Community Experience Distilled

Starling Game Development Essentials

Develop and deploy isometric turn-based games using Starling

Juwal Bose

[PACKT] open source*
community experience distilled
PUBLISHING

Starling Game Development Essentials

Develop and deploy isometric turn-based games using Starling

Juwal Bose

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Starling Game Development Essentials

First published: December 2013

Production Reference: 1171213

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

ISBN 978-1-78398-354-4

`www.packtpub.com`

Cover Image by Aniket Sawant (`aniket_sawant_photography@hotmail.com`)

Credits

Author
Juwal Bose

Reviewers
Daniel Sperl
Devon O. Wolfgang
Mariam Dholkawala

Acquisition Editors
Meeta Rajani
Erol Staveley
Rebecca Youe

Commissioning Editor
Sruthi Kutty

Technical Editors
Shruti Rawool
Shali Sasidharan

Project Coordinator
Aboli Ambardekar

Proofreader
Paul Hindle

Indexers
Tejal Soni
Mehreen Deshmukh
Monica Ajmera Mehta

Graphics
Yuvraj Mannari

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

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

Juwal has been actively involved with Adobe technologies, manages the Adobe user group for his state, and maintains a close relationship with the experts at Adobe in the Indian subcontinent. He has been developing games since 2004 using multiple technologies including ActionScript, Objective-C, Java, Unity, LibGDX, Cocos2D, OpenFL, Blender, and Starling. His team has created more than 350 online Flash games to date, and many of the job management games are listed at the top of leading portals worldwide.

Juwal writes game development tutorials for GameDevTuts+ and manages the blog of Csharks games. His isometric tutorial for GameDevTuts+ is well received and is considered a thorough guide for developing tile-based isometric games. Juwal also takes part in game jams and recently won the Nasscom GDC's game jam with his Starling-based game, Mosquito Defense. This is Juwal's first book, but he aims to keep writing and sharing his nine years of game development experience through more books.

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

Acknowledgments

I would like to thank my wife, Dr. Nidhina Haridas, for inspiring me to aim higher and try harder. My deep thanks go to Mr. Eldhose P Mathew, CEO of Csharks, for allowing me to devote my professional time for the creation of this book. The incredible graphics for the sample game was created by Mr. Sugith Vijayan, one of Csharks' awesome artists. A big thanks to him for the same and congratulations on a work well done at such a short notice. Sincere thanks to Mr. Abhilash CV for his patience on testing all the cross-platform builds of the sample game and to the whole team at Csharks.

I wish to extend my thanks to Ville Koskela for his awesome rectangle packing algorithm, without which the real multiple screen support would never have been possible. I thank Devon O. Wolfgang, a well-known ActionScript rockstar, for helping with the technical reviewing of the book as well as his valuable advice on code revision. My sincere thanks to Mariam Dholkawala—the veteran and expert game developer—for taking her time out for reviewing the code and for providing her valuable input also at a very short notice.

Finally, the most important person, Daniel Sperl, the creator of Starling, demands my extreme gratitude. He had supported me all along with wise tips, insights, and frequent encouragement. Although being incredibly busy, he took time out to review the code and was the real inspiration behind the `DynamicTextureAtlasCreator` class. I am humbled by his helpfulness and inspired by this opportunity to be associated with him.

I extend my thanks to Meeta Rajani, Aboli Ambardekar, and Sruthi Kutty along with others at Packt Publishing for being extremely patient with me and helping me out at each step.

About the Reviewers

Daniel Sperl is a longtime game developer and has created numerous casual games in ActionScript, C#, and Objective-C. Unsatisfied with the existing 2D frameworks of the time, he created the Sparrow Framework for iOS, mimicking the tried and tested Flash API in Objective-C. Ironically, Adobe was looking for exactly such a framework for Flash when they introduced Stage3D in 2011, and so the Starling Framework was born. In 2012, he cofounded the company Gamua to be able to work full-time on his frameworks.

When he is not developing Starling or giving support in the forum, Daniel loves to play the latest Zelda or Super Mario title with his wife, or ride his bike along the countryside of Austria. Yes, he loves birds just like his cat.

Devon O. Wolfgang hails from "the most dangerous city in America" (Business Insider, 2013), Flint, Michigan. He spent six years working as an electronics technician for the U.S. Navy. Since his honorable discharge from the military, he has worked continuously in the world of Flash and ActionScript development; from various freelance projects, to the high-paced arena of an advertising agency, to the gaming industry, and now works as Senior Software Engineer in the entertainment sector with Meez. Currently, he resides in Dublin, Ireland, with his beautiful wife. You can find his infrequent ramblings on ActionScript, technology, and life in general, on his blog at `http://blog.onebyonedesign.com/`.

Mariam Dholkawala is an independent gaming professional living in Mumbai. She has spent 11 years developing games across mobile and web platforms. She has worked as a Studio Head and Producer with leading gaming companies in India, and currently freelances the clients for their end to end game development requirements. Her development skills include programming games in AS3/Flash, HTML5/JavaScript, and using frameworks such as Starling, Box2D, and Citrus Engine.

In addition to creating games, she has written gaming articles on Adobe Developer Connection and is a freelance writer for a leading technology magazine in India. When not working on games, Mariam likes traveling to new countries and learning about new cultures and languages. She blogs at `http://www.mariamdholkawala.com`.

Table of Contents

Preface

With iOS and Android, everyone is a gamer and there is a huge demand for cross-platform games. Creating games is always fun as well as challenging, and Starling, with Adobe AIR, makes it very easy for an ActionScript developer to create games. This book will guide you through the development of a cross-platform Starling tile-based, isometric, turn-based game with AI, multiplayer capability, and its own level editor, which you will publish on the Web, App Store, and Google Play.

Starling Game Development Essentials takes you step-by-step through the development of a complicated isometric game, where you will create a level editor, AI logic for an enemy, multiplayer functionality on the same device, integrate particle effects, AdMob ANE, Flox leaderboard, and finally deploy to the Web, App Store, and Google Play. You will explore `RenderTexture` for dynamically creating game levels and you will later upgrade to the exceptional `QuadBatch` for deploying on devices. You will use Starling Particle Extension for explosion effects. You will develop a simple AI manager to help the enemy make decisions and use `pathfinder` to facilitate grid-based path finding. With the help of `DynamicTextureAtlasCreator`, you will support multiple screens with a single texture atlas.

Creating games is never easy, but *Starling Game Development Essentials*, with the help of the Flag Defense game source code, will help you in the process and make you capable of creating your next Starling cross-platform game.

What this book covers

Chapter 1, *Our Capture the Flag Game*, introduces the game idea for the Flag Defense game which we will be developing during the course of the book. This chapter covers the game loop, game states, and the tile-based game development approach which we will use to build the game engine.

Chapter 2, *Going Isometric*, helps you to understand the isometric projection and details the relationship between the Cartesian and isometric coordinates. With the help of a helper class created in this chapter, we will do isometric movement as well as effective depth swapping. With the help of sample code, we will form a level data structure to follow for the game.

Chapter 3, *Isometric-level Designer*, explains how we need to create a level editor to speed things up, and using FeathersUI, we will create one which adheres to our level data structure. We will use it to export the level data and also to change the registration points of nonstandard tiles so that everything looks correct in our game level.

Chapter 4, *Just Finding Your Way*, is the chapter where it gets more serious as we implement our game level, add soldiers, add UI, and use user interaction to move the soldiers. We will use a path finding algorithm for enemy soldiers to proceed to their destination. We will be implementing basic turn-based AI and get introduced to an alternate approach for displaying animated items in Starling.

Chapter 5, *Boom Boom, Explosions!*, is where we introduce particle effects and add explosions to some demos. We will use frame-based particle effects as well as the Starling particle extension. The online particle designer tool is used for creating the PEX files for the explosion effect.

Chapter 6, *Going Cross Platform*, explains the complete game code, both for the Web version as well as the device version. The AI decision making logic is discussed in detail and multiscreen support is analyzed, resulting in an approach which requires only a single set of art. Local leaderboard, integration of ANE, deployment to various app stores, and the different tips and tricks of device development are explained in detail.

Chapter 7, *Flox – Leaderboards, Analytics, and More*, introduces the set of services provided via Flox.cc which enable us to have global leaderboards, game analytics, persistent data, user tracking, and much more across all devices. Sample code is provided for each case which can be altered for use in any custom game.

What you need for this book

You will require a good ActionScript 3 editor/compiler, which ideally should be Flash Builder from Adobe. You may also use FDT or Flashdevelop as well, but you may need to make some adjustments for the cross-platform setup as the book explains the process from the Flash Builder perspective. All sample codes should compile and execute once correctly wired up in all the previous mentioned software.

Additionally, we have a few framework dependencies for some chapters which are Starling, Starling Particle Extension, Starling Graphics Extension, AS3 Signals, TweenLite, Flox, Pathfinder class (provided in code), and Rectangle Packing.

Who this book is for

This book is for ActionScript developers who are trying to create Starling cross-platform games. Prior knowledge of Starling will help, but is not necessary. The flag defense game covers some complicated topics in game development which is beneficial even for those who are already creating games with Starling. JavaScript experts who do not know ActionScript but have the willingness to try can also use this book, but it will be a steep learning curve.

I personally believe that if you are an expert at any single programming language, then you should have no trouble learning another one very easily. What matters in game development is the right application of logic, the willingness to learn, and to update, not the language of choice.

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 are shown as follows: "The second part of the first phase is the `EnterFrame` event."

A block of code is set as follows:

```
var tile:Image = Image.fromBitmap("tile"+levelData[5][4]+".png");
addChild(tile);
tile.x=5*tileWidth;
tile.y=4*tileWidth;
```

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

```
var m:Matrix = new Matrix(1,0.5,-1,0.5,0,0);
```

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: "The default texture in the Particle Editor is **Circle**".

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.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `http://www.packtpub.com/sites/default/files/downloads/3544OS_ColoredImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you 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
Our Capture the Flag Game

Welcome to Starling deep dive, where we will focus on creating an interesting game using the excellent Starling framework. We will learn about the major aspects of a typical game including game states and game loops, which will eventually help us to create our own Starling-based game engine. We will be proceeding on the assumption that you already know the concepts of **ActionScript 3** programming and have a basic knowledge of the Starling framework. If you are good in ActionScript 3 but have never tried Starling, this is the opportunity to get started. Starling opens up the possibility of deploying your ActionScript-based games onto the touch-based devices including Android and iOS with excellent performance.

In this chapter, we will cover the following topics:

- Learn about the components that make up a game engine
- Learn about the tile-based design approach
- Get introduced to the tools of trade—applications which help in development
- Explain and understand the game idea which we will be implementing
- Implement an array-based level data onto a top-down game scene

Understanding a game

So what really is a game? It seems to be a difficult question, as we can imagine a million kinds of answers depending on whom you would ask. But particularly for a developer, a game can be explained as a very simple and straightforward logic block. When stripped of all its glory, a game program essentially becomes what we programmers call a **loop**. You may find it hard at this point to even consider to imagine that the ultra-realistic 3D racing game that you had just played is indeed just a glorified programming loop.

This loop is known as the game loop and it helps in executing the game logic, which is separated into logical blocks known as **states** or **game states**.

Introducing the game loop and game states

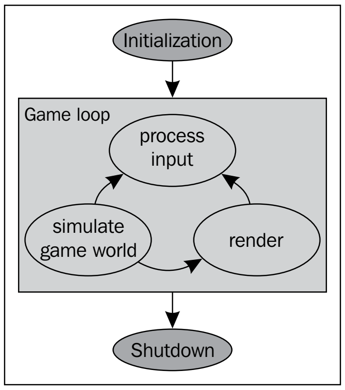
Once the game is started, the program will enter the game loop and remains in the loop until we exit the game. During this time, the game loop gets executed at a predetermined time, every second. Sometimes this maybe called the **FPS** (**frames per second**). In typical cases, we try to maintain a stable FPS throughout the game for the perfect gameplay experience. In every frame, the logic will execute a set of instructions as per the current state of the game. The state of a game is defined by game states. This may sound a bit confusing if you have not tried developing games yet; please read ahead for more clarification.

Game state

Let us try to understand a game state in layman's terms. A typical game starts off in an introduction (`init`) state, moves onto a menu state of some kind, and then to an actual gameplay state. When you get defeated, the game moves onto a game over state, and then moves towards the menu state where the process can be restarted. The traditional way of handling multiple states is with a series of `if` statements, switches, and loops. The program begins in the introduction state and loops until a key is pressed or a certain time has elapsed. Then, the menu state starts displaying a set of buttons and loops until a selection is made. Later, the game begins where we loop through the game logic until we reach a level up or till a game is over. In every frame through the game loop, the program checks to see if it should switch to the next state or simply draw the next frame from the current state.

Game loop

In its very basic form, the game loop is a state machine which switches between the different game states based on a game logic. Here is a simple illustration of the basic game loop with very basic game states: **Initialization**, **Game loop**, and **Shutdown** (exit). The following diagram shows the basic game loop:



The game loop receives user interactions and inputs, executes the game logic, updates the game world, and finally renders the scene onto our screen. The loop gets repeated based on the FPS. We need to break down our own game into different game states and implement the game engine using an efficient game loop, which logically executes through these states based on the user interaction or logic results. Do not think that this is a complicated process, as we will be breaking down our game idea into manageable game states and implementing it into a loop using an efficient technique that is already provided to us by Flash.

Flash's inherent game loop

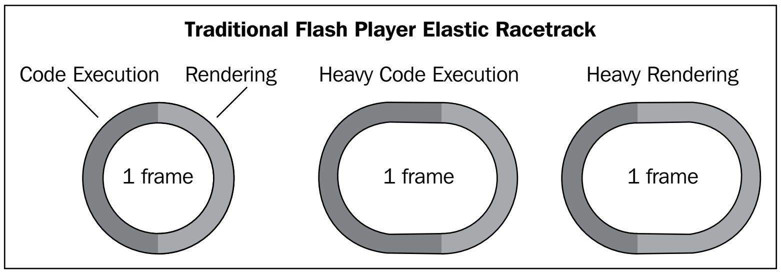
Flash already has a loop implemented which is working perfectly for us. All we need is to tap into this rendering loop. Flash has an FPS setting, based on which the Flash Player tries to render each frame one after the other. We can use this for our game loop as we have the `Event.ENTER_FRAME` event, which gets dispatched on every frame. We can easily write our core game logic as a listener to this event, and it will get executed for every frame which serves the purpose of the game loop. In addition to this, Starling has a new `EnterFrameEvent`, which also brings in more efficiency as well as functionality. With this event, we get access to the elapsed time between each individual `EnterFrame` events. This time difference is crucial in implementing time-based animations and delivering the same performance when the FPS cannot remain the same over the duration of execution.

Flash Player tries to render frame after frame while trying to maintain a stable FPS. The following diagram—the famous Flash Player elastic runtime diagram—explains how Flash Player executes the code and does the rendering for every frame. Each frame loop consists of two phases, divided into the following three parts:

- Events
- The `EnterFrame` event
- Rendering

The first phase includes two parts (events and the `EnterFrame` event), both of which potentially results in your code being called. In the first part of the first phase, runtime events arrive and are dispatched. The runtime never speeds up the frame rate due to lack of activity. If events occur during other parts of the execution cycle, the runtime queues up those events and dispatches them in the next frame. The second part of the first phase is the `EnterFrame` event. This event is distinct from the others because it is always dispatched once per frame.

The following diagram shows **Flash Player Elastic Racetrack**:



Once all the events are dispatched, the rendering phase of the frame loop begins. At that point, the runtime calculates the state of all visible elements on the screen and draws them to the screen. Then, the process repeats itself, like a runner going around a racetrack.

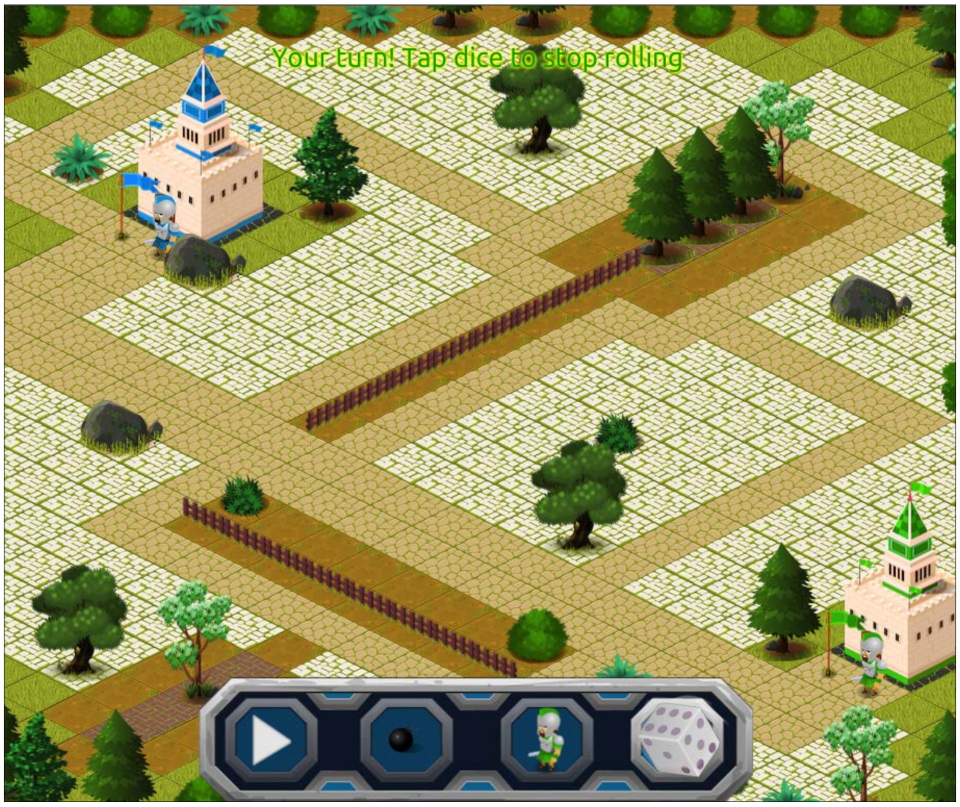
Capture the Flag

It's time to talk about a game idea that we will be trying to implement in this book. We will be creating an isometric game based on tile-based logic where the objective is to capture the enemy's flag while defending our own. The game will be turn based, where each turn is determined by the dice throws. The game involves enemies with **AI** (**Artificial Intelligence**), who will try to avoid obstacles and find paths to reach their destination. It also involves the creation of levels externally, which will adhere to a specific format suitable to be used in the game. We will include the ability for two players to play against each other by sharing the gaming device.

The game idea

The game is played out between two teams in an open terrain which will vary depending on the level loaded. The two teams are color coded among which one will be controlled by the player, while the other will be controlled by the computer. Later we will try to make both teams player controlled when we implement the same device multiplayer. The terrain is divided into square grids, which can be occupied by the soldiers of either team. Each team has their base to the extremes of the playing field away from each other. Each team base will have three soldiers and a colored flag, which they should defend. The objective is to capture the opponent team's flag and bring it to your base while keeping your flag intact. The team which has both the flags at their base wins the game.

The following screenshot shows the finished Capture the Flag game level:



Rules of the game

The gameplay is controlled by throwing a dice. Each team gets a turn/chance to throw the dice after the other team has completed its move, and hence it is a turn-based game. For a turn-based game, only one player can make a move at any particular time; when it is his/her turn, other players need to wait for their turn. A famous example would be the classic chess game.

Dice values and their corresponding results in our game are as follows:

- If the dice value is 1, the player can add a soldier on a grid next to their base
- If the dice value is 6, the player can add a bomb on a grid next to their base
- For any dice values, the player can move his/her soldiers as many grids as the value

Player can move multiple soldiers if he/she has more than one soldier on the grid. The total number of moves made by all the soldiers should be equal to the dice value. A soldier can only be moved once per dice throw.

The rules of the game play are as follows:

- Any grid can be occupied by a single soldier only.
- Soldiers can enter the grid occupied by an opponent soldier, thereby resetting the opponent soldier to their base. If a soldier ends up on a grid with a bomb, both the soldier and the bomb gets reset to base.
- Soldier entering a grid with a flag picks it up.
- Collecting the flag doesn't require an exact dice value.
- Bombs automatically move towards the nearest enemy soldier one step at each turn.

A valid move is executed by the player by dragging from the soldier to a destination grid, if the current dice value permits the same. The user interface of the game will also have a die which can be clicked on to stop the dice roll. This results in a new dice value for which the moves can be executed. For grid-based games, we have a well-established school of game development logic called tile-based design. We will explore the same in the next section.

The tile-based approach

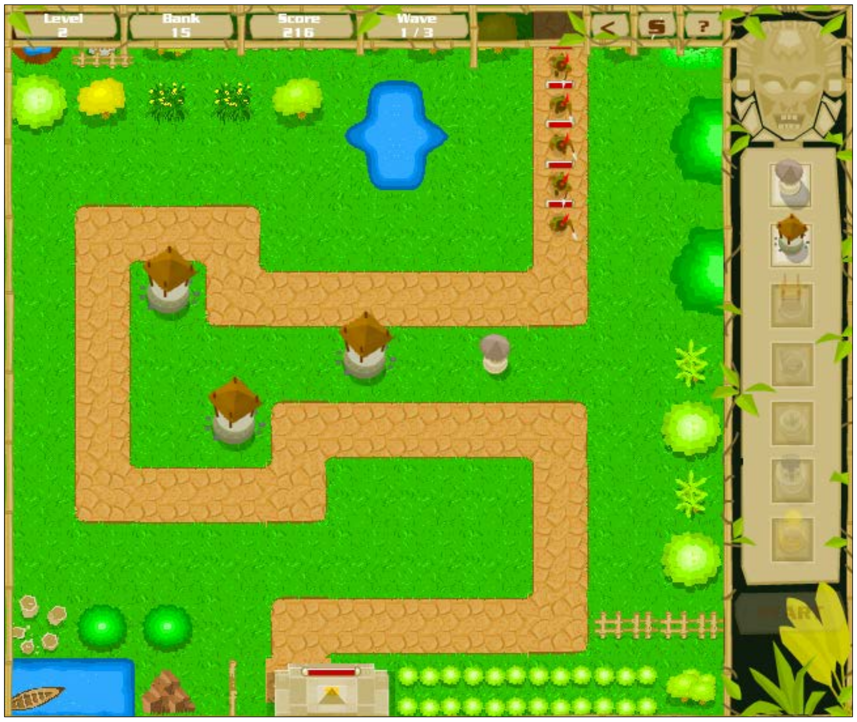
Tile-based game design involves breaking the level art into pieces known as tiles, which will then be arranged in a two - dimensional grid to create complicated levels. These tiles will be of equal dimensions, which means that the height and width will be the same for all of the tiles. Care should be taken while designing such tiles, as they need to be joined with other tiles on all four sides without creating a discontinuity in the overall level graphic created by all of the tiles arranged in the grid. Hence, a tile will need to be seamless, and we should be able to stack an infinite number of them adjacent to each other in order to create complicated-level designs.

The major advantages of using such an approach are as follows:

- We can get rid of huge level sheets
- Only a limited number of tiles are capable of creating complicated levels
- Tile-based level and logic can be implemented using standard data structures
- Collision detection and path finding becomes comparatively easier
- Scrolling levels are much easier to implement

- Multiple views can be supported: top-down, side, and isometric
- Easy serializing and deserializing of level data

The following screenshot shows an example of a top-down tile-based game level:



Now let us see how a simple two-dimensional array can be used to facilitate the implementation of the tile-based design.

An array-based implementation of a level

We have understood that a tile-based level is actually a two-dimensional grid of tiles. This can be logically represented using a 2D array, where the value stored at any index can be the ID of the corresponding tile. For example, say we have 20 different tiles making up a level of 50 x 50 tiles. This means that if an individual tile has dimensions A x B, then our level will be 50 x A wide and 50 x B high. Here, A is called `tileWidth` whereas B is called `tileHeight`. In many cases, both of these values will be the same resulting in square tiles. For representing these 20 different tiles, we can give them sequential IDs starting from 1 to 20. So, tile number 12 will have an ID as 12. This means that our 50 x 50 level data can be stored as a two-dimensional array with tile IDs as values.

The following example shows the level data for an 8 x 5 tile grid:

```
[[5,1,1,1,1,1,1,6],
[4,0,0,0,0,9,0,2],
[4,0,9,0,0,0,0,2],
[4,0,0,0,0,0,0,2],
[8,3,3,3,3,3,3,7]]
```

This level data can be explained as follows:

- ID 0: Sand tile, walkable
- ID 1, 2, 3, and 4: Wall tiles on top, right, bottom, and left respectively
- ID 5, 6, 7, and 8: Wall corner tile on top - left, top - right, bottom - right, and bottom - left respectively
- ID 9: Sand tile with a rock formation

Relationship between screen space and level array

Now that we have represented our level data using a 2D array, let us explore how we can formulate a relationship between the graphic in screen space and their corresponding value in the array. Let us consider a case where we have square tiles with the dimension `tileWidth` x `tileWidth`. Also, our level array is M x N, that is, M columns and N rows. We can place the first tile at `[0][0]` on screen at (0,0) as:

```
var tile:Image = Image.fromBitmap("tile"+levelData[0][0]+".png");
addChild(tile);
```

The source uses a better asset management by using a texture atlas, and then an image will be created using the following code:

```
tile=new Image(texAtlas.getTexture(paddedName(levelData
[i][j])));
```

`LevelData[0][0]` will return the stored tile ID value and it is appended to the string to form the correct name of the tile. For example, if `levelData[0][0]` is 5, then the tile image name will be `tile5.png`. We create an image from this particular bitmap and add it to the scene. As this tile is situated at the top-left extreme, there is no need to explicitly set the `x` and `y` values because by default they will be `0, 0`. For any other tile, we will need to set the `x` and `y` values for rightly positioning it on the screen.

Let us take the example of `levelData[5][4]`, which means the tile is in the sixth column and fifth row. For placing this tile, we will need to offset the tile by relevant multiples of `tileWidth`.

```
var tile:Image = Image.fromBitmap("tile"+levelData[5][4]+".png");
addChild(tile);
tile.x=5*tileWidth;
tile.y=4*tileWidth;
```

So for any tile at an arbitrary position `i, j` in the array, the screen position will be `x = i * tileWidth` and `y = j * tileWidth`. This is the forward relationship from level array to screen space, but what about the reverse? How will we find the corresponding level array value from any particular point in screen space?

For this, we will need to reverse the preceding formula, that is, `i = x / tileWidth` and `j = y / tileWidth`. Care is to be taken to make sure that we only consider the integer value of the calculated values, as the original result will be a float with decimal values. This is because AS3 array indices are always integer values starting at a value of 0.

So let us write down our formulas for any level array at the position `[i][j]`:

```
x = offsetX + (i * tileWidth);
y = offsetY + (j * tileWidth);
```

Here, `offsetX` and `offsetY` are the space we need to leave out in the left and top respectively, because in normal cases our level art won't start at (0, 0) screen position, but at an arbitrary (`offsetX`, `offsetY`). Reversing the preceding formula, we calculate the level array indices as follows:

```
i = int ((x - offsetX) / tileWidth);
j = int ((y - offsetY) / tileWidth);
```

There can be times at which the calculated values may fall out of valid array dimensions and may need to be ignored. Now let us try implementing a simple top-down view array-based level using Starling, based on the knowledge we have acquired so far.

Prerequisites for using the shared code

There are certain prerequisites that are necessary for following the programming done in this book. You will need to perform the following setup properly:

- ActionScript development IDE such as Flash Builder 4.7, FDT, FlashDevelop, or Flash IDE is required. Flash Builder is recommended, but if you are concerned with licensing, then the free FlashDevelop is ideal for the Windows platform.

- The latest source or compiled SWC of Starling should be added to the class path or should be linked. At the time of writing, the latest Starling version was 1.4.1.
- Download the art assets and the sample code for this book from the Packt website.

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.

Creating a top-down game scene

Let us try to implement a sample Starling-based top-down level with provision to detect the tile on which we touch. We will use the source and the tile sheet given in the `Chapter 1` folder from the shared source. The code for the `TopDownScene` class is shared in the following code snippet. You will need to initialize this class from the Starling document class using `new Starling(TopDownLevel, stage)`.

```
package com.csharks.juwalbose
{
  import starling.display.DisplayObjectContainer;
  import starling.display.Image;
  import starling.events.Event;
  import starling.events.Touch;
  import starling.events.TouchEvent;
  import starling.textures.Texture;
  import starling.textures.TextureAtlas;
  public class TopDownLevel extends DisplayObjectContainer
  {
    //Texture Atlas image
    [Embed(source="../../../../assets/assets.png")]
    public static const AssetTex:Class;
    //Texture Atlas XML
    [Embed(source = "../../../../assets/assets.xml", mimeType =
      "application/octet-stream")]
  private static const AssetXml:Class;
    private var texAtlas:TextureAtlas;
    private var tileWidth:uint=50;
    private var borderX:uint=25;
    private var borderY:uint=50;
    private var touch:Touch;
    //LevelData as 2 dimensional array
    private var levelData:Array=
      [["14","21","21","21","21","21","21","13","21","21","21",
        "21","21","21","17"],
      ["18","12","7","2","2","8","2","3","2","5","2",
        "2","7","13","20"],
      ["18","3","3","2","2","2","2","2","2","2","2",
        "2","3","2","20"],
      ["18","3","3","2","2","2","9","2","2","2","3",
        "2","3","3","20"],
      ["18","5","2","2","5","2","2","2","4","2","2",
        "2","2","5","20"],
      ["10","2","2","2","2","3","2","2","2","2","2",
        "2","7","2","12"],
      ["18","2","8","2","2","2","2","3","2","5","2",
        "2","2","5","20"],
      ["18","2","2","2","4","2","2","2","2","2","4",
        "2","2","2","20"],
      ["18","11","2","3","2","2","2","3","2","2","2",
        "2","2","10","20"],
      ["15","19","19","19","19","19","19","13","19",
        "19","19","19","19","19","16"]];
  public function TopDownLevel()
  {
    super();
    this.addEventListener(Event.ADDED_TO_STAGE, init);
  }

  private function init(e:Event):void
  {
  this.removeEventListener(Event.ADDED_TO_STAGE, init);
  //Let us create a texture atlas
  var tex:Texture =Texture.fromBitmap(new AssetTex(),false);
  var img:Image;
  texAtlas=new TextureAtlas(tex,XML(new AssetXml()));
  //Loop through 2D array & add images to stage
  for(var i:int=0;i<levelData.length;i++){//i is for rows
  for(var j:int=0;j<levelData[0].length;j++){//j is for columns
  img=new Image(texAtlas.getTexture(paddedName(levelData[i][j])));
  addChild(img);
  //calculate position depending on i & j
  img.x=j*tileWidth+borderX;
  img.y=i*tileWidth+borderY;
  }
```

```
    }
    //Listener for touch
    this.addEventListener(TouchEvent.TOUCH, onTouch);
    }

    private function paddedName(id:String):String{
      //function to return '0' padded name
      var offset:uint=10000;
      offset+=int(id);
      var str:String="tiles";
      str+=offset.toString().substr(1,4);
      return str;
    }
    private function detectTile(valueX:int, valueY:int):void{
      //function to detect tile at given position
      var tileX:int=int(valueX-borderX)/tileWidth;
      var tileY:int=int(valueY-borderY)/tileWidth;
      trace(levelData[tileY][tileX]);
    }
    protected function onTouch(event:TouchEvent):void
    {
      for (var i:int = 0; i < event.getTouches(this).length; ++i)
        {
          touch = event.getTouches(this)[i];
          detectTile(touch.globalX,touch.globalY);
        }
      }
    }
  }
```

Here we are using the Starling `TextureAtlas` class created from embedded assets. The embedded XML and PNG files are created using the `TexturePacker` application using the Sparrow/Starling export feature. The `PaddedName` function adds the zero padding to the tile names and the `onTouch` listener function traces out the tile value in the `levelData` array based on the touch position. The registration point of an image is at its top-left corner by default. This benefits us for placing them properly in the scene. Try working with this on your own by playing around with the level data. You can change the values in the level data and see the level scene updates accordingly. We have a total of 22 tiles to play with.

Applying the RenderTexture approach

In the preceding code, we have placed each tile as individual images on the scene. This will look fine but it is not necessarily the best approach. This creates a lot of independent entities on the scene, which may become a performance hog for a complicated game where we may need to manipulate each and every tile on every frame. An alternate approach is to use a `RenderTexture` class.

A `RenderTexture` class can be considered as a dynamically changeable Starling texture, which can be altered anytime via code. We can use methods to clear it, draw items onto it, or to fill it up completely. For our previous level scene, we can simplify everything by creating a single image based on a `RenderTexture` class, and then drawing all tiles onto it. This results in the scene having only one entity, thereby minimizing the effort on the Flash Player. Let us take a look at the changes in the code:

```
rTex=new RenderTexture(stage.stageWidth,stage.stageHeight);
var rTexImage:Image= new Image(rTex);
addChild(rTexImage);
for(var i:int=0;i<levelData.length;i++){
  for(var j:int=0;j<levelData[0].length;j++){
    img=new Image(texAtlas.getTexture
      (paddedName(levelData[i][j])));
    img.x=j*tileWidth+borderX;
    img.y=i*tileWidth+borderY;
    //draw to texture instead of adding new image
    rTex.draw(img);
  }
}
```

The `detectTile` function is also slightly altered to ignore invalid touch points which fall out at the side level area:

```
private function detectTile(valueX:int, valueY:int):void{
  var tileX:int=int(valueX-borderX)/tileWidth;
  var tileY:int=int(valueY-borderY)/tileWidth;
  if(tileX<0 ||tileY<0 ||tileX>levelData[0].length-1
  ||tileY>levelData.length-1)
  { //outside of tile area
    return;
  }
  trace(levelData[tileY][tileX]);
}
```

Use `RenderTexture` wherever possible for efficient resource management as well as performance boost. There is an additional method called `RenderTexture.drawBundled`, which can further speed things up considerably. We will be using this function a lot for our screen update.

Tools of the trade

We have successfully created our level scene based on a level data 2D array. There are different tools which we may need or which may help us to speed up our development. Let me mention a few of these which every game developer should consider having in their tool box:

- TexturePacker: This is a tool for packing images into texture atlases. It is very handy for resource management, size reduction, and memory optimization. A free version is available. Please note that the latest Flash Professional CC has an inbuilt spritesheet creator.
- Hiero: This is a free online Java-based application for creating bitmap fonts. Bitmap fonts will help to speed things up for game text which need frequent updating. You may also use standalone tools such as BMFont for Windows or Glyph Designer for OS X.
- Adobe Scout: This is a wonderful tool for profiling your AS3 applications and games from the stables of Adobe. This is a must have for any AS3 developer. It is a part of the Adobe Gaming SDK. You can get it for free now.
- Adobe ATF tools: These are a set of texture packing tools from Adobe, which help to minimize the memory consumption on handheld devices. This will be needed for our cross-platform deployment segment. It is a part of Adobe Gaming SDK.
- DragonBones framework: This is a Flash IDE extension for creating skeleton-based animations. It is a part of Adobe Gaming SDK.
- FeathersUI latest source or compiled SWC should be added to class path or should be linked. We will be using this excellent framework for all our UI needs.
- External debuggers such as Arthropod or De MonsterDebugger will facilitate bug spotting and fixing.

Summary

In this chapter, we learned about the basic structure of a game and got introduced to the game loop and game states. We learned that the Flash Player facilitates game development by providing its own game loop hook and doing scene updates per frame. I also explained to you the game idea, which we will be trying to develop throughout this book. The idea is a tile-based, isometric, and turn-based game. We understood about the turn-based game and got in-depth understanding of the tile-based approach.

We wrote code to create a level using the tile-based approach where the level data is stored as a 2D array. We also found out the relationship between screen space and array indices. Let us use the tile-based approach we have learned in this chapter to implement the isometric view in the next chapter. We will learn how a normal 2D view relates to the interesting pseudo 3D isometric view.

2
Going Isometric

The top-down view is fun, but there are other kinds of visual representations that can take our games to the next level. If we are able to render a complete 3D scene, then our game becomes much more realistic and engaging. Creating 3D games is a topic for another book, but don't worry, we have the next best thing—**Isometric Game View**. An isometric view can be considered as a pseudo-3D implementation of our tile-based game, which adds 3D-like depth to the scene.

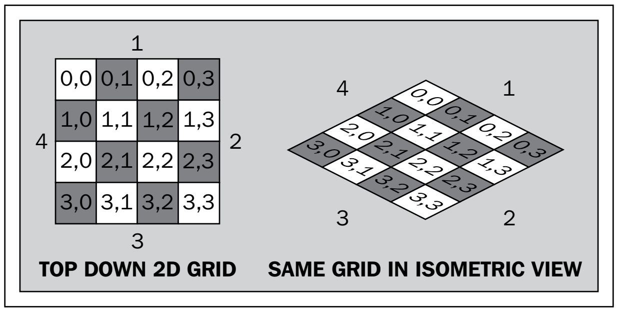
In this chapter, we will cover the following topics:

- Learning about isometric view
- Learning about the relation between 2D Cartesian and isometric coordinates
- Implementing a sample level using conversion equations and helper classes
- Explaining and understanding isometric movement and depth swapping
- Creating a level data structure to effectively store isometric level details

What is an isometric view?

A simple explanation can be that an isometric view is the view of a camera mounted at the top corner of a room, with the camera tilted 30 degrees towards the floor and 45 degrees from either wall. A perfect square room will be rhombus shaped, which will be exactly twice the height wide.

The following figure explains the view and helps you understand how an indexed array will look in an isometric projection:



If you have never tried creating isometric views, you may be wondering how it can be done. At first look, there won't be any obvious solution for placing tiles onto an isometric grid, and you will be thinking of using complicated equations involving complex trigonometry for this purpose. Do not think too much, as the equations are very simple, although not very obvious unless explained.

Cartesian to isometric equations

A very important thing to understand here is that the level data still remains the same 2D array, and we will be altering only the rendering process. Later on, we will need to update the level data to accommodate large tiles, which will contain items that are bigger than the current tile size. Our two - dimensional top-down coordinates for a tile can be called **Cartesian coordinates**. The relationship between Cartesian and isometric coordinates is shown in the following code:

```
//Cartesian to isometric:
x_Iso = x_Cart - y_Cart;
y_Iso = ( x_Cart + y_Cart ) / 2;

//Isometric to Cartesian:

x_Cart = ( 2 * y_Iso + x_Iso ) / 2;
y_Cart = ( 2 * y_Iso - x_Iso ) / 2;
```

Now that is very simple isn't it? We will use an `IsoHelper` class for this conversion where we can pass through a point and get back to the converted point.

An isometric view via a matrix transformation

Although the equations are simple and straightforward, the art needed for an isometric tile is a bit complicated. The artist needs to create the rhombus-shaped tile art with pixel precision and mostly tileable in all four directions. An alternative approach is to use the square tile itself and skew them dynamically using the corresponding code. Let us try to create the isometric view for the level data explained in the previous chapter with the same tiles using this approach.

The transformation matrix for isometric transformation is as follows, which is essentially a rotation of 45 degrees and scaling by half in Y axis:

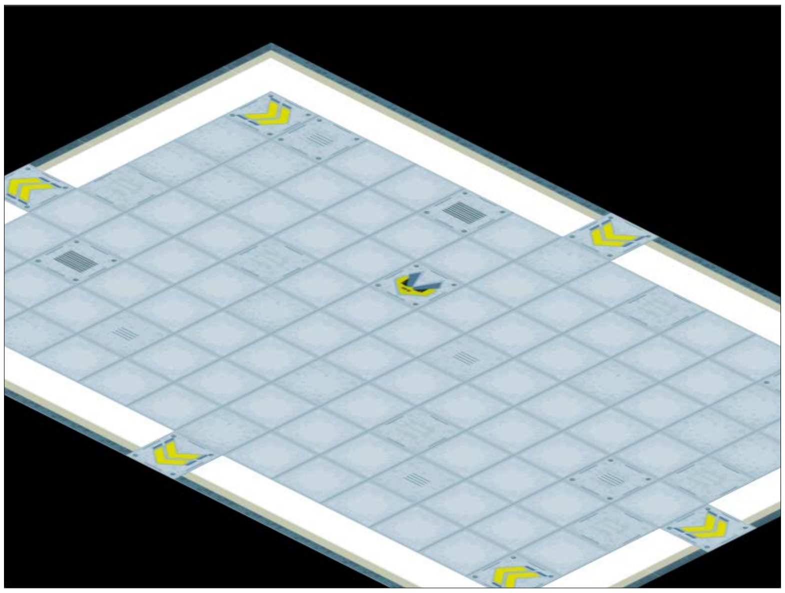
```
var m:Matrix = new Matrix(1,0.5,-1,0.5,0,0);
```

[You can explore more on matrices at the Adobe docs available at `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/geom/Matrix.html`.]

The code for the `IsometricLevel` class, which is also in the `Chapter 1` folder, is shared as follows. You should initialize this class from the Starling document class using `new Starling (IsometricLevel, stage)`. The following approach just applies the isometric transformation matrix to the `RenderTexture` image. Minor changes in the `init` function are shown in the following code:

```
var m:Matrix = new Matrix(1,0.5,-1,0.5,0,0);
for(var i:int=0;i<levelData.length;i++){
  for(var j:int=0;j<levelData[0].length;j++){
    img=new
      Image(texAtlas.getTexture(paddedName(levelData[i][j])));
    img.x=j*tileWidth+borderX;
    img.y=i*tileWidth+borderY;
    rTex.draw(img);
  }
}
m.translate( 300, 0 );
rTexImage.transformationMatrix = m;
```

We apply the transformation matrix to the `RenderTexture` image and translate it by 300 pixels so that the whole of it is visible. Skewing will make a part of the image to be out of the visible area of the screen. We will get the following result:



An alternate approach is to apply the transformation matrix to each individual tile image, find the corresponding isometric coordinates, and move and place individual tiles accordingly as shown in the following code:

```
var m:Matrix = new Matrix(1,0.5,-1,0.5,0,0);
var pt:Point=new Point();
for(var i:int=0;i<levelData.length;i++){
  for(var j:int=0;j<levelData[0].length;j++){
    img=new
      Image(texAtlas.getTexture(paddedName(levelData[i][j])));
    img.transformationMatrix = m;
    pt.x=j*tileWidth+borderX;
    pt.y=i*tileWidth+borderY;
    pt=IsoHelper.cartToIso(pt);
    img.x=pt.x+300;
    img.y=pt.y;
    rTex.draw(img);
  }
}
```

Here, we use the convenient `cartToIso(pt)` conversion function of our `IsoHelper` class to find the corresponding isometric coordinates to our Cartesian coordinates. We are offsetting the drawing by 300 pixels to handle the skewing offset for the image.

This approach will work in some cases, but not all top-down tiles can be simply skewed and made into an isometric tile. For example, consider a tree in the top-down view, it will simply look like a skewed tree graphic after we apply the isometric transformation. So, the right approach is to create an isometric tile art specifically and use isometric equations to place them correctly. Let us use the isometric tiles provided in the assets pack to create a sample level.

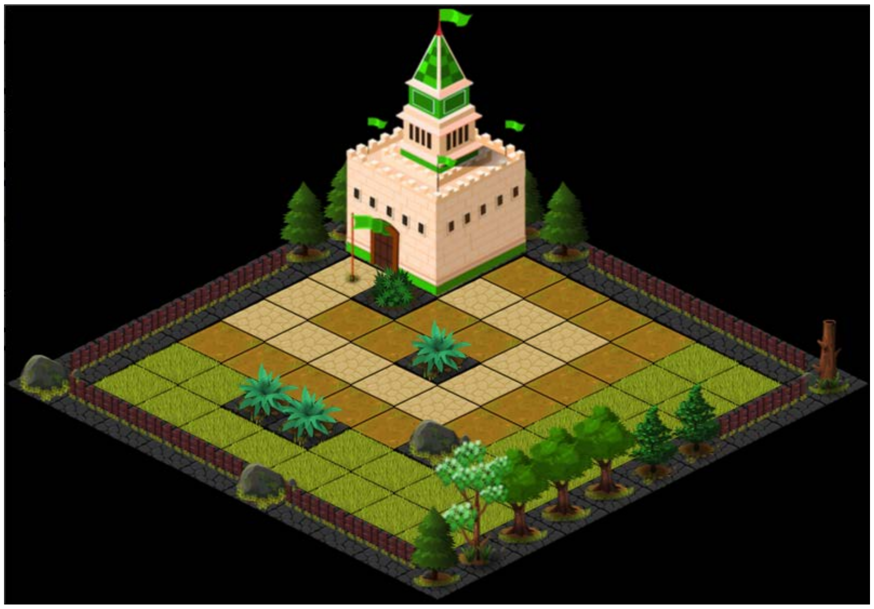
Implementing the isometric view via isometric art

Please refer to the `SampleIsometricDemo` source folder, which implements a sample level of our game using isometric art and the previously mentioned equations. There are some differences in the approach that I will be explaining in the following sections. Most of it has to do with the change in level data, altering the registration point of larger tiles, and handling depth. We also need to offset the image drawing so that it fits in the screen area. We use a variable called `screenOffset` for this purpose. The render code is as follows:

```
var pt:Point=new Point();
for(var i:int=0;i<groundArray.length;i++){
  for(var j:int=0;j<groundArray[0].length;j++){
    //draw the ground
    img=new
      Image(texAtlas.getTexture(String(groundArray[i][j]).split
        (".")[0]));
    pt.x=j*tileWidth;
    pt.y=i*tileWidth;
    pt=IsoHelper.cartToIso(pt);
    img.x=pt.x+screenOffset.x;
    img.y=pt.y+screenOffset.y;
    rTex.draw(img);
    //draw overlay
    if(overlayArray[i][j]!="*"){
      img=new
        Image(texAtlas.getTexture(String(overlayArray[i][j]).
          split(".")[0]));]));
      img.x=pt.x+screenOffset.x;
      img.y=pt.y+screenOffset.y;
```

```
          if(regPoints[overlayArray[i][j]]!=null){
            img.x+=regPoints[overlayArray[i][j]].x;
            img.y-=regPoints[overlayArray[i][j]].y;
          }
          rTex.draw(img);
        }
      }
    }
```

The result is shown in the following screenshot:



Level data structure

The level data for our isometric level is not just a simple 2D array with index numbers any more, but a combination of multiple data structures. We have a 2D array for the ground tiles, another 2D array for overlay tiles, and a dictionary to store altered registration points of the overlay tiles. Ground tiles are those tiles which exactly fit the isometric tile dimensions, which in this case is 80 x 40, and makes up the bottom-most layer of the isometric level. These tiles won't take part in any depth sorting as they are always rendered below all other items that populate the level.

Overlay tiles are items which may not fit into the isometric tile dimensions and have height, for instance, buildings, trees, bushes, rocks, and so on. Some of these can be fit into tile dimensions, but are kept as such that we have various advantages using the following approach:

- We are free to place an overlay tile over any ground tile, which adds to flexibility
- We would need a lot of tiles if we try to fit overlay tiles and ground tiles together for all permutations and combinations
- Effects such as tinting can be applied independently to the overlay tiles
- Depth handling becomes much easier
- Overlay tiles which are smaller than the tile size reduce the game size

Altering registration points

Starling considers all images as rectangular blocks with their registration point at the top-left corner. The registration point is the point which can be considered as the (0,0) of that image. Traditional Flash had given us the capability to alter the registration points by embedding images inside `Sprite` or `MovieClip`. We can still do the same, but it will require unnecessary creation of a lot of `Sprites`. Alternately, we can use the `pivotX` and `pivotY` properties of Starling objects for the same result too. In our isometric level, we will need to precisely place overlay tiles inside the isometric grid space. An overlay tile does not have any standard size as it can be any item— a tree, building, character, and so on. So, placing them correctly is a tricky thing and very specific to the tile concerned.

This leads us to have independent registration points for each overlay tile. We use a dictionary structure to save these values and use those values as offsets while placing overlay tiles. For example, we need to place a bush image, `nonwalk0009.png`, exactly at the middle of an isometric grid, which means moving it 12 pixels to the left and 19 pixels to the top for proper alignment. We save (12,19) as a new point inside our dictionary for ID `nonwalk0009.png`, as follows:

```
regPoints["nonwalk0009.png"]=new Point(12,19);
```

Finding a tile's precise placement point needs to involve visual interaction; hence, we will build a level editor, which makes this easier.

Depth sorting

An isometric view needs us to handle the depth of items manually. For ground tiles, there is no depth issue as they always form the lowest layer over which all the overlay items and characters are drawn. But overlay tiles and characters need to be drawn at specific depths for it to look appropriate. By depth, I mean the order at which the images are drawn. An image drawn later will overlap the one drawn earlier, thereby making it seem in front of the latter. For a level which does not change or without any moving items, we need to find the depth only once for the initial render. But for a level with moving characters or vehicles, we need to find every frame in the game loop and render.

The current sample level does not change over time, so we can simply render the level by looping through the array. Any overlay item placed at a higher I or J value will be rendered later, and hence will be shown in front, where I and J are array indices. Thus, items placed at higher indices appear closer to the camera, that is, for the same I, a higher J is closer to the camera and vice versa.

[You may go ahead and explore the painter's algorithm available at `http://en.wikipedia.org/wiki/Painter's_algorithm`.]

When we have a moving item, we need to find the corresponding array position it occupies based on its current screen position. By using these new found array indices, we can compare with the overlay tile's indices and decide on the drawing sequence. The code to find array indices from the screen position is as follows:

```
//capture screen position
var screenPos:Point=new Point(hero.x,hero.y);
//convert to cartesian coordinates
var cartPos:Point=IsoHelper.isoToCart(screenPos);
//find tile indices from cartesian values
var tilePos:Point=IsoHelper.getTileIndices(screenPos,tileWidth);
```

Understanding isometric movement

Isometric movement is very straightforward to implement. All we need to do is move the item in top-down Cartesian coordinates and draw it on the screen after converting into isometric coordinates. For example, if our character is at a point, `heroCart` in the Cartesian system, then the following code moves him/her to the right:

```
heroCart.x+=heroSpeed;
//convert to isometric coordinates
heroIso=IsoHelper.cartToIso(heroCart);
```

```
heroImage.x=heroIso.x;
heroImage.y=heroIso.y;
rTex.draw(heroImage);
```

Detecting isometric collision

Collision detection for any tile-based game is done based on tiles. When designing, we will make sure that certain tiles are walkable while certain tiles are nonwalkable, which means that the characters can move over some tiles but not over others. So, when we calculate the movements of any character, we first make sure that the character won't end up on a nonwalkable tile. Thus, after each movement, we check if the resulting position falls in a nonwalkable tile by finding array indices as mentioned previously. If the result is true, we will ignore the movement, else we will proceed with the movement and update the on-screen character position.

```
heroCart.x+=heroSpeed;
//find new tile point
var tilePos:Point=IsoHelper.getTileIndices(heroCart,tileWidth);
//this checks if new tile position is occupied, else proceeds
if(checkWalkable(tilePos)){
  //convert to isometric coordinates
  heroIso=IsoHelper.cartToIso(heroCart);
  heroImage.x=heroIso.x;
  heroImage.y=heroIso.y;
  rTex.draw(heroImage);
}
```

You may be wondering that the hero character should need some special considerations to be drawn correctly as the right depth, but by the way we draw things, it gets handled automatically. We do not allow the hero to move onto a nonwalkable tile, that is, bushes, trees, and so on. So, any tile remains walkable or nonwalkable. The character gets drawn on top of a walkable tile, which does not contain any overlay items, and hence it will occupy the right depth.

In this method, a full tile has to be made either walkable or nonwalkable, but this may not be the case for all games. We may need to have tiles, which block entry from a specific direction or block exit in a particular direction as a fence along one border of a tile. In such cases, the tile is still walkable, but valid movement is also checked by tracking the direction in which the character is moving. For our game, the first method will be used along with the four-way freedom of movement.

In an isometric view, movement can be either in four directions or eight directions, which in turn is called a four-way movement or an eight-way movement respectively. A four-way movement is when we move along the X or Y axis alone on the Cartesian space. An eight-way movement happens when, in addition to four - way, we also move the item diagonally. Logic still remains the same.

> You are advised to check out my online tutorials on isometric world creation available at **GameDevTuts+** for more details:
>
> `http://gamedevelopment.tutsplus.com/tutorials/creating-isometric-worlds-a-primer-for-game-developers--gamedev-6511`
>
> `http://gamedevelopment.tutsplus.com/tutorials/creating-isometric-worlds-a-primer-for-gamedevs-continued--gamedev-9215`

Summary

In this chapter, we learned about the isometric projection and the equations that help us to implement it based on the simpler Cartesian system. We implemented a sample isometric level using isometric art as well as learned about matrix-based fake isometric rendering. We analyzed the `IsoHelper` class, which facilitates easy conversion between Cartesian and isometric coordinates and also helps in finding array indices.

We learned why altering the registration points is essential for perfectly placing the overlay tiles and we found that our level data needs to track these registration points as well. We also learned how depth sorting, collision detection, and isometric movement are done based on our tile-based approach. Moving forward, we will try to create a level editor to simplify our level creation process.

3
Isometric-level Designer

Isometric games look cool and almost make us believe that it is a 3D game. Creating these isometric levels can be a very demanding process, unless you rely on some tools to make things easier for you. There are tools such as **Tiled**, **Mappy**, **TileStudio**, and **DAME** which helps us with level designing, but there are not many professional tools for isometric-level designing. Dame and Tiled are the ones capable of handling isometric designs among which Tiled is the easiest one to understand. Let us try creating our own simple solution, which can be easily updated for use with any flat isometric-level design.

In this chapter, we will cover the following topics:

- Befriending **Feathers UI**
- Creating AIR desktop applications, which will work on Windows, Linux, and Mac
- Creating the isometric-level designer
- Adding the capability to export level data and capture designed scenes
- Using Starling graphics extension to create isometric grids
- Adding the capability to visually edit registration points

The approach for creating our level designer

We will create an AIR desktop application, which can work across all machines supporting AIR. Using AIR helps us get access to files that are loading and saving. We will be using file access to:

- Dynamically load all tile images from a folder
- Save the level data in the form of a text file

- Capture the designed level as screen grab for comparing with a coded level later on

From the previous chapter, we know that we need a ground-layer array, an overlay-layer array, a registration points dictionary, and a screen offset point value for effectively displaying any flat isometric level. By flat, I mean that our isometric level has a single ground level and there are no elevated tiles. For this purpose, the level editor will require the following minimal features:

- Ability to load all tile images from a folder recursively
- Ability to paint the ground layer
- Ability to paint the overlay layer
- Ability to fill a layer with any selected tile, which is called **flood filling**
- Ability to export created level data
- Ability to screen grab the design
- Ability to reset each layer
- A visible isometric grid for reference
- Ability to edit the registration point of nonstandard tiles and export that data too

The code for the designer app is available in the `Chapter 4` folder.

Befriending Feathers UI

If you have not heard about Feathers UI, then I implore you to check it out at `http://feathersui.com/`. Feathers UI provides Flex or fl.controls - like components which save a lot of work. Created by Josh Tynjala, who is a very active figure in the Starling community—this one is a must have for everyone's code repository. In the code, we have added Feathers UI and the aeon desktop theme source files as a dependency and also copied the graphic assets into the assets folder. Once the Starling root is created, we initialize the theme to set it up, as follows:

```
new AeonDesktopTheme();
```

We will be using `Button`, `ImageLoader`, `ScrollContainer`, `Panel`, and `Radio` controls from Feathers UI.

Creating an isometric grid with the Starling graphics extension

The Starling graphics extension enables drawing APIs, which were originally available in the flash display list `Sprite` class. We will use the `Shape` class to create our isometric grid from the `IsoHelper` class, as shown in the following code:

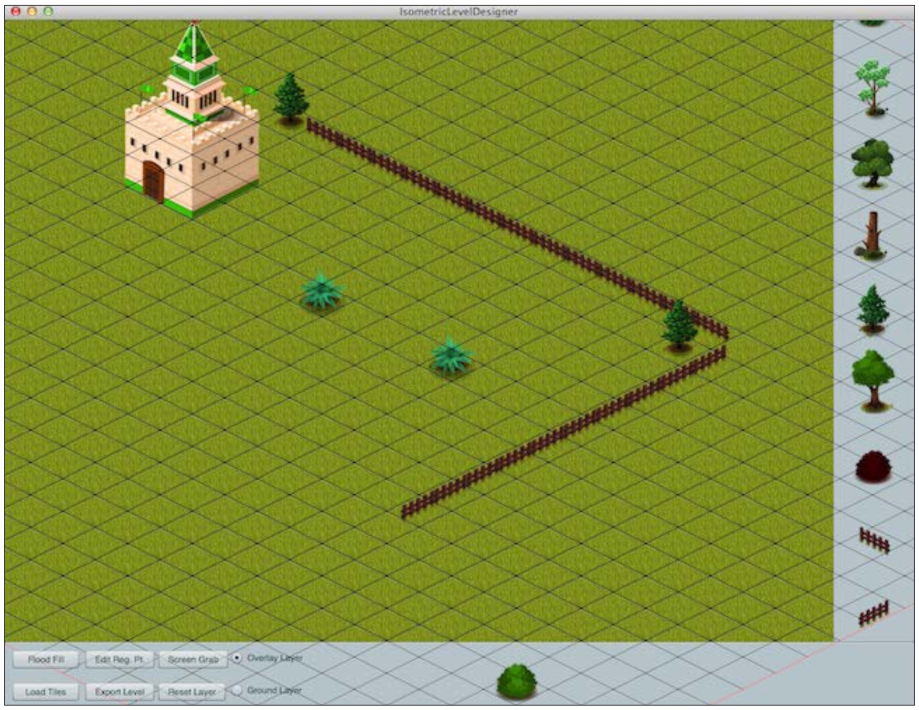
```
public static function createIsoGrid(cols:uint,rows:uint,tileWidth:Num
ber,tileHeight:Numb
  er):Shape{
    var tempSprite:Shape=new Shape();
    var d:Point=new Point(0,tileHeight*rows);
    var c:Point=new Point(tileWidth*cols,tileHeight*rows);
    var b:Point=new Point(tileWidth*cols,0);
    var a:Point=new Point(0,0);

    tempSprite.graphics.lineStyle(1,0x000000);
    var srcPt:Point=new Point(0,a.y);
    var destPt:Point=new Point(0,0);
    for(var i:uint=0;i<=cols;i++){
      if(i==0||i==cols){
        tempSprite.graphics.lineStyle(1,0xff0000);
      }else{
        tempSprite.graphics.lineStyle(1,0x000000);
      }

      srcPt.x=a.x+(i*tileWidth);
      srcPt.y=a.y;
      destPt.x=d.x+(i*tileWidth);
      destPt.y=d.y;
      srcPt=IsoHelper.cartToIso(srcPt);
      destPt=IsoHelper.cartToIso(destPt);
      tempSprite.graphics.moveTo(srcPt.x,srcPt.y);
      tempSprite.graphics.lineTo(destPt.x,destPt.y);
      }
      for(i=0;i<=rows;i++){
        if(i==0||i==rows){
          tempSprite.graphics.lineStyle(1,0xff0000);
        }else{
          tempSprite.graphics.lineStyle(1,0x000000);
        }
      srcPt.x=d.x;
      srcPt.y=a.y+(i*tileHeight);
      destPt.x=b.x;
```

```
          destPt.y=b.y+(i*tileHeight);;
          srcPt=IsoHelper.cartToIso(srcPt);
          destPt=IsoHelper.cartToIso(destPt);
          tempSprite.graphics.moveTo(srcPt.x,srcPt.y);
          tempSprite.graphics.lineTo(destPt.x,destPt.y);
        }
      return(tempSprite);
    }
```

In the preceding code, we find the four corners of the isometric plane and draw lines updating the columns and rows in a loop. There is another helper function called `createFilledSingleIsoGrid`, which returns a single filled isometric grid.

Developing a designer app

The `com.adobe` package in the source code has the necessary classes for PNG encoding, which will help us to do our screen grab. The `com.csharks.juwalbose` package contains the `IsoHelper` class and the `Designer_v4` class, which is the application. The `v4` is just for effective management at my end for code tracking as we internally use a similar tool which is designated `v3` that uses Flex for the same purpose, but with much enhanced features. Have a look at the following screenshot:



We use two `Rendertexture` instances for the two layers that constitute our level scene, the `groundCanvas` and `topCanvas classes` that correspond to the two arrays in our level data, `groundArray` and `overlayArray`. These are added to the scene while we initialize their corresponding arrays with a dummy data, `*` for all positions, which essentially means that unless there is a tile, the value stored will be `*`. The size of the arrays is 33 x 33 to effectively cover our 1024 x 768 screen real estate.

Loading files recursively

We need to load all image files from within a folder into a container from which we can easily select any tile to draw our level. Images may be in the JPG or PNG format and may reside within the subfolders. The code will consider these situations as follows:

```
private function getFilesRecursive(folder:File):void
{
  var contents:Array = folder.getDirectoryListing();
  var tempS:String;
  for (var i:uint = 0; i < contents.length; i++)
  {
    if (contents[i].isDirectory)
    {
      if (contents[i].name != "."  && contents[i].name != "..")
      {//it's a directory
        getFilesRecursive(contents[i]);
      }
    }
    else
    {
      tempS = contents[i].].name.toLowerCase();
      if (tempS.substr(tempS.length - 3,3) == "jpg" ||
        tempS.substr(tempS.length - 3,3) == "png")
      {
        folderContents.push(contents[i]);
      }
    }
  }}
```

File references are pushed into the `folderContents` array for later use. We can use the extension property to determine the file type.

Displaying tiles

Once the process is complete, we use the `populateTiles` method to load the images into an `ImageLoader` class which gets added to the right of `ScrollContainer`. Each `ImageLoader` instance is named as the index of the file in the `folderContents` array, enabling a tricky tracking logic.

```
for (var i:uint=0; i<folderContents.length; i++)
  {
    imgLoader=new ImageLoader();
    imgLoader.width=imgLoader.height=80;
    imgLoader.source=folderContents[i].url;
    container.addChild(imgLoader);
    imgLoader.name=i.toString();

    if(i==0){
      selectItem(imgLoader);
    }
  }
```

By default, we use the first image as the default stencil texture for painting the level. Stencils are created when we tap on any of the `ImageLoader` instances in the `ScrollContainer` control by calling the `selectItem` method as shown in the preceding code. The associated code also highlights the selected tile and adds it at the bottom for us to see.

```
private function selectItem(target:ImageLoader):void
{
  if(selectedItem!=null){
    selectedItem.color=0xffffff;
  }
  selectedItem=target;
  selectedItem.color=0x00ff0033;
  loadStencil(selectedItem);
}
private function loadStencil(img:ImageLoader):void
{
  urlReq = new URLRequest(String(img.source));
  loader.load(urlReq);
}
private function imgLoaded(e:flash.events.Event):void
{
  stencil.dispose();
  stencil=Texture.fromBitmap(loader.content as Bitmap,
  false,true);
```

```
    stencilImage.texture=stencil;
    stencilImage.readjustSize();
    stencilLoaded = true;
    stencilImage.y=800-stencilImage.height/2;
  }
```

The `ScrollContainer` instance has the interaction mode set for the mouse as follows:

```
container.interactionMode=Scroller.INTERACTION_MODE_MOUSE;
```

[You may also use a list for displaying the tiles.]

Painting the level

Level drawing is initiated from the `TouchEvent` listener, `onTouch`, which proceeds only if we have a stencil loaded and have touched inside the painting area. This method also switches stencils when we tap on the tiles on the right. The touch point is converted to find the corresponding tile indices in the currently selected layer, as shown in the following code:

```
var pt:Point = touch.getLocation(grid);
pt = IsoHelper.getTileIndices(IsoHelper.isoToCart(pt),tileWidth);
changeArray(pt);
```

The `ChangeArray` method stores the stencil file name in the array position indicated by the converted point and calls the `drawTile` method, as shown in the following code:

```
private function changeArray(pt:Point):void
{
  if (pt.x > -1  && pt.x < groundArray[0].length  && pt.y > -1  &&
  pt.y < groundArray.length)
  {
    var savedItemString="\"" +
      folderContents[selectedItem.name].name + "\"";
  if(bottomLayer.isSelected){
    groundArray[pt.y][pt.x] =  savedItem;
  }else if(topLayer.isSelected){
    topArray[pt.y][pt.x] =  savedItem;
  }
  drawTile(pt.x,pt.y,true);
  }
}
```

Note the way we use the name of `selectedItem`, which is the `ImageLoader` control to point to the file reference saved in the `folderContents` array.

The `drawTile` function performs the actual drawing onto the render textures as shown in the preceding code.

```
if(bottomLayer.isSelected){
  groundCanvas.draw(stencilImage,matrix);
}else if(topLayer.isSelected){
  topCanvas.draw(stencilImage,matrix);
}
```

Flood filling is done by setting all values in the corresponding layer array to the tile name and drawing the tile at all valid tile positions. The `ResetLayer` method resets all array values to `*` and clears the corresponding `RenderTexture` image.

Saving the level

Level saving is done as a simple text file by concatenating string values from the corresponding data in the `saveTxtData` method. This would have been more professional if done as a JSON file or XML, but I will leave it for you to decide.

Screen grabbing makes use of a latest feature in Starling where we can save the stage as a `BitmapData` object. The code is simple, as follows:

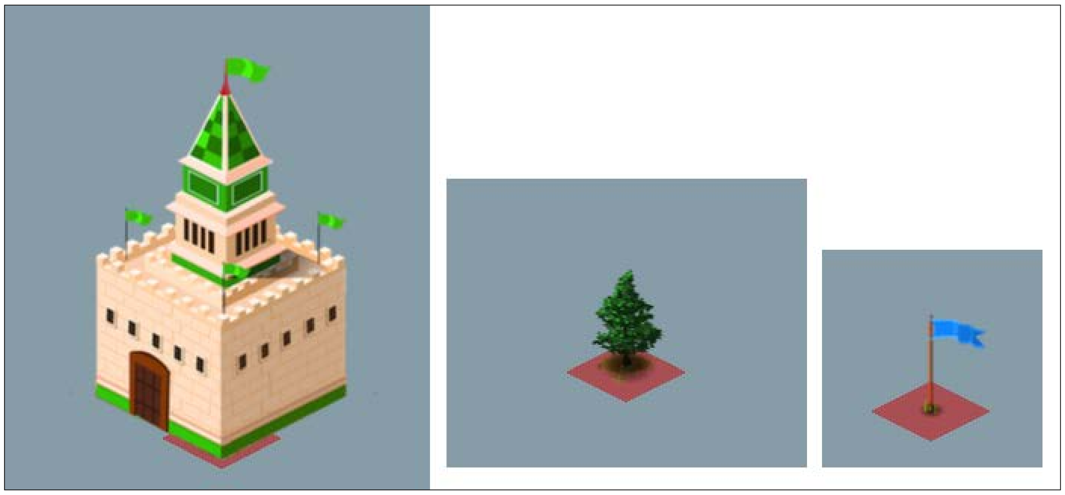
```
var imageByteArray:ByteArray =
  PNGEncoder.encode(this.stage.drawToBitmapData());
```

Editing registration points

For altering the registration point, we toggle visuals on the screen to hide the painting area and add a single isometric grid with the nonstandard tile at its default position altered by the already stored registration point value from the `regDict` dictionary. The single isometric grid serves as a reference point for proper placement. Dragging the tile will make it move to the desired position. While in this mode, the `onTouch` listener will perform differently by moving the nonstandard tile based on the mouse movement.

```
nonStdTile.x = touch.globalX - nonStdTile.width / 2;
nonStdTile.y = touch.globalY - nonStdTile.height / 2;
```

Nonstandard tiles need to be placed in such a way that they look correctly standing on the ground, as shown in the following figure:



When we hit the done button, we save the current altered registration point position as shown in the following code, where `regPtZero` indicates a fixed position on the stage which is considered as the origin (0,0).

```
regDict[folderContents[selectedItem.name].name]=new
  Point(int((nonStdTile.x - regPtZero.x)*10)/10,int((regPtZero.y -
    nonStdTile.y)*10)/10);
```

Summary

In this chapter, we created an isometric-level designer application with which we can create our level visually and interactively. We explored how to load files recursively and how to use the new stage grab feature of Starling. We used Feathers UI for our UI components and created Shapes using the Starling graphics extension.

You found out how we need to alter the registration points of each tile and store them in a dictionary to be exported along with the level data as a plain text file. This application can be considered as a base to expand on adding more features and eventually creating your own custom full-fledged level designer. I would suggest you to add the ability to load a level data and edit it. Moving forward, we will get started with developing our game and learning path findings and basic AI.

4
Just Finding Your Way

With the help of the level editor, we are able to create any number of levels with minimal effort. All we need to do is let the level designer visually create the levels and export the data in an usable format. Our game plays out in a single level specifically tailored to have the maximum space to move about with two color-coded bases at two distant places. I have shared the source for a demo in the `Chapter 4` folder, which has the game scene basics implemented.

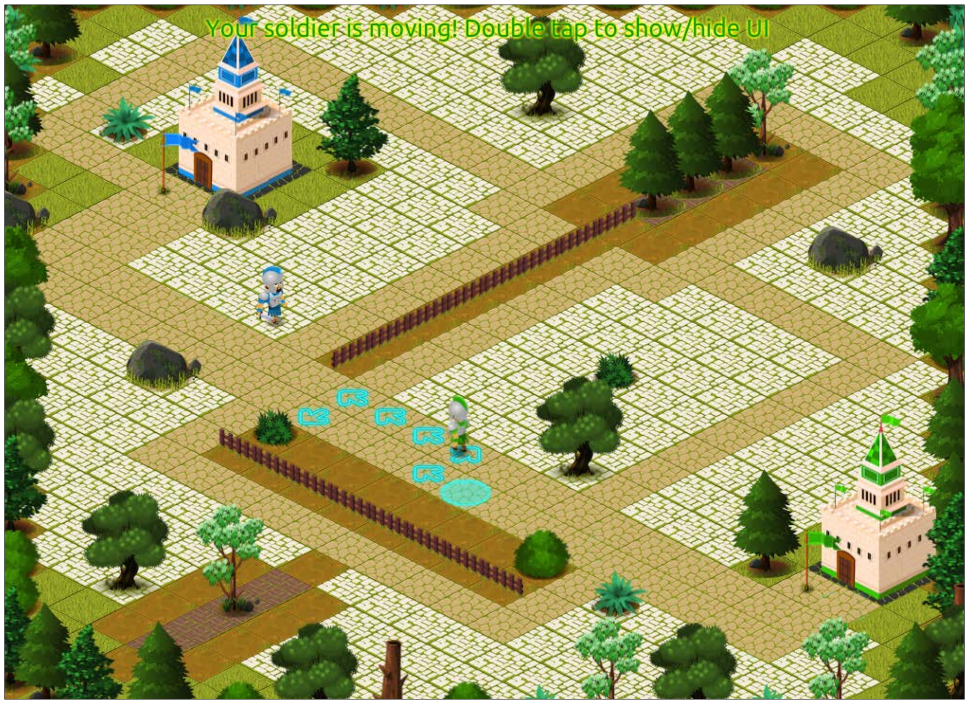
In this chapter, we will cover the following topics:

- Implementing our isometric level
- Placing soldiers, drawing paths for them, and making them follow the paths
- Learning about the path findings
- Using the `pathfinder` class to find a path and making the enemy soldiers to follow it
- Implementing the game UI
- Implementing the whole turn-based game logic based on the game states
- Implementing very basic **AI** (**Artificial Intelligence**)
- Learning to create an alternative class for `MovieClip` for better control

The Flag Defense game level

Using the level editor, I have exported the level data in the form of two arrays for ground and overlay tiles, and a dictionary for storing altered registration points.

The following screenshot shows the game level:



We won't be going through the whole code line by line as that would be a lot of code and much of it will be self-explanatory for AS3 developers. I will explain the concepts and the tricky parts of the code, and would expect all of you to go through the whole project as well as all the associated classes to understand the complete implementation.

Our game is named as **Flag Defense**. We will start with the implementation of the Web version of the game first and later on we will re-use the code to create a cross-platform version based on the Adobe AIR technology for iOS and Android.

The game can be checked out at the following links:

- Android: `https://play.google.com/store/apps/details?id=air.com.csharks.flagdefense&hl=en`
- Web: `http://onlineindiangames.com/oig/game/2774/Flag-Defense.html`
- iOS: `https://itunes.apple.com/us/app/flag-defense/id754518884?ls=1&mt=8`

Game settings

Every game involves a bit of homework and planning before we even start the project. For our game, we need to decide on the resolution, size of the tile, size of the array, and the corresponding art. As you may have already seen the game art, you may have deduced our approach, but the process is always the other way around. We have decided to set a 1024 x 768 resolution for our web game. Deciding on a 2D top-down tile size of 40 in turn means an 80 x 40 tile in an isometric space.

Note that the isometric tile is two times wide and of the same height as the top-down tile. This can be tested by applying the isometric matrix transformation to a top-down tile as detailed in the *Chapter 2*, *Going Isometric*.

The preceding numbers mean that in order to create a level which fills the entire screen, we will need a very specific number of tiles.

```
Number of columns = 1024 / 80 = 12.8 ~= 13
Number of rows = 768 / 40 = 19.2 ~= 20
```

This means that we will need level data with arrays of `13` x `20`.

We have also used an FPS of 30, which should be fine for a cross-platform project. The background color is chosen as `#004400` for matching with the predominant grass color used for the ground tiles. This helps to fill up any minor gaps that may pop up due to the pixel variations in the tile sizes.

The support classes used

If you have not yet explored the project files that were provided, please do it now. The following is the list of the major classes used and their basic purposes:

- `IsoHelper`: This is a helper class for easy isometric and Cartesian conversion
- `ResourceManager`: This class loads assets using the default `AssetManager` class
- `GameEvent`: This is a custom event which helps in passing a custom parameter
- `CustomAnimatedItem`: This class is an alternative to the `MovieClip` class
- `CustomIsometricCharacter`: This is a base class for isometric characters based on the `CustomAnimatedItem` class
- `GameStates`: This class includes reference of all game states

- `EmbeddedAssets`: This class embeds the texture sheet, Atlas XML, and TrueType font
- `MainLevel`: This is the Starling root class which controls the gameplay
- `WorldLayer`: This class allows `RenderTexture`-based game level implementation
- `GroundLayer`: This class allows `RenderTexture`-based ground implementation

Other classes are self-explanatory as it is evident from their names and they mostly deal with the graphic elements used in the game. Please make sure that you look into the source code provided while going through the chapters from here on. It is advised to have the code open in your editor while you are reading for your easy reference.

The interesting point to note is that we have two different `RenderTexture` classes rather than one as used in earlier examples. The reason is a matter of efficiency and optimization. The ground layer seldom changes and won't need redrawing most of the time. The overlay layer will require a redraw each time some soldier moves. If we only had a single `RenderTexture` class, then we would need to redraw everything. Now with two `RenderTexture` classes, we can keep the whole ground layer on one and draw it onto the world layer while updating the scene. This saves us a lot of time as we don't need to draw the ground tile by tile again.

Faster drawing for RenderTextures

Drawing tiles onto a `RenderTexture` class is a heavy process and we would want to speed it up as much as possible as this may need to be done on every frame. The `RenderTexture` class has a convenience function named `drawBundled()`, which can speed things up, as explained in the following code, from the `GroundLayer` class's drawing function:

```
public function drawGround():void {
  this.drawBundled(function():void{
    for(var i:int = 0; i<rows; i++) {
      for(var j:int = 0; j<cols; j++) {
        img = new Image(assetsManager.
          getTexture(String(groundArray[i][j]).split(".")[0]));
        pt.x = j*tileWidth;
        pt.y = i*tileWidth;
        pt = IsoHelper.cartToIso(pt);
        img.x = pt.x + screenOffset.x;
        img.y = pt.y + screenOffset.y;
```

```
            this.draw(img);
          }
        }
      });
    }
```

The variable `pt` is a global point instance. The same is applied in the `WorldLayer` class's `render` function which draws the ground layer, the soldiers, and the overlying tiles.

The CustomAnimatedItem class

For the usual purpose of displaying an animated item, we all can use the default Starling `MovieClip` class. For creating an animated character, we will need to make sure that we have a method to control the different animations it can perform. For example, our character needs the animations such as idle, walk, attack, die, idle with blue flag, idle with green flag, walk with blue flag, walk with green flag, attack with blue flag, and attack with green flag. All these animations are needed for four directions as our game is isometric. Considering that we have two types of characters, green and blue, this amounts to handling a lot of movie clips per character. So, I have created a new class which skips the use of `MovieClip` and uses the `Image` class instead.

`CustomAnimatedItem` is a core class which extends Starling's `Image` class and changes its texture according to the frame rate of the animation. It will dispatch an event when a particular animation is complete. We can call its `update` function per frame to advance the animation. It also has a provision to have multiple animations, which can be swapped at ease as we did with the `CustomIsometricCharacter` class. The following code explains the creation of a `CustomAnimatedItem` class:

```
var character:CustomAnimatedItem = new
  CustomAnimatedItem("char",70,50,4,0.05,true);
```

The parameters are respectively image name, image width, image height, image name padding, frame delay, and whether to dispatch an event or not.

The images for each frame will follow a particular naming convention for this to work correctly; in the previous case it will be `char0001.png` or `char0006.png` as we have `4` as the zero padding value.

The CustomIsometricCharacter class

The `CustomIsometricCharacter` class extends the `CustomAnimatedItem` class to add much more functionality required for our isometric character. We will be following the given naming convention for our character's animation frame images:

```
newFrameName =
  baseImageName+"_"+state+"_"+facing+"_"+paddedString;
```

For example, for a character named Mario with a frame for walking down with a zero padding of `4`, the naming will be `mario_walk_down_0001.png`. This enables us to easily switch the animations by changing the `state` property of the class and also change the facing direction by changing the `facing` property.

It has a 2D tile coordinate named `paintPoint`, which indicates the tile coordinates that it occupies in the 2D array. The corresponding Cartesian coordinate is `paintPoint2D` and the respective isometric coordinate is `paintPointIso`. The `update` method calculates `paintPointIso` as per the current `paintPoint2D` value. For moving the character, we just increment/decrement the `paintPoint2D` value and the rest will be calculated automatically via `update`. The following code shows how `CustomIsometricCharacter` is created and animated:

```
character = new CustomAnimatedCharacter
  ("soldier",imageWidth,imageHeight,tileWidth,
    paintPoint,offsetPoint,padding,framedelay,dispatchEvent);
character.setLabels(new
  Array(["walk",6],["jump",4],["attack",8]));
character.state = "walk";
character.facing = "right";
character.gotoAndPlay(character.state+"_"+character.facing);
```

An interesting parameter is `offsetPoint`, which actually deals with the altering of a registration point as with the case of our overlay tiles. As noted earlier, you can also use the pivot property of a Starling image for the same implementation. These character images are usually bigger than our isometric tile dimensions and hence, drawing them at their default top-left corner registration point will make their legs fall out of the tile on which they are standing. So, we will need to offset the drawing point to make sure that the characters look right and feel that they are standing on their intended `paintPoint` tile.

Time-based animation versus frame-based animation

Frame-based animations depend entirely on the frame rate of the `Shockwave Flash` document. For example, the following code is a frame-based animation of simple sidewise movement:

```
this.paintPoint2D.x += speed;
```

Each frame in the `paintPoint2D` value is incremented in `x` by the `speed` value. This has a severe flaw as the animation will slow down on a low-end machine where the frame rates are low. It may also speed up on a super fast machine as well. Such a feature is undesirable and sometimes may even break our game logic. Hence, it is always advisable to have time-based animations instead of frame-based ones. A time-based variant of the previous code can be found in the `Soldier` class, shown as follows:

```
this.paintPoint2D.x += speed*delta;
```

Here, `delta` is the time elapsed after the last frame. This value can either be manually found out by finding the difference of time between the frames or can be found from the `passedTime` property of `EnterFrameEvent`. Such time-based animations are not affected by the frame rates and will always run at the same speed.

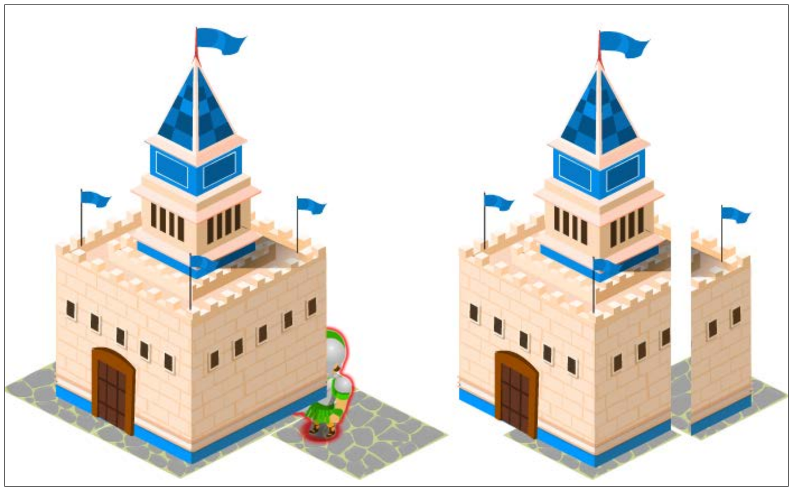
Depth sorting issues

Depth sorting is a very tricky thing to do with huge isometric worlds with multiple moving items. For all nonmoveable tiles, which include the ground tiles and overlay tiles, all we need to do is to sequentially draw them while parsing the respective level data arrays. Ground tiles can be omitted from depth considerations as they are always below the characters and overlay tiles. For overlay tiles, the ones with higher `X` or `Y` values are drawn on top of those with lower values. Drawing happens in increasing order of the rows and in each row it happens in increasing order of columns—a loop within a loop. This effectively handles the depth sorting automatically as we do not have any moving tiles.

But things get complicated when we consider the characters moving across the ground. Each character, being a `CustomIsometricCharacter` instance, will have a corresponding `paintPoint`, which is the tile coordinate it occupies. So, we can easily compare this value while going through the previous loop and draw them at the right instance. For example, a character on row `2` and column `4` can be drawn at exactly that position within the loop. After each move, the `update` method of the `CustomIsometricCharacter` class finds out the current `paintPoint` value, which essentially points to the row and the column it occupies. In our game, we have made sure that any tile with an overlay item is nonwalkable and hence, we do not have to think about two items occupying same row and column.

The only issue that you will face is with the larger tiles which occupy multiple rows. A tile occupying multiple columns but a single row will always render correctly. But the one which occupies multiple rows will break our automatic depth sorting when a moveable item occupies a tile next to it. This happens when the moveable item occupies the lower row than the big tile but the next column. The right end of the bigger tile which also occupies the same row as the moveable tile will be drawn on top of it. The reason being that the big tile exists at a higher row than the moveable tile and hence drawn later, thereby overlapping the moveable tile. This is wrong, as although the moveable tile is on a lower row, it still is in the next column which in real-world space means that it should be drawn on top of the big tile in that particular row. This can only be explained via a visual representation. See how the castle and the soldier are neighbors while the soldier is in a lower row than the castle. The castle gets drawn after the soldier and hence its right part overlaps the soldier.

Check the following figure to understand this issue:



The solution is to split such tiles per row as displayed in the previous figure to the right. This essentially makes sure that each piece is rendered per row, and hence an item in a higher column is always drawn later and won't get masked.

Implementing the game level

The Flag Defense game is essentially an event-based game, which only needs a very simple game loop. This is because in this turn-based game, each action occurs sequentially and is based on events. So, our game loop essentially is very simple and involves only advancing of animation frames, moving characters, rendering the updated level, and switching few game states.

Consider the following code in the `MainLevel` class:

```
private function loop(e:EnterFrameEvent):void {
  Starling.juggler.advanceTime
    (e.passedTime);diceAnim.update(e.passedTime);
  worldLayer.render(e.passedTime, groundImage);
  if(gameState == GameStates.GREEN_MOVE &&
    worldLayer.noneWalking()){
    clearPath();
    switchGameState(GameStates.BLUE_DICE);
  }else if(gameState == GameStates.BLUE_MOVE &&
    worldLayer.noneWalking(true)){
    clearPath();
    switchGameState(GameStates.GREEN_DICE);
  }
}
```

The `worldLayer.render` method calls the `update` method of all soldiers, moves them if their `walking` property is `true`, and then draws the game world.

The game states

The following are the different game states that we have used in the game and their functionalities:

- `GREEN_DICE`: This indicates that the dice is thrown by the green player, tap to stop dice rolling
- `GREEN_PATH`: This indicates the path drawing done by the green player by touching and dragging
- `GREEN_PATHCOMPLETE`: This indicates the path completed by the green player, that is, the path now has the same number of moves as the dice value
- `GREEN_MOVE`: This indicates the automatic movement for the green player on the drawn path
- `BLUE_DICE`: This indicates an automatic dice throw for the blue player
- `BLUE_DICESTOP`: This indicates an automatic dice stop for the blue player
- `BLUE_PATH`: This indicates automatic pathfinding, path drawing for the blue player
- `BLUE_MOVE`: This indicates the automatic movement of the blue player on the found path

All game states starting with `BLUE` constitute the AI for this game. The `GREEN_MOVE` game state can also be considered as an AI as the soldier needs to follow the path automatically.

The touch listener

The `MainLevel` class listens to the touch events and makes sure that it responds accordingly based on the current game state. Double-tapping anywhere on the screen will toggle the UI on and off. For the game states `GREEN_PATH` and `GREEN_PATHCOMPLETE`, we track touches for drawing the path by a user. If we touch a green soldier, we start the path drawing and as we move the touch, the path array for that soldier is updated if the new tile is directly walkable from the previous touched tile. Path drawing stops when we touch a nonwalkable tile, when we release our touch, or when the path length becomes higher than the dice value. Arrows are drawn on the ground layer while the path is updated. It also makes sure that the game state switches to `GREEN_PATHCOMPLETE` when we have enough path points.

Following the AI path

In the game, we will need the soldiers to follow the path we draw for them as well as the enemy soldiers to find their own path. Let us first see how path following is done.

The soldier class (`CustomIsometricCharacter`) is already wired to move in the direction specified by its `facing` property if its `walking` property is `true`. For following a path, we set the `walking` property to `true`, find the next destination from its `path` array, find the new `facing` value by comparing the current position and destination. The `path` array of a soldier will have tile coordinates in a sequential order. So, once the soldier reaches the current destination, we will update the destination to the next point from the `path` array and the process is repeated till there are no more points in the array. Then the `walking` property is turned off as we have followed our path to reach the final destination.

The important methods implementing the preceding functionality in the `soldier` class are `markDestination()`, `findDirection()`, and `move(delta:Number)`.

Path finding

For the green soldiers, path finding is direct as the player himself does it by drawing the path. The code just makes sure that the newly marked tile is walkable and is either vertically or horizontally next to the previous valid position saved in the `path` array. In our engine, we don't allow the soldiers to walk diagonally between the tiles.

For the blue soldiers, we need to find the path using a path finding algorithm. There are many standard algorithms for the same which can be applied to a 2D array to find a shortest path. Two of the major candidates are **Dijkstra's** algorithm and **A *** algorithm. There are far too many standard implementations of these algorithms in AS3 that you are not advised to reinvent the wheel, but use one which suits the purpose. Essentially, most of all the implementations will need a collision array with `0` value for a walkable tile and `1` for an obstacle. We create the same in the `createLevel` method of the `MainLevel` class, shown as follows:

```
for(var i:int=0; i<groundLayer.rows; i++){
  worldLayer.collisionArray[i] = new Array();
  for(var j:int=0; j<groundLayer.cols; j++){
    if(worldLayer.overlayArray[i][j] != "*" ||
      groundLayer.groundArray[i][j] == "tile4.png")
    {
      worldLayer.collisionArray[i].push(1);
    }else{
      worldLayer.collisionArray[i].push(0);
    }
  }
}
```

The array has a value, `1` when there is an overlay item in that tile position or if the ground has `tile4.png` which is the base for our castles. The ground tile is considered as the castle tile occupies multiple tiles, but can be an overlay value in only one of those tile positions.

We are using the `Pathfinder` class by Joe Hocking, which is an A * implementation found at `http://www.newarteest.com/flash/astar.html` with some slight modification to remove the diagonal path finding.

We can call the `Pathfinder` class to return an array of points in sequential order from a starting point to an end point by using the following code:

```
path=PathFinder.go(startPoint.x,startPoint.y,endPoint.x,
  endPoint.y,collisionArray);
```

For the blue soldiers, we calculate the path from their `paintPoint` position to the green flag position, and splice the resulting array to only include as many points as decided by the dice throw. Once the path is found, the path following part is the same for all the soldiers as explained before.

Making AI decision

Simulating AI-based decision making requires a bit of extra effort. We need to purposefully introduce delays in each step for making it realistic and to introduce a human element. Most of the AI decisions can be made instantly which does break the illusion of the human player that we are trying to convey. In our game, each AI decisions are made via methods which are called with a delay using the `juggler` class as follows:

```
Starling.juggler.delayCall(switchGameState,2,
  [GameStates.BLUE_DICESTOP]);
```

Summary

In this chapter, we learned to implement our game level using the approaches learned earlier. We got to know about some custom classes, which help us implement our soldiers in a much better way. Basic AI decision making and path following was explored. I introduced you to the excellent `Pathfinder` class, which helped us to use the A * algorithm to find paths between points. I explained to you some of the depth sorting issues and their solutions, which we have implemented in this game.

We also learned about time-based animations and their advantages. I have explained to you the various game states and their corresponding functionalities as well as explained the purpose of our touch listener. I could only cover as much in this chapter while the project has a lot of code, which should be thoroughly investigated. The concepts explained in this chapter make it very easy for fully understanding the game code. The game logic will be further fleshed out in the upcoming chapters, but the current engine lays the solid base on top of which everything else will be built. In the next chapter, we will try to include bombs in our level and make them explode with wonderful particle effects. We will create the effects using a particle editor tool that is available online.

5
Boom Boom, Explosions!

We now have a minimal version of our engine which can be used as a base to create our final game. The core game mechanics of the game will be fleshed out in the next chapter, and we will focus on the particle effects in this chapter.

Special effects bring the much needed eye candy to the game. It adds polish to the game and makes the experience all the more rich. There is a certain balance that needs to be maintained while adding the effects, as almost all of the elements in a game can be enhanced with the help of these effects. We can add particle effects to the very basic action of pressing a button, the awarding of a bonus, life loss, picking up items, level up, and the list is endless. If overdone, it will become too much and distract the user, or even drive them away from the game. So, special effects should be used to enhance the gameplay in a well thought-out manner. These effects may not seem mandatory, as some wise people have said, if present, these may not be noticed, but if not present, they will be.

In this chapter, we will cover the following topics:

- Image-based particle effects
- Starling Particle Extension
- Online Particle Designer tool
- Particle Designer application
- Adding explosions to our game

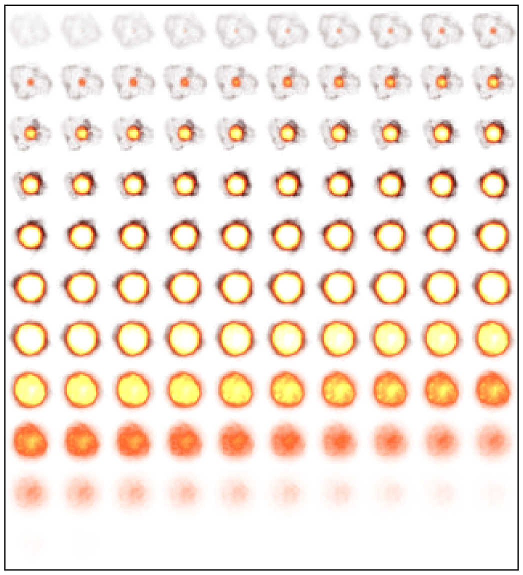
Adding some effects

In our game, we will need to add an explosion effect which should happen when the user steps on a bomb. Bombs will be released into the game area and they will slowly move towards the enemy soldiers. When a bomb and an enemy soldier occupy the same tile, the bomb will explode, resetting the soldier to his/her castle. For the sake of simplicity, we will have a sample project which will just add the explosions to wherever we click on the game level. The source code for this can be found in the `Chapter 5` folder.

An explosion can be created in two different ways. The easiest and most straightforward way is to use a frame-based particle effect. The more complicated but visually rich approach will be to use **Particles**.

An image-based explosion

This is just our regular Starling `MovieClip` class or our `CustomAnimatedItem` instance without any special changes. We need to have images or sprite sheets with a sequence of explosions, shown as follows:



Creating such a sprite sheet involves a lot of talent when using tools such as Photoshop, and it may be very difficult for normal artists. There are some tools which can help; one among them that stands out is **Explosion Generator 3** and it is free. Also, it has a lot of additional options that are available for Mac and Windows. You can get it at `http://www.deepblueapps.com/explosion-generator-3/`.

Now once you have your sprites, you just need to pack them into `TextureAtlas` and create an explosion `MovieClip` based on them. Although this is easy, it has the issue of being monotonous, as you end up seeing the same explosion over and over again each time. You can add some amount of variation to the explosion by altering the properties such as tint, rotation, and speed of the `MovieClip` class, or even independent frames if you are using a custom approach such as our `CustomAnimatedItem` class.

Explore the source in the `Static Explosion` folder to see how easy it is to add a frame-based explosion. Just click on the screen to add an explosion to the scene. The code uses the Starling `Juggler` class as follows:

```
private function addExplosion(tp:Point):void
  {
    var tex:Vector.<Texture>= texAtlas.getTextures("Explosion_");
    var exp:MovieClip=new MovieClip(tex);
    addChild(exp);
    Starling.juggler.add(exp);
    exp.x=tp.x-exp.width/2;
    exp.y=tp.y-exp.height/2;
    exp.addEventListener(Event.COMPLETE,animDone);
  }
private function animDone(e:Event):void
  {
    var exp:MovieClip=e.target as MovieClip;
    Starling.juggler.remove(exp);
    exp.removeFromParent(true);
  }
```

[The `Juggler` function controls all the Starling animations. Its `advanceTime` method needs to be called for every frame.]

Particle systems

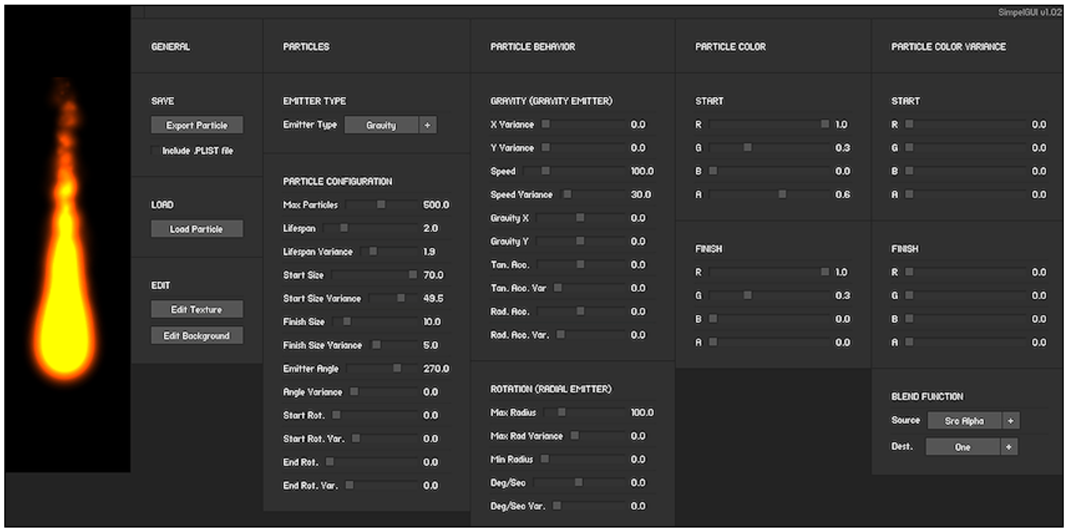
A particle system works by using a large number of images to simulate the particle effects, which will have a good performance with the use of `QuadBatch`. We will learn more about `QuadBatch` in the next chapter. As each particle in the physical world has some properties depending on its physical composition, the same properties when given to an image can be used to simulate that kind of particle. For example, when simulating water, each particle in the sprite sheet is given all the physical properties of a water droplet such as color, mass, and velocity. Similarly, while rendering an explosion, each particle in a sprite is given the characteristics of a particle in an explosive flame.

In order to impart those physical characteristics, each sprite is rendered with a certain set of properties such as particle size, particle color, particle velocity, particle acceleration, and particle rotation. Now, programmatically configuring these parameters is painful. Hence, there are software solutions that one can use to tweak these configurations in a WYSIWYG editor and save the configuration in a `PEX` file. The `PEX` file can later be imported in Starling using the **Particle System Extension** that draws particles according to the given configuration for the specified amount of time.

The Starling Extension Particle System or Starling Particle Extension, can be located at `https://github.com/PrimaryFeather/Starling-Extension-Particle-System/`.

The `PEX` format is just a simple XML variant to store and reproduce the essential data to create the particle effect. There are two tools which can be used to create the particle systems. The tool that I have used to generate the `PEX` files is a particle editor by Devon O. Wolfgang (**ONEBYONE** design). You can access it at `http://onebyonedesign.com/flash/particleeditor/`. The Particle Designer tool by 71Squared is another good tool to generate the `PEX` files. In fact, it also offers a collection of such `PEX` files, but is a paid application which is available only for Mac. Check out the interfaces for both the tools.

The interface for the Particle Editor online tool by Devon O. Wolfgang looks similar to the following screenshot:



PARTICLES

EMITTER TYPE

Emitter Type: Gravity

PARTICLE CONFIGURATION

Max Particles 500.0
Lifespan 2.0
Lifespan Variance 1.9
Start Size 70.0
Start Size Variance 49.5
Finish Size 10.0
Finish Size Variance 5.0
Emitter Angle 270.0
Angle Variance 0.0
Start Rot. 0.0
Start Rot. Var. 0.0
End Rot. 0.0
End Rot. Var. 0.0

PARTICLE BEHAVIOR

GRAVITY (GRAVITY EMITTER)

X Variance 0.0
Y Variance 0.0
Speed 100.0
Speed Variance 30.0
Gravity X 0.0
Gravity Y 0.0
Tan. Acc. 0.0
Tan. Acc. Var 0.0
Rad. Acc. 0.0
Rad. Acc. Var. 0.0

ROTATION (RADIAL EMITTER)

Max Radius 100.0
Max Rad Variance 0.0
Min Radius 0.0
Deg/Sec 0.0
Deg/Sec Var. 0.0

PARTICLE COLOR

START

R 1.0
G 0.3
B 0.0
A 0.6

FINISH

R 1.0
G 0.3
B 0.0
A 0.0

PARTICLE COLOR VARIANCE

START

R 0.0
G 0.0
B 0.0
A 0.0

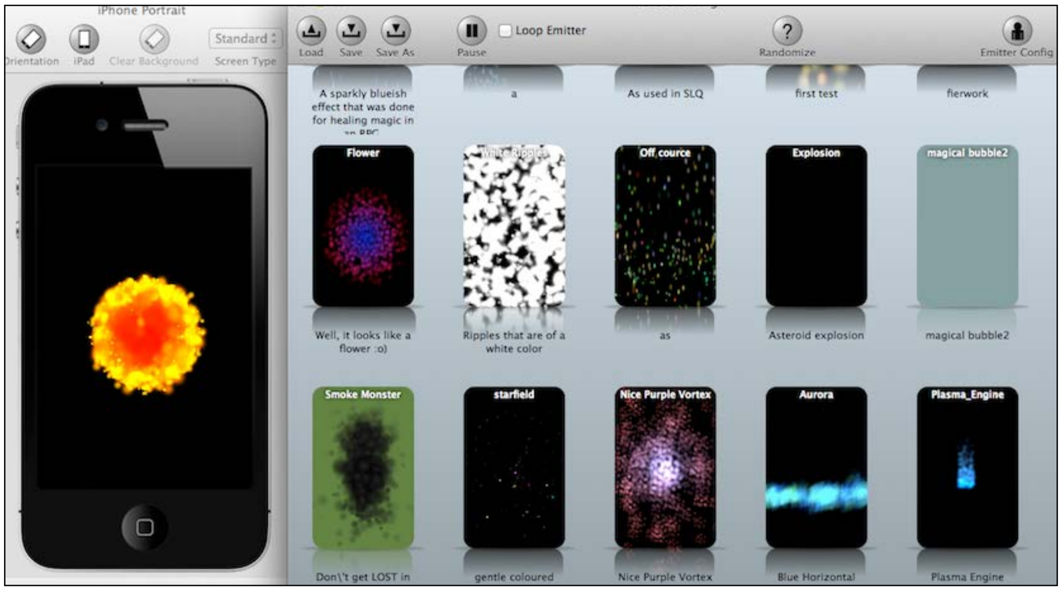
FINISH

R 0.0
G 0.0
B 0.0
A 0.0

BLEND FUNCTION

Source: Src Alpha
Dest.: One

The interface for the Particle Designer tool by 71Squared looks similar to the following screenshot:



Creating the explosion using the Particle Editor tool

We will use the free online tool to create our explosion. Let us take a look at the configuration of particles for rendering the explosion.

For an explosion, since the particles can spread in all the directions, we will first set **Angle Variance** to `360`. Variance is the parameter that accounts for randomness. If the variance of any parameter is set to zero, it will have a constant value. However, since we have set **Angle Variance** to `360`, it will have a value between `0` and `360`, thereby making the explosion to start at the center rather than at a point.

Next, we need the color of the flame to be a little towards red when the flame dies. Hence, we increase the red component in the **PARTICLE FINISH** color. Notice how the flame becomes reddish towards the edges. The orange flame looks more realistic.

One can create various kinds of flames using the editor. You should try tweaking all the parameters to find the most suitable explosion.

Another important parameter in the editor is the blending setting. It determines how rendering of the explosion blends with the background. By default, the source is set to source alpha while the destination is set to one. This allows for a small blending between the flame and the background. You can try tweaking those parameters to achieve the desired blending.

The final exported `PEX` configuration looks as follows:

```
<particleEmitterConfig>
  <texture name="texture.png"/>
  <sourcePosition x="300.00" y="300.00"/>
  <sourcePositionVariance x="0.00" y="0.00"/>
  <speed value="100.00"/>
  <speedVariance value="30.00"/>
  <particleLifeSpan value="1"/>
  <particleLifespanVariance value="0"/>
  <angle value="270.00"/>
  <angleVariance value="360"/>
  <gravity x="0.00" y="0.00"/>
  <radialAcceleration value="0.00"/>
  <tangentialAcceleration value="0.00"/>
  <radialAccelVariance value="0.00"/>
  <tangentialAccelVariance value="0.00"/>
  <startColor red="0.60" green="0.51" blue="0.00" alpha="0.62"/>
  <startColorVariance red="0.00" green="0.00" blue="0.00"
    alpha="0.00"/>
  <finishColor red="0.60" green="0.11" blue="0.00" alpha="0.00"/>
  <finishColorVariance red="0.00" green="0.00" blue="0.00"
    alpha="0.00"/>
  <maxParticles value="412.35"/>
  <startParticleSize value="55.660000000000004"/>
  <startParticleSizeVariance value="49.53"/>
  <finishParticleSize value="30.3"/>
  <FinishParticleSizeVariance value="5.00"/>
  <duration value="-1.00"/>
  <emitterType value="0"/>
  <maxRadius value="100.00"/>
  <maxRadiusVariance value="0.00"/>
  <minRadius value="0.00"/>
  <rotatePerSecond value="0.00"/>
  <rotatePerSecondVariance value="0.00"/>
  <blendFuncSource value="770"/>
  <blendFuncDestination value="1"/>
  <rotationStart value="0.00"/>
```

```
    <rotationStartVariance value="0.00"/>
    <rotationEnd value="0.00"/>
    <rotationEndVariance value="0.00"/>
  </particleEmitterConfig>
```

Adding smoke

The explosion that we have created looks rather simple. We can make it more realistic by blending two or more particle effects together. The other particle systems can be used to render smoke, debris, flash, and so on.

So, let us go ahead and try to render smoke using the Particle Editor. The default texture in the Particle Editor is **Circle**. When we hit the **Edit Texture** button, the Particle Editor takes us to a menu where we can select the texture being used to render one particle. Since smoke is a little cloudy in nature, let us select **Blob** instead of **Circle**.

Also, we should set **Angle Variance** to `360` to produce a centralized effect. Since smoke is something that expands unlike flame, we should set the particle **Start Size** smaller than the particle **Finish Size**. The start color should be set to dark gray and the end color should be set to light gray. Note that this can also be achieved by changing alpha only.

The final configuration of the smoke `PEX` file looks as follows:

```
  <particleEmitterConfig>
    <texture name="texture.png"/>
    <sourcePosition x="300.00" y="300.00"/>
    <sourcePositionVariance x="0.00" y="0.00"/>
    <speed value="100.00"/>
    <speedVariance value="30.00"/>
    <particleLifeSpan value="1.25"/>
    <particleLifespanVariance value="0.38"/>
    <angle value="270.00"/>
    <angleVariance value="360"/>
    <gravity x="0.00" y="0.00"/>
    <radialAcceleration value="0.00"/>
    <tangentialAcceleration value="0.00"/>
    <radialAccelVariance value="0.00"/>
    <tangentialAccelVariance value="0.00"/>
    <startColor red="0" green="0" blue="0.00" alpha="0.49"/>
    <startColorVariance red="0" green="0" blue="0.00" alpha="0.00"/>
    <finishColor red="0.18" green="0.19" blue="0.18" alpha="0.19"/>
    <finishColorVariance red="0.00" green="0.00" blue="0.00"
      alpha="0.00"/>
    <maxParticles value="1000"/>
    <startParticleSize value="20.16"/>
    <startParticleSizeVariance value="0"/>
    <finishParticleSize value="70"/>
    <FinishParticleSizeVariance value="0"/>
    <duration value="-1.00"/>
    <emitterType value="0"/>
    <maxRadius value="100.00"/>
    <maxRadiusVariance value="0.00"/>
    <minRadius value="0.00"/>
    <rotatePerSecond value="0.00"/>
    <rotatePerSecondVariance value="0.00"/>
    <blendFuncSource value="770"/>
    <blendFuncDestination value="1"/>
    <rotationStart value="0.00"/>
    <rotationStartVariance value="0.00"/>
    <rotationEnd value="0.00"/>
    <rotationEndVariance value="0.00"/>
  </particleEmitterConfig>
```

Adding the explosion to the stage

Explore the source in the `Dynamic Explosion` folder to see how we create an explosion system and render it. We have two particle systems each, one for explosive fire and the other for the smoke. Since we want the smoke to persist for sometime after the flame has finished, we make the flame particles last for a smaller duration as compared to the smoke particles. The `ResourceManager` class is updated to supply new flame and smoke particle system, as follows:

```
var f:PDParticleSystem = new PDParticleSystem(new XML(new
  flame_xml()), Texture.fromBitmap(new flame()));
var s:PDParticleSystem = new PDParticleSystem(new XML(new
  smoke_xml()), Texture.fromBitmap(new smoke()));
```

The new `Explosion` class uses these to create a new explosion, mixing them together as seen in the following code:

```
public function Explosion(tilePoint:Point,isoPoint:Point)
  {
    f = ResourceManager.flameFx;
    s = ResourceManager.smokeFx;
    paintPoint.copyFrom(tilePoint);
    paintPointIso.copyFrom(isoPoint);
    s.addEventListener(Event.COMPLETE, removeS);
    f.addEventListener(Event.COMPLETE, removeF);
```

```
        f.start(0.05);
        s.start(0.1);
        disposeCount = 0;
        completed = false;
      }
    public function update(delta:Number):void{
        f.advanceTime(delta);
        s.advanceTime(delta);
    }
    private function removeS(event:Event=null):void
      {
        s.removeEventListener("complete", removeS);
        s.stop(true);
        disposeCount++;
        if(disposeCount == 2){
          completed = true;
        }
      }
```

It tracks both the particle systems for completion and sets a `completed` flag to `true` so that we can remove it from rendering. It stores the tile point and the isometric position so that the render code compares the tile point to draw it at the correct depth, for example, behind the bushes. The `WorldLayer` class has an `explosion` vector which stores all instances and its render loop has the following code segment which does the rendering:

```
for(var i:int=0; i<rows; i++){
  for(var j:int=0; j<cols; j++){
    for(var id:int=0; id<explosions.length; id++){
      explosion = explosions[id] as Explosion;
      if(explosion.paintPoint.x == j && explosion.paintPoint.y ==
        I) {
        if(explosion.completed){
          explosions.splice(id,1);
          explosion.destroy();
          explosion = null;
        }else{
          explosion.update(delta);
          mat.tx = explosion.paintPointIso.x;
          mat.ty = explosion.paintPointIso.y;
          draw(explosion.s, mat);
          draw(explosion.f, mat);
        }
      }
    }
  }
}
```

Summary

In this chapter, we added special effects to the Starling game scene. We learned about the frame-based particle effects which are basically a Starling `MovieClip`. I also shared tools with which you can create frames with effects. I introduced you to the Starling Particle Extension, which is used to create dynamic explosions comprising of multiple particle systems including a flame and smoke.

The next chapter is the most important part of this book where everything comes together to form the complete game. We use everything that we have learned so far to make this happen. The code and the packaging will get refactored to make everything much more well organized.

6
Going Cross Platform

So, we are at the sixth chapter of the book, and you may be wondering why everything was so simple. You may be wondering where the really complicated parts are hiding out. I will be talking about all the complicated aspects of the flag defense game in this chapter. The source code for the completed web game can be found in the `FlagDefenseComplete` folder and the source code for the completed cross-platform game (Android and iOS) can be found in the `FlagDefenseDevice` folder.

In this chapter, we will cover the following topics:

- Complete details on the implementation of the web version
- In-depth analysis of the AI decision-making
- Supporting multiple screen sizes
- Tricks used by the game for supporting multiple devices
- Creating dynamic texture atlases
- iOS and Android specifics
- Local leaderboards
- Integrating ANE for AdMob
- Publishing to the Google Play Store and the Apple App Store

The finished game

If you have not yet played the game, please go ahead, compile the `FlagDefenseComplete` project, and give it a go. The code base has the usual single player game as well as a same device multiplayer game, where you can play against a friend by sharing the mouse on the same PC. It is not possible to explain the whole code in this chapter; I am focusing on explaining the complicated aspects only.

Let me briefly explain the organization of the code base. We have a few dependencies such as the Starling framework, the Starling Particles extension, AS3 signals, TweenLite, Flox, and `sounds.swc`, which resides in the `assets` folder and contains all the sounds used in the game. The default application is `Web_Preloader.as`, which shows the loading progress and eventually loads our main entry class. This compiler argument sets up the `StartupFrame FlagDefenseComplete` frame.

The `libs` package has the `Pathfinder` class and `com.csharks` has all the game-specific classes. The `ResourceManager` class handles playing sounds, accessing support files for particle effects, and populating the `AssetManager` instance with all the textures. We are using a single 1024 x 768-based texture atlas embedded in the `EmbeddedAssets` class. The `GameStates` class has two new states to facilitate the automatic movement of bombs per turn that are mentioned as follows:

- `BLUE_BOMBS_MOVE`: This moves blue bombs automatically
- `GREEN_BOMBS_MOVE`: This moves green bombs automatically

The `com.csharks.flagdefense.entities` package has the `Bomb` class and the `Explosion` class, which were introduced earlier. The `com.csharks.flagdefense.helpers` package contains the following classes:

- The `AiManager` class: The core decision-making is done by this class assisted by other helper classes
- The `LeveUtils` class: This stores the level data and creates the collision array
- The `SceneManager` class: This is the starting Starling class, which swaps out the different game scenes

The `com.csharks.flagdefense.scenes` package has the `MainLevel` class and the `MultiSamePC` class, which respectively handle the single player and multiplayer modes of gameplay. Additional classes, `AiStates` and `AiDecisions`, help `AiManager` to go through the process of AI decision-making.

It is interesting to note that the `MultiSamePC` class actually extends the `MainLevel` class and overrides some key functions, which easily transforms the game from the single player mode to multiplayer mode; of course, with no small help from the `AiManager` class.

The MainLevel game loop

Now that you have gone through the code base, I am sure that you can easily understand the logic flow and realize how the game works. Let me reconfirm what you have already found out. The `MainLevel` class is the starting point, which initializes an instance of the `WorldLayer` class, spawns one soldier each for each side, sets the initial game state, sets up the UI, adds the game loop listener, adds the game UI listener, and a touch listener. The following list explains what the game loop does:

- Updates the changes in the game world, runs the AI process, and renders the scene
- Checks if all the blue soldiers have completed moving and then changes the state
- Checks if all the green soldiers have completed moving and then changes the state
- Checks if all the blue bombs have completed moving and then changes the state
- Checks if all the green bombs have completed moving and then changes the state
- Checks if any side has won

The touch listener handles the following logic:

- If a touch is started, it checks if it was on a soldier and starts path drawing
- If a touch has moved, it checks if it has a valid neighbor, and appends it to the path
- Ends the path drawing when the value is used up or placed at the wrong position
- Toggles the bottom game UI

The touch listener is overridden by `MultiSamePC` to make it work for both sides.

The most important part among all this is the following line of code in the game loop, which actually advances the game logic:

```
worldLayer.render(e.passedTime,groundImage);
```

AiManager

The single line code given at the end of the last section just makes the `WorldLayer` instance render to its `RenderTexture` as was explained in *Chapter 2*, *Going Isometric*, but now it has another very simple looking single line of code which makes all the difference, as follows:

```
aiManager.step(delta);
```

The `step` method within `AiManager` performs the following functions:

- Loops through the blue and green soldier vectors, removes dead soldiers, and calls the `action` and `update` methods of each live soldier
- If any character has just come into a new tile, then it calls the `checkForAxn` method with that soldier
- Loops through the blue and green bomb vectors, removes exploded bombs, and calls the `action` and `update` methods of the rest of the bombs
- If any bomb has come into a new tile, then it calls the `checkForExplosion` method with that bomb
- Loops through all the explosion particle effects stored in the explosions vector, removes those which are done, and calls the `update` method on others
- Checks soldier vector length to see if any side has won

The `Action` and `update` methods of the `Soldier` class and the `Bomb` class just let them move forwards based on the stored path. These help update their paint points so that we can render them correctly. The `CheckForAxn` method is very important as it handles all the following tasks:

- If the soldier has walked into an enemy bomb, well, it explodes!
- If the soldier has walked into the flag point, then it determines whether to take the flag, reset the flag, or win the game
- If there is an enemy soldier in the next tile in our path, then it attacks him. After a delay of 4 seconds, it kills him and continues on the path

All the other methods in the `AiManager` class are easy to understand and help out independently as support methods, but the most crucial of them are `makeDecision` and `addAiPath`.

Making the AI decisions

For the single player game, the most complicated part is the AI decision-making; what to do when it has a dice value. This happens in the `MainLevel` class at the `switchGameState` method, precisely at the `GameStates.BLUE_PATH` case.

```
aiDecision=aiManager.makeDecision(diceAnim.value);
```

The return value is checked to see if the AI has decided to spawn a bomb or spawn a soldier, or move its soldiers. But the action happens in the `makeDecision` method in the `AiManager` class. This method does the following in the exact sequence looping through the blue soldiers vector:

- Checks if a soldier has an enemy bomb as a neighbor and sets the soldier to move away using the `addAiPath` and `stepAway` methods. This gets the highest priority.
- Checks if any of the castles are nearby and sets course towards them if the current objective demands it. This is done by setting the soldier's `aiDestination` method and calling the `addAiPath` method.
- Checks to see if any enemy soldiers are close by within the killable distance and sets course towards them. This has the lowest priority.
- If the preceding priority checks return `false` and checks if it should spawn a soldier or a bomb.
- If none of the above is happening, then we are in free roaming states where soldiers need to just get to destinations. The AI determines if it needs to get the green flag, return the blue flag, or kill the enemy soldier with any flag. This is executed by using any random available soldier, until the dice value runs out.
- The tricky part in this process is that, at times, there won't be any valid paths to follow or we may reach the assigned destination without exhausting the dice value completely, or we may have to intentionally step on a bomb as there will be no other way. The `addAiPath` method has provisions for the same where it may, at times, let the soldier just roam around randomly or step into the next tile of an enemy bomb, which means suicide.
- An AI is always a work in progress, as there is always room for improvement at any stage. It depends on how intelligent you want to make your AI. Currently, our AI is pretty basic and just gets the work done.

Cross platform

It's time to let you explore the web code base on your own and figure out the rest of the code. With the help of the Adobe AIR technology, we are able to use the same ActionScript code base to target desktops, Android, iOS, and Blackberry along with the Web. This is what we call cross-platform development, and Starling helps us with its optimized architecture and touch-based interactivity.

Moving on, let us open up the `FlagDefenseDevice` project, which is actually a Flash builder ActionScript mobile project. The code base remains almost the same with some very subtle but important changes, which we will discuss later on. We have an additional dependency, the `RectanglePacking` algorithm by *Ville Koskela*, which is used for the creation of dynamic texture atlases.

[More information on this can be found at `http://villekoskela.org/2012/08/12/rectangle-packing/`.]

We have the `com.Adobe` package, which helps us save the dynamically created atlas for future use. We no longer need a `preloader` class as the entry point. Let us explore the `FlagDefenseDevice` document class. The first important change which you may actually miss is the SWF directive, which sets the width and height to `100%`.

```
  [SWF(backgroundColor = "#004400", frameRate = "30", width "100%",
height = "100%")]
```

This enables us to make the game fit any screen on which it is displayed. The `com.csharks.juwalbose.utils` package now has a different `ResourceManager` class and a new class called `DynamicTextureAtlasCreator`, which is a slightly altered version from the following link:

[`DynamicTextureAtlasCreator` can be found on GitHub at `https://github.com/juwalbose/DynamicAtlasCeatorAirDemo`.]

Another major difference is that the art assets are no longer embedded and they are dynamically loaded at runtime. This is a very important step, which can save you some significant memory usage as embedding takes up a lot of our precious and available memory. An important thing to notice is that we are using two texture atlases based on the 2048 x 1536 iPad retina screen size. These can be found in the `media` folder inside the `bin-debug` folder.

Supporting multiple screens

Supporting multiple screens is kind of the holy grail in cross-platform development. There are multiple approaches which we can follow, but all have their own flaws and there is no single solution at this point. Let me briefly explain the different possible approaches in order of increasing complexity.

The easiest one is to just provide one single art asset designed for one-screen size, say, 800 x 480, which has the most prevalent aspect ratio and stretch to fit the current screen. The main flaw is that the art will be distorted depending on the aspect ratio of the screen. Also, the quality of the art will be poor in high-resolution as well as low-resolution screens.

We can alternately provide multiple sets of art to be used for multiple sets of screen sizes as it is done in Android with LDPI, MDPI, HDPI, XDPI, and XHDPI. This will certainly address the quality loss to a certain extent, but the aspect ratio distortion will persist if we allow the canvas to stretch to fit the screen.

Yet another alternative, which is preferred by many game developers as well as the Starling creator, is to use **letterboxing**. Letterboxing tries to fit the canvas to the screen without changing the aspect ratio. This makes the game display empty areas on its borders that fall outside the strict aspect.

Rolf Ruiz has an excellent Starling `Viewport` class which can help with letterboxing. More information on this is available at `https://github.com/alesys/StarlingViewPort`.

The most efficient method is to create needed textures from an asset dynamically at runtime based on the current screen size. This can be achieved by using Flash vector assets along with Emibap's exceptional **Dynamic Texture Atlas** and **Bitmap Font Generator** available at `https://github.com/emibap/Dynamic-Texture-Atlas-Generator`.

The only problem with this approach is that this breaks the way we normally create games, which involves spritesheets and raster atlases. An artist may be more comfortable creating a spritesheet, or rather a development pipeline in place will facilitate the same. Ideally, game development is spritesheet-based and handling more platforms, such as HTML5 will demand the same. So, it is advised to have a spritesheet-based development pipeline to support as many platforms as possible.

So, I have created a solution, which creates textures dynamically based on the current screen size, but is still raster atlas-based. Enter the `DynamicTextureAtlasCreator` class, which works in tandem with `ResourceManager` and the Rectangle packing algorithm by *Ville Koskela* to create atlases dynamically at runtime.

Check out the `DeviceManager` class, which tries to detect if it is an iOS or an Android device.

Dynamic Texture Atlas Creator

This class uses a 2048 iPad retina-based texture atlas to create textures for the current screen size by scaling the images with respect to the aspect ratio. The created images are then packed into a new atlas and saved in the `ApplicationStorageDirectory` folder for use for any subsequent runs. The crux of it is the following function, which scales the images:

```
private static function scaleBitmapData(ARG_object : BitmapData, ratio
: Number):BitmapData {
  var bmpd : BitmapData = new
    BitmapData(Math.round(ARG_object.width * ratio),
  Math.round(ARG_object.height * ratio), true, 0x000000);
  var scaleMatrix:Matrix = new Matrix();
  scaleMatrix.scale(ratio, ratio);
  var colorTransform:ColorTransform = new ColorTransform();
// draw the object to the BitmapData, applying the matrix to scale
  bmpd.draw( ARG_object, scaleMatrix
    ,colorTransform,null,null,true);
  ARG_object.dispose();
  ARG_object = null;
  return bmpd;
}
```

The `ResourceManager` class initiates the creation of the dynamic atlas by calling the following:

```
DynamicAtlasCreator.createFrom(XhdpiPng.bitmapData,data,scaleRatio,ass
ets,atlasName);
```

It passes the 2048-based texture details along with the scale ratio to which we need to scale down, an instance to the asset manager, and the name in which we would be saving the new atlas. `ScaleRatio` needs to be calculated based on the current screen.

Finding the ScaleRatio value

The good thing about developing a tile-based game is that it becomes much easier to scale the game scene up or down as it is just a matter of increasing or decreasing the tile width. In our game too, we just need to find a new tile width, based on which we can re-align our game without affecting the logic and create a new atlas based on the new tile width. The `FlagDefenseDevice` class calls up an `onResize` method when a Starling root is created. `OnResize` calls up the `rescaleAndRedraw` method of the `SceneManager` class, passing in the current stage width and height as parameters.

Have a look at the following `rescaleAndRedraw` code from the `SceneManager` class:

```
public function rescaleAndRedraw(newWidth:Number,newHeight:Number)
:void{
  if(ResourceManager.assets){
    ResourceManager.assets.purge();
  }
  var visibileTiles:Point = new Point(13,20);
  var newDimensionX:uint =
  1 + Math.round(newWidth / visibileTiles.x);
  var newDimensionY:uint =
  1 + Math.round(newHeight / visibileTiles.y);

  if(newDimensionX > 2 * newDimensionY){
    newDimensionX = 2 * newDimensionY;
  }else{
    newDimensionY = newDimensionX / 2;
    newDimensionX = 2 * newDimensionY;
  }
  var scaleRatio:Number = Math.round(newDimensionY / 8) / 10;
  ResourceManager.initialise(scaleRatio);
    addEventListener(Event.ENTER_FRAME,waitForReload);
}
private function waitForReload(e:Event):void{
  if(ResourceManager.initialised){
  removeEventListener(Event.ENTER_FRAME,waitForReload);
  launchMenu();
  }
}
```

We need to show 13 tiles horizontally and 20 tiles vertically to complete the display of our effective game area. Here, we essentially divide the screen width and height with these values to find the best size for our new tile width. This is a misnomer as we are actually using the `newDimensionY` height value as the tile width. The `scaleRatio` value is calculated by dividing the new tile width by the original tile width, which is 80. `ResourceManager` is called to initialize and thereby create a new texture atlas and we wait for the process to be completed. The menu gets launched once the textures are in place.

Positioning items properly

Now, as the screen size can be of any value, we need to find a way to properly position items. The safest way is to position them based on the screen size and the item size itself.

```
helpBtn.x = stage.stageWidth / 2 - helpBtn.width / 2;
```

This code places the button in the middle of any screen horizontally. For absolute placement we need to use the `scaleRatio` value.

```
soundBtn.y = (20 * ResourceManager.scaleRatio);
```

This will place it at 20 pixels for iPad retina and 10 pixels for iPad nonretina.

Now, in order to middle align our active game area on any screen size, we need to do some calculations. This is done within the `MainLevel class' init` method.

```
sr=ResourceManager.scaleRatio;
var newDimensionY:uint = sr * 80;
var newDimensionX:uint = 2 * newDimensionY;
viewPort = RectangleUtil.fit(
  new Rectangle(0, 0, 2*newDimensionY*visibileTiles.x,
    newDimensionY*visibileTiles.y),
  new Rectangle(0, 0, stage.stageWidth, stage.stageHeight),
    ScaleMode.SHOW_ALL);
screenOffset.x = 2 * newDimensionY * screenOffset.x / (2 *
  tileWidth);
screenOffset.y = newDimensionY * screenOffset.y / tileWidth;
screenOffset.x += viewPort.x;
screenOffset.y += viewPort.y;
tileWidth = newDimensionY;
```

This finds the rectangle fitting value to centralize the active area and adds it to the `screenoffset` value which makes all the drawing to get offset, thereby centralizing the game scene.

Essential tips and tricks

We employ a bunch of tricks for our game to be supported on all devices that are explained as follows:

- For startup images, we show the iOS default image when the game launches for the period of initializing the new texture atlas and remove it once the menu launches.
- The background of all the scenes except the game scene is set to stretch and fit. This will distort the image, but will make sure there is no letter boxing black areas. Choose the better evil.
- We use 2048-based texture assets to make sure most of the current devices won't have any quality loss for images.

- The ground layer is modified to draw grass tiles to fill out the entire visible area. We then align the effective game area in the middle of the screen. This makes sure there are no letterboxing black areas even without stretching out the canvas.

We need to minimize the number of texture atlases that we use, and I personally try to stick with two. In order to facilitate this, I had to make some hard decisions. I reduced the number of animation frames for all animations to three, which actually is overdoing it. I also used 1024 iPad nonretina-based textures for the menu background, the wooden base, game win, game over, and title. This does result in minor loss in quality, but it helped me fit everything into two 2048 atlases, and I specifically packed all soldier frames and game tiles into one and the rest into the other, which again helps with batching. Eventually, I ended up loading the menu background and the wood base images individually and removed them from the texture atlas.

> Notice that the texture atlases are not completely filled, which is inefficient to use. But this is done to make sure the dynamic packing always gets done within a maximum-sized 2048 atlas.

Device-specific changes

The `Application Descriptor` XML file is of great importance when we go cross platform. We need to provide a unique ID, which is usually the package name of your game, which is `com.csharks.flagdefense`. We need to set the following:

```
<autoOrients>false</autoOrients>
<aspectRatio>landscape</aspectRatio>
<renderMode>direct</renderMode>
<fullScreen>true</fullScreen>
<visible>true</visible>
```

We need to point to all the needed icon files, which are different for Android, desktop, and iOS, a safer way being to populate them all. There are some scripts out there which can help us create all the icons needed. We will also need to add the ANE file details if we are using any.

A major change in the code is the replacement of `RenderTextures` with `QuadBatch`. When creating a game for all devices, we need to compromise a bit to handle the lack of resources in many of those devices. Some may lack enough memory, some may not have the graphics capabilities, and some may not be fast enough. Although our render texture-based approach works on most of the current generation devices, it has the tendency to fail with older generation devices, especially the iPad1. Most experts would advise you to quit supporting these old generation devices, but knowing how to support them will be key information. `RenderTexture` needs a lot of memory, and when it is marked as persistent, it uses up double the memory. We will run out of texture memory and will face a blank screen when running on a low-memory device.

QuadBatch to the rescue

`QuadBatch` helps us add quads or images to a scene and render them really fast with exceptional performance. The catch is to have all the images read from the same texture that we already have. Thankfully, our game is architectured to make sure that this swap needs minimal work. All we need to do is:

- Add the `QuadBatch` instance to the stage, not image-based on `RenderTexture`
- Instead of drawing to `RenderTexture`, just add them to the `QuadBatch` of every frame

The following code demonstrates a slight change for `groundlayer`:

```
for(i=0; i<rows; i++){
for(j=0; j<cols; j++){
  str = groundArray[i][j];
  img.texture =
    assetsManager.getTexture(str.split(".")[0]);
  pt.x = j * tileWidth;
  pt.y = i * tileWidth;
  pt = IsoHelper.cartToIso(pt);
  img.x = pt.x + screenOffset.x;
  img.y = pt.y + screenOffset.y;
  glQb.addImage(img);
}
}
```

Handling touch input

In the web version, it is much easier to select any soldier and draw a path precisely using the mouse. But on a touch-based device, it becomes very hard to precisely do the same, especially with small-screen devices like phones. We need to alter our control mechanism a bit to handle this limitation. An alternate approach is to tap to select any soldier, and then tap on the screen to add a path depending on the screen position. For example, if we tap on the top of the screen, we will add a path upwards. We need to be liberal in finding the soldier when we tap as the touch point, which is a finger, is pretty inaccurate and big. So, we need to find if we have a soldier on the tapped tile and also on any tile around it.

iOS specifics

For targeting iOS, we need to provide all the default launch images in the root folder. Currently they are:

- `Default-Landscape~ipad.png`
- `Default-Landscape~ipad@2x.png`
- `Default.png`
- `Default@2x.png`
- `Default-568h@2x.png`

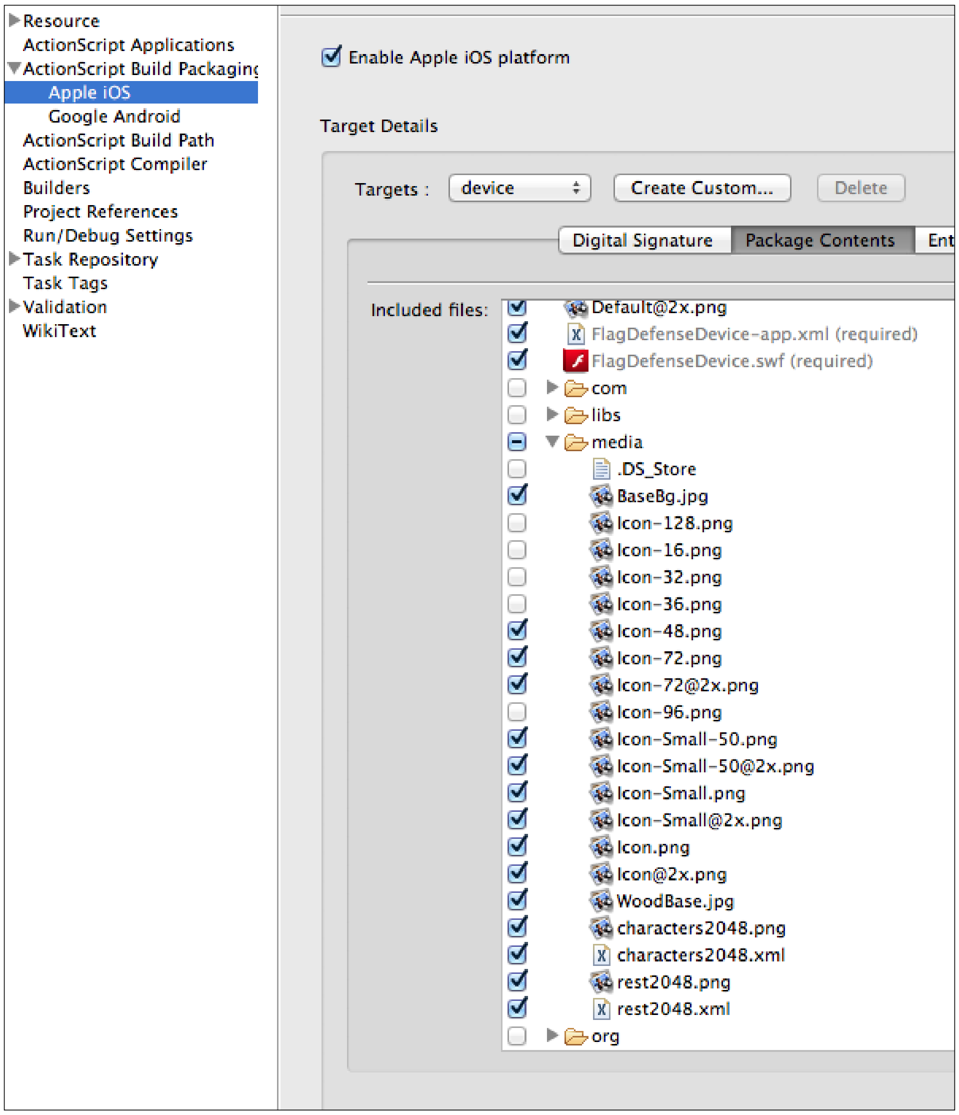
For targeting all iOS devices, that is, to build a universal game, we need to have the following code in the `infoAdditions` section:

```
<key>UIDeviceFamily</key>
  <array>
    <string>1</string>
    <string>2</string>
  </array>
```

Also, the following code line is needed to enable retina display access:

```
<requestedDisplayResolution>high</requestedDisplayResolution>
```

Also, it becomes essential to pack only those files that are needed for a particular platform in the project properties. The following screenshot shows the needed files for iOS, which can be identified from the tick marks to their left:



- Resource
- ActionScript Applications
- ActionScript Build Packaging
  - Apple iOS
  - Google Android
- ActionScript Build Path
- ActionScript Compiler
- Builders
- Project References
- Run/Debug Settings
- Task Repository
- Task Tags
- Validation
- WikiText

[✓] Enable Apple iOS platform

Target Details

Targets : device | Create Custom... | Delete

Digital Signature | Package Contents | Ent

Included files:

- [✓] Default@2x.png
- [✓] FlagDefenseDevice-app.xml (required)
- [✓] FlagDefenseDevice.swf (required)
- [ ] com
- [ ] libs
- [-] media
  - [ ] .DS_Store
  - [✓] BaseBg.jpg
  - [ ] Icon-128.png
  - [ ] Icon-16.png
  - [ ] Icon-32.png
  - [ ] Icon-36.png
  - [✓] Icon-48.png
  - [✓] Icon-72.png
  - [✓] Icon-72@2x.png
  - [ ] Icon-96.png
  - [✓] Icon-Small-50.png
  - [✓] Icon-Small-50@2x.png
  - [✓] Icon-Small.png
  - [✓] Icon-Small@2x.png
  - [✓] Icon.png
  - [✓] Icon@2x.png
  - [✓] WoodBase.jpg
  - [✓] characters2048.png
  - [✓] characters2048.xml
  - [✓] rest2048.png
  - [✓] rest2048.xml
- [ ] org

Android specifics

There are some different settings that you need to be aware of for targeting Android devices. We need to set the right permissions as follows:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_
STORAGE"/>
<uses-permission android:name="android.permission.DISABLE_KEYGUARD"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

Without the write permission, our dynamic texture atlas won't be able to save the created atlas. Also, it is always good to enable storage in the SD card feature when available.

```
<manifest android:installLocation="preferExternal">
```

Another important addition to Android builds is handling the back button.

The Android back button

This is specific to Android devices only as iOS devices do not have a back button. The `FlagDefenseDevice` document class has a key listener added for this purpose. The following code is for the listener:

```
private function handleKeyDown(ev:flash.events.KeyboardEvent):void{
  switch(ev.keyCode){
  case Keyboard.ESCAPE://for testing on PC :)
  case Keyboard.BACK:
  ev.preventDefault();
  var gameRef:SceneManager=Starling.current.root as SceneManager;
  gameRef.tryBackPress();
  break;
  case Keyboard.MENU:
  ev.preventDefault();
  break;
  case Keyboard.SEARCH:
  ev.preventDefault();
  break;
  }
}
```

This calls the `tryBackPress` method in the `SceneManager` class, which in turn calls the `backButtonPressed` method in the running scene. All running scenes have this method as they all now extend the `SkeletonScene` class, which has this public method. Please notice that we are also capturing the menu and search keys found in some Android devices too as we do not have any use for those in a typical game.

All of these devices are primarily phones and there is a possibility of an incoming call at any time. So, we need to listen to any such event and pause our game at that time. This can also happen when another app comes to the foreground or when the user presses the home button. For this purpose, the `FlagDefenseDevice` class has listeners added.

```
NativeApplication.nativeApplication.addEventListener(flash.events.
Event.ACTIVATE,
  function (e:flash.events.Event):void { starling.start();
    ResourceManager.startMusic();});

NativeApplication.nativeApplication.addEventListener(flash.events.
  Event.DEACTIVATE,
  function (e:flash.events.Event):void { starling.
stop();ResourceManager.stopMusic();});

NativeApplication.nativeApplication.systemIdleMode=SystemIdleMode.
KEEP_AWAKE;
```

The `DEACTIVATE` event is triggered when the app goes to the background and the `ACTIVATE` event is triggered when it comes to the foreground. The final line of code is to keep the screen from turning off.

Saving data locally

We will need to store persistent data locally on the device. This can be level details to start the game where the user left off the last time they played, or can be preferences data such as the sound volume, sound on/off, or even the current local leaderboard. Saving the current game state occasionally to pick up where we left off is a recommended feature for any mobile game. Local data can be stored in multiple ways by using `SharedObject`, `EncryptedLocalStore`, a custom file, or even an `sqlLite` database.

Data access using a `SharedObject` method can be done in the following manner:

```
private static var so:SharedObject=SharedObject.getLocal("some unique
name");
public function store(gameData:Object):void{
  so.data[gameName]=gameData;
  so.flush();
}
public function retrieve():Object{
  return(so.data[gameName]);
}
```

With `EncryptedLocalStore` it is a bit different, as shown in the following code:

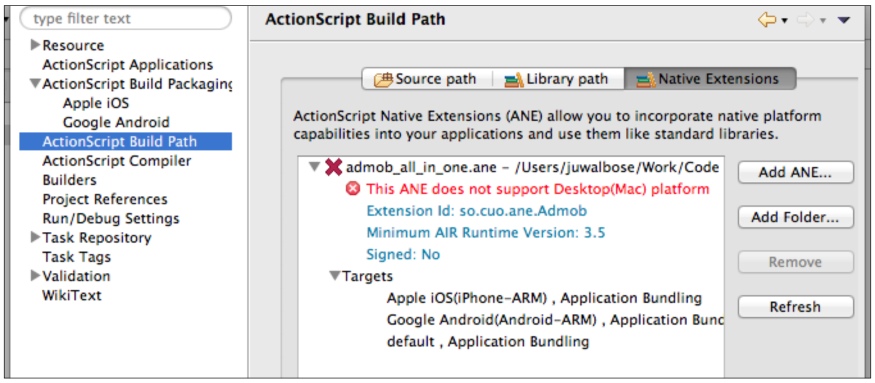
```
private function saveItem(key:String, value:String):void
{
  var bytes:ByteArray = new ByteArray();
  bytes.writeUTFBytes(value);
  EncryptedLocalStore.setItem(gameName+key, bytes);
}
private static function  loadItem(key:String):String {
  var storedValue:ByteArray =
    EncryptedLocalStore.getItem(gameName+key);
  return(storedValue.readUTFBytes(storedValue.length));
}
```

You may use FscoreBoard by *Sean McCracken* (@Seantron) and *Jesse Freeman* (@CodeBum) for creating a local leaderboard (`https://github.com/gamecook/FScoreboard`).

Using the `SqlLite` database is a bit more complicated and is to be considered when there is a lot of data to be stored. You can check the Adobe documentation with the link `http://help.adobe.com/en_US/as3/dev/WS5b3ccc516d4fbf351e63e3d118676a5497-7fb4.html`.

AdMob ANE integration

AIR Native Extensions (**ANE**s) are modules capable of extending the capabilities of AIR, depending on the platform for which it is intended for. It allows the developer to access platform-specific device APIs, which are not available via AIR. For example, there are ANEs for GameCenter, Network Info, Geolocation, and so on. AdMob ANEs are available as *all in one*, which means you can target iOS and Android with a single ANE. Integrating an ANE is as simple as adding it in the **ActionScript Build Path**, as shown in the following screenshot:



ActionScript Build Path

ActionScript Native Extensions (ANE) allow you to incorporate native platform capabilities into your applications and use them like standard libraries.

admob_all_in_one.ane - /Users/juwalbose/Work/Code
This ANE does not support Desktop(Mac) platform
Extension Id: so.cuo.ane.Admob
Minimum AIR Runtime Version: 3.5
Signed: No
Targets
Apple iOS(iPhone-ARM) , Application Bundling
Google Android(Android-ARM) , Application Bund
default , Application Bundling

The code for using an AdMob ANE is different for different versions available out there. As AdMob keeps on upgrading and releasing versions, it is advisable to use the latest version and follow the instructions by its author.

Publishing to the app stores

Finally we are here and we have a completed game ready to be published to the Apple App Store and Google Play Store. Adobe has released articles detailing every aspect of this and I will just list them out. You are advised to read more about them from online resources.

Apple App Store

Let us list out the steps for releasing to the Apple App Store, as follows:

- Get developer certificates and provisioning profiles from the Apple Provisioning Portal for debugging and testing. For release to iTunes, you will need a distribution certificate.
- Create an entry for your game in iTunes connect.
- Hit Export Release Build from Flash Builder and add the necessary details. Make sure you package all the necessary files along with the right icons, launch images, and ANE. Select the option indicating Apple App Store.
- Use Application Loader to upload the created `ipa` file to the App Store for review.

Google Play

Let us list out the steps for releasing to Google Play:

- Create a self-signed certificate using Flash Builder or one from an authority to digitally sign your application.
- Hit Export Release Build from Flash Builder and add the necessary details. Make sure you package all the necessary files along with the right icons, launch images, and ANE.
- Select Export Application with Captive Runtime to package necessary AIR framework files along with the application so that the end user does not need to have AIR runtime installed on their device to use your game.
- The created `.apk` file can be uploaded to any Android App Store including Google Play.

Summary

In this chapter we learned about all the complicated logic that happens within our game. I explained about `AiManager`, which single handedly manages the whole AI logic and much more. We then explored the cross-platform deployment of the same code base by creating a new ActionScript mobile project. The main takeaway from the chapter would be how to support multiple screen sizes for our game without breaking our logic or changing much of our code.

I have explained to you the tips and tricks employed to make our game work in the same way across all screens and devices. We explored a simple ANE integration for some monetization using AdMob ANE. Finally, we found out how we can publish our game to the Apple App Store as well as to the Google Play store.

Eventually it would be seen that we couldn't use exactly the same code base for all the platforms. But on the contrary, we can use the device code base for the Web also, which then makes it a single code base for all. Yet I wanted to show how we can iterate easily and adapt to platforms without major changes. Also, it enabled me to show you how things can be done with `RenderTexture` as well as `QuadBatch`, as both of which, you will put to good use in your own exploits.

Next up we will get to know Flox from Gamua that can take your game to the next level with global leaderboards, tracking, analytics, and much more.

7
Flox – Leaderboards, Analytics, and More

Our cross-platform game is now complete and is ready to be published. So, we will need a cross-platform framework which can track its penetration, log custom events, have a global leaderboard, and so on. There is no other technology which does all this and much more than **Flox**, which is specifically tailored to be cross-platform with a straightforward AS3 API. Flox comes from **Gamua**, the exceptional team behind Sparrow and Starling, who need no introduction to the audience of this book. In this chapter, we will go through all the features offered by Flox, such as:

- Global leaderboard
- User tracking
- Game analytics and custom event tracking
- Persistent game data
- Social integration

We will not be adding these features directly to the Flag Defense game, as many of these features are game-specific and will change for each game. I will demonstrate the code snippets for each so that you can easily integrate it into your game.

Understanding Flox

Flox is the one-stop server side for your games. It is a highly scalable backend that offers game-focused features such as analytics; logs and error introspections; leaderboards and high scores; comprehensive support for players; and a nifty entity system to store, load, and query your custom game data. It's a bit too much but that's not all, head over to `http://gamua.com/flox/features/` to check all of the features.

There are no upfront costs, no monthly commitments, and no minimum fees. Sign up at `http://gamua.com/flox/` and get 1000000 operations for free, as of the time of writing. The backend is designed to be highly flexible and scalable, and it lets you focus on what matters the most—your game!

Getting started with Flox

Once you have signed up with Flox, you get access to your dashboard and the Flox documentation. You will need to download the Flox SWC and link it with your game project. You can explore the documentation provided at `www.flox.cc/docs/getting-started`, which is more than enough for all basic integration. Let me explain the integration in the following simple steps:

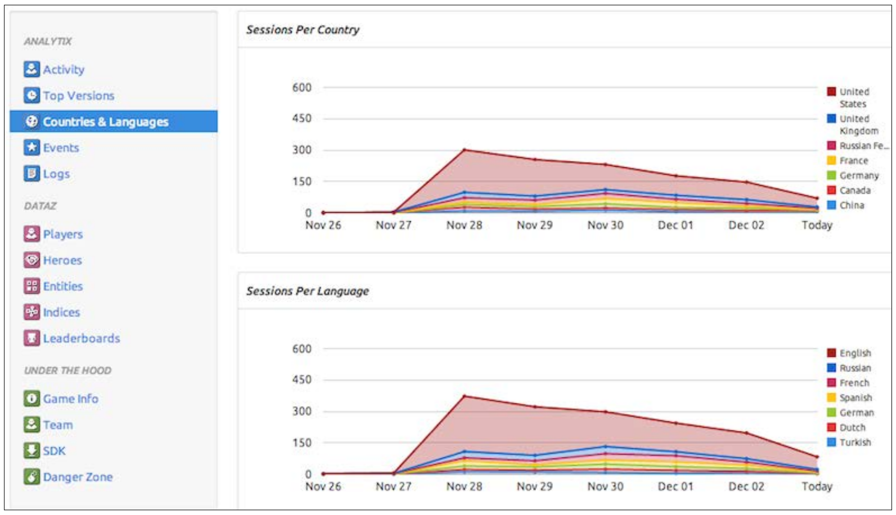
- Add your game using the **Games** menu at the top of the Flox site dashboard
- Initialize Flox using the following code:

```
Flox.init("INSERT_GAME_ID_HERE", "INSERT_GAME_KEY_HERE",
  gameVersion);
```

That's it! Flox is all set to do all the heavy lifting now. You will get your unique ID and an unique KEY while you register your game, as it is an unique game per user. This initialization already tracks the gameplay and creates a default guest user on each device that is used for playing your game.

Flox game analytics

The basic analytics get logged when you initialize Flox, and it can be accessed via the Flox web interface.



For tracking further, we can let our game submit more data to Flox. The main issues faced while developing a cross-platform game is that there will be at least one device which may have a problem with our code. Such exceptions can be tracked and logged so that we can swiftly come up with a fix and update our game. The following code, which adds an event to the `loaderInfo` property of the `Document` class, does just that:

```
loaderInfo.uncaughtErrorEvents.addEventListener(UncaughtErrorEvent
  .UNCAUGHT_ERROR, function(event:UncaughtErrorEvent):void
  {
    Flox.logError(event.error, "Uncaught Error: " +
      event.error.message);
  }
);
```

The interesting part of the code is `Flox.logError`, which does the actual logging. This kind of logging should only be done for the most critical cases, as these get a special treatment apart from the other logging methods. There are other logging methods, as mentioned, which you can use for alternate purposes:

```
Flox.logInfo("Player lost a life.", player);
Flox.logWarning("Something fishy is going on!");
```

The `Flox.logInfo` method is your best friend for simple tracking, as it can replace the `trace()` method with the added benefit of being remote. Flox suggests you to keep logging within necessary limits and also keep an eye on the number of operations. Also, keeping the minimum number of messages saves the data charges for users using a mobile device.

Tracking the events

Any event can be tracked using Flox; even the custom events can be tracked. Consider the following code:

```
Flox.logEvent("MyCustomEvent");
```

This code will create a new event named `MyCustomEvent` and will start collecting statistical data about it. Flox will prepare this data in numbers and charts for you. You can even do more with the events by using the event properties. You'll encounter the event properties as a key/value mapping that you can submit to each event.

```
Flox.logEvent("LevelStarted", { TimeRemaining: 12.0, GameMode:
  "DeathMatch" });
```

The Flox web interface will list all these activity logs for your perusal.

[The data is sent in bulk on game launch, and hence you will receive your logs only after the second launch of the game.]

The following are the few things that you should keep in mind:

- If you submit a `String` as an event value, Flox will show you the total number of times this string has been submitted to an event.
- If you submit a `Number` as an event value, Flox will be able to do some math work with it and calculate several statistical values. You will be able to see the minimum, maximum, average, and sum of the values that have been submitted.

The Flox leaderboard

Using Flox, it is easy to add top-score leaderboards. You will need to first create a leaderboard for your game in the **Leaderboards** section in the Flox web interface. You will need to provide a name, ID, and the natural sort order for the leaderboard. The sort order indicates whether the highest value comes at the top or the lowest value. Usually, the scores need to have the highest value at the top, but the level time should have the lowest value at the top of the list. The leaderboard ID will be used to submit and retrieve the scores. The following are the scores that are submitted:

```
Flox.postScore(leaderboardid, score, "Johnny");
```

Similarly, it is easier to retrieve the submitted scores, as follows:

```
Flox.loadScores(leaderboardid, TimeScope.THIS_WEEK,
  function onComplete(scores:Array):void {
    trace("retrieved " + scores.length + " scores", scores);
  },
  function onError(error:String, cachedScores:Array):void {
    trace("error loading scores: " + error);
  }
);
```

We can use a FeathersUI-based interface to display the list of scores. The `OnError` function also returns the locally stored scores. Flox handles syncing of scores if your device is offline, and it uses `applicationStorageDirectory` for storing some temporary data. You are advised not to change anything there unless absolutely necessary.

[You may notice that we have a `TimeScope` option, using which we can retrieve the scores based on different scopes. It can fetch scores for `All_TIME`, `TODAY`, or `THIS_WEEK`.]

It is always better to use a try-catch safety envelope when dealing with Internet interfaces, as is the case with the leaderboards. Things may go wrong due to various unpredictable reasons, and you do not want your game to crash due to the same.

A way in which the leaderboards can be made more interesting is to show the scores of the user's Facebook friends.

User tracking

It becomes more important to track the user when you can play the game on any number of different devices. Every player who plays a Flox-enabled game has a player ID that is automatically assigned to him/her. This ID is unique to every player and even on every device, unless the user logs in using alternate credentials. You can access the player as follows:

```
var currentPlayer:Player = Player.current;
```

The player's ID is stored close to the player and in a way fitting for your game's medium (for example, a cookie or an AS3-shared object). This approach allows Flox to bind each request to a specific player, thus creating the basis for all possible player-related requests. The different user authentication methods are listed as follows:

- Guest authentication: The guests are the players who are bound to a single installation of your game.
- Key authentication: The players are identified by a single foreign `key` string that you are able to submit. Use this key to integrate your Flox players with a foreign authentication system, such as the iOS GameCenter or Facebook. These players can be recovered and they will have access to their game data from different devices, as long as the `key` string is the same.
- E-mail authentication: The players who authenticated themselves using their e-mail address are recoverable as well. Also, this is the only form of authentication that allows the Flox backend to ensure that the player is the one he/she pretends to be.

[As soon as this ID is gone/lost, the data bound to it will also be lost.]

The methods that we can use are as follows:

```
Player.login(...);
Player.loginWithKey(...);
Player.loginWithEmail(...);
```

Among these, it is easier to use the e-mail authentication method. In reality, people are not ready to trust you with any personal data unless you prove yourselves worthy of it. So, even asking for an e-mail in a modal window when the game launches, may not be a good idea. It is advised to do a thorough market study before you decide to add a player authentication.

By using the e-mail authentication, Flox can make sure that your players are who they pretend to be, by sending them confirmation e-mails. This basically works as follows:

- Your game offers an e-mail authentication and prompts for the player's e-mail address
- The player enters his/her e-mail address
- The Flox server sends a confirmation e-mail to this address
- The player clicks on the confirmation link present within the e-mail
- Flox confirms the game installation, and tells the player that they are now ready to play on
- The player switches back to your game and is now able to access his/her game data

The Flox e-mail authentication system automatically authenticates the first game installation that a player logs in to. This means that for each player, the first `Flox.loginWithEmail(...)` call will succeed without prompting a player to check his/her e-mail account. Subsequent logins on different devices/installations will send confirmation e-mails. You can use the following code to build your authentication system:

```
Player.loginWithEmail(email, function
  onLoginComplete(player:Player)
  {
    //Yay! The player is logged in.
  }, function onLoginFailed(error:String,
    confirmationEmailSent:Boolean)
  {
    if(confirmationEmailSent)
      {
        /*The player is playing on this device for the first time.
```

```
          He has been sent a confirmation email and needs to click
          on the contained confirmation link.
          You should now tell the player that he needs to check his
          emails, click the confirmation link and return to the
          game. When the player returns to your game, you should
          proceed by calling loginWithEmail() again. This time the
          device your player is playing on will be authorized and
          the login attempt will succeed. */
        } else {
          /*Darn! Something unexpected went wrong during the
          authentication attempt. You should probably log the error
          and tell your player about it. */
        }
    } );
```

Social integration with Facebook or other platforms can also help us to authenticate the player. We can use a social network's API to get the user's unique ID and use it to log in. We can later use the same to get his/her friends list and other social attributes. The basic sample code can be as follows:

```
var facebookID:String = ...;
var floxPlayerKey:String = createUID(16, facebookID);
Player.loginWithKey, floxPlayerKey,
    function onComplete(player:Player):void { ... },
    function onError(error:String):void { ... }
);
```

The Flox entities

Once you have your player authenticated, you can store data for later retrieval. This data is persistent and can be used as saved games, level progression, score, preferences, and so on. Such data is represented as an **entity**. You can think of entities as simple objects within your game code. Flox allows you to save those entities to the Cloud and retrieve them from there at anytime. You can have your own custom items which extends the `Entity` class as follows:

```
public class Savegame extends Entity {
  ...
}
var savegame:Savegame = new Savegame();
savegame.progress = 0.5;
savegame.saveQueued();
```

In order to allow Flox to save/load your custom objects, they need to be prepared to work with the Flox SDK. As you'll see, only minor changes will be necessary, such as:

- Your custom objects need to extend the `com.gamua.flox.Entity` class
- Your custom objects need to have a default/no-argument constructor
- The properties of your custom objects need to be readable and writable, or have getters and setters

Behind the scenes, Flox will be engaged in saving the entity immediately, but you may decide to save it later, if necessary. The SDK adds the `savegame` object to its internal save-queue and processes the entities in the queue as soon as possible. This means that your `savegame` is usually saved in an instant, but may be saved at a later point in time if, for example, your player's device is offline at that very moment. If you need much more precise control over the saving mechanism, then you may use the following code:

```
var savegame:Savegame = new Savegame();
savegame.id = "my-savegame";
savegame.inGameProgress = 0.1;
  //save it to Flox
savegame.save( function onComplete(savegame:Savegame): void {
  //The savegame has been saved successfully.
}, function onError(error:String):void {
  //An error occurred while saving the entity: The device may
    be offline.
} );
```

It is advised to stick to the `saveQueued()` method, as it will handle the offline failure situation.

Loading the entities

The saved entities can be retrieved by using the IDs assigned to them, as follows:

```
var savegameId:String = "my-savegame";
//load the associated entity of type Savegame Entity.load(Savegame,
savegameId, function onComplete(savegame:Savegame):void {
  /*Everything worked just fine and an entity has been
  retrieved from the server. This savegame is never null. */
}, function onError(error:String, httpStatus:HttpStatus):void {
  if(httpStatus == HttpStatus.NOT_FOUND) {
  /*There's no entity on the server that matches the given type
    and ID. */
  } else {
  /*Something went wrong during the load operation: The
  player's device may be offline. */
  }
} );
```

Destroying an entity is similar to saving it, as it provides different approaches based on whether you have an `entity` instance or not. Consider the following code:

```
entity.destroyQueued();
//or
entity.destroy( function onComplete(entity:Entity):void {
  //Yay! Deleting the entity did succeed.
}, function onError(error:String):void {
  //Oops! Deleting the entity did fail.
} );
//or
Entity.destroyQueued(Savegame, "my-savegame");
//or
Entity.destroy(Savegame, "my-savegame", function onComplete():void {
  //Yay! Deleting the entity did succeed.
}, function onError(error:String):void {
  //Oops! Deleting the entity did fail.
} );
```

Using queries for entities

Entities can be accessed using their IDs, but we can also use the query system to access them. **Queries** are used when it becomes necessary to find groups of entities that match a certain set of conditions without knowing their exact IDs. The following code gets all the stored entities of the type `Human`:

```
var query:Query = new Query(Human);
query.find( function onComplete(humans:Array):void {
  //The humans array contains all humans
}, function onError(error:String):void {
  //Something went wrong. The player's device may be offline.
} );
```

For limiting the amount of data fetched, it is advised to set limits to your query. Also, effective pagination can be done by using the `offset` value, as follows:

```
//set a reasonable limit to this query
query.limit = 20;
//set the offset for the paging
query.offset = 20;
```

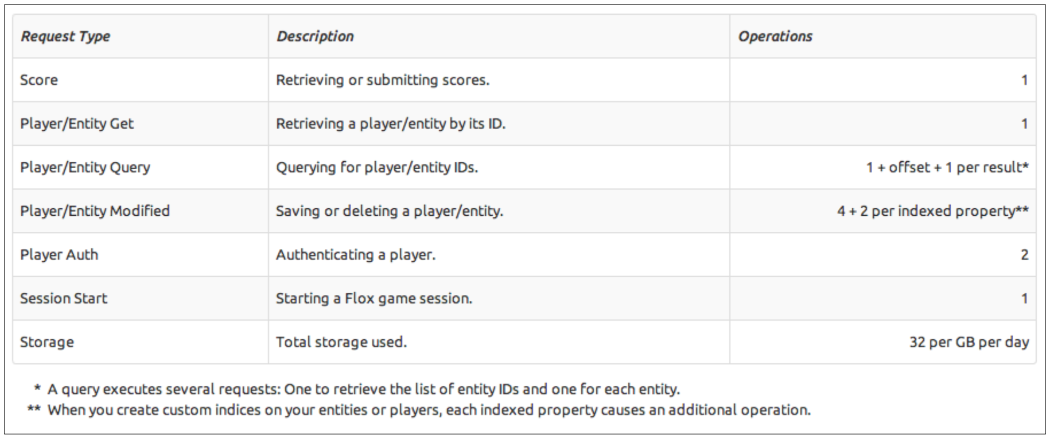
Queries can do a lot more than this, and you can explore the documentation at `www.flox.cc/docs/queries`, to know more about them.

The Flox operations

Flox uses a terminology called **operations** to track the use of service. Let us see how Flox operations work, and how you can calculate and estimate the number of operations that your games will require. The following paradigms apply:

- **Buy server operations only when you need them**: You do not have to commit to any form of monthly payments.
- **Pay only for the amount of server operations you actually use**: There are no minimum fees and no obligations.
- **Operations never expire**: Once you've bought operations, they'll reside at your account until they are put to use. If the need arises, they'll remain in your account forever.

Different server actions cost different amounts of operations. The following table visualizes the basic operations that you can execute on the Flox servers:



Request Type	*Description*	*Operations*
Score	Retrieving or submitting scores.	1
Player/Entity Get	Retrieving a player/entity by its ID.	1
Player/Entity Query	Querying for player/entity IDs.	1 + offset + 1 per result*
Player/Entity Modified	Saving or deleting a player/entity.	4 + 2 per indexed property**
Player Auth	Authenticating a player.	2
Session Start	Starting a Flox game session.	1
Storage	Total storage used.	32 per GB per day

* A query executes several requests: One to retrieve the list of entity IDs and one for each entity.
** When you create custom indices on your entities or players, each indexed property causes an additional operation.

Using the web interface for browsing your game data (for example, analytics and log files) is free. The cost of operations and their count may vary in the near future. You can get the current status from this link: `https://www.flox.cc/docs/operations`

Summary

In this final chapter, we learned all about Flox. We now know how easy it is to add a global leaderboard system into our game. I showed you how to authenticate the players across devices and store persistent data using the Flox entities. Flox's powerful logging system, event tracking, and analytics were also introduced, which will help you to take your game to the next level by issuing game updates based on the data collected.

Most of the time, it is the smaller things that may matter the most. A great follow up of the user comments, fixing the levels where the players struggle the most, creating the right balance within the game, and so on, can take your game to the next level. Analytics serve a very important role to keep your user community entertained, satisfied, as well as getting things fixed on the go. With its cross-platform dynamic nature, flexibility, scalability, and developer-friendly pricing structure, Flox is really made for all game developers. With Starling, you get the chance to use it to 100 percent of its potential.

We have reached the end of the book. I hope you had a great time reading through and checking out the code base. I believe that, at this point, you will be all set to create the next Starling hit game. I did choose an isometric game for multiple reasons. It required managing lot more art, updating the full stage for every frame, having elements with multiple states for each direction, and also having the mystic element of a pseudo 3D display. This book proves that the art can be managed, so that we do not run out of the notorious texture memory on devices such as iPad1, and the full screen rendering per frame will still give us decent FPS. Everything depends on the way you approach the code and optimize it. I hope it is clear how we iterated from the web build to the device builds. Now get started. Level up!

Index