);
    this->addChild(menu);

    return true;
}
```

Menus in the Pause scene should now look like the following screenshot:

Coding the menus in the Game Over scene

The Game Over scene requires a menu system to restart the game and to go back to the main menu. Menu item images will be used to display the buttons that the user can interact with to perform the actions mentioned.

Add the following code to the `init()` method:

```
auto menuTitle =
MenuItemImage::create("GameOverScreen/Game_Over.png",
"GameOverScreen/Game_Over.png");

auto retryItem =
MenuItemImage::create("GameOverScreen/Retry_Button.png",
"GameOverScreen/Retry_Button(Click).png",
CC_CALLBACK_1(GameOver::GoToGameScene, this));

auto mainMenuItem =
MenuItemImage::create("GameOverScreen/Menu_Button.png",
"GameOverScreen/Menu_Button(Click).png",
CC_CALLBACK_1(GameOver::GoToMainMenuScene, this));

auto menu = Menu::create(menuTitle, retryItem, mainMenuItem,
NULL);

menu->alignItemsVerticallyWithPadding(visibleSize.height / 4);
this->addChild(menu);
```

The first statement creates a menu title using the menu item image, but the callback is not specified as it is static and the image that is tapped on is its default image, providing the illusion of a menu title that doesn't change.

The next two statements create the menu item images with the parameters mentioned in the previous sections.

The fourth statement creates the menu with the items created in the previous statements.

The fifth statement aligns the items vertically with padding.

The sixth statement adds the menu as a child to the scene. To add children to the scene, the `addChild()` method requires a node that is essentially menus, sprites, labels, and so on.

The `init()` method should give you the following screenshot:

```
// on "init" you need to initialize your instance
bool GameOver::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    auto menuTitle = MenuItemImage::create("GameOverScreen/Game_Over.png",
                                           "GameOverScreen/Game_Over.png");
    auto retryItem = MenuItemImage::create("GameOverScreen/Retry_Button.png",
                                           "GameOverScreen/Retry_Button(Click).png",
                                           CC_CALLBACK_1(GameOver::GoToGameScene, this));
    auto mainMenuItem = MenuItemImage::create("GameOverScreen/Menu_Button.png",
                                              "GameOverScreen/Menu_Button(Click).png",
                                              CC_CALLBACK_1(GameOver::GoToMainMenuScene, this));
    auto menu = Menu::create(menuTitle, retryItem, mainMenuItem, NULL);
    menu->alignItemsVerticallyWithPadding(visibleSize.height / 4);
    this->addChild(menu);

    return true;
}
```

The Game Over screen should now look like the following screenshot (testing this screen may not be immediately possible; this can be achieved by temporarily making the pause button call the `GameOver` function, but remember to change it back):

Summary

This chapter covered how to create menus in different scenes with menu items and how to position these menus. These menus provided the functionality to switch between scenes for the game. The next chapter will show you how to implement transitions so that the scenes feel more alive when you switch between them. Transitions allow the scenes to switch using an animation, and Cocos2d-x provides several animations from fading to zooming.

4
Scene Transitions

This chapter will guide you through the process of implementing transitions to switch between scenes, making the process more dynamic and less static.

The topics covered in this chapter are as follows:

- The fade transition
- Additional transitions

Transitions allow an animation to be played when you switch between scenes, making the game look more dynamic and attractive for the player. Transitions can be used when you want to push or replace scenes.

The fade transition

The game that will be developed through the course of this book will incorporate the fade transition, as it is a universal transition that is easily recognized; however, Cocos2d-x provides many more transitions, which will be covered briefly. To learn more about the various transitions offered by Cocos2d-x, visit `http://www.cocos2d-x.org/reference/native-cpp/V3.0/da/d00/group__transition.html`.

An example syntax of scene transition is as follows:

```
Director::getInstance()->replaceScene
(TransitionName::create(transition properties));
```

As the preceding example shows, transitions are incorporated within the existing scene code and allow the scenes to be manipulated very easily but provide great end results.

The fade transition fades between two different scenes. There are two instances of the fade transition. The first instance allows you to set the duration and scene that you want to transition to; the second provides the same functionality but also allows the color of the fade transition to be set. The first instance will be used for this example. To learn more about the different transitions, visit the URL mentioned earlier.

Replace all the lines of code within the project that push a scene or replace a scene with the following code:

```
Director::getInstance()->replaceScene(TransitionFade::create
(1.0, scene));
```

Alternatively, you can also use the following code:

```
Director::getInstance()->pushScene(TransitionFade::create
(1.0, scene));
```

For example, the `GoToGameScene` function within the `MainMenuScene.cpp` file should look similar to the following screenshot:

```
47  void MainMenu::GoToGameScene(cocos2d::Ref *pSender)
48  {
49      auto scene = GameScreen::createScene();
50
51      Director::getInstance()->replaceScene(TransitionFade::create(1.0, scene));
52  }
```

Although the game will only incorporate the fade transition, the remaining sections of this chapter will briefly cover some of the other popular transitions that Cocos2d-x provides.

Additional transitions

This section will cover some of the additional transitions that are provided by Cocos2d-x and how to incorporate them. Different transitions allow your game to be more dynamic. For more in-depth information, visit `http://www.cocos2d-x.org/reference/native-cpp/V3.0/da/d00/group__transition.html`. The following list details the additional transitions that are provided by Cocos2d-x:

- **Flip transitions**: These transitions flip around a specified axis, with the current scene moving away and the new scene coming in.

  The example code for this transition is as follows:

  ```
  Director::getInstance()->replaceScene
  (TransitionFlipX::create(3, scene));
  ```

- **Zoom transitions**: These transitions flip the scenes in a similar way to flip transitions but also allow them to be zoomed.

  The example code for this transition is as follows:

```
Director::getInstance()->replaceScene
(TransitionZoomFlipY::create(3, scene));
```

- **Page transitions**: These transitions move between the scenes as if they were pages in a book.

  The example code for this transition is as follows:

```
Director::getInstance()->replaceScene
(TransitionPageTurn::create(3, scene, false));
```

The `popScene()` function cannot take a scene transition.

Summary

This chapter covered scene transitions, which can be used to liven up the switching between scenes. Cocos2d-x provides several built-in transitions, which aid in the development. Although Cocos2d-x offers several different transitions, most of the successful games use only a few selected transitions if not one. Using several different transitions can make the game look unprofessional like the early websites that contained loads of animations. The next chapter will cover how to add sprites to games. Sprites allow bitmaps to be drawn; for example, obstacles and player characters, which are the foundations of what the user sees.

5
The Game Sprites

This chapter will guide you through the process of using sprites to display game objects such as the background, player characters, and **Non-playable Characters** (**NPCs**).

Sprites are computer graphics that are used to represent objects to be displayed within a scene; for example, a player would be a sprite within the game environment. These player sprites would interact with other sprites that are present.

The topics that are covered in this chapter are as follows:

- Adding sprites
- Positioning sprites
- Moving sprites

Sprites simplify the process of displaying game objects—you don't need to create a bitmap with assigned memory, position the bitmap, and draw it. Sprites allow all of this to be done behind the scenes. Cocos2d-x has built-in methods to create, manipulate, and display sprites.

Each section will cover how to add the game sprites within their appropriate scenes and manipulate them if necessary, for example, moving the background sprite to provide a scrolling effect. The following is an example of the sprite declaration to add a sprite to a scene:

```
auto backgroundSprite = Sprite::create
("MainMenuScreen/Background.png");
```

The preceding line of code creates a sprite variable called `backgroundSprite`, which is created from a PNG image. It is recommended that you use PNG images as they allow transparency. The basic process of displaying a sprite is as follows:

1. Declare a sprite.

Declare the sprite in the header if it needs to be manipulated after the declaration.

2. Initialize the sprite.
3. Position the sprite.
4. Set any other properties, such as the scale and rotation.
5. Draw a sprite that is handled via Cocos2d-x.

Adding the Main Menu sprites

The only sprite that needs to be displayed in the Main Menu scene is the background. Add the following code below the menu code in the `init()` method within the `MainMenuScene.cpp` file:

```
auto backgroundSprite = Sprite::create
("MainMenuScreen/Background.png");

backgroundSprite->setPosition(Point((visibleSize.width / 2) +
origin.x, (visibleSize.height / 2) + origin.y));


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

The first statement of the preceding code declares and initializes a sprite called `backgroundSprite` with a PNG image.

The second statement sets the position of the background sprite at the center of the screen using the visible size, which is the area of the screen that is available on a particular device, as there are different screen sizes. The origin is used as some devices (mainly Android) don't start at (0, 0) because of onscreen buttons and so on. This will help to overcome positioning issues.

The third statement adds the sprite as a child to the scene and also sets the z-index value, which states the order of the items. As the background sprite should be below all the other items, the z-index is set to `-1` since by default the other objects are set to `0`. Although the z-index was set to `-1`, a little thing to note is that any node added after it will automatically be on top of the previous nodes if the z-index isn't set.

The `init()` method should give you the following screenshot:

```
// on "init" you need to initialize your instance
bool MainMenu::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    auto menuTitle = MenuItemImage::create("MainMenuScreen/Game_Title.png",
                                           "MainMenuScreen/Game_Title.png");
    auto playItem = MenuItemImage::create("MainMenuScreen/Play_Button.png",
                                          "MainMenuScreen/Play_Button(Click).png",
                                          CC_CALLBACK_1(MainMenu::GoToGameScene, this));

    auto menu = Menu::create(menuTitle, playItem, NULL);
    menu->alignItemsVerticallyWithPadding(visibleSize.height / 4);
    this->addChild(menu);

    auto backgroundSprite = Sprite::create("MainMenuScreen/Background.png");
    backgroundSprite->setPosition(Point((visibleSize.width / 2) + origin.x, (visibleSize.height / 2) + origin.y));
    this->addChild(backgroundSprite, -1);

    return true;
}
```

The GUI after adding the Main Menu sprites will look as shown in the following screenshot:

Adding the Game Over sprites

The only sprite that needs to be displayed in the Game Over scene is the background. Add the following code below the menu code in the `init()` method within the `GameOverScene.cpp` file, as we did in the previous section:

```
auto backgroundSprite = Sprite::create
("GameOverScreen/Game_Over_Screen_Background.png");

backgroundSprite->setPosition(Point((visibleSize.width / 2) +
origin.x, (visibleSize.height / 2) + origin.y));


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

The GUI after adding the Game Over sprites should look like what is shown in the following screenshot:

Adding the Pause sprites

The only sprite that needs to be displayed in the Pause scene is the background. Add the following code below the menu code in the `init()` method within the `PauseScene.cpp` file, as we did in the previous section:

```
auto backgroundSprite = Sprite::create
("PauseScreen/Pause_Background.png");
```

```
backgroundSprite->setPosition(Point((visibleSize.width / 2) +
origin.x, (visibleSize.height / 2) + origin.y));

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

The GUI after adding the Pause sprites should look like what is shown in the following screenshot:

Adding the Game sprites

Unlike in the previous backgrounds, the Game scene's background will be moving, providing a scrolling effect. Before you dive into the background code, an `update` function needs to be scheduled to be called in every frame using the following steps:

1. Schedule the `update` function in the `init()` method using the following code:

   ```
   this->scheduleUpdate();
   ```

2. Declare the `update` function in the `GameScene.h` file using the following code:

   ```
   void update(float dt);
   ```

3. Implement the `update` function in the `GameScene.cpp` file using the following code:

```
void GameScreen::update(float dt)
{

}
```

The `update` function is used to handle the game's logic and takes a `float` parameter called delta time, which is the time between frames. As all devices are not capable of running at 60 fps, this is factored in during gameplay to provide a smooth experience across a wide variety of devices.

The following are the steps to implement a scrolling background:

1. Declare a sprite array for the background consisting of two items. As the background will be scrolling, two sprites will be used to provide the illusion of a consistent background. However, once a sprite has gone off the screen, it will reset back. The following code needs to be added within the header:

```
cocos2d::Sprite *backgroundSpriteArray[2];
```

2. Initialize the sprite array and set the position of the sprites; so, one is displayed on the screen initially, and the other one below the screen will be displayed as the background scrolls upwards. Then, add these to the scene as children; all this will be done within the `init()` method:

```
for (int i = 0; i < 2; i ++)
{
    backgroundSpriteArray[i] = Sprite::create
    ("GameScreen/Game_Screen_Background.png");
    backgroundSpriteArray[i]->setPosition
    (Point((visibleSize.width / 2) + origin.x, (-1 *
    visibleSize.height * i) + (visibleSize.height / 2) +
    origin.y));
    this->addChild(backgroundSpriteArray[i], -2);
}
```

3. Next, move the sprites within the `update()` method by factoring in the delta time (0.75 is just the speed that can be changed, possibly by the user in a settings scene, which could be implemented as a side task, and it is multiplied by the screen height to keep a constant speed on different devices). You will also need to add a `visibleSize` variable, similar to what we did in the constructor:

```
for (int i = 0; i < 2; i ++)
    {
      backgroundSpriteArray[i]->setPosition
```

```
        (Point(backgroundSpriteArray[i]->getPosition().x,
        backgroundSpriteArray[i]->getPosition().y + (0.75 *
        visibleSize.height * dt)));
    }
```

4. Although the backgrounds scroll, they do not reset once they have gone above the screen. Add the following code above the previous code:

```
Size visibleSize = Director::getInstance()->
getVisibleSize();
Point origin = Director::getInstance()->getVisibleOrigin();
for (int i = 0; i < 2; i ++)
{
    if (backgroundSpriteArray[i]->getPosition().y >=
    visibleSize.height + (visibleSize.height / 2) -1)
    {
      backgroundSpriteArray[i]->setPosition
      (Point((visibleSize.width / 2) + origin.x, (-1 *
      visibleSize.height) + (visibleSize.height / 2)));
    }
}
```

The background is now scrolling and resets once it has gone off the screen.

Now, the scrolling asteroids will be implemented. The asteroids will spawn every 2.5 seconds randomly in the *x* axis just below the screen. Once an asteroid goes off the screen, it will be destroyed. Add the following code to implement scrolling asteroids:

1. Declare the spawn asteroid method in `GameScene.h` (the scheduler requires a `float` parameter for delta time, but doesn't need it to be used):

```
void spawnAsteroid(float dt);
```

2. Declare the asteroids that will be stored as a vector of sprites as they can be dynamically added and removed:

```
std::vector<cocos2d::Sprite *> asteroids;
```

3. Implement the scheduler within the `init()` method inside the `GameScene.cpp` file (as the game will spawn asteroids every 1.0 second. 1.0 is specified as the interval time, but if it isn't specified, the method will be run for every frame):

```
this->schedule
(schedule_selector(GameScreen::spawnAsteroid), 1.0);
```

4. Implement the `spawnAsteroid(float dt)` method:

```
void GameScreen::spawnAsteroid(float dt)
{
    Size visibleSize = Director::getInstance()->
    getVisibleSize();
```

```
    Point origin = Director::getInstance()->
    getVisibleOrigin();

    int asteroidIndex = (arc4random() % 3) + 1;
    __String* asteroidString = __String::
    createWithFormat("GameScreen/Asteroid_%i.png",
    asteroidIndex);

    Sprite *tempAsteroid = Sprite::create(asteroidString->
    getCString());

    int xRandomPosition = (arc4random() % (int)
    (visibleSize.width - (tempAsteroid->
    getContentSize().width / 2))) + (tempAsteroid->
    getContentSize().width / 2);

    tempAsteroid->setPosition(Point(xRandomPosition +
    origin.x, -tempAsteroid->getContentSize().height +
    origin.y));
    asteroids.push_back(tempAsteroid);
    this->addChild(asteroids[asteroids.size() - 1], -1);
}
```

5. Before implementing the code to check whether an asteroid has gone off the screen, let's go over what the `spawnAsteroid(float dt)` method does:
    - Firstly, the visible size and origin are declared and initialized
    - Then, `asteroidIndex` stores a random number that determines which asteroid will be spawned (as there are three different asteroids)
    - Next, the filename for the random asteroid is created using Cocos2d-x's built-in `__String` object, and using `this`, a temporary asteroid sprite is declared and initialized
    - The asteroid is then positioned below the screen and randomly along the *x* axis
    - Then, the asteroid is added to the asteroids' vector
    - The asteroid is added as a child to the scene

6. Finally, add the following code to the `update(float dt)` method, which will move the asteroids at the same speed as the speed at which the background is moving. Then, it will check whether the asteroid has moved off the screen; if so, it will remove the asteroid from the scene and from the asteroid's vector:

```
for (int i = 0; i < asteroids.size(); i++)
{
    asteroids[i]->setPosition(Point(asteroids[i]->
    getPosition().x, asteroids[i]->getPosition().y +
    (0.75 * visibleSize.height * dt)));

    if (asteroids[i]->getPosition().y >
    visibleSize.height * 2)
    {
      this->removeChild(asteroids[i]);
      asteroids.erase(asteroids.begin() + i);
    }
}
```

The final sprite to be added is the player's spaceship; use the following steps:

1. Declare the player sprite in the `GameScene.h` file:

```
cocos2d::Sprite *playerSprite;
```

2. Initialize the player sprite within the `init()` method in the `GameScene.cpp` file:

```
playerSprite = Sprite::create("GameScreen/Space_Pod.png");
```

3. Position the player sprite below the pause button centered along the *x* axis:

```
playerSprite->setPosition(Point(visibleSize.width / 2,
pauseItem->getPosition().y - (pauseItem->
getContentSize().height / 2) - (playerSprite->
getContentSize().height / 2)));
```

4. Now add the player sprite as a child to the scene:

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

Now that the sprites have been implemented, here are the images of how the `GameScene.h` and `GameScene.cpp` files should look.

The following screenshot shows how the `GameScene.h` file should look after all the code has been added by following the previous steps:

```
#ifndef __GAME_SCENE_H__
#define __GAME_SCENE_H__

#include "cocos2d.h"

class GameScreen : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(GameScreen);

    void GoToPauseScene(Ref *pSender);
    void GoToGameOverScene(Ref *pSender);

    void update(float dt);

    void spawnAsteroid(float dt);

    std::vector<cocos2d::Sprite *> asteroids;

    cocos2d::Sprite *backgroundSpriteArray[2];
    cocos2d::Sprite *playerSprite;
};

#endif // __GAME_SCENE_H__
```

The following screenshot shows how the `init()` method inside `GameScene.cpp` should look after all the code has been added by following the previous steps to initialize the game's sprites:

```
// on "init" you need to initialize your instance
bool GameScreen::init()
{
    //////////////////////////////
    // 1. super init first
    if ( !Layer::init() )
    {
        return false;
    }

    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    auto pauseItem = MenuItemImage::create("GameScreen/Pause_Button.png",
                                           "GameScreen/Pause_Button(Click).png",
                                           CC_CALLBACK_1(GameScreen::GoToPauseScene, this));
    pauseItem->setPosition(Point(pauseItem->getContentSize().width - (pauseItem->getContentSize().width / 4) + origin.x,
                                 visibleSize.height - pauseItem->getContentSize().height + (pauseItem->getContentSize().width / 4) + origin.y));

    auto menu = Menu::create(pauseItem, NULL);
    menu->setPosition(Point::ZERO);
    this->addChild(menu);

    for (int i = 0; i < 2; i ++)
    {
        backgroundSpriteArray[i] = Sprite::create("GameScreen/Game_Screen_Background.png");
        backgroundSpriteArray[i]->setPosition(Point((visibleSize.width / 2), (-1 * visibleSize.height * i) + (visibleSize.height / 2)));
        this->addChild(backgroundSpriteArray[i], -2);
    }

    playerSprite = Sprite::create("GameScreen/Space_Pod.png");
    playerSprite->setPosition(Point(visibleSize.width / 2, pauseItem->getPosition().y - (pauseItem->getContentSize().height / 2) - (playerSprite->
        getContentSize().height / 2)));
    this->addChild(playerSprite, -1);

    this->scheduleUpdate();

    this->schedule(schedule_selector(GameScreen::spawnAsteroid), 1.0);

    return true;
}
```

The following screenshot shows how the `update()` and `spawnAsteroid()` methods inside `GameScene.cpp` should look after all the code has been added by following the previous steps in order to add the asteroids randomly to provide a gameplay element to the game:

```
void GameScreen::update(float dt)
{
    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    for (int i = 0; i < 2; i ++)
    {
        if (backgroundSpriteArray[i]->getPosition().y >= visibleSize.height + (visibleSize.height / 2) -1)
        {
            backgroundSpriteArray[i]->setPosition(Point((visibleSize.width / 2) + origin.x, (-1 * visibleSize.height) + (visibleSize.height / 2)));
        }
    }

    for (int i = 0; i < 2; i ++)
    {
        backgroundSpriteArray[i]->setPosition(Point(backgroundSpriteArray[i]->getPosition().x, backgroundSpriteArray[i]->getPosition().y + (0.75 *
            visibleSize.height * dt)));
    }

    for (int i = 0; i < asteroids.size(); i++)
    {
        asteroids[i]->setPosition(Point(asteroids[i]->getPosition().x, asteroids[i]->getPosition().y + (0.75 * visibleSize.height * dt)));

        if (asteroids[i]->getPosition().y > visibleSize.height * 2)
        {
            this->removeChild(asteroids[i]);
            asteroids.erase(asteroids.begin() + i);
        }
    }
}

void GameScreen::spawnAsteroid(float dt)
{
    Size visibleSize = Director::getInstance()->getVisibleSize();
    Point origin = Director::getInstance()->getVisibleOrigin();

    int asteroidIndex = (arc4random() % 3) + 1;
    __String* asteroidString = __String::createWithFormat("GameScreen/Asteroid_%i.png", asteroidIndex);

    Sprite *tempAsteroid = Sprite::create(asteroidString->getCString());

    int xRandomPosition = (arc4random() % (int)(visibleSize.width - (tempAsteroid->getContentSize().width / 2))) + (tempAsteroid->getContentSize().width
        / 2);

    tempAsteroid->setPosition(Point(xRandomPosition + origin.x, -tempAsteroid->getContentSize().height + origin.y));
    asteroids.push_back(tempAsteroid);
    this->addChild(asteroids[asteroids.size() - 1], -1);
}
```

In this tutorial, although speed was implemented as a magic number for general development purposes, this should be avoided and `#defines` or variables should be used.

Finally, this is how our GUI will look:

Summary

This chapter covered sprites, which allow game objects to be added and drawn within a scene. The sprites added in this chapter will serve as the basis for the gameplay that will be implemented over the next few chapters. The next chapter will cover sprite actions that allow sprites to be manipulated, for example, rotation and scaling, all of which are provided via Cocos2d-x.

6
Implementing Actions

This chapter will cover the actions that allow the modification of node properties, for example, rotating a sprite or moving a menu item.

The contents of this chapter will not be used for the game being developed through the course of this book, but it's important to have an understanding of these features.

As a personal task, you can figure out how to implement the actions from this chapter to enhance the game's experience. Here is a start: rotate the asteroids as they move. This chapter can be thought of as a general-purpose reference guide with no additional dependencies, but the features that are learned could be implemented as a separate task in the game to make it more interesting.

The topics covered in this chapter are as follows:

- Actions to manipulate nodes
- Repeating actions
- Sequencing actions
- Animating actions

More information regarding any of the actions covered in this chapter can be found at `http://www.cocos2d-x.org/wiki/Reference`.

Actions

Actions allow Cocos2d-x's nodes to be manipulated, such as scaling a sprite. These actions can be performed once or several times and can be sequenced one after another. Any object that is a node can use the actions that will be covered throughout this chapter.

The general code structure for running an action is as follows:

```
auto action = actionName::create(actionProperties);
nodeName->runAction(actionName);
```

Moving

The movement action, as the name suggests, allows a node to move around the scene over a set period of time that can be specified. There are two types of movement actions:

- `MoveBy`: This action moves the specified node relative to its current position. For example, assume that the node is at (400, 500) and the `MoveBy` action is applied with (30, 40) with a duration of 2 seconds; the node will then travel to (430, 540) from its original position of (400, 500) over a period of 2 seconds in a linear manner.

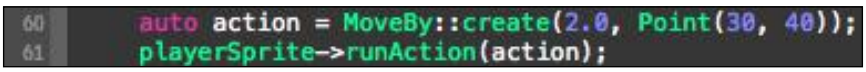
  The following is the syntax:

  ```
  auto action = MoveBy::create(duration, Point
  (xPosition, yPosition));
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = MoveBy::create(2.0, Point(30, 40));
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:

- `MoveTo`: This action moves the specified node relative to the specified position from its current position. Take this simple example: the node is at (400, 500) and the `MoveTo` action is applied with (30, 40) with a duration of 2 seconds; the node will travel to (30, 40) from its original position of (400, 500) over a period of 2 seconds in a linear manner.

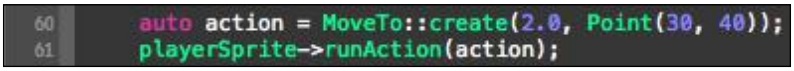
The following is the syntax:

```
auto action = MoveTo::create(duration, Point
(xPosition, yPosition));
playerSprite->runAction(action);
```

The following is the example's code:

```
auto action = MoveTo::create(2.0, Point(30, 40));
playerSprite->runAction(action);
```

The following is the example's screenshot:



This section covered the use of the movement actions, with some simple examples, and the process of implementing them within a game in order to allow the nodes to move around the scene.

Jumping

The jumping action, as the name suggests, allows a node to jump around the scene over a set period of time that can be specified. There are two types of jumping actions:

- `JumpBy`: This action makes the specified node jump relative to its current position. Take this example: the node is at (400, 500) and the `JumpBy` action is applied with (60, 70) with a duration of 2 seconds; jumping five times, with each jump reaching a height of 45 pixels, the node will jump to (460, 570) from its original position of (400, 500) over a period of 2 seconds in a linear manner.

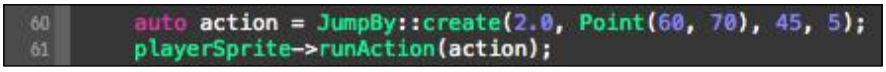
  The following is the syntax:

  ```
  auto action = JumpBy::create(duration, Point
  (xPosition, yPosition), heightOfEachJump, numberOfJumps);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = JumpBy::create(2.0, Point(60, 70), 45, 5);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:



- `JumpTo`: This action makes the specified node jump to a specified position from its current position. For instance, the node is at (400, 500) and the `JumpTo` action is applied with (60, 70) with a duration of 2 seconds; jumping 5 times, with each jump reaching a height of 45 pixels, the node will jump to (60, 70) from its original position of (400, 500) over a period of 2 seconds in a linear manner.

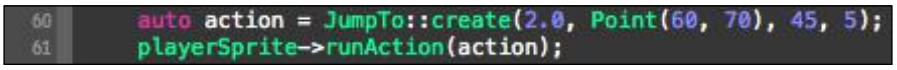
  The following is the syntax:

```
auto action = JumpTo::create(duration, Point
(xPosition, yPosition), heightOfEachJump, numberOfJumps);
playerSprite->runAction(action);
```

  The following is the example's code:

```
auto action = JumpTo::create(2.0, Point(60, 70), 45, 5);
playerSprite->runAction(action);
```

  The following is the example's screenshot:



This section covered the use of the jumping actions, with some simple examples, and the process of implementing them within a game in order to allow nodes to jump around the scene.

Bezier actions

The `Bezier` action allows a node to curve around points in a Bezier manner in the scene over a set period of time that can be specified. There are two types of Bezier actions:

- `BezierBy`: This action makes the specified node curve around points relative to its current position. Take this simple example: the node is at (400, 500) and the `BezierBy` action is applied with an end position of (60, 70) with a duration of 2 seconds around the (0, 300) and (300, 200) points to finish at (460, 570) from its original position of (400, 500) over a period of 2 seconds in a linear manner.

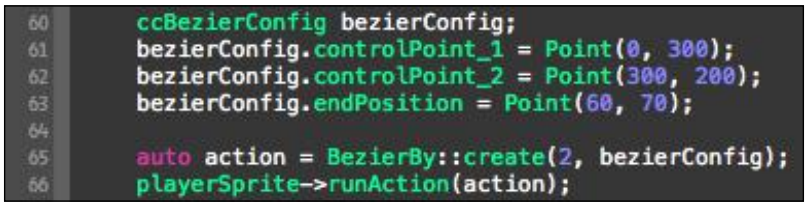
  The following is the syntax:

```
ccBezierConfig configName;
configName.controlPoint_1 = Point(xPosition, yPosition);
configName.controlPoint_2 = Point(xPosition, yPosition);
configName.endPosition = Point(xPosition, yPosition);
auto action = BezierBy::create(duration, configName);
playerSprite->runAction(action);
```

The following is the example's code:

```
ccBezierConfig bezierConfig;
bezierConfig.controlPoint_1 = Point(0, 300);
bezierConfig.controlPoint_2 = Point(300, 200);
bezierConfig.endPosition = Point(60, 70);
auto action = BezierBy::create(2, bezierConfig);
playerSprite->runAction(action);
```

The following is the example's screenshot:



- `BezierTo`: This action makes the specified node curve around points to end up at the specified position. Take this simple example: the node is at (400, 500) and the `BezierTo` action is applied with an end position of (60, 70) with a duration of 2 seconds around the (0, 300) and (300, 200) points to finish at (60, 70) from its original position of (400, 500) over a period of 2 seconds in a linear manner.

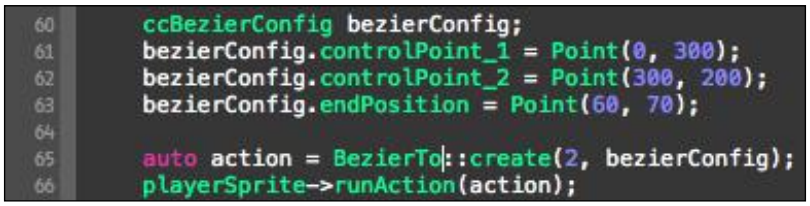
  The following is the syntax:

```
ccBezierConfig configName;
configName.controlPoint_1 = Point(xPosition, yPosition);
configName.controlPoint_2 = Point(xPosition, yPosition);
configName.endPosition = Point(xPosition, yPosition);
auto action = BezierTo::create(duration, configName);
playerSprite->runAction(action);
```

  The following is the example's code:

```
ccBezierConfig bezierConfig;
bezierConfig.controlPoint_1 = Point(0, 300);
bezierConfig.controlPoint_2 = Point(300, 200);
bezierConfig.endPosition = Point(60, 70);
auto action = BezierTo::create(2, bezierConfig);
playerSprite->runAction(action);
```

The following is the example's screenshot:



```
60      ccBezierConfig bezierConfig;
61      bezierConfig.controlPoint_1 = Point(0, 300);
62      bezierConfig.controlPoint_2 = Point(300, 200);
63      bezierConfig.endPosition = Point(60, 70);
64
65      auto action = BezierTo::create(2, bezierConfig);
66      playerSprite->runAction(action);
```

This section covered the use of the Bezier actions, with some simple examples, and the process of implementing them within a game to curve nodes around points.

Placing

The place action allows a node to be placed at a specific point within the scene. There is a single place action—`Place`. This action allows you to specify where the node should be placed and places it at that location. For example, a node needs to be placed at (40, 50) within the scene.

The following is the syntax:

```
auto action = Place::create(Point(xPosition, yPosition));
playerSprite->runAction(action);
```

This example can be accomplished using the following code:

```
auto action = Place::create(Point(40, 50));
playerSprite->runAction(action);
```

The following is the example's screenshot:

```
60      auto action = Place::create(Point(40, 50));
61      playerSprite->runAction(action);
```

This section covered the use of the `Place` action, which allows nodes to be instantly positioned to a set location, essentially the same as setting the node's position.

Scaling

The scaling action, as the name suggests, allows a node to be scaled, essentially resizing it over a set period of time that can be specified, from its current size. There are two types of scaling actions as follows:

- `ScaleBy`: This action scales the specified node relative to its current size. Take this simple example: if the node is at size (100, 200) and the `ScaleBy` action is applied with (2, 3) with a duration of 2 seconds, the node will scale to (200, 600) from its original size of (100, 200) over a period of 2 seconds in a linear manner:

  The following is the syntax:

  ```
  auto action = ScaleBy::create(duration, xScale, yScale);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = ScaleBy::create(2, 2, 3);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:

- `ScaleTo`: This action scales the specified node relative to its original size. For example, if the node is at size (100, 200) and the `ScaleTo` action is applied with (2, 3) with a duration of 2 seconds, the node will scale to (200, 600) from its original size of (100, 200) over a period of 2 seconds in a linear manner. If the action is applied again, node's size will not change, as the `ScaleTo` method scales relative to the node's original size and not the current size:

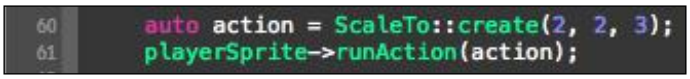
  The following is the syntax

  ```
  auto action = ScaleTo::create(duration, xScale, yScale);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = ScaleTo::create(2, 2, 3);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:

If the `ScaleBy` method was applied several times, it would scale the node each time. However, the `ScaleTo` method wouldn't as this method is relative to the node's original size, so applying the same scale over and over again wouldn't change the node's scale unless it had been explicitly changed via another action.

This section covered the use of the scaling actions, with some simple examples, and the process of implementing them within a game to allow nodes to be scaled.

Rotation

The rotation action, as the name suggests, allows a node to be rotated over a set period of time that can be specified. There are two types of rotation actions:

- `RotateBy`: This action rotates the specified node relative to its current rotation. Take this simple example: if the node is rotated by 45 degrees and the `RotateBy` action is applied with an angle of 90 degrees and a duration of 2 seconds, the node will rotate by 135 degrees from its original rotation of 45 degrees over a period of 2 seconds in a linear manner.

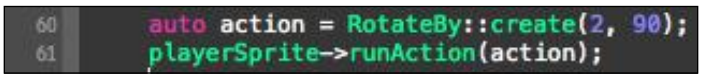
  The following is the syntax:

  ```
  auto action = RotateBy::create(duration, rotationAngle);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = RotateBy::create(2, 90);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:

- `RotateTo`: This action rotates the specified node relative to its original rotation. Take this simple example: the node is rotated by 45 degrees and the `RotateTo` action is applied with an angle of 90 degrees and a duration of 2 seconds. The node will rotate to 90 degrees from its original rotation of 45 degrees over a period of 2 seconds in a linear manner.

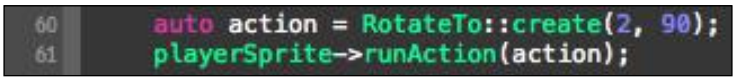
  The following is the syntax:

  ```
  auto action = RotateTo::create(duration, rotationAngle);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = RotateTo::create(2, 90);
  playerSprite->runAction(action);
  ```

The following is the example's screenshot:



If the `RotateBy` method was applied several times, it would rotate the node each time. However, the `RotateTo` method wouldn't, as this method is relative to the node's original rotation, so applying the same rotation over and over again wouldn't change the node's rotation unless it had been explicitly changed via another action.

This section covered the use of the rotation actions, with some simple examples, and the process of implementing them within a game to allow nodes to be rotated.

Tinting

The tinting action, as the name suggests, allows a node's color to be tinted over a set period of time that can be specified. There are two types of tinting actions:

- `TintBy`: This action tints the specified node relative to its current tint color. For instance, the node has a tint value of (10, 10, 10) RGB color, and the `TintBy` action is applied with the color (10, 20, -5) for a duration of 2 seconds. The node will tint to (20, 30, 5) from its current tint of (10, 10, 10) over a period of 2 seconds in a linear manner.

  The following is the syntax:

  ```
  auto action = TintBy::create(duration, redValue,
  greenValue, blueValue);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = TintBy::create(2, 10, 20, -5);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:

- `TintTo`: This action tints the specified node relative to its original tint color. For example, the node has a tint value of (10, 10, 10) RGB color, with an original tint value of (0, 0, 0), and the `TintTo` action is applied with the color (10, 20, 5) and a duration of 2 seconds. The node will tint to (10, 20, 5) from its current tint of (10, 10, 10) over a period of 2 seconds in a linear manner.

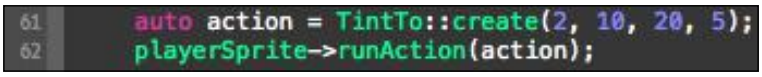
  The following is the syntax:

```
auto action = TintTo::create(duration, redValue,
greenValue, blueValue);
playerSprite->runAction(action);
```

  The following is the example's code:

```
auto action = TintTo::create(2, 10, 20, 5);
playerSprite->runAction(action);
```

  The following is the example's screenshot:



If the `TintBy` method was applied several times, it would tint the node each time. However, the `TintTo` method wouldn't, as this method is relative to the node's original tint, so applying the same tint over and over again wouldn't change the node's tint unless it had been changed explicitly via another action.

This section covered the use of the tinting actions, with some simple examples, and the process of implementing them within a game to allow nodes to be tinted.

Fading

The fading action, as the name suggests, allows a node's opacity to be changed over a set period of time that can be specified. There are three types of fading actions:

- `FadeTo`: This action fades the specified node relative to its original opacity, which is 255 (fully visible, with 0 being invisible). Take this simple example: the node has a `FadeTo` action applied with the opacity value of 100 and a duration of 2 seconds. The node will fade to an opacity value of 100 from its original opacity value of 255 over a period of 2 seconds in a linear manner.

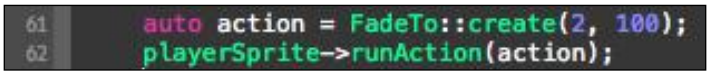
  The following is the syntax:

```
auto action = FadeTo::create(duration, opacity);
playerSprite->runAction(action);
```

The following is the example's code:

```
auto action = FadeTo::create(2, 100);
playerSprite->runAction(action);
```

The following is the example's screenshot:



- `FadeIn`: This action fades the specified node to full opacity from its current opacity; this can be used to introduce nodes within a scene. For example, the node with a current opacity value of 200 has a `FadeIn` action applied with a duration of 2 seconds. The node will fade to an opacity value of 255 (fully visible) from its current opacity of 200 over a period of 2 seconds in a linear manner.

  The following is the syntax:

  ```
  auto action = FadeIn::create(duration);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = FadeIn::create(2);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:

- `FadeOut`: This action fades the specified node to an opacity of 0 (no longer visible) from its current opacity; this can be used to remove nodes from a scene. For instance, the node with a current opacity value of 200 has a `FadeOut` action applied with a duration of 2 seconds. The node will fade to an opacity value of 0 (no longer visible) from its current opacity value of 200 over a period of 2 seconds in a linear manner.

  The following is the syntax:

  ```
  auto action = FadeOut::create(duration);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = FadeOut::create(2);
  playerSprite->runAction(action);
  ```

The following is the example's screenshot:

This section covered the use of the fading actions, with some simple examples, and the process of implementing them within a game to allow nodes to be faded.

Skewing

The skewing action allows a node to be skewed by affecting its *x* and *y* skew properties over a set period of time that can be specified. There are two types of skewing actions:

- `SkewBy`: This action skews the specified node relative to its current *x* and *y* skew properties. Take this simple example: the node is at a skew value of (2, 1) and the `SkewBy` action is applied with the skew values of (2, 4) and a duration of 2 seconds. The node will skew to (4, 4) from its current skew of (2, 1) over a period of 2 seconds in a linear manner.

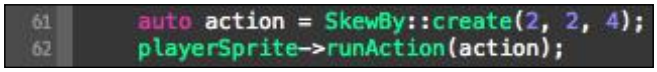
  The following is the syntax:

  ```
  auto action = SkewBy::create(duration, skewX, skewY);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto action = SkewBy::create(2, 2, 4);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:

- `SkewTo`: This action skews the specified node relative to its original *x* and *y* skew properties. Take this simple example: the node is at a skew value of (2, 1) and the `SkewTo` action is applied with the skew values of (2, 4) and a duration of 2 seconds. The node will skew to (2, 4) from its current skew of (2, 1) over a period of 2 seconds in a linear manner.

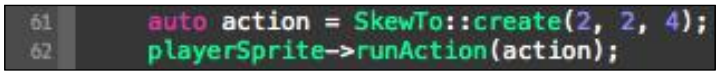
  The following is the syntax:

  ```
  auto action = SkewTo::create(duration, skewX, skewY);
  playerSprite->runAction(action);
  ```

The following is the example's code:

```
auto action = SkewTo::create(2, 2, 4);
playerSprite->runAction(action);
```

The following is the example's screenshot:



If the `SkewBy` method was applied several times, it would skew the node each time. However, the `SkewTo` method wouldn't, as this method is relative to a node's original skew properties, so applying the same skew over and over again wouldn't change the node's skew unless it had been changed explicitly via another action.

This section covered the use of the skewing actions, with some simple examples, and the process of implementing them within a game to allow nodes to be skewed.

Repeating

The repeat action, as the name suggests, allows a single action to be repeated. There are two types of repeat actions:

- `Repeat`: This action repeats an action for a set number of times which is specified. For example, a node at a rotation of 10 degrees has a `Repeat` action applied, which consists of a `RotateBy(2, 45)` action and is set to be repeated three times; after 6 seconds, the node will be at a rotation of 145 degrees:

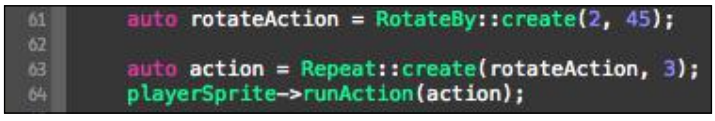
  The following is the syntax:

  ```
  auto action = Repeat::create(actionToRepeat,
  numberOfTimesToRepeat);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto rotateAction = RotateBy::create(2, 45);
  auto action = Repeat::create(rotateAction, 3);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:



- `RepeatForever`: This action repeats an action forever. Take this simple example: a node at a rotation of 10 degrees has a `RepeatForever` action applied, which consists of a `RotateBy(2, 45)` action; the node will rotate 45 degrees every 2 seconds:

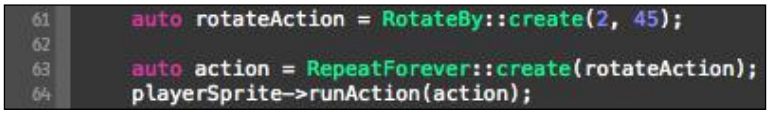
  The following is the syntax:

  ```
  auto action = RepeatForever::create(actionToRepeat);
  playerSprite->runAction(action);
  ```

  The following is the example's code:

  ```
  auto rotateAction = RotateBy::create(2, 45);
  auto action = RepeatForever::create(rotateAction);
  playerSprite->runAction(action);
  ```

  The following is the example's screenshot:



This section covered the use of the repeat actions, with some simple examples, and the process of implementing them within a game to allow node actions to be repeated.

Sequencing

If you were to run multiple actions one after the other, they would all start at the same time; this, in certain circumstances, may produce the desired effect. But in certain instances, the need arises for an action to be performed only once the previous action has finished, for example, moving a sprite to a specified location and then scaling it to fit its container. This can be achieved using the `Sequencing` action, which essentially contains any combination of the previous actions covered. There is a single sequence action as follows:

- `Sequence`: This action performs each action specified one after another, after the previous one has finished. Take this simple example: a node has a sequence action applied, which consists of a rotate action and a movement action. First, the sprite will rotate to the desired angle over the set time period and then move to the set location over the set period of time. The crucial aspect is that an action can only start if the previous action has finished.

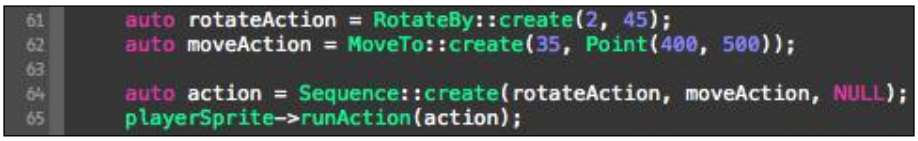
The following is the syntax:

```
auto action = Sequence::create(action1, action2, NULL);
playerSprite->runAction(action);
```

The following is the example's code:

```
auto rotateAction = RotateBy::create(2, 45);
auto moveAction = MoveTo::create(35, Point(400, 500));
auto action = Sequence::create(rotateAction,
moveAction, NULL);
playerSprite->runAction(action);
```

The following is the example's screenshot:



This section covered how to sequence actions one after another to allow multiple actions to be applied, but only one of them is run at any given time.

Animation

Until now, all the actions that have been applied have been done so in a linear fashion. For example, when a `MoveBy` action was applied, it would move at a constant speed throughout; this could be overcome by sequencing several movement actions to provide the illusion of momentum, for instance. However, this is very inefficient and extremely tedious. Cocos2d-x provides an immense number of animations, all of which can be found in Cocos2d-x's API guide at `http://www.cocos2d-x.org/wiki/Reference`. This section will cover a single example, which will be a great basis for moving forward:

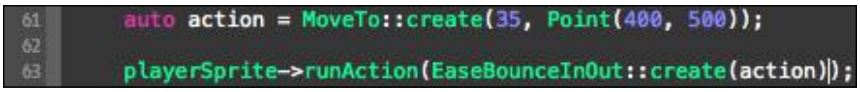
The following is the syntax:

```
auto action = ActionName::create(actionProperties);
playerSprite->runAction(Animation(action));
```

The following is the example's code:

```
auto action = MoveTo::create(35, Point(400, 500));
playerSprite->runAction(EaseBounceInOut::create(action));
```

The following is the example's screenshot:



```
61      auto action = MoveTo::create(35, Point(400, 500));
62
63      playerSprite->runAction(EaseBounceInOut::create(action));
```

The preceding example will move the node, but it will bounce at the start and at the end of the action. These animations can be useful to provide the illusion of gravity or momentum, for example. This section covered what animations are in relation to actions and how they can be implemented.

Summary

This chapter covered a lot of in-depth topics around the concept of the actions that can be applied to nodes. These actions allow the properties of a node to be manipulated; although these were not implemented within the game, they are still important. Here is a little task: implement some of the actions in the game, such as making the asteroids rotate or even scale. Experiment and see what can be achieved using these actions. Until now, the game has no gameplay interaction, but in the next chapter, touch controls will be implemented to provide the user with the functionality to move the space pod.

7
Moving the Space Pod Using Touch

This chapter will cover how to set up touch events within our game. So far, the game has had no user interaction from a gameplay perspective. This chapter will rectify this by adding touch controls in order to move the space pod and avoid the asteroids.

The topics that will be covered in this chapter are as follows:

- Implementing touch
    - Single-touch
    - Multi-touch
- Using touch locations
- Moving the spaceship when touching the screen

There are two main routes to detect touch provided by Cocos2d-x:

- **Single-touch**: This method detects a single-touch event at any given time, which is what will be implemented in the game as it is sufficient for most gaming circumstances
- **Multi-touch**: This method provides the functionality that detects multiple touches simultaneously; this is great for pinching and zooming; for example, the Angry Birds game uses this technique

Though a single-touch will be the approach that the game will incorporate, multi-touch will also be covered in this chapter so that you are aware of how to use this in future games.

The general process for setting up touches

The general process of setting up touch events, be it single or multi-touch, is as follows:

1. Declare the touch functions.
2. Declare a listener to listen for touch events.
3. Assign touch functions to appropriate touch events as follows:
    - When the touch has begun
    - When the touch has moved
    - When the touch has ended
4. Implement touch functions.
5. Add appropriate game logic/code for when touch events have occurred.

Single-touch events

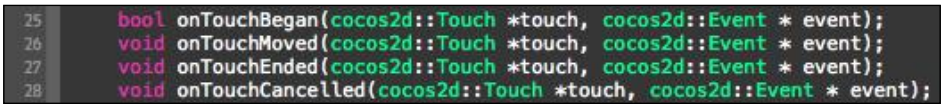
Single-touch events can be detected at any given time, and for many games this is sufficient as it is for this game.

Follow these steps to implement single-touch events into a scene:

1. Declare touch functions in the `GameScene.h` file as follows:

```
bool onTouchBegan(cocos2d::Touch *touch, cocos2d::
Event * event);
void onTouchMoved(cocos2d::Touch *touch, cocos2d::
Event * event);
void onTouchEnded(cocos2d::Touch *touch, cocos2d::
Event * event);
void onTouchCancelled(cocos2d::Touch *touch, cocos2d::
Event * event);
```

   This is what the `GameScene.h` file will look like:



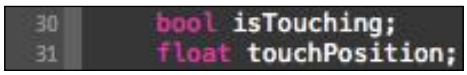
The previous functions do the following:

- The `onTouchBegan` function detects when a single-touch has occurred, and it returns a Boolean value. This should be `true` if the event is swallowed by the node and `false` indicates that it will keep on propagating.
- The `onTouchMoved` function detects when the touch moves.
- The `onTouchEnded` function detects when the touch event has ended, essentially when the user has lifted up their finger.
- The `onTouchCancelled` function detects when a touch event has ended but not by the user; for example, a system alert. The general practice is to call the `onTouchEnded` method to run the same code, as it can be considered the same event for most games.

2. Declare a Boolean variable in the `GameScene.h` file, which will be `true` if the screen is being touched and `false` if it isn't, and also declare a float variable to keep track of the position being touched:

```
bool isTouching;
float touchPosition;
```

This is how it will look in the `GameScene.h` file:



3. Add the following code in the `init()` method of `GameScene.cpp`:

```
auto listener = EventListenerTouchOneByOne::create();
listener->setSwallowTouches(true);
listener->onTouchBegan = CC_CALLBACK_2
(GameScreen::onTouchBegan, this);
listener->onTouchMoved = CC_CALLBACK_2
(GameScreen::onTouchMoved, this);
listener->onTouchEnded = CC_CALLBACK_2
(GameScreen::onTouchEnded, this);
listener->onTouchCancelled = CC_CALLBACK_2
(GameScreen::onTouchCancelled, this);
this->getEventDispatcher()-
>addEventListenerWithSceneGraphPriority(listener, this);
isTouching = false;
touchPosition = 0;
```

This is how it will look in the `GameScene.cpp` file:

```
60      auto listener = EventListenerTouchOneByOne::create();
61      listener->setSwallowTouches(true);
62
63      listener->onTouchBegan = CC_CALLBACK_2(GameScreen::onTouchBegan, this);
64      listener->onTouchMoved = CC_CALLBACK_2(GameScreen::onTouchMoved, this);
65      listener->onTouchEnded = CC_CALLBACK_2(GameScreen::onTouchEnded, this);
66      listener->onTouchCancelled = CC_CALLBACK_2(GameScreen::onTouchCancelled, this);
67
68      this->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, this);
69
70      isTouching = false;
71      touchPosition = 0;
```

4. There is quite a lot of new code in the previous code snippet, so let's run through it line by line:
    - The first statement declares and initializes a listener for a single-touch
    - The second statement prevents layers underneath from where the touch occurred by detecting the touches
    - The third statement assigns our `onTouchBegan` method to the `onTouchBegan` listener
    - The fourth statement assigns our `onTouchMoved` method to the `onTouchMoved` listener
    - The fifth statement assigns our `onTouchEnded` method to the `onTouchEnded` listener
    - The sixth statement assigns our `onTouchCancelled` method to the `onTouchCancelled` listener
    - The seventh statement sets the touch listener to the event dispatcher so the events can be detected
    - The eighth statement sets the `isTouching` variable to `false` as the player won't be touching the screen initially when the game starts
    - The final statement initializes the `touchPosition` variable to `0`
5. Implement the touch functions inside the `GameScene.cpp` file:

```
bool GameScreen::onTouchBegan(cocos2d::Touch *touch,
cocos2d::Event * event)
{
    isTouching = true;
    touchPosition = touch->getLocation().x;

    return true;
}

void GameScreen::onTouchMoved(cocos2d::Touch *touch,
```

```
cocos2d::Event * event)
{
    // not used for this game
}

void GameScreen::onTouchEnded(cocos2d::Touch *touch,
cocos2d::Event * event)
{
    isTouching = false;
}

void GameScreen::onTouchCancelled(cocos2d::Touch *touch,
cocos2d::Event * event)
{
    onTouchEnded(touch, event);
}
```

The following is what the `GameScene.cpp` file will look like:

```
76  bool GameScreen::onTouchBegan(cocos2d::Touch *touch, cocos2d::Event * event)
77  {
78      isTouching = true;
79      touchPosition = touch->getLocation().x;
80
81      return true;
82  }
83
84  void GameScreen::onTouchMoved(cocos2d::Touch *touch, cocos2d::Event * event)
85  {
86      // not used for this game
87  }
88
89  void GameScreen::onTouchEnded(cocos2d::Touch *touch, cocos2d::Event * event)
90  {
91      isTouching = false;
92  }
93
94  void GameScreen::onTouchCancelled(cocos2d::Touch *touch, cocos2d::Event * event)
95  {
96      onTouchEnded(touch, event);
97  }
```

6. Let's go over the touch functions that have been implemented previously:
    - The `onTouchBegan` method will set the `isTouching` variable to `true` as the user is now touching the screen and is storing the starting touch position
    - The `onTouchMoved` function isn't used in this game but it has been implemented so that you are aware of the steps for implementing it (as an extra task, you can implement touch movement so that if the user moves his/her finger from one side to another direction, the space pod gets changed)

- The `onTouchEnded` method will set the `isTouching` variable to `false` as the user is no longer touching the screen
- The `onTouchCancelled` method will call the `onTouchEnded` method as a touch event has essentially ended

7. If the game were to be run, the space pod wouldn't move as the movement code hasn't been implemented yet. It will be implemented within the `update()` method to move left when the user touches in the left half of the screen and move right when user touches in the right half of the screen. Add the following code at the end of the `update()` method:

```
// check if the screen is being touched
if (true == isTouching)
{
    // check which half of the screen is being touched
    if (touchPosition < visibleSize.width / 2)
    {
      // move the space pod left
      playerSprite->setPosition().x(playerSprite-
      >getPosition().x - (0.50 * visibleSize.width * dt));

      // check to prevent the space pod from going off
the screen (left side)
      if (playerSprite->getPosition().x <= 0 +
      (playerSprite->getContentSize().width / 2))
      {
        playerSprite->setPositionX(playerSprite-
        >getContentSize().width / 2);
      }
    }
    else
    {
      // move the space pod right
      playerSprite->setPosition().x(playerSprite-
      >getPosition().x + (0.50 * visibleSize.width * dt));

      // check to prevent the space pod from going off the
      screen (right side)
      if (playerSprite->getPosition().x >=
      visibleSize.width - (playerSprite->
      getContentSize().width / 2))
      {
        playerSprite->setPositionX(visibleSize.width -
        (playerSprite->getContentSize().width / 2));
      }
    }
}
```

The following is how this will look after adding the code:

```
// check if the screen is being touched
if (true == isTouching)
{
    // check which half of the screen is being touched
    if (touchPosition < visibleSize.width / 2)
    {
        // move the space pod left
        playerSprite->setPositionX(playerSprite->getPosition().x - (0.50 * visibleSize.width * dt));

        // check to prevent the space pod from going off the screen (left side)
        if (playerSprite->getPosition().x <= 0 + (playerSprite->getContentSize().width / 2))
        {
            playerSprite->setPositionX(playerSprite->getContentSize().width / 2);
        }
    }
    else
    {
        // move the space pod right
        playerSprite->setPositionX(playerSprite->getPosition().x + (0.50 * visibleSize.width * dt));

        // check to prevent the space pod from going off the screen (right side)
        if (playerSprite->getPosition().x >= visibleSize.width - (playerSprite->getContentSize().width / 2))
        {
            playerSprite->setPositionX(visibleSize.width - (playerSprite->getContentSize().width / 2));
        }
    }
}
```

The preceding code performs the following steps:

1. Checks whether the screen is being touched.
2. Checks which side of the screen is being touched.
3. Moves the player left or right.
4. Checks whether the player is going off the screen and if so, stops him/her from moving.
5. Repeats the process until the screen is no longer being touched.

This section covered how to set up single-touch events and implement them within the game to be able to move the space pod left and right.

Multi-touch events

Multi-touch is set up in a similar manner of declaring the functions and creating a listener to actively listen out for touch events.

Follow these steps to implement multi-touch into a scene:

1. Firstly, the multi-touch feature needs to be enabled in the `AppController.mm` file, which is located within the `ios` folder. To do so, add the following code line below the `viewController.view = eaglView;` line:

```
[eaglView setMultipleTouchEnabled: YES];
```

The following is what the `AppController.mm` file will look like:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // Override point for customization after application launch.

    // Add the view controller's view to the window and display.
    window = [[UIWindow alloc] initWithFrame: [[UIScreen mainScreen] bounds]];

    // Init the CCEAGLView
    CCEAGLView *eaglView = [CCEAGLView viewWithFrame: [window bounds]
                                        pixelFormat: kEAGLColorFormatRGB565
                                        depthFormat: GL_DEPTH24_STENCIL8_OES
                                 preserveBackbuffer: NO
                                         sharegroup: nil
                                      multiSampling: NO
                                    numberOfSamples: 0];

    // Use RootViewController manage CCEAGLView
    _viewController = [[RootViewController alloc] initWithNibName:nil bundle:nil];
    _viewController.wantsFullScreenLayout = YES;
    _viewController.view = eaglView;
    [eaglView setMultipleTouchEnabled: YES];

    // Set RootViewController to window
    if ( [[UIDevice currentDevice].systemVersion floatValue] < 6.0)
    {
        // warning: addSubView doesn't work on iOS6
        [window addSubview: _viewController.view];
    }
    else
    {
        // use this method on ios6
        [window setRootViewController:_viewController];
    }

    [window makeKeyAndVisible];

    [[UIApplication sharedApplication] setStatusBarHidden:true];

    // IMPORTANT: Setting the GLView should be done after creating the RootViewController
    cocos2d::GLView *glview = cocos2d::GLView::createWithEAGLView(eaglView);
    cocos2d::Director::getInstance()->setOpenGLView(glview);

    cocos2d::Application::getInstance()->run();

    return YES;
}
```

2. Declare the touch functions within the game scene header file (the functions do the same thing as the single-touch equivalents but enable multiple touches that can be detected simultaneously):

```
void onTouchesBegan(const std::vector<cocos2d::Touch *>
&touches, cocos2d::Event *event);
void onTouchesMoved(const std::vector<cocos2d::Touch *>
&touches, cocos2d::Event *event);
void onTouchesEnded(const std::vector<cocos2d::Touch *>
&touches, cocos2d::Event *event);
void onTouchesCancelled(const std::vector<cocos2d::Touch *>
&touches, cocos2d::Event *event);
```

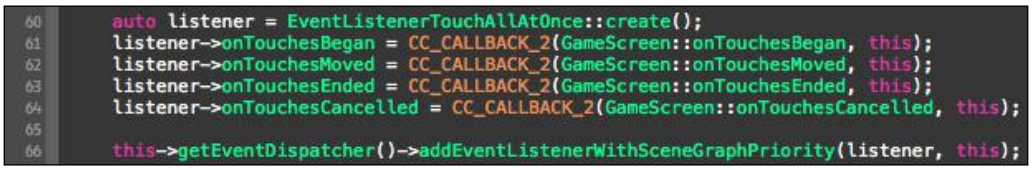
The following is what the header file will look like:

```
    void onTouchesBegan(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event);
    void onTouchesMoved(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event);
    void onTouchesEnded(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event);
    void onTouchesCancelled(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event);
```

3. Add the following code in the `init()` method of the `scene.cpp` file to listen to the multi-touch events that will use the `EventListenerTouchAllAtOnce` class, which allows multiple touches to be detected at once:

```
auto listener = EventListenerTouchAllAtOnce::create();
listener->onTouchesBegan = CC_CALLBACK_2
(GameScreen::onTouchesBegan, this);
listener->onTouchesMoved = CC_CALLBACK_2
(GameScreen::onTouchesMoved, this);
listener->onTouchesEnded = CC_CALLBACK_2
(GameScreen::onTouchesEnded, this);
listener->onTouchesCancelled = CC_CALLBACK_2
(GameScreen::onTouchesCancelled, this);
this->getEventDispatcher()-
>addEventListenerWithSceneGraphPriority(listener, this);
```

The following is how this will look:



4. Implement the following multi-touch functions inside the `scene.cpp`:

```
void GameScreen::onTouchesBegan(const std::
vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    CCLOG("Multi-touch BEGAN");
}

void GameScreen::onTouchesMoved(const std::
vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    for (int i = 0; i < touches.size(); i++)
    {
        CCLOG("Touch %i:  %f", i, touches[i]-
        >getLocation().x);
    }
}

void GameScreen::onTouchesEnded(const std::
vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    CCLOG("MULTI TOUCHES HAVE ENDED");
}
```

```
void GameScreen::onTouchesCancelled(const std::
vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    CCLOG("MULTI TOUCHES HAVE BEEN CANCELLED");
}
```

The following is how this will look:

```
void GameScreen::onTouchesBegan(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    CCLOG("MULTI TOUCH BEGAN");
}

void GameScreen::onTouchesMoved(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    for (int i = 0; i < touches.size(); i++)
    {
        CCLOG("Touch %i:  %f", i, touches[i]->getLocation().x);
    }
}

void GameScreen::onTouchesEnded(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    CCLOG("MULTI TOUCHES HAVE ENDED");
}

void GameScreen::onTouchesCancelled(const std::vector<cocos2d::Touch *> &touches, cocos2d::Event *event)
{
    CCLOG("MULTI TOUCHES HAVE BEEN CANCELLED");
}
```

5. The multi-touch functions just print out a log, stating that they have occurred, but when touches are moved, their respective *x* positions are logged.

This section covered how to implement core foundations for multi-touch events so that they can be used for features such as zooming (for example, zooming into a scene in the *Clash Of Clans* game) and panning. Multi-touch wasn't incorporated within the game as it wasn't needed, but this section is a good starting point to implement it in future games.

Summary

This chapter covered how to set up touch listeners to detect touch events for single-touch and multi-touch. We incorporated single-touch within the game to be able to move the space pod left or right, depending on which half of the screen was being touched. Multi-touch wasn't used as the game didn't require it, but its implementation was shown so that it can be used for future projects. The next chapter will cover how to implement collision detection to add the final gameplay element so that the asteroids can collide with the player.

8
Collision Detection

This chapter will cover how Cocos2d-x handles collision detection. This will be used to detect collisions between the space pod and asteroids. There are several methods that can be used but fortunately, over the years, Chukong Technologies has refined Cocos2d-x to incorporate a great physics engine. This is built on top of the existing and popular **Chipmunk engine** while simplifying its implementation, but the extensively used physics engine **Box2d** can also be used with Cocos2d-x.

Topics covered in this chapter are as follows:

- What is collision detection?
- Setting up collision detection
- Implementing collision detection

Collision detection

Before implementing collision detection in our game, let's go over what collision detection actually is. Collision detection is the process of determining whether two or more objects have collided using mathematical computations. Fortunately, we won't have to implement this manually.

Player collision detection

The only collision detection that needs to be performed is between the players, the space pod, and the moving asteroids that the player has to avoid. If the player collides with an asteroid, then the game will transition to the Game Over scene.

Setting up collision detection

In the `GameScene.h` file, add the following lines:

```
void setPhysicsWorld(cocos2d::PhysicsWorld* world)
{
  mWorld = world;
  mWorld->setGravity(cocos2d::Vect(0, 0));
}
bool onContactBegin(cocos2d::PhysicsContact& contact);
cocos2d::PhysicsWorld* mWorld;
```

The first statement declares and initializes the physics world method that assigns the physics world and sets the gravity to `0` because the game doesn't require gravity as it's in space. So, the physics engine will simply be used for collision detection.

The second statement declares the `onContactBegin()` method that will be called when a collision has taken place.

The third statement creates the local physics world.

The `GameScene.h` file should look like the following screenshot once the preceding code has been added to the header:

```
#ifndef __GAME_SCENE_H__
#define __GAME_SCENE_H__

#include "cocos2d.h"

class GameScreen : public cocos2d::Layer
{
public:
    // there's no 'id' in cpp, so we recommend returning the class instance pointer
    static cocos2d::Scene* createScene();

    // Here's a difference. Method 'init' in cocos2d-x returns bool, instead of returning 'id' in cocos2d-iphone
    virtual bool init();

    // implement the "static create()" method manually
    CREATE_FUNC(GameScreen);

    void GoToPauseScene(Ref *pSender);
    void GoToGameOverScene(Ref *pSender);

    void update(float dt);

    void spawnAsteroid(float dt);

    bool onTouchBegan(cocos2d::Touch *touch, cocos2d::Event * event);
    void onTouchMoved(cocos2d::Touch *touch, cocos2d::Event * event);
    void onTouchEnded(cocos2d::Touch *touch, cocos2d::Event * event);
    void onTouchCancelled(cocos2d::Touch *touch, cocos2d::Event * event);

    bool isTouching;
    float touchPosition;

    void setPhysicsWorld(cocos2d::PhysicsWorld* world){mWorld = world; mWorld->setGravity(cocos2d::Vect(0, 0));}
    bool onContactBegin(cocos2d::PhysicsContact& contact);

    std::vector<cocos2d::Sprite *> asteroids;

    cocos2d::Sprite *backgroundSpriteArray[2];
    cocos2d::Sprite *playerSprite;

private:
    cocos2d::PhysicsWorld* mWorld;

};

#endif // __GAME_SCENE_H__
```

Implementing the collision detection

The next step is to implement the physics into the `GameScene.cpp` file using the following steps:

1. Modify the `init()` method as follows:
    - Change `auto scene = Scene::create();` to `auto scene = Scene::createWithPhysics();`
    - Add `layer->setPhysicsWorld(scene->getPhysicsWorld());` below the layer declaration

    The following screenshot is how the `GameScene.cpp` file will look after these modifications:

```
Scene* GameScreen::createScene()
{
    // 'scene' is an autorelease object
    auto scene = Scene::createWithPhysics();

    // 'layer' is an autorelease object
    auto layer = GameScreen::create();
    layer->setPhysicsWorld(scene->getPhysicsWorld());

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

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

2. Add the following code lines after the player sprite initialization inside the `init()` method:

```
auto body = PhysicsBody::createCircle
(playerSprite->getContentSize().width / 2);
body->setContactTestBitmask(true);
body->setDynamic(true);
playerSprite->setPhysicsBody(body);
```

    The following screenshot is what the updated `GameScene.cpp` file will look like:

```
    playerSprite = Sprite::create("GameScreen/Space_Pod.png");
    auto body = PhysicsBody::createCircle(playerSprite->getContentSize().width / 2);
    body->setContactTestBitmask(true);
    body->setDynamic(true);
    playerSprite->setPhysicsBody(body);
    playerSprite->setPosition(Point(visibleSize.width / 2,
                                    pauseItem->getPosition().y - (pauseItem->getContentSize().height / 2)
                                    - (playerSprite->getContentSize().height / 2)));
    this->addChild(playerSprite, -1);
```

Let's go over the preceding code lines that have just been added:

- The first statement creates a physics body for the space pod, which will be a circle for the purpose of this game
- The second statement sets the body up for collision detection
- The final statement assigns the body to the player's sprite

3. Add the following code lines before the `return` statement inside the `init()` method:

```
auto contactListener =
EventListenerPhysicsContact::create();
contactListener->onContactBegin =
CC_CALLBACK_1(GameScreen::onContactBegin, this);
this->getEventDispatcher()->
addEventListenerWithSceneGraphPriority(contactListener, this);
```

The following screenshot is how this will look in the file:

```
80      auto contactListener = EventListenerPhysicsContact::create();
81      contactListener->onContactBegin = CC_CALLBACK_1(GameScreen::onContactBegin, this);
82      this->getEventDispatcher()->addEventListenerWithSceneGraphPriority(contactListener, this);
```

4. The previous code adds a listener to actively listen for collisions; if a collision is detected, the `onContactBegin` method will be called.
5. Using the following code, add a physics body for the asteroids every time they are spawned. Add this after pushing the asteroid to the asteroids' vector.

```
auto body =
PhysicsBody::createCircle(asteroids[asteroids.size() - 1]->
getContentSize().width / 2);
body->setContactTestBitmask(true);
body->setDynamic(true);
asteroids[asteroids.size() - 1]->setPhysicsBody(body);
```

The following screenshot is how the file will look after adding the preceding code:

```
202     tempAsteroid->setPosition(Point(xRandomPosition + origin.x, -tempAsteroid->getContentSize().height + origin.y));
203     asteroids.push_back(tempAsteroid);
204     auto body = PhysicsBody::createCircle(asteroids[asteroids.size() - 1]->getContentSize().width / 2);
205     body->setContactTestBitmask(true);
206     body->setDynamic(true);
207     asteroids[asteroids.size() - 1]->setPhysicsBody(body);
```

Nothing is done differently for the asteroid body than the space pod apart from changing the size and position.

6. The final step is to implement the `onContactBegin` method, which will call the `GoToGameOverScene` method that was implemented earlier in the book:

```
bool GameScreen::onContactBegin(PhysicsContact& contact)
{
    GoToGameOverScene(this);

    return true;
}
```

Though only circular collision detection was used in this chapter as it was sufficient, when using more complex objects and shapes, the need for better and bespoke shapes can arise, and then tools such as PhysicsEditor can be used to create custom shapes. PhysicsEditor is an amazing tool that allows you to create custom shapes using the built-in tracer or manipulating the physics body shape using a mouse.

Summary

This chapter covered how Cocos2d-x handles physics using its own physics implementation built on top of the Chipmunk physics engine. Then, the scene was modified to run using physics. Once the physics world was set up, we implemented physics listeners and bodies to detect collisions between the player's space pod and the asteroids. The next chapter will cover how to add audio to the game when certain conditions are triggered, for example, when the player dies by colliding with an asteroid.

9
Adding Audio to the Game

This chapter will cover how Cocos2d-x handles the implementation of audio within a game. We will also cover the different types of audio available for use and how to implement them using sound files that will be provided. The sound will be played when certain conditions are met and methods are triggered.

The topics covered in this chapter are as follows:

- Loading sound effects
- Playing sound effects
- Loading background music
- Playing background music
- Additional sound effect features
- Additional background music features

Cocos2d-x along with many other engines supports two main types of audio implementation. They are as follows:

- **Sound effects**: This is used to play a short file, which could be repeated several times. For example, clicking on a button or collecting pickups.
- **Music**: This is used to play music files that are large and mainly designed for background music, and usually only one is played at any given time.

Firstly, add the sound files to your `Resources` folder inside a sound folder (the sound folder isn't necessary but helps with file organization). Then, add the sound folder to the game project. This process is very similar to adding the images as we did earlier in the book.

The next few sections will cover audio implementation within our project (Cocos2d-x only supports MP3 and WAV formats for cross-platform support but the OGG format is supported on Android).

Loading and playing sound effects

This section will cover how to use Cocos2d-x to load and play back sound effects within our game when certain conditions that have already been implemented throughout the course of this book are triggered.

The following are the main steps to play sound effects:

1. Include sound header.
2. Preload sound effect file (usually in the `init()` method).
3. Play sound effect when appropriate.

Adding sound effects

There will be two main sound effects that will be played throughout the game:

- **Button-click sound effect**: This will be played whenever a menu button is clicked excluding the pause button
- **Crash sound effect**: This will be played when the player's space pod collides with an asteroid

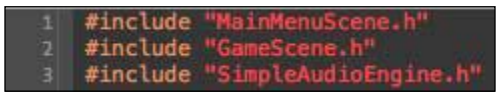
Adding the menu-button-click sound effect

Let's add the menu-button-click sound effect in the `MainMenuScene.cpp` file using the code explained in the following steps, which follows the steps given in the *Loading and playing sound effects* section:

1. Add the sound header to the top of the file:

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

   This is how the `MainMenuScene.cpp` file will look after adding the preceding header file:



2. Preload the sound effect inside the `init()` method (make sure when the resource files are added, they are added as a reference and not as a group):

```
CocosDenshion::SimpleAudioEngine::getInstance()->
preloadEffect("audio/ButtonClick.wav");
```

This is how the preceding code will look in the `MainMenuScene.cpp` file:

```
49      CocosDenshion::SimpleAudioEngine::getInstance()->preloadEffect("audio/ButtonClick.wav");
```

The audio engine doesn't belong to the Cocos2d-x namespace as it is an optional module. This allows other sound engines such as FMOD to be used.

3. Play the sound effect in the `GoToGameScene` method:

```
CocosDenshion::SimpleAudioEngine::getInstance()->
playEffect("audio/ButtonClick.wav");
```

This is how the preceding code will look in the `MainMenuScene.cpp` file:

```
54  void MainMenu::GoToGameScene(cocos2d::Ref *pSender)
55  {
56      CocosDenshion::SimpleAudioEngine::getInstance()->playEffect("audio/ButtonClick.wav");
57
58      auto scene = GameScreen::createScene();
59
60      Director::getInstance()->replaceScene(TransitionFade::create(1.0, scene));
61  }
```

Before adding the crash sound effect, add all the other instances of the button-click sound effect using the preceding steps. The following is a list of all the places it needs to be played:

- The play button in the Main Menu scene (we just implemented it)
- The resume, main menu, and retry buttons in the Pause scene
- The retry and main menu buttons in the Game Over scene

Adding the crash sound effect

Now let's add the crash sound effect using the same steps for the button-click sound effect. Add the code shown in the following steps in the `GameScene.cpp` file:

1. Add the sound header to the top of the file:

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

This is how it will look in the `GameScene.cpp` file:

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

2. Preload the sound effect inside the `init()` method:

```
CocosDenshion::SimpleAudioEngine::getInstance()->
preloadEffect("audio/Crash.wav");
```

This is how the preceding code will look in the `GameScene.cpp` file:

```
    CocosDenshion::SimpleAudioEngine::getInstance()->preloadEffect("audio/Crash.wav");
```

3. Play the sound effect in the `onContactBegin()` method:

```
CocosDenshion::SimpleAudioEngine::getInstance()->
playEffect("audio/Crash.wav");
```

This is how the preceding code will look in the `GameScene.cpp` file:

```
bool GameScreen::onContactBegin(PhysicsContact& contact)
{
    CocosDenshion::SimpleAudioEngine::getInstance()->playEffect("audio/Crash.wav");

    GoToGameOverScene(this);

    return true;
}
```

This section covered how to preload sound effect files and play them using Cocos2d-x. Then, button-click and crash sound effects were implemented in the game. There are other features that can be used for sound effects that will be covered in the next section.

Additional sound effect features

The previous section covered loading and playing sound effect files, but there are a few other things that can be done with sound effects. Some of them will be covered in this section, with the rest being accessible via Cocos2d-x's reference page.

Though sound effect files are small, they can be paused and resumed using the following code:

```
CocosDenshion::SimpleAudioEngine::getInstance()->
pauseEffect(soundID);
CocosDenshion::SimpleAudioEngine::getInstance()->
resumeEffect(soundID);
```

The `soundID` variable is an unsigned integer, which is assigned the return value of a sound effect being played, for example:

```
unsigned int id;
id = CocosDenshion::SimpleAudioEngine::getInstance()-
>playEffect("audio/ButtonClick.wav");
CocosDenshion::SimpleAudioEngine::getInstance()->pauseEffect(id);
CocosDenshion::SimpleAudioEngine::getInstance()->resumeEffect(id);
```

The volume of a sound effect playback can be set ranging from 0 to 1, using the following code:

```
CocosDenshion::SimpleAudioEngine::getInstance()->
setEffectsVolume(0.5);
```

This section was an overview of some of the additional features for sound effects provided by Cocos2d-x.

Loading and playing background music

This section will cover how to use Cocos2d-x to load and play background music within our game to add an extra dimension to the game.

The following are the main steps to play background music:

1. Include sound header.
2. Preload background music file (usually in the `init()` method).
3. Play background music, usually at the start of a scene in the `init()` method.

Adding background music

Let's add background music to the `MainMenuScene.cpp` file:

1. Add the sound header to the top of the file (we have already added it for the button-click sound effect):

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

The following is how it will look in the `MainMenuScene.cpp` file:

```
#include "MainMenuScene.h"
#include "GameScene.h"
#include "SimpleAudioEngine.h"
```

2. Check whether the background music is already playing. If it isn't, preload the background music file and play it on repeat (add `true` to repeat, after specifying the audio file in the play method). Add the following code inside the `init()` method:

```
if (CocosDenshion::SimpleAudioEngine::getInstance()->
isBackgroundMusicPlaying() == false)
{
    CocosDenshion::SimpleAudioEngine::getInstance()->
    preloadBackgroundMusic("audio/Music.mp3");
    CocosDenshion::SimpleAudioEngine::getInstance()->
    playBackgroundMusic("audio/Music.mp3", true)
}
```

The following is how it will look in the `MainMenuScene.cpp` file:

```
51  if (CocosDenshion::SimpleAudioEngine::getInstance()->isBackgroundMusicPlaying() == false)
52  {
53      CocosDenshion::SimpleAudioEngine::getInstance()->preloadBackgroundMusic("audio/Music.mp3");
54      CocosDenshion::SimpleAudioEngine::getInstance()->playBackgroundMusic("audio/Music.mp3", true);
55  }
```

This section covered how to preload a background music file and play it using Cocos2d-x by first checking whether the music is already playing. There are other features that can be used for background music. These will be covered in the next section.

Additional background music features

The previous section covered loading and playing background music, but there are a few other features provided by Cocos2d-x for background music. Some of them will be covered in this section, with the rest being accessible via Cocos2d-x's reference page.

Background music can be paused and resumed as well through Cocos2d-x. This needs to be done when the application goes to the background. iOS automatically prevents the music from playing but Android OS doesn't. Modify the `AppDelegate.cpp` file using the following steps:

1. Add the following code to the `applicationDidEnterBackground()` method to pause the music when the application goes to the background. For example, when the home button is tapped:

   ```
   CocosDenshion::SimpleAudioEngine::getInstance()->
   pauseBackgroundMusic();
   ```

   The following is what the preceding code will look like in the `AppDelegate.cpp` file:

```
98  void AppDelegate::applicationDidEnterBackground() {
99      Director::getInstance()->stopAnimation();
100
101     // if you use SimpleAudioEngine, it must be pause
102     CocosDenshion::SimpleAudioEngine::getInstance()->pauseBackgroundMusic();
103 }
```

2. Add the following code to the `applicationWillEnterForeground()` method to resume the music when the application returns to the foreground. For example, when the user navigates back to the application from the device's home screen:

   ```
   CocosDenshion::SimpleAudioEngine::getInstance()->
   resumeBackgroundMusic();
   ```

   The following is what the preceding code will look like in the `AppDelegate.cpp` file:

```
106 void AppDelegate::applicationWillEnterForeground() {
107     Director::getInstance()->startAnimation();
108
109     // if you use SimpleAudioEngine, it must resume here
110     CocosDenshion::SimpleAudioEngine::getInstance()->resumeBackgroundMusic();
111 }
```

3. The volume of the background music playback can be set to range from 0 to 1, using the following code:

   ```
   CocosDenshion::SimpleAudioEngine::getInstance()->
   setBackgroundMusicVolume(0.5);
   ```

   The following is what the preceding code will look like in the `AppDelegate.cpp` file:

```
102     CocosDenshion::SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(0.5);
```

This section was an overview of some of the additional features provided by Cocos2d-x for background music. These additional features can be used to enhance the gameplay and create an immersive atmosphere within the game.

Summary

This chapter covered how Cocos2d-x handles audio loading and playback along with some of the additional features provided such as setting the audio volume. Sound effects were applied with background music constantly being played in the background. The next chapter will cover the use of an accelerometer to enable more dynamic and engaging gameplay for the user.

10
Implementing Accelerometer Support

This chapter will cover how to set up accelerometer support within a Cocos2d-x project using the built-in functions. This chapter can be thought of as a general purpose reference guide with no additional dependencies.

The contents of this chapter will not be used for the game that is being developed through the course of this book, but it is still important to have an understanding of accelerometer support using Cocos2d-x, enabling you to implement it in future projects.

The topics covered in this chapter are as follows:

- Setting up the accelerometer
- Detecting motion
- Getting the motion's direction

Many existing games make use of accelerometer support to enhance the gaming experience. Accelerometers allow the phone/application to detect motion, for example, tilting a device. Fortunately, Cocos2d-x provides built-in functionalities to set up and handle motion through the accelerometer. Most devices have accelerometers, so if an application only supported accelerometer gameplay, it wouldn't have an issue running on most, if not all, devices.

This isn't to say that other input techniques such as touch should be omitted, as not all users like motion or touch. Providing multiple avenues of input is the ideal route to go down when developing games. The touch functionality should be provided as a minimum requirement as users are more familiar with this kind of input. Accelerometer support will not be implemented in the game created throughout this book; you can implement it as an additional feature after reading this chapter.

Setting up the accelerometer

This section will cover how to use Cocos2d-x to set up the accelerometer support within a scene in order to enable motion detection. Setting up the accelerometer is very similar to setting up touch events, as it requires a listener to actively check for motion that calls the appropriate functions.

The following steps will guide you through the process of implementing accelerometer support within a game:

1. Open the scene header file that you would like to add accelerometer support to, and add the acceleration method that will be called every time motion is detected:

   ```
   void onAcceleration(cocos2d::Acceleration *acc,
   cocos2d::Event *event);
   ```

   The following is how the file will look after adding the preceding code line:

2. Add the following code lines inside the `init()` method of the scene's implementation file:

   ```
   Device::setAccelerometerEnabled(true);
   auto listener = EventListenerAcceleration::create
   (CC_CALLBACK_2(MainMenu::onAcceleration, this));
   Director::getInstance()->getEventDispatcher()->
   addEventListenerWithSceneGraphPriority(listener, this);
   ```

   The following is how the file will look after adding the preceding code lines:

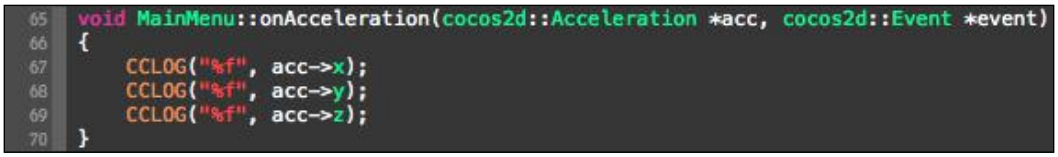
   Let's go over the code added to the `init()` method:

   - The first statement enables the accelerometer
   - The second statement sets up an acceleration listener to actively check for motion and call the `onAcceleration()` method when motion is detected
   - The third statement sets the listener to the event dispatcher to pick up the motion event

3. Add the following implementation for the `onAcceleration()` method:

```
void MainMenu::onAcceleration(cocos2d::Acceleration *acc,
cocos2d::Event *event)
{
    CCLOG("x = %f", acc->x);
    CCLOG("y = %f", acc->y);
    CCLOG("z = %f", acc->z);
}
```

   The following screenshot is how the file will look after adding the preceding code lines:



The `onAcceleration()` method that was just implemented simply prints out the different axes' acceleration values. Using these, you could keep a track of previous values and check what direction it has moved, to possibly move the space pod. Use this chapter and implement the accelerometer support into the game as an additional feature.

Using the preceding code, you can test the accelerometer on a device. The values received can be stored and compared to see how they change depending on how the device is moved. This will enable you to determine the range for the implementation of the desired motion effect within the game, which could be used to move the space pod using the accelerometer. This could be achieved by storing the acceleration value and comparing it to the current acceleration and then moving in a particular direction.

Summary

This chapter covered how Cocos2d-x handles motion through the accelerometer support. However, it wasn't implemented within the game. This chapter serves as a good foundation for implementing it within the game to move the space pod using motions such as tilting the device. The next chapter will cover how to solve some of the common problems you might face when using Cocos2d-x and where to go from there.

11
Problem Solving and What's Next

Congratulations! We have completed the game development. Together, we created a space game, which tasks the player with avoiding the oncoming asteroids using touch. This chapter won't add anything to the game in particular but will show you some of the common pitfalls you will most likely face when developing games using Cocos2d-x and how to overcome them.

We will cover the following topics in this chapter:

- Common issues when developing games using Cocos2d-x
- Overcoming the issues
- What to do with the skills acquired through the course of the book

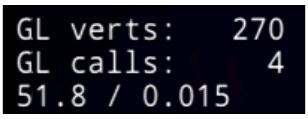
Problem solving

This section will cover some of the issues that may plague a Cocos2d-x project; it will also cover the methods to overcome them.

Removing debug information

A new project, by default, has the debug information situated in the bottom-left corner of the screen; this information should be disabled before publishing a game. However, it is still important, so it should only be disabled once the game development is complete.

The following is what the debug information looks like:



To remove the debug information, use the following steps:

1. Open `AppDelegate.cpp`.
2. Change the line `director->setDisplayStats(true);` to `director->setDisplayStats(false);`, as shown in the following screenshot:

Positioning on different devices

There are several devices the user can use; therefore, developing for a single device isn't a practical route, as it would alienate many potential users, thus reducing revenue. The main problem to avoid is developing using magic numbers (arbitrary values within the code). For example, for positioning, use relative positioning instead; this will be dynamic.

Let's look at an example. A background needs to be added to a scene and should be centered. The developer might be using an iPhone in landscape mode with a resolution of 480 x 320 pixels; in this case, the developer could set the background position to (240, 160). This will work well on their device but not on iPhone Retina, iPad, or most Android devices. Instead, the best method would be to position the background at `(screenSize.width / 2, screenSize.height / 2)`. Using this method, the background will be positioned well on all devices. A similar system should be used for all assets.

Moving an object on different devices

When moving objects on different devices, there can be speed differences; this can be due to two main problems:

- **Frame rate**: This problem arises when more powerful devices can run the game at higher frame rates. This can be overcome by factoring in delta time using an `update()` function, as we did when scrolling the game's background.

- **Screen size**: Different screen sizes can cause issues as a moving object could take longer to move on an iPad Retina compared to an iPad. This can be overcome by factoring in the screen resolution; width or height can be used, but it's important to be consistent. This was demonstrated earlier in the book when scrolling the background.

Trouble generating new projects

When generating new projects, several issues can arise; one of the main causes is that the setup wasn't successful. Confirm that all the environment variables have been added by running the `setup.py` command again.

Reusing actions

If you try and reuse actions on a node, the game will crash. The only way to overcome this is to create a new action or clone an action (you need to autorelease it for Cocos2d-x 2.x.x) for each instance instead of trying to reuse it.

Sequencing actions

When running several actions, you might want them to run one after another, that is, when the previous one has finished running. If multiple actions are run using the `runAction` method on a node, they will start as soon as they are called. However, to run them one after another, the sequence action should be run; this has already been covered in this book.

Running your application on simulators

A general rule of thumb is to not run applications on Android simulators as they are terrible; run them on an actual device. For iOS, the simulators are a lot better but still not as good as a real device, so you should run the application on a real device if applicable. This is due to certain features such as motion missing from a simulator and simulators having poor performance. A very good alternative to the Android simulator is BlueStacks, which is free.

Application size

Many users think that because the generated application's folder is over 200 MB, the size of their application will also be 200 MB; it won't as there is all the extra stuff. An easy way to find it out is to build the project and check the application's size. However, this doesn't mean you shouldn't try and reduce the application size. Use the following tips to reduce the application's size:

- Use sprite sheets instead of separate images
- Reduce image size, as high-resolution assets aren't always required for mobile devices
- Make sure that all the unnecessary files are removed, including duplicates
- Try to avoid big images if they are not needed (images above 1024 x 1024 pixels resolution are not good for cross-platform games)

Breakpoints

This is more of a general tip: use breakpoints when you are having issues, as it allows individual lines to be tested while being able to check the status of variables using the console. Using logs (CCLOG) is also a useful technique for debugging.

What's next?

Until now, we discussed some of the issues that may occur when developing using Cocos2d-x. The best way to overcome them is to check out the tests that are provided by Cocos2d-x; check out the Cocos2d-x forum, stack overflow, and our YouTube channel (`https://www.youtube.com/user/sonarsystemslimited`), which contains hundreds of tutorials.

Together we created a game from scratch, providing you with the foundation to create games. The game developed through the course of this book will be available for iOS and Android for free; it is titled *Space Drop Free*.

So, what's next now that this book is finished? First, this book is a great reference guide for Cocos2d-x, so it can be used over and over again when implementing features and trying to recap how they work. Here is a list of things that you could do as learning tasks:

- Implement accelerometer support into the game
- Add a scoring feature
- Increase the speed of the background as the game progresses, similar to the *Super Jet Bunny* game
- Add multiple levels
- Implement a power up feature

Index

D

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

Z

Thank you for buying
Cocos2d-x Game Development Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

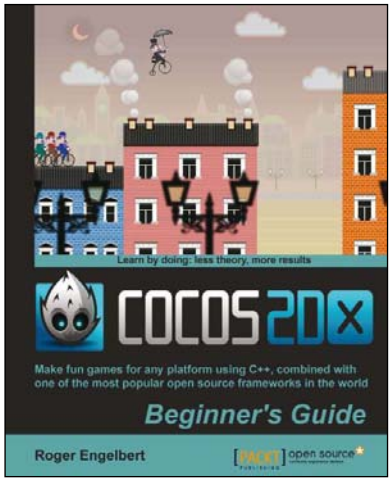
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced 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, 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

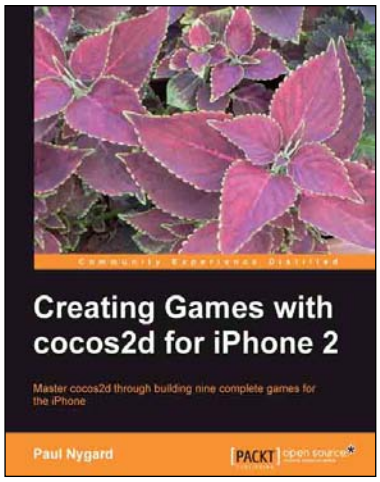


Cocos2d-x by Example Beginner's Guide

ISBN: 978-1-78216-734-1 Paperback: 246 pages

Make fun games for any platform using C++, combined with one of the most popular open source frameworks in the world

1. Learn to build multi-device games in simple, easy steps, letting the framework do all the heavy lifting.
2. Spice things up in your games with easy-to-apply animations, particle effects, and physics simulation.
3. Quickly implement and test your own gameplay ideas, with an eye for optimization and portability.



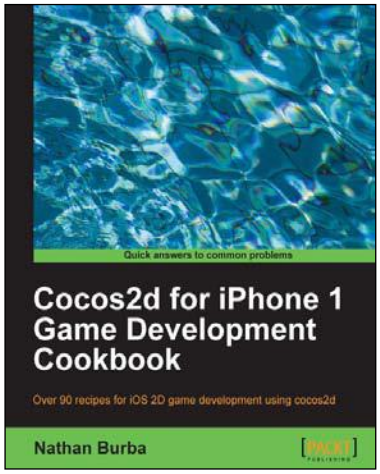
Creating Games with cocos2d for iPhone 2

ISBN: 978-1-84951-900-7 Paperback: 388 pages

Master cocos2d through building nine complete games for the iPhone

1. Games are explained in detail, from the design decisions to the code itself.
2. Learn to build a wide variety of game types, from a memory tile game to an endless runner.
3. Use different design approaches to help you explore the cocos2d framework.

[PACKT] open source
community experience distilled
PUBLISHING

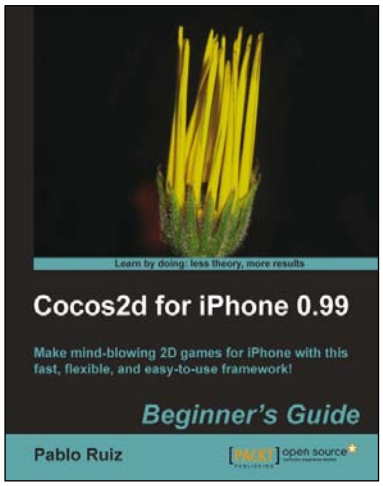


Cocos2d for iPhone 1 Game Development Cookbook

ISBN: 978-1-84951-400-2 Paperback: 446 pages

Over 90 recipes for iOS 2D game development using cocos2d

1. Discover advanced Cocos2d, OpenGL ES, and iOS techniques spanning all areas of the game development process.
2. Learn how to create top-down isometric games, side-scrolling platformers, and games with realistic lighting.
3. Full of fun and engaging recipes with modular libraries that can be plugged into your project.



Cocos2d for iPhone 0.99 Beginner's Guide

ISBN: 978-1-84951-316-6 Paperback: 368 pages

Make mind-blowing 2D games for iPhone with this fast, flexible, and easy-to-use framework!

1. A cool guide to learning Cocos2d with iPhone to get you into the iPhone game industry quickly.
2. Learn all the aspects of Cocos2d while building three different games.
3. Add a lot of trendy features such as particles and tilemaps to your games to captivate your players.
4. Full of illustrations, diagrams, and tips for building iPhone games, with clear step-by-step instructions and practical examples.