LibGDX Cross-Platform Development Blueprints

Develop four exciting cross-platform games with increasing complexity using LibGDX and understand its key concepts

Indraneel Potnis

[PACKT] open source
PUBLISHING

LibGDX Cross-Platform Development Blueprints

Table of Contents

LibGDX Cross-Platform Development Blueprints

Credits

About the Author

About the Reviewers

www.PacktPub.com

- Support files, eBooks, discount offers, and more
  - Why subscribe?
  - Free access for Packt account holders

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
  - Downloading the example code
  - Downloading the color images of this book
  - Errata
  - Piracy
  - Questions

1. Monty Hall Simulation

- Setting up
  - Prerequisites
  - Installing the Gradle plugin
  - Using the setup app
  - Importing projects
- Introduction to the game
  - General flow of the game
  - Summary of classes

- Making the initial screen
  - Implementing the Door class
  - Implementing the GameManager class
  - Implementing the Monty class
- Taking input
  - Updating the GameManager class
  - Implementing the InputManager class
- Adding game logic
  - Finding doors with goats
  - Adding game states
- Displaying text and implementing restart
  - Displaying text
  - Implementing restart
  - Displaying the background
- Summary

2. Whack-A-Mole
- Making the initial screen
  - Implementing the Mole class
  - Implementing the GameManager class
  - Implementing the WhackAMole class
- Adding some color
  - Adding the background
  - Implementing the holes
  - Adding moles in holes
- Animating the mole
  - Jumping up and down
  - Waiting underground
- Adding randomness and taking input
  - Randomizing wait times
  - Taking input
- Adding more effects

- Stunning the mole
- Adding the stun sign
- Keeping scores and adding sounds
- Keeping scores
- Adding sound effects
- Summary

3. Catch the Ball

- Making a moving basket
- Implementing the Basket class
- Implementing the GameManager class
- Implementing the CatchTheBall class
- Moving the basket
- Throwing the ball
- Making the ball
- Adding movement
- Adding gravity
- Detecting collisions
- Colliding with the ground
- Colliding with the basket
- Throwing multiple balls
- Throwing the balls after specific intervals
- Randomizing and optimizing
- Keeping the score and maintaining the high score
- Keeping the score
- Custom fonts
- Saving high scores
- Implementing screens
- Implementing the menu screen
- Implementing screen transitions
- Implementing the Back button
- Catching the Back button

- Adding sound effects and music
  - Adding sound effects
  - Adding background music
- Summary

4. Dungeon Bob
- Creating the player
  - Implementing the Bob class
  - Implementing the GameManager class
  - Implementing the GameScreen class
- Moving the player
  - Bob’s movement on desktop
  - Continuous movement
  - Bob’s movement on mobile
- Character animation
  - Walking Bob 1
  - Walking Bob 2
- Summary

5. Using the Tiled Map Editor
- Installation and basics
  - Installing and setting up Tiled
  - Map layers and drawing
- Miscellaneous
  - Custom properties
  - Drawing objects
  - Tile animations and images
- Summary

6. Drawing Tiled Maps
- Asset management
  - Texture packer
  - The AssetManager class
- Rendering maps

Basic map rendering
Reading the map
Map objects
Summary
7. Collision Detection
Scaling objects and adding a secondary camera
Integrating Bob
Camera control
Integrating game objects
Physics and collision
Adding physics
Collision detection – 1
Collision detection – 2
Jumping
Summary
8. Collectibles and Enemies
Collecting items and detecting hazards
Collecting objects
Displaying the score and adding hazards
Enemies
Adding enemies
Adding enemies through Tiled
Enemy motion
Summary
9. More Enemies and Shooting
Skeletons and chasing
Skeletons
Chasing Bob
Shooting and stars
Stars
Shooting

Summary
10. More Levels and Effects
Multiple levels
Adding the door
Changing levels
Respawning Bob
Particle effects
Editor setup and basics
The Effect Emitters section
The Emitter Properties section
Loading the effect into the game
The loading screen
Game states
Integrating the screen in the game
Summary
Index

LibGDX Cross-Platform Development Blueprints

LibGDX Cross-Platform Development Blueprints

First published: December 2015

Production reference: 1161215

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78398-206-6

www.packtpub.com

Credits

Author

Indraneel Potnis

Reviewers

Si Fleming PhD

Stéphane Meylemans

Commissioning Editor

Kartikey Pandey

Acquisition Editor

Subho Gupta

Content Development Editors

Anand Singh

Deepti Thore

Technical Editor

Taabish Khan

Copy Editor

Rashmi Sawant

Project Coordinator

Paushali Desai

Proofreader

Safis Editing

Indexer

Hemangini Bari

Production Coordinator

Shantanu N. Zagade

Cover Work

Shantanu N. Zagade

About the Author

Indraneel Potnis is a mobile developer who lives in Mumbai. He has worked in diverse areas of the IT industry, such as web development, QA, and mobile application development.

Since childhood, he has been interested in playing computer games, and he became interested in making them in college. He made a card game called *Mendhicoat* with a friend on the Android platform and released it on the Google Play store.

I would like to thank my parents for their support in writing this book. Special thanks go to Amey Kshirsagar from Arcoiris Labs for proofreading and working on some of the art used here.

About the Reviewers

Si Fleming PhD is a principal engineer with experience in working with Java and PHP for over a decade. He holds a PhD in computer science from the University of Sussex, where his research focused on distributed systems, ad hoc social networks, Q&A, security, and privacy.

Stéphane Meylemans has a bachelor’s degree in information technology. He worked in web development for eight years and then decided to move on to mobile development (games and apps). He has learned Unreal Engine, Unity, and Android app development and is currently working on several Android apps and games as a freelancer. He is also an instructor in new technologies and works as a community manager/event organizer for a game store in Belgium.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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

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

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

Why subscribe?

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

Free access for Packt account holders

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

Preface

LibGDX is a game framework with which people can make efficient games that run on all the platforms (mobile/web/desktop) with a single code base. Games are also a big source of monetization in the mobile market. The programming language is Java, which is widely used everywhere, and is very easy to learn.

This book will focus on practical things by introducing a different variety of game projects in each chapter. This book will expose you to different areas, types, techniques, and tactics of game development.

What this book covers

Chapter 1, *Monty Hall Simulation*, discusses how to set up LibGDX and how to create a simple but a complete game from scratch.

Chapter 2, *Whack-A-Mole*, discusses some more concepts along with a game of Whack-A-Mole. These concepts include animation, stun, and sound effects.

Chapter 3, *Catch the Ball*, discusses how to make a game called Catch the Ball and covers some concepts. These concepts include motion physics, collision detection, and implementing a menu screen.

Chapter 4, *Dungeon Bob*, discusses a platformer game called Dungeon Bob and covers concepts such as character motion and character animation.

Chapter 5, *Using the Tiled Map Editor*, discusses a tool called Tiled, used to make and design 2D levels/maps.

Chapter 6, *Drawing Tiled Maps*, discusses how to draw Tiled maps in the game and covers asset management, among other things.

Chapter 7, *Collision Detection*, discusses map collision detection, camera control, and jumping effects, among other things, as we progress through the game.

Chapter 8, *Collectibles and Enemies*, discusses how to add collectibles, hazards, and enemies to our game, among other things.

Chapter 9, *More Enemies and Shooting*, discusses how to add more enemy types with intelligence and shooting, among other things.

Chapter 10, *More Levels and Effects*, discusses how to make multiple levels, a loading screen, and particle effects, among other things.

What you need for this book

The following things are required for this book:

- A computer with Internet connection
- JDK
- Eclipse with the ADT plugin installed
- Android SDK

Who this book is for

This book is for people who have a good knowledge of Java and are familiar with LibGDX. You will learn about the development of different types of games with the help of relevant concepts. You will also learn how to put these concepts in practice while making fully functional games along the way.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: “Lastly, we need to dispose `restartTexture` as well.”

A block of code is set as follows:

```
static Sprite restartSprite;
static Texture restartTexture;
static final float RESTART_RESIZE_FACTOR = 5500f;
```

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

```
public static void initialize(float width,float height){
    // other code excluded
    TextManager.initialize(width, height);
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: “Select all of the projects and click on **Finish**.”

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Downloading the color images of this book

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

Errata

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

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

Piracy

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

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

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

Questions

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

Chapter 1. Monty Hall Simulation

In this book, we are going to learn about developing games in LibGDX. LibGDX is a cross-platform Java game framework that can deploy games to desktops as well as mobile devices. We will learn the concepts and we will make four games along the way. With each game, we will learn more concepts. You need to have basic knowledge of Java and LibGDX.

We will cover the following topics in this chapter:

- Setting up
- Introduction to the game
- Creating the initial screen
- Taking inputs
- Adding the game logic
- Displaying text and implementing restart

Setting up

I’m going to briefly cover how to set up our projects in this section. So, let’s get started.

Prerequisites

Let’s see how to set up our project. We are going to work with Eclipse as the IDE. These are the prerequisites required before we set up the project. You should have them installed:

- The JDK
- Eclipse with the ADT plugin installed
- The Android SDK

Installing the Gradle plugin

We need to install the Gradle plugin for Eclipse as the LibGDX setup application is Gradle-based.

Gradle is a project management tool that allows you to configure and build projects. It handles dependencies of the projects by downloading them automatically and keeping them in a central repository on the local disk. You can easily add or remove dependencies by editing its configuration file.

To install the Gradle plugin for Eclipse, go to **Help** | **Install New Software**. A window for **Available Software** will come up. Click on the **Add** button, next to the **Work with** section. In the **Add Repository** window, type `Gradle` in the **Name** field and `http://dist.springsource.com/release/TOOLS/gradle` in the **Location** field.

Note

This link might not work on old versions of Eclipse. You can use `http://dist.springsource.com/milestone/TOOLS/gradle` instead.

Once you click on **OK**, wait until the software component list to install comes up. Select all the boxes and click on **Next**. Continue and confirm any subsequent steps. When the plugin is installed, Eclipse will have to be restarted to take effect. After restarting, the plugin will be active.

Using the setup app

Go to the LibGDX website and download the setup app from http://bitly.com/1i3C7i3. LibGDX has a setup app that downloads the framework and related libraries and creates skeleton projects for you to work with. Once you have downloaded it, you will get a JAR file. Double-click on it to open the setup.

These are the inputs that you need to provide the app:

- **Name**: This is the name of your game. We will name it `MontyHall`.
- **Package**: This is the main package of your project. We will name it `com.packtpub.montyhall`.
- **Game class**: This is the name of the main game class. We will name it `Monty`.
- **Destination**: This is the folder where you will create the projects. Give the path to a folder in your drive where you want the projects.
- **Android SDK**: This is the path where you have installed the Android SDK.
- **LibGDX Version**: Here, select the default version.
- **Sub Projects**: These checkboxes will allow you to create projects for the specified targets. Android is required as we need to keep the assets in the Android project.
- **Extensions**: These are the additional libraries that you may require in your game.

To enable the Eclipse project generation, click on **Advanced**, tick the checkbox near **Eclipse**, and click on **Save**. Click on **Generate** to start creating the projects. My setup looks like this:


Libgdx Project Generator
libGDX
PROJECT SETUP
Name: MontyHall
Package: com.packtpub.montyhall
Game class: Monty
Destination: F:/projects/MontHall
Browse
Android SDK G:\adt\sdk
Browse
LibGDX Version Release 1.5.3
Sub Projects
Desktop
Android
Ios
Html
Extensions
Bullet
Freetype
Tools
Controllers
Box2d
Box2dlights
Ashley
Ai
Advanced
Generate

Importing projects

Once the projects have been created for you, it is now time to import them in Eclipse. Open Eclipse and go to **File** | **Import** | **Gradle** | **Gradle Project**. On the **Import Gradle Project** screen, give the folder path where you want your projects to be stored. It's better if you create a new folder and give its location. Click on **Build Model** and give it some time to download the dependencies and configure the projects.

After the download is complete, you will see the projects listed. Select all of the projects and click on **Finish**. After the loading screen completes, your projects would have been successfully imported in Eclipse. That's it, you have configured the projects!

To test them out, right-click on the desktop project and go to **Run As** | **Java Application**. A window will ask you to select the Java application; select **DesktopLauncher** and click on **OK**.

Tip

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.

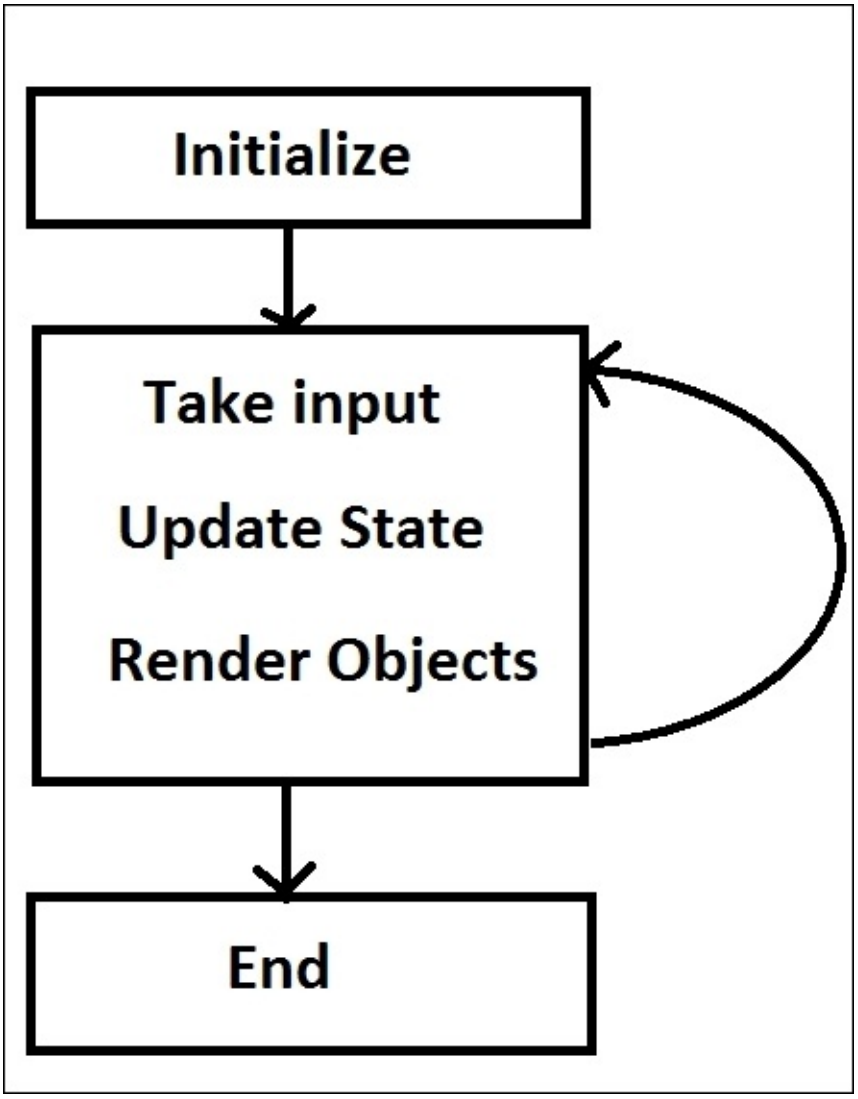
Introduction to the game

The game is a classic Monty Hall simulation. You are presented with three closed doors. Two of these doors contain a goat behind them. One door has a car behind it. The objective of the game is to win the car by correctly guessing the door. You are asked to select a door once. Once you select it, one of the doors containing the goat is opened.

You are then asked to either keep your choice or switch your selection. After you make the choice, the door you've selected is opened and you will come to know whether you have won the car or not:

General flow of the game

The following flowchart shows the general workflow of the game:



Summary of classes

The core game classes that we are going write are described in short, as follows:

- `Door`: This class is for a door that we are going to display. The player can click/tap on it to open the door.
- `InputManager`: This class handles the input detection for the game and updates appropriate structures as the game logic dictates.
- `TextManager`: This class handles all the text messages the user sees on the screen.
- `GameManager`: This class handles the initialization/reinitialization of game objects and miscellaneous game logic.

Making the initial screen

Here, we will make a basic game screen to display the door.

Implementing the Door class

To implement the `Door` class, we need to perform the following steps:

1. Create a new package in the core projects and name it `com.packtpub.gameobjects`.
2. Create a new Java class in the `com.packtpub.gameobjects` package and name it `Door`.

   Type the following code in the file:

```
package com.packtpub.gameobjects;

import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Vector2;

public class Door {

    public Sprite openSprite; //Represents sprite to display when the
door is open
    public Sprite closeSprite; //Represents sprite to display when the
door is closed
    public boolean isOpen = false; //Whether the door is open or not
    public Vector2 position = new Vector2(); // position of the door
    //door dimensions
    public float height;
    public float width;

    public void render(SpriteBatch batch){

        if(isOpen){
            openSprite.draw(batch);
        }
        else{
            closeSprite.draw(batch);
        }

    }
}
```

In this class, we declared two sprites that represent the images for the open and the closed positions of the door. We declared a Boolean variable called `isOpen` to denote the status of the door, which is initially set to closed. A Boolean variable called `isGoat` denotes whether there is a goat behind this door. The `position` variable is used to set the position where the door will be drawn; the `height` and `width` variables represent the door dimensions.

We declared a method called `render` (which takes a `SpriteBatch` to draw) that will draw the open or closed image of the door, depending on the state of `isOpen`.

Note

Whenever we draw something on the screen, we use the instance of `SpriteBatch` to do it.

The `SpriteBatch` does the job of uploading the textures to the GPU for drawing. Uploading textures is generally a computationally costly operation. `SpriteBatch` optimizes by grouping textures and uploading them to the GPU in batches.

Implementing the GameManager class

Create a new package called `com.packtpub.managers`. Create a new file called `GameManager.java` in this package. Type the following content:

```
package com.packtpub.managers;

import com.packtpub.gameobjects.Door;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.utils.Array;

public class GameManager {
    static Array<Door> doors ; // array of the 3 doors
    static Texture doorTexture; // texture image for the door
    private static final float DOOR_RESIZE_FACTOR = 2500f;
    private static final float DOOR_VERT_POSITION_FACTOR = 3f;
    private static final float DOOR1_HORIZ_POSITION_FACTOR = 7.77f;
    private static final float DOOR2_HORIZ_POSITION_FACTOR = 2.57f;
    private static final float DOOR3_HORIZ_POSITION_FACTOR = 1.52f;
    static float width,height;

    public static void initialize(float width,float height){
        GameManager.width = width;
        GameManager.height= height;
        doorTexture = new
Texture(Gdx.files.internal("data/door_close.png"));
        initDoors();
    }

    public static void renderGame(SpriteBatch batch){
        // Render(draw) each door
        for(Door door : doors)
            door.render(batch);
    }

    public static void dispose() {
        // dispose of the door texture to ensure no memory leaks
        doorTexture.dispose();
    }

    public static void initDoors(){
        doors = new Array<Door>();

        // instantiate new doors and add it to the array
        for(int i=0;i<3;i++){
            doors.add(new Door());
        }

        // set the doors' display position

doors.get(0).position.set(width/DOOR1_HORIZ_POSITION_FACTOR,height/DOOR_VER
```

```
T_POSITION_FACTOR);

doors.get(1).position.set(width/DOOR2_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

doors.get(2).position.set(width/DOOR3_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

        for(Door door : doors){
            // instantiate sprite for the //closed door with the texture of
it
            door.closeSprite = new Sprite(doorTexture);

            door.width = door.closeSprite.getWidth()*
(width/DOOR_RESIZE_FACTOR);
            door.height = door.closeSprite.getHeight()*
(width/DOOR_RESIZE_FACTOR);
            door.closeSprite.setSize(door.width, door.height);
            door.closeSprite.setPosition(door.position.x,door.position.y);
         }
    }
}
```

First, we declare an array of `door` objects, which are going to hold the instances of the three doors for us. Then, we declare a texture image of the door. We declare a constant named `DOOR_RESIZE_FACTOR` that we will use to resize the doors' dimensions. Then, we have some constants related to door positioning. We have an `initialize()` method, which we are going to use to implement the initialization logic. This method takes `width` and `height` as arguments, which are our game's viewport dimensions.

We instantiate and initialize `doorTexture` with the `door_close.png` image:

```
doorTexture = new  Texture(Gdx.files.internal("data/door_close.png"));
```

For this to work, copy the image of the closed door to your Android projects' `assets/data` folder and refresh your project. We declare a function named `initDoors()`, where we will write the initialization logic for the doors. In this function, after the `door` objects are added to the doors' arrays, we set the positions of the individual doors:

```
doors.get(0).position.set(width/DOOR1_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

doors.get(1).position.set(width/DOOR2_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

doors.get(2).position.set(width/DOOR3_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);
```

You need not use the exact same values that I have used. You can do a little bit of trial and error to set the values you want.

Next, we set the sprite for each door and its dimensions:

```
  door.closeSprite = new Sprite(doorTexture);
  door.width = door.closeSprite.getWidth()*(width/DOOR_RESIZE_FACTOR);
```

```
door.height = door.closeSprite.getHeight()*(width/DOOR_RESIZE_FACTOR);
door.closeSprite.setSize(door.width, door.height);
door.closeSprite.setPosition(door.position.x, door.position.y);
```

In the `renderGame()` method, we simply iterate each door in the `doors` array and render them. This method takes `SpriteBatch` as an argument, which is used to draw the sprites. In the `dispose()` method, we dispose of our texture to ensure that there are no memory leaks.

Implementing the Monty class

For the `Monty` class, type the following code:

```
package com.packtpub.montyhall;

import com.badlogic.gdx.ApplicationListener;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.managers.GameManager;

public class Monty implements ApplicationListener {

    private OrthographicCamera camera;
    private SpriteBatch batch;
    //viewport width and height
    private float w,h;

    @Override
    public void create() {
        // set our viewport to window dimensions
        w = Gdx.graphics.getWidth();
        h = Gdx.graphics.getHeight();
        // instantiate the camera and set the viewport
        camera = new OrthographicCamera(w,h);
        // center the camera at w/2,h/2
        camera.setToOrtho(false);

        batch = new SpriteBatch();
        //initialize the game
        GameManager.initialize(w,h);
    }

    @Override
    public void dispose() {
        //dispose the batch and the texture
        batch.dispose();
        GameManager.dispose();
    }

    @Override
    public void render() {
        // Clear the screen
        Gdx.gl.glClearColor(1, 1, 1, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

        // set the spritebatch's drawing view to the camera's view
        batch.setProjectionMatrix(camera.combined);

        // render the game objects
        batch.begin();
        GameManager.renderGame(batch);
        batch.end();
```

```
    }

    @Override
    public void resize(int width, int height) {
    }

    @Override
    public void pause() {
    }

    @Override
    public void resume() {
    }

}
```

We implement the `ApplicationListener` interface in our class and implement its methods. Since our game is going to be 2D, we will use `OrthographicCamera`. The camera is basically like an eye through which we can see the game world. We declare a `SpriteBatch` that aids us in drawing sprites. Next, we set the viewport to the dimensions of our game window, which we configured in our starter class with the following lines:

```
w = Gdx.graphics.getWidth();
h = Gdx.graphics.getHeight();
```

We then instantiate the camera and set its viewport. The camera is centered at viewport *width(w)/2* and viewport *height(h)/2* by the following line:

```
camera.setToOrtho(false);
```

The `GameManager` class' `initialize()` method is called to set up game objects and their properties. In the `render` method. we first clear the screen with a white color with the following code:

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

We set the spritebatch's drawing view to that of the camera's view with the following code:

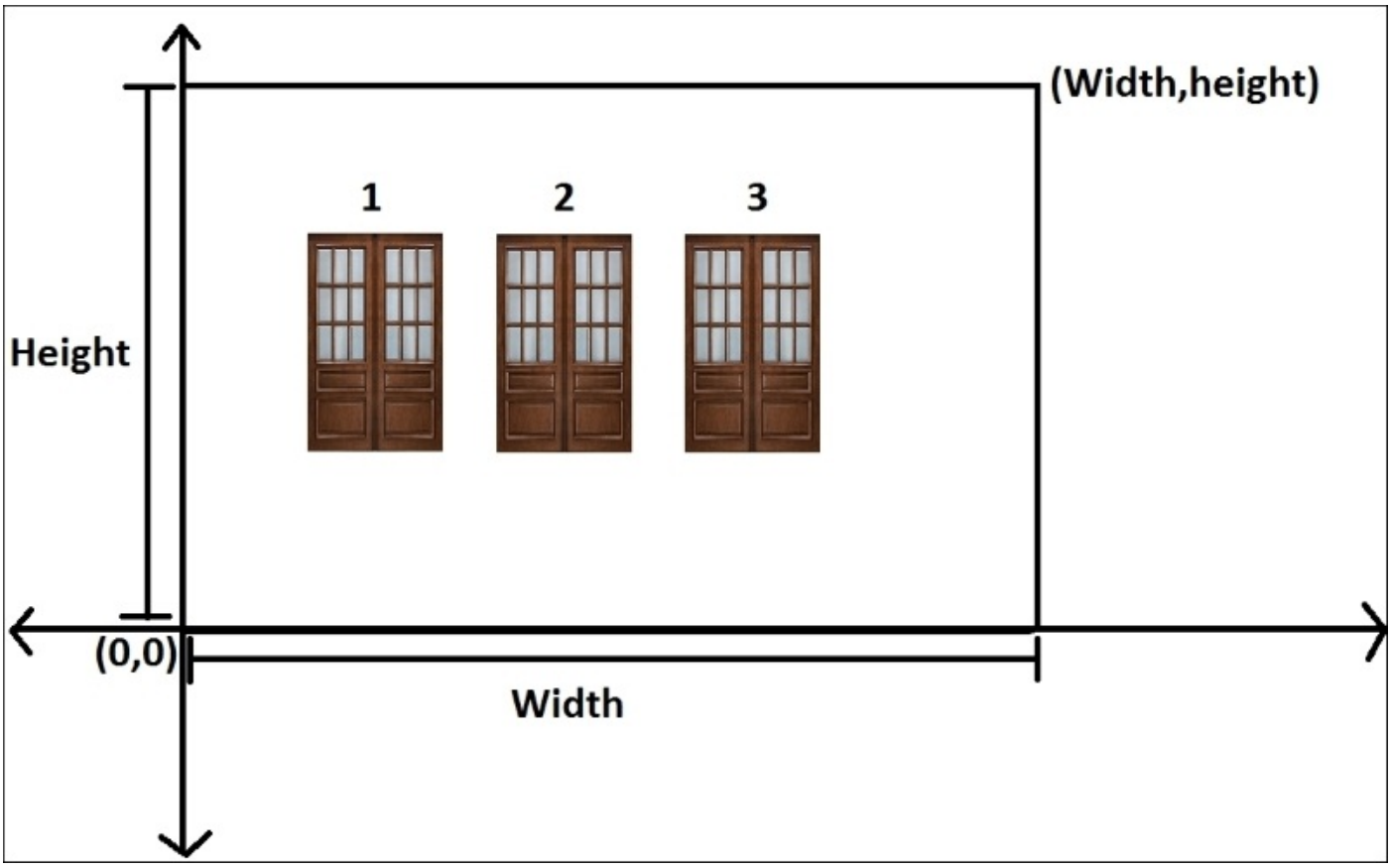
```
batch.setProjectionMatrix(camera.combined);
```

The `GameManager` class' `render()` method is called between the spritebatch's `begin()` and `end()` calls to actually draw the images.

Tip

Draw calls should only be made for a particular `SpriteBatch` between its `begin()` and `end()` calls.

Lastly, in the `dispose()` method, we dispose of the `SpriteBatch` and call the `GameManager` class' `dispose()` method to clean up the memory when the game is exiting. The following figure is a representation of how the game would look conceptually in the coordinate point of view:


(Width,height)
1
2
3
Height
(0,0)
Width

Taking input

We are now going to see how to capture the touch/click input and open a door when it has been touched.

Updating the GameManager class

We have to make changes to the structure of our `GameManager` class. The changed/added lines are highlighted in the following code:

```
package com.packtpub.managers;

import com.packtpub.gameobjects.Door;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Vector3;
import com.badlogic.gdx.utils.Array;

public class GameManager {
    static Array<Door> doors ; // array of the 3 doors
    static Texture doorTexture; // texture image for the door
    static Texture carTexture;  // texture image for the car
    static Texture goatTexture; // texture image for the goat
    static Vector3 temp = new Vector3(); // temp vector to store input
coordinates

    private static final float DOOR_RESIZE_FACTOR = 2500f;
    private static final float DOOR_VERT_POSITION_FACTOR = 3f;
    private static final float DOOR1_HORIZ_POSITION_FACTOR = 7.77f;
    private static final float DOOR2_HORIZ_POSITION_FACTOR = 2.57f;
    private static final float DOOR3_HORIZ_POSITION_FACTOR = 1.52f;
    static float width,height;

    public static void initialize(float width,float height){
        GameManager.width = width;
        GameManager.height= height;
        doorTexture = new
Texture(Gdx.files.internal("data/door_close.png"));
        carTexture = new
Texture(Gdx.files.internal("data/door_open_car.png"));
        goatTexture = new
Texture(Gdx.files.internal("data/door_open_goat.png"));

        initDoors();
    }

    public static void renderGame(SpriteBatch batch){
        // Render(draw) each door
        for(Door door : doors)
            door.render(batch);
    }

    public static void dispose() {
        // dispose of the textured to ensure no memory leaks
        doorTexture.dispose();
        carTexture.dispose();
        goatTexture.dispose();
```

```
    }

    public static void initDoors(){
        doors = new Array<Door>();

        // instantiate new doors and add it to the array
        for(int i=0;i<3;i++){
            doors.add(new Door());
        }

        // set the doors' display position

doors.get(0).position.set(width/DOOR1_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

doors.get(1).position.set(width/DOOR2_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

doors.get(2).position.set(width/DOOR3_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

        for(Door door : doors){
            // instantiate sprite for //the closed door with the texture of
it
            door.closeSprite = new Sprite(doorTexture);

            door.openSprite = new Sprite();
            door.width = door.closeSprite.getWidth()*
(width/DOOR_RESIZE_FACTOR);
            door.height = door.closeSprite.getHeight()*
(width/DOOR_RESIZE_FACTOR);
            door.closeSprite.setSize(door.width, door.height);
            door.closeSprite.setPosition(door.position.x,door.position.y);
            //set the dimensions for the open door
            door.openSprite.setSize(door.width, door.height);
            door.openSprite.setPosition(door.position.x, door.position.y);
        }
        //setting the textures for the open doors
        doors.get(0).openSprite.setRegion(goatTexture);
        doors.get(1).openSprite.setRegion(carTexture);
        doors.get(2).openSprite.setRegion(goatTexture);
    }
}
```

We declared two more textures: one for the image of an open door with a goat behind it and the other for the image of an open door with a car behind it. We declared a vector to store the input coordinates of the point on the screen where the user has touched or clicked. This will help us determine which door has been selected:

```
static Vector3 temp = new Vector3();
```

Next, we load the textures with the images of a goat behind the door and a car behind the door in the `initialize()` method. We then set the size and positions of the corresponding sprites for each door in `initDoors()`.

In the `dispose()` method, we dispose of the car and goat texture when they are not in use.

Implementing the InputManager class

Create a new `InputManager` class in the `com.packtpub.managers` package and type in the following code:

```
package com.packtpub.managers;

import com.packtpub.gameobjects.Door;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.OrthographicCamera;

public class InputManager {

    public static void handleInput(OrthographicCamera camera){
        // Check if the screen is touched
        if(Gdx.input.justTouched()){
            // Get input touch coordinates// and set the temp vector with
these values
            GameManager.temp.set(Gdx.input.getX(),Gdx.input.getY(), 0);
            //get the touch coordinates //with respect to the camera's
viewport
            camera.unproject(GameManager.temp);

            float touchX = GameManager.temp.x;
            float touchY=  GameManager.temp.y;

            //iterate the doors array and //check if we
tapped/touched/clicked on any door
            for(int i=0;i<GameManager.doors.size;i++){
                Door door = GameManager.doors.get(i);
                // only check if the door is closed
                if(!door.isOpen){

                    if(handleDoor(door,touchX,touchY)){
                        break;
                    }
                }
            }

        }
    }

    public static boolean handleDoor(Door door,float touchX,float touchY){

        //check whether the touch //coordinates lie on the door's bounds
        if((touchX>=door.position.x) && touchX<=
(door.position.x+door.width) && (touchY>=door.position.y) && touchY<=
(door.position.y+door.height) ){
            //open the door if it is touched/clicked
            door.isOpen=true;
            return true;

        }
    return false;
    }
```

```
}
```

We have two methods in this class: `handleInput()` and `handleDoor()`. In `handleInput()`, we first check whether the game screen has just been touched using the following line of code:

```
Gdx.input.justTouched()
```

This function checks whether any new touch/click event has been detected on the game screen. If it is detected, we start processing it. We get the input touch/click coordinates using the following methods:

- `Gdx.input.getX()`
- `Gdx.input.getY()`

We set the temporary vector with these *x* and *y* values. This is a 3D vector, and since we are dealing with 2D here, we set the *z* coordinate to `0`:

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

Now, you might wonder why we need to store 2D touch coordinates in a 3D vector and what the need of storing them in a vector is anyway. The answer comes from the way LibGDX gives us input coordinates. It gives us the pixel coordinates that correspond to the actual size of the screen. In our case, the viewport size matches the screen size exactly. But you may have viewports of different dimensions other than the screen. We will get erroneous results in that case.

To understand this, imagine the screen is 800 x 600 px and we have set up the game world to 400 x 300 units, that is, half the size. Now, if we click on the center, we might expect to get the touch coordinates as (200, 150). But instead, LibGDX will give us (400, 300) since it is the actual center of the game screen pixel-wise. To avoid this, we convert screen coordinates to viewport coordinates using the following method:

```
camera.unproject(GameManager.temp);
```

This method needs a 3D vector as an argument. It converts the coordinates and stores them in the same vector, so we can then use them in our input handling logic. This method is called on the game's camera, so it is passed to the `handleInput()` method. Next, we iterate over the `doors` array to check which door we have touched. Here, we only check closed doors using the `handleDoor()` function:

```
handleDoor(door,touchX,touchY))
```

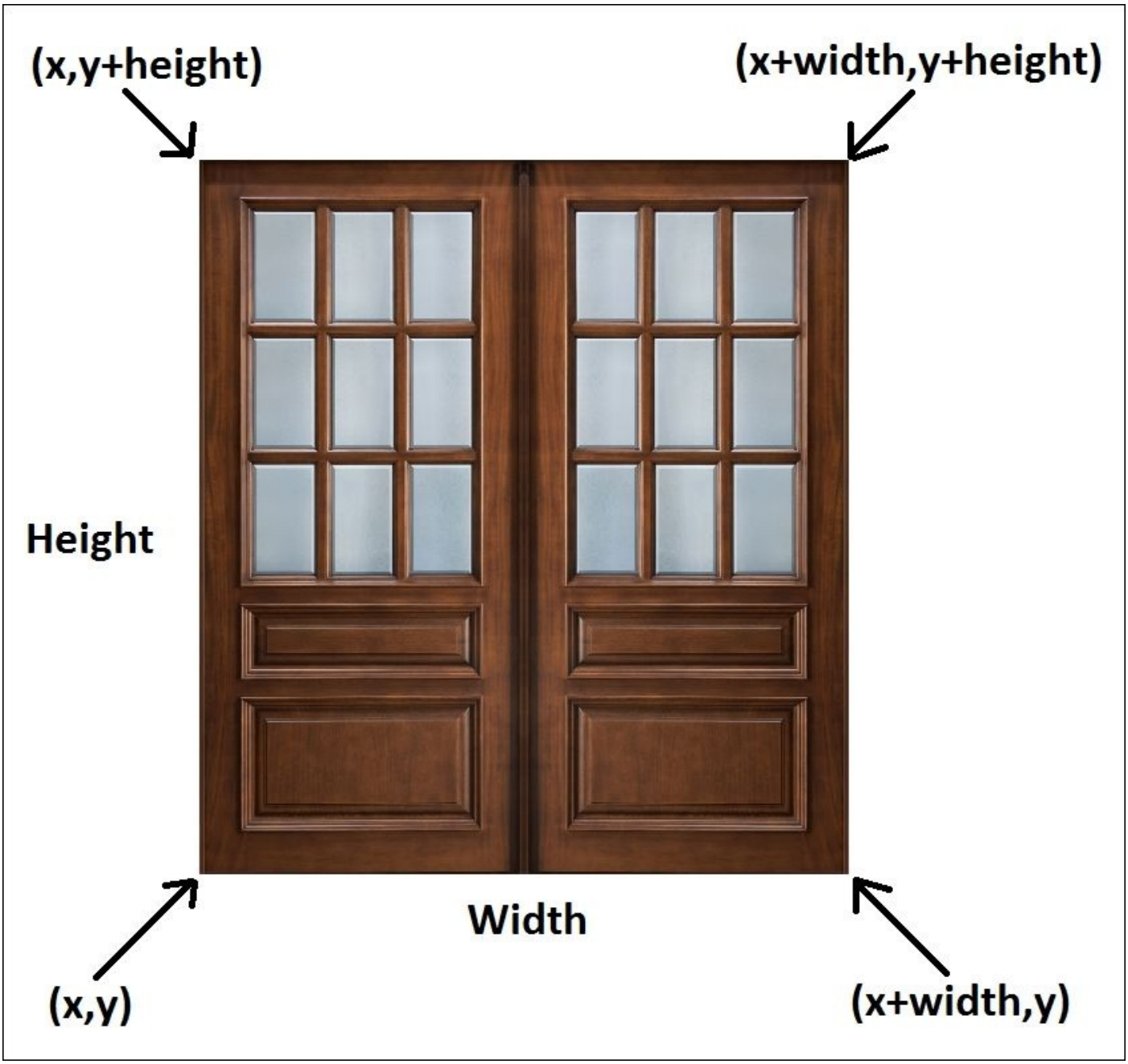
In this method, we check whether the touch point lies on a particular door. We do this by checking horizontal bounds, as follows:

```
(touchX>=door.position.x) && touchX<=(door.position.x+door.width)
```

We do this by checking vertical bounds, as follows:

```
(touchY>=door.position.y) && touchY<=(door.position.y+door.height)
```

This is explained in the following figure:



If the point lies on the door, we change its state to open so that we can display the image when the door is open. This method returns `true` if a door was touched and `false` otherwise. This is handled by the `handleInput()` method that stops iterating over the doors via `break`. Since we are not detecting multiple touches, only one door can be touched at a time; we stop detecting touches for other doors once we find the door that we have touched.

We just need to make a small addition to the `render()` method of our `Monty` class to include the input detection by calling the `InputManager` class' `handleInput()` method:

```
batch.setProjectionMatrix(camera.combined);
InputManager.handleInput(camera);
  // render the game objects
  batch.begin();
```

Adding game logic

Here, we will add some logic to the game such as searching for doors that have goats behind them and the addition of game states for easy management of the code.

Finding doors with goats

First of all, we need to declare a Boolean variable in our `Door` class called `isGoat` so that we can determine whether a goat is behind the door:

```
  public boolean isGoat = false; // indicates whether a goat is behind the
door
  public Vector2 position =new Vector2(); // position of the door
  //door dimensions
  public float height;
```

Next, we need to set this variable for each door appropriately in the `initDoors()` method of the `GameManager` class:

```
  doors.get(0).openSprite.setRegion(goatTexture);
  doors.get(0).isGoat= true;
  doors.get(1).openSprite.setRegion(carTexture);
  doors.get(1).isGoat= false;
  doors.get(2).openSprite.setRegion(goatTexture);
  doors.get(2).isGoat= true;
```

When you select a door, another door that has a goat behind it is revealed. If there are two doors remaining that have goats behind them, a door is chosen among them in random. To find this, we are going to write a function called `getGoatIndices()`. In this method, we find door indices that have goats and store them in an array of integers, which is declared in our `GameManager` class:

```
   static Texture goatTexture; // texture image for the goat
   static Vector3 temp = new Vector3(); // temp vector to store input
coordinates
   static IntArray goatIndices; // array of integer to store door indices
with goats
```

Tip

We use the `IntArray` class to store a list of integers. This is a LibGDX type that provides utility functions similar to `ArrayList`, but at the same time, it is faster and more memory-efficient.

We will instantiate it in the `initialize()` method of the `GameManager` class:

```
       goatIndices = new IntArray();
```

We now define the `getGoatindices()` method in `GameManager`. This method takes the index of the door that is currently selected as an argument:

```
/** Find doors containing goats from the remaining doors */
public static IntArray getGoatindices(int selectedDoorIndex){
    goatIndices.clear(); // remove all previous values from the array

    for(int i=0;i<doors.size;i++){
        // exclude selected door
        if(i!=selectedDoorIndex && doors.get(i).isGoat) {
            goatIndices.add(i);
        }
```

```
    }

    return goatIndices;
}
```

We go over the elements of the `doors` array to the left-hand side of the selected element and add the index if a goat is found behind the door. The same goes for the right-hand side. We return the array of indices at the end of the method.

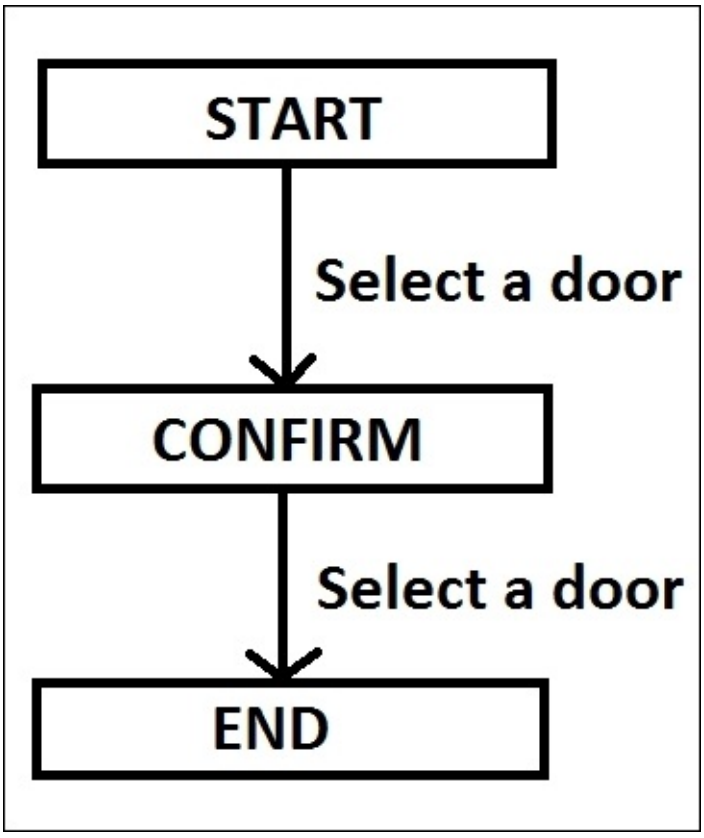
Adding game states

We have three states in the game: `START`, `CONFIRM`, and `END`:

- `START`: This is the initial state where all the doors are closed. Once the user selects a door, a different door that has a goat behind it is opened. The state changes to `CONFIRM`.
- `CONFIRM`: In this state, the user has the option to stay (select the same door again) or switch (select the remaining closed door). Once a door has been selected, it is opened. At this point, the game's logic checks whether the user has won or lost and the state changes to `END`.
- `END`: In this state, the game is basically over and the user just sees the message whether he has won or lost.

At the `START` state, which is the initial state, the initial screen is shown with all the three doors closed, where the user is supposed to select a door. Once you select a door, the game goes into the `CONFIRM` state, where another door with a goat behind it is opened. At this time, you have the option to stay with your decision (the door you chose) or switch your decision and choose the other closed door.

Once you make the decision and choose a door, it is opened, and you will know whether you have won or lost. The game goes into the `END` state and it is over:



We will declare the three states by an enum and a variable level in the `GameManager` class:

```
public static enum Level {
	START,
```

```
        CONFIRM,
        END
    }
static Level level;
```

We initialize the level with `START` in the `initialize()` method of the `GameManager` class:

```
level = Level.START;
```

We are going to slightly change the signature of the `handleDoor()` method to include the index of the door that we want to check in the `InputManager` class:

```
public static boolean handleDoor(Door door,float touchX,float touchY,int doorIndex)
```

This index is passed on by the `handleInput()` method:

```
handleDoor(door, touchX,touchY,i)
```

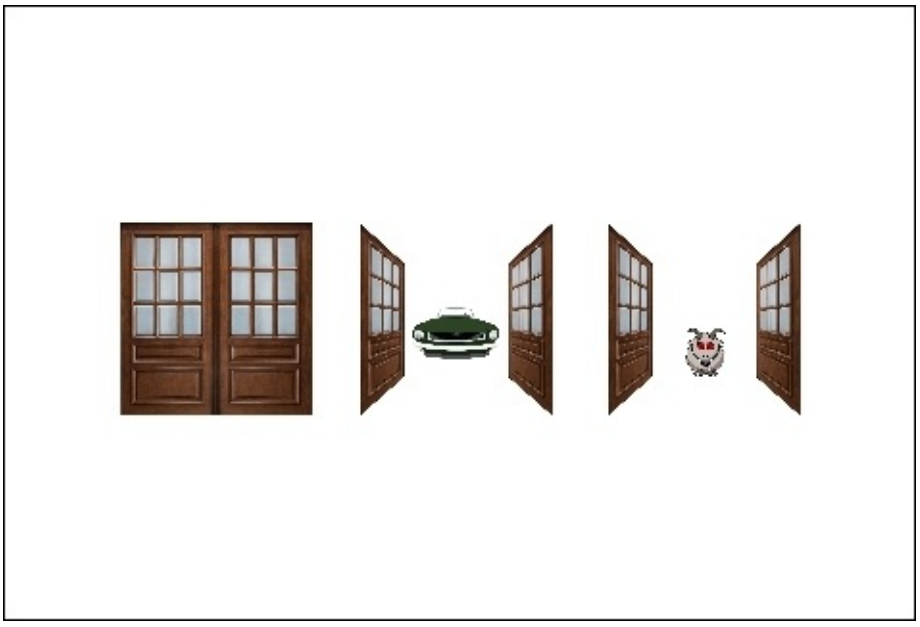
We will implement further logic by updating the `handleDoor()` function:

```
public static boolean handleDoor(Door door,float touchX,float touchY,int doorIndex){

    // check whether the touch coordinates lie on the door's bounds
    if((touchX>=door.position.x) && touchX<=(door.position.x+door.width) && (touchY>=door.position.y) && touchY<=(door.position.y+door.height) ){

        switch(GameManager.level){
            case START:
                // open a random door from the //remaining doors once the user selects a door

GameManager.doors.get(GameManager.getGoatindices(doorIndex).random()).isOpen=true;
                // change the state to confirm
                GameManager.level = GameManager.Level.CONFIRM;
                break;
            case CONFIRM:
                door.isOpen=true; // open the selected door
                GameManager.level = GameManager.Level.END; // change the state to end
                break;
        }

        return true;

    }

    return false;
}
```

We write a `switch` statement that checks the game state. If the state is `START`, then the user hasn't selected any door as of yet. So, after the user makes his selection, we use the `getGoatindices()` function to get the door indices, which have a goat behind them. Then, we randomly select one of them and get the door that corresponds to it from the `doors`

array. We then open this door and set the game state to `CONFIRM`.

If the state is `CONFIRM`, then the user has confirmed his selection. In this case, we open the door that he has selected and change the state to `END`:



Displaying text and implementing restart

Here, we will learn how to display text messages on the screen and implement the restart functionality. The messages give a visual cue to the user about the state of the game and what action he is supposed to take. The restart functionality will allow the user to restart the game from any point.

Displaying text

We have created the logic and different game states, but we haven't shown the user whether he has won or lost. Also, we don't show which state he is in and what action he is supposed to take. In this section, we are going to learn how to display text. LibGDX has a class called `BitmapFont` that enables us to draw text on the screen.

We are going to create a class called `TextManager`, which is going to handle all the text rendering. But first, we will make some changes to the code. We will introduce a new Boolean variable in the `GameManager` class called `hasWon` to indicate whether the user has won the game or not:

```
static Level level;
static boolean hasWon=false;
static Array<Door> doors ; // array of the 3 doors
```

To set the variable, we are going to implement the logic in `InputManager`. We will highlight the changes in the `handleDoor()` function:

```
case CONFIRM:
    door.isOpen=true; // open the selected door
    GameManager.level = GameManager.Level.END; // change the state to end
    if(!door.isGoat){
        GameManager.hasWon=true;
    }
    break;
```

Now, create a new class in the `com.packtpub.managers` package and name it `TextManager` and type the following code:

```
package com.packtpub.managers;

import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

public class TextManager {
    static BitmapFont font; // we draw the text to the screen //using this
variable
    // Texts corresponding to different states
    static String start = "Select a door";
    static StringBuffer confirm;
    static String win = "You Win!";
    static String lose = "You Lose!";
    // viewport width and height
    static float width,height;

    public static void initialize(float width,float height){

        TextManager.width = width;
        TextManager.height= height;
        //set the font color to cyan

        font = new BitmapFont();
```

```
        font.setColor(Color.CYAN);

        //scale the font size according to screen width
        font.scale(width/1600f);

        confirm = new StringBuffer( (String) "You selected door no.Do you
want to switch or stay?");

    }

    public static void displayMessage(SpriteBatch batch){
        // draw the text based on the game state
        switch(GameManager.level){
        case START:
            // calculations to center the text on the screen
            font.draw(batch, start, (width/2 -
font.getBounds(start).width/2),
GameManager.doors.first().closeSprite.getY()/2 +
font.getBounds(start).height/2 );
            break;
        case CONFIRM:
            font.draw(batch, confirm, (width/2 -
font.getBounds(confirm).width/2),
GameManager.doors.first().closeSprite.getY()/2 +
font.getBounds(confirm).height/2 );
            break;
        case END:
            // draw win/lose text based on the status
            if(GameManager.hasWon)
              font.draw(batch, win,(width/2 - font.getBounds(win).width/2),
GameManager.doors.first().closeSprite.getY()/2 +
font.getBounds(win).height/2);
            else
              font.draw(batch,lose,(width/2 –
font.getBounds(lose).width/2),
GameManager.doors.first().closeSprite.getY()/2 +
font.getBounds(lose).height/2);
            break;
        }

    }

    public static void setSelectedDoor(int doorIndex){
        // insert selected door number into confirm display text
        confirm.insert(confirm.indexOf("door no")+"door no".length(), " "+
(doorIndex+1));
    }

}
```

We declare a `BitmapFont` instance that we are going to use. `BitmapFont` is the class used to display text in LibGDX. Next, we will define the strings that we are going to display when the user is in the `START` state, that is, either he wins or he loses.

We are going to use `StringBuffer` to store the `CONFIRM` state's text. This is because we need to modify it in between so that we can include the door number the user has selected.

We declare the `width` and `height` of the screen. These variables will be required to correctly position our text.

We create a function called `initialize()` to set initial values of some variables. This method takes `height` and `width` as arguments, which are passed to it by `GameManager`:

```
public static void initialize(float width,float height)
```

We set the dimensions of our viewport at the first two lines in this method. Next, we instantiate our font. By default, the font used is 15pt Arial:

```
font = new BitmapFont();
```

We can also use custom fonts and styles, which we will explore in later chapters. We set the font's color to Cyan. You can set any color you want:

```
font.setColor(Color.CYAN);
```

We can also scale the font according to our requirement using the `font.scale()` method:

```
font.scale(width/1600f);
```

This method is called in the `GameManager` class' `initialize()` method and the viewport's dimensions are passed to it:

```
public static void initialize(float width,float height){
    // other code excluded
    TextManager.initialize(width, height);
}
```

We created a method called `setSelectedDoor()` to insert the door number the user has selected in the `CONFIRM` state's text. This is done so that we can let the user know which door he has selected. We insert the door index of the selected door after the words *door no*.

This method is called in the `InputManager` class' `handleDoor()` function, where we check whether a door has been selected by the user:

```
GameManager.level = GameManager.Level.CONFIRM; // change the state // to
confirm
TextManager.setSelectedDoor(doorIndex);
```

We declare a new method called `displayMessage()` that takes a `SpriteBatch` used to draw. In this method, we are going to use the `switch` case to determine which text to draw based on the state of the game.

If the state is `START` or `CONFIRM`, we display the corresponding text. If it is `END`, we display the win/lose message, depending on the status:

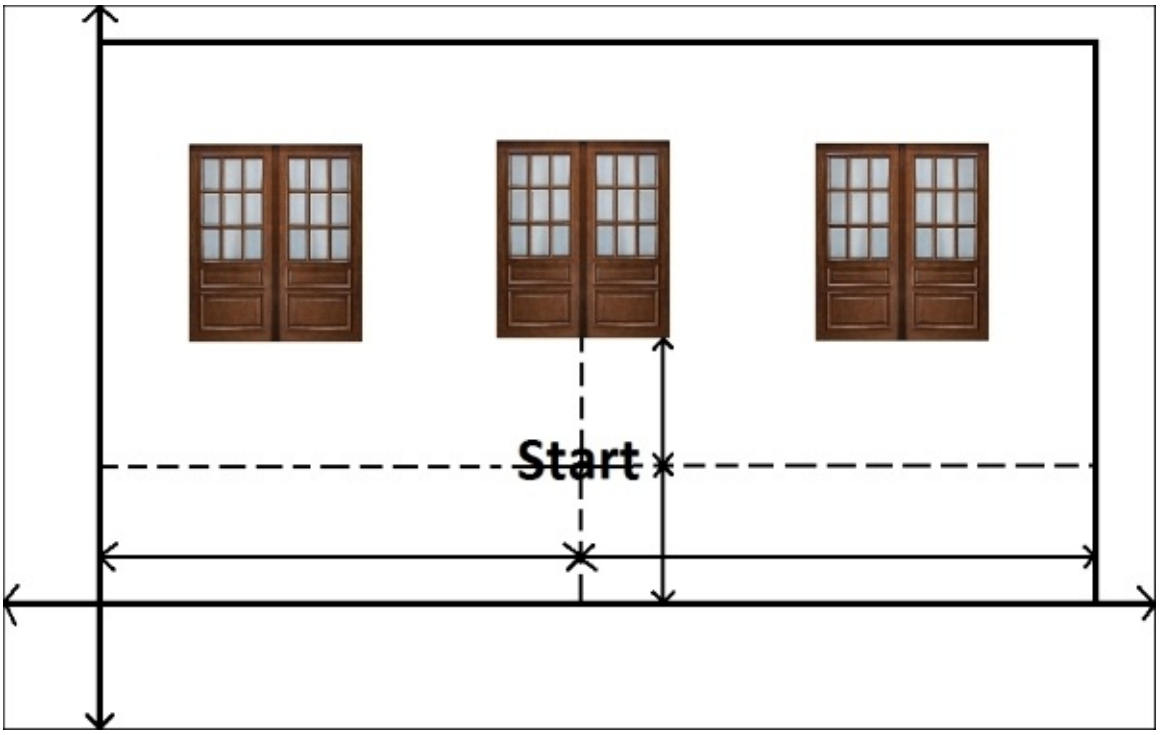
```
public static void displayMessage(SpriteBatch batch){
       // draw the text based on the game state
      switch(GameManager.level){
      case START:
               // calculations to center the text on the screen
               font.draw(batch, start,(width/2 –
```

```
font.getBounds(start).width/2), (height+
GameManager.doors.first().closeSprite.getY()+GameManager.doors.first().clos
eSprite.getHeight())/2 - font.getBounds(start).height/2);
                break;
      case CONFIRM:
                font.draw(batch, confirm,(width/2 –
font.getBounds(confirm).width/2), (height+
GameManager.doors.first().closeSprite.getY()+GameManager.doors.first().clos
eSprite.getHeight())/2 - font.getBounds(confirm).height/2);
                break;
      case END:
                // draw win/lose text based on the status
                  if(GameManager.hasWon)
                font.draw(batch, win,(width/2 –
font.getBounds(win).width/2), (height+
GameManager.doors.first().closeSprite.getY()+GameManager.doors.first().clos
eSprite.getHeight())/2 - font.getBounds(win).height/2);
                else
                font.draw(batch, lose,(width/2 –
font.getBounds(lose).width/2), (height+
GameManager.doors.first().closeSprite.getY()+GameManager.doors.first().clos
eSprite.getHeight())/2 - font.getBounds(lose).height/2);
                break;
      }

   }
```

We are using a `font.draw()` method, which takes four arguments: the `SpriteBatch` (used to draw), the message text, and the *x* and *y* positions of the displayed text. The *x* and *y* positions are calculated in such a way that the text displayed is always in the center. To correctly position the text, we use the `BitmapFont` class' `getBounds()` method. This method is called on the particular font instance and the text to be displayed is passed to it. It calculates the size of the text and returns it. An example of this is as follows:

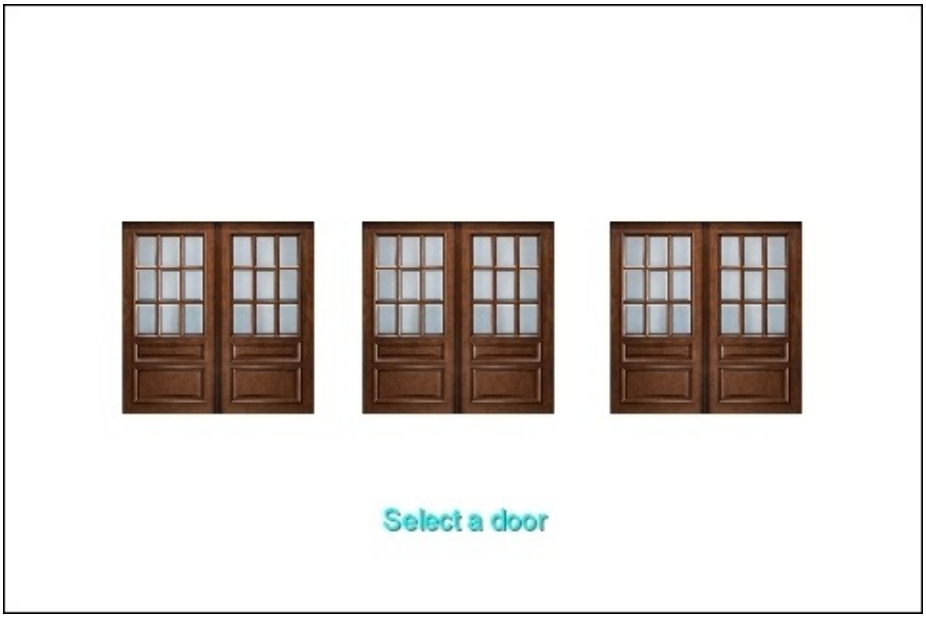
```
font.draw(batch, win,(width/2 - font.getBounds(win).width/2),
GameManager.doors.first().closeSprite.getY()/2 +
font.getBounds(win).height/2);
```

Let's take a look at the following figure:



We can also display text on multiple lines. We have to use `\n` in our strings where we want the line break. To display this, we have to use the `font.drawMultiline()` method instead of `font.draw()`. Finally, we need to call the `displayMessage()` of `TextManager` in the `GameManager` class' `renderGame()` method. Add the following line at the end of the `renderGame()` method:

```
TextManager.displayMessage(batch);
```



Implementing restart

Let's add a restart functionality to our game. We will display a button for restart, and when the user clicks on it, our game will be restarted. First, let's copy the image that we will use for the restart button to our project. I am using `restart.png` and this image needs to be copied to the Android project's `assets/data` folder. All the assets would be available for you in the code bundle accompanied with the book.

Let's declare a sprite called `restartSprite` and a corresponding texture in our `GameManager` class. We also declare a constant named `RESTART_RESIZE_FACTOR`. This will be used to resize the restart sprite based on screen dimensions. We will initialize the sprite and set its parameters in the `initialize()` method of the `GameManager` class:

```
static Sprite restartSprite;
static Texture restartTexture;
static final float RESTART_RESIZE_FACTOR = 5500f;
```

The `initialize()` method will be as follows:

```
restartTexture = new Texture(Gdx.files.internal("data/restart.png"));
restartSprite = new Sprite(restartTexture);
restartSprite.setSize(restartSprite.getWidth()*width/RESTART_RESIZ
E_FACTOR,restartSprite.getHeight()*width/RESTART_RESIZE_FACTOR);
restartSprite.setPosition(0 ,0);
```

The drawing of `restartSprite` will occur in the `renderGame()` method:

```
restartSprite.draw(batch);
```

Let's write a function for handling the user clicks/touches on the restart button. We will create a function called `handleRestart()` in the `InputManger` class:

```
public static void handleRestart(float touchX,float touchY){
    // determine if the user has clicked/touched the restart button
    if((touchX>=GameManager.restartSprite.getX()) && touchX<=
(GameManager.restartSprite.getX()+GameManager.restartSprite.getWidth()) &&
(touchY>=GameManager.restartSprite.getY()) && touchY<=
(GameManager.restartSprite.getY()+GameManager.restartSprite.getHeight()) ){

        GameManager.restartGame();
    }
}
```

This method is called by the `handleInput()` function and takes inputs *x* and *y* coordinates as arguments:

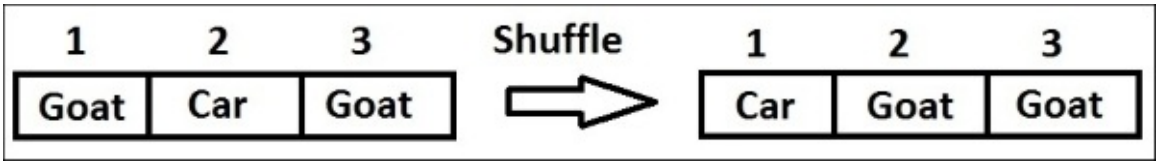
```
handleRestart(touchX,touchY);  // in handleInput() method.
```

Once it is determined that the button is touched using bounds checking, as described previously, we will call the `restartGame()` function, which will be defined in the `GameManager` class:

```
public static void restartGame(){
    // shuffle the positions of the doors inside the doors array
```

```
        doors.shuffle();

    // reset the door positions

doors.get(0).position.set(width/DOOR1_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

doors.get(1).position.set(width/DOOR2_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

doors.get(2).position.set(width/DOOR3_HORIZ_POSITION_FACTOR,height/DOOR_VER
T_POSITION_FACTOR);

    for(int i=0;i<GameManager.doors.size;i++){
        GameManager.doors.get(i).isOpen=false;
        // reset the sprite positions

GameManager.doors.get(i).closeSprite.setPosition(GameManager.doors.get(i).p
osition.x, GameManager.doors.get(i).position.y);

GameManager.doors.get(i).openSprite.setPosition(GameManager.doors.get(i).po
sition.x, GameManager.doors.get(i).position.y);

    }
    GameManager.hasWon=false;
    // reset the level
    GameManager.level = GameManager.level.START;
    TextManager.confirm = new StringBuffer( (String) "You selected door
no.Do you want to switch or stay?");
}
```

We call the `shuffle()` function on our `doors` array. This is done so that every time we restart, the doors will be positioned in a different order:



Next, we set the `isOpen` status of each door to `false`, marking them closed, which was the initial configuration. The positions of the sprites are reset to their original positions as well. Finally, we reset `hasWon` to `false`, the `CONFIRM` state's string to its original text, and the game's level to `START`.

Lastly, we need to dispose of `restartTexture` as well. So, we call it in the `dispose()` method, as follows:

```
restartTexture.dispose();
```

Displaying the background

We have done everything in our game except the background. To finish off, we will quickly add a background image to the game to make it look better. In the `GameManager` class, type the following code:

```
static Texture backtexture;
static Sprite backSprite;
```

In the `initialize()` method of the `GameManager` class, add the following code:

```
backtexture = new Texture(Gdx.files.internal("data/background.jpg"));
backSprite = new Sprite(backtexture);
backSprite.setSize(width, height);
backSprite.setPosition(0,0f);
```

We will set `width` and `height` to the same values as that of our viewport to cover the whole screen. Let’s draw our background in the `renderGame()` method of the same class:

```
backSprite.draw(batch);
// Render(draw) each door
```

Remember to draw the background before drawing anything else. Otherwise, it will overlay on top of our game objects. Lastly, we need to dispose of the texture in the `dispose()` method:

```
backtexture.dispose();
```

Summary

In this chapter, we learned how to set up LibGDX, and we created a simple but complete game from scratch. We learned some of the concepts in LibGDX, such as the following:

- Displaying images
- Capturing mouse/touch inputs
- Displaying text

In the next few chapters, we will learn more about LibGDX and game development concepts progressively while making awesome games!

Chapter 2. Whack-A-Mole

In this chapter, we are going to learn how to make a Whack-A-Mole style game. The objective of the game is to whack as many moles as possible. The moles come up and go down from the ground through holes. The user can tap on the moles to hit/whack them. The user gets points on whacking the moles and the points are displayed on the screen.

We will cover the following topics in this chapter:

- Making the initial screen
- Adding some color
- Animating the mole
- Adding randomness and taking inputs
- Adding more effects
- Keeping scores and adding sounds

Making the initial screen

We will implement a basic game screen with some moles.

Implementing the Mole class

We will set up a new project with the LibGDX setup, which is similar to the following:

Let’s see how to implement the mole in our game. Create a new package in the core projects and name it `com.packtpub.whackamole.gameobjects`. Create a new Java class in this package and name it `Mole`.

Type the following code in the file:

```
package com.packtpub.whackamole.gameobjects;

import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Vector2;
```

```
public class Mole {
    public Sprite moleSprite; //sprite to display the mole
    public Vector2 position =new Vector2();// The mole's position
    public float height,width;  // the mole's dimensions

    public void render(SpriteBatch batch){

    moleSprite.draw(batch);

    }
}
```

This class defines the mole. We declare the sprite for the mole, its position, and its dimensions. In the `render` method, we draw the mole using the `SpriteBatch`. The code for this class is just the start, and we are going to add more members and methods to it as we go further.

Implementing the GameManager class

Create a new package called `com.packtpub.whackamole.managers`. Create a new file named `GameManager.java` in this package. Type the following content:

```
package com.packtpub.whackamole.managers;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.utils.Array;
import com.packtpub.whackamole.gameobjects.Mole;

public class GameManager {
    static Array<Mole>moles; // array of the moles
    static Texture moleTexture; // texture image for the mole

    private static float MOLE_RESIZE_FACTOR = 2500f;

    private static float MOLE_VERT_POSITION_FACTOR = 3f;
    private static float MOLE1_HORIZ_POSITION_FACTOR = 5.8f;
    private static float MOLE2_HORIZ_POSITION_FACTOR = 2.4f;
    private static float MOLE3_HORIZ_POSITION_FACTOR = 1.5f;

    public static void initialize(float width,float height){

        moles = new Array<Mole>();
        moleTexture = new Texture(Gdx.files.internal("data/mole.png"));
        // instantiate new moles and add it to the array
        for(int i=0;i<3;i++){
            moles.add(new Mole());
        }

        // set the mole's display position

moles.get(0).position.set(width/MOLE1_HORIZ_POSITION_FACTOR,height/MOLE_VER
T_POSITION_FACTOR);

moles.get(1).position.set(width/MOLE2_HORIZ_POSITION_FACTOR,height/MOLE_VER
T_POSITION_FACTOR);

moles.get(2).position.set(width/MOLE3_HORIZ_POSITION_FACTOR,height/MOLE_VER
T_POSITION_FACTOR);

        for(Mole mole : moles){
            // instantiate sprite for the mole with the texture of it
            mole.moleSprite = new Sprite(moleTexture);

            //set the dimensions for the mole
            mole.width = mole.moleSprite.getWidth()*
(width/MOLE_RESIZE_FACTOR);
            mole.height = mole.moleSprite.getHeight()*
(width/MOLE_RESIZE_FACTOR);
            mole.moleSprite.setSize(mole.width, mole.height);
```

```
            mole.moleSprite.setPosition(mole.position.x, mole.position.y);

        }
    }

    public static void renderGame(SpriteBatch batch){

        // Render(draw) each mole
        for(Mole mole : moles)
            mole.render(batch);

    }
    public static void dispose() {
        // dispose of the mole texture to ensure no memory leaks
        moleTexture.dispose();

    }
}
```

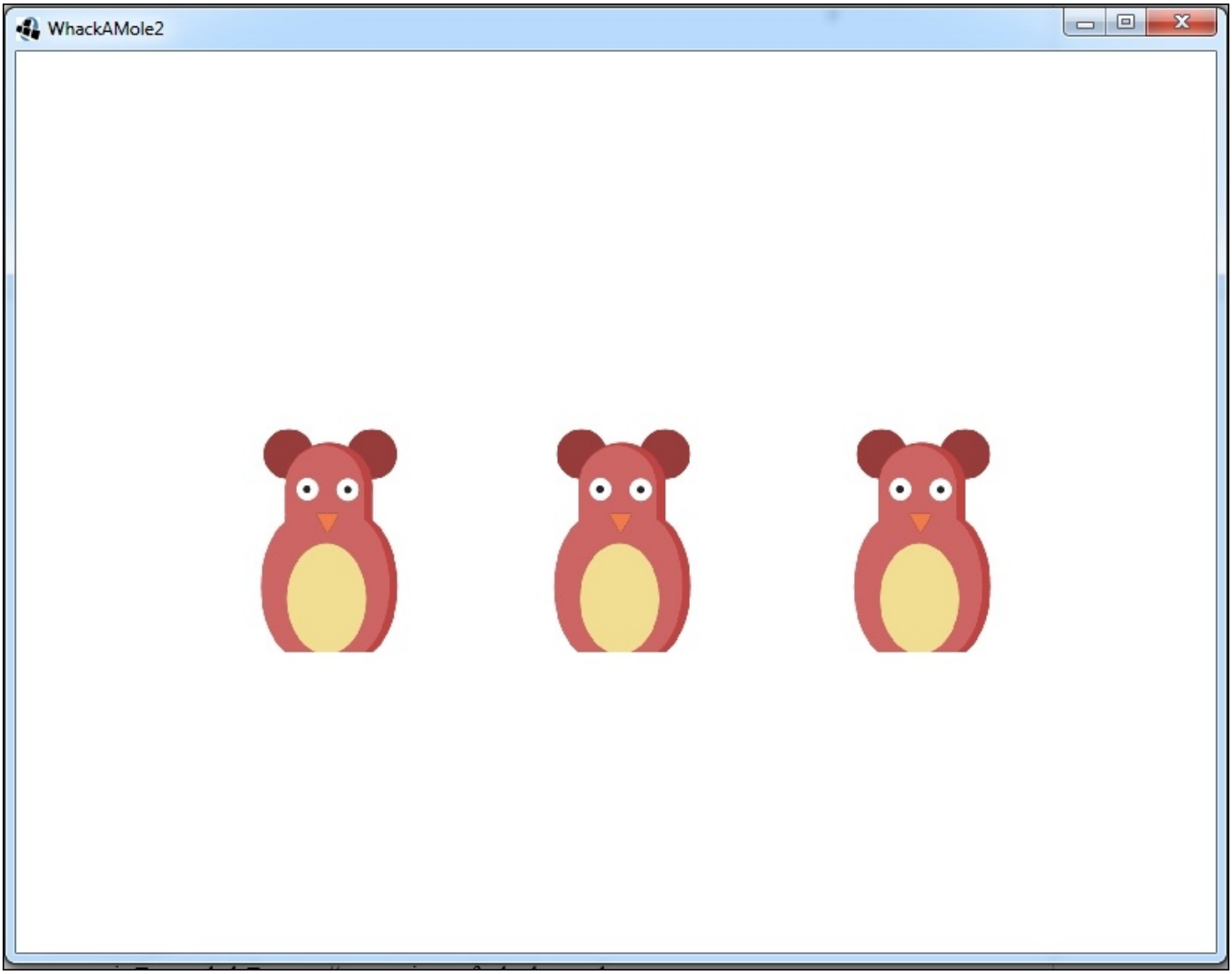
Here, we are doing exactly what we did in the previous chapter. We are going to create three mole instances, initialize them, display them, and at the end, we will dispose of the texture used. The code is pretty self-explanatory.

Implementing the WhackAMole class

Edit the `WhackAMole.java` file and type the following code:

```
package com.packtpub.whackamole;

import com.badlogic.gdx.ApplicationAdapter;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.whackamole.managers.GameManager;

public class WhackAMole extends ApplicationAdapter {
    SpriteBatch batch; // spritebatch for drawing
    OrthographicCamera camera;
    @Override
    public void create () {
        // get window dimensions and set our viewport dimensions
        float height= Gdx.graphics.getHeight();
        float width = Gdx.graphics.getWidth();
        // set our camera viewport to window dimensions
        camera = new OrthographicCamera(width,height);
        // center the camera at w/2,h/2
        camera.setToOrtho(false);

        batch = new SpriteBatch();
        //initialize the game
        GameManager.initialize(width, height);
    }

    @Override
    public void dispose() {
        super.dispose();
        //dispose the batch and the textures
        batch.dispose();
        GameManager.dispose();
    }

    @Override
    public void render () {
        // Clear the screen
        Gdx.gl.glClearColor(1, 1, 1, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        // set the spritebatch's drawing view to the camera's view
        batch.setProjectionMatrix(camera.combined);

        // render the game objects
        batch.begin();
        GameManager.renderGame(batch);
        batch.end();

    }
}
```

This class is also very similar to the one in the previous chapter. It has some differences though. Instead of implementing the `ApplicationListener` interface, we are now extending the `ApplicationAdapter` class. This allows us to override the methods we want instead of implementing all of them. We have overridden `create()`, `render()`, and `dispose()`, which perform the same functions as before. If you run the game, you'll get three moles on the screen against a white background:



Adding some color

This topic will include how to add the background and the hole sprites for the mole to come up and down.

Adding the background

Let’s quickly add a background to our game. In the `GameManager` class, we will add these variables:

```
static Texture backgroundTexture; // texture image for background
static Sprite backgroundSprite;   // sprite for background
```

In the `GameManager` class’ `initialize()` method, we load the texture and initialize the sprite to cover the whole screen:

```
backgroundTexture = new Texture(Gdx.files.internal("data/ground.jpg"));
backgroundSprite = new Sprite(backgroundTexture); //set background sprite
// set background sprite's dimensions and position
backgroundSprite.setSize(width, height);
backgroundSprite.setPosition(0,0f);
```

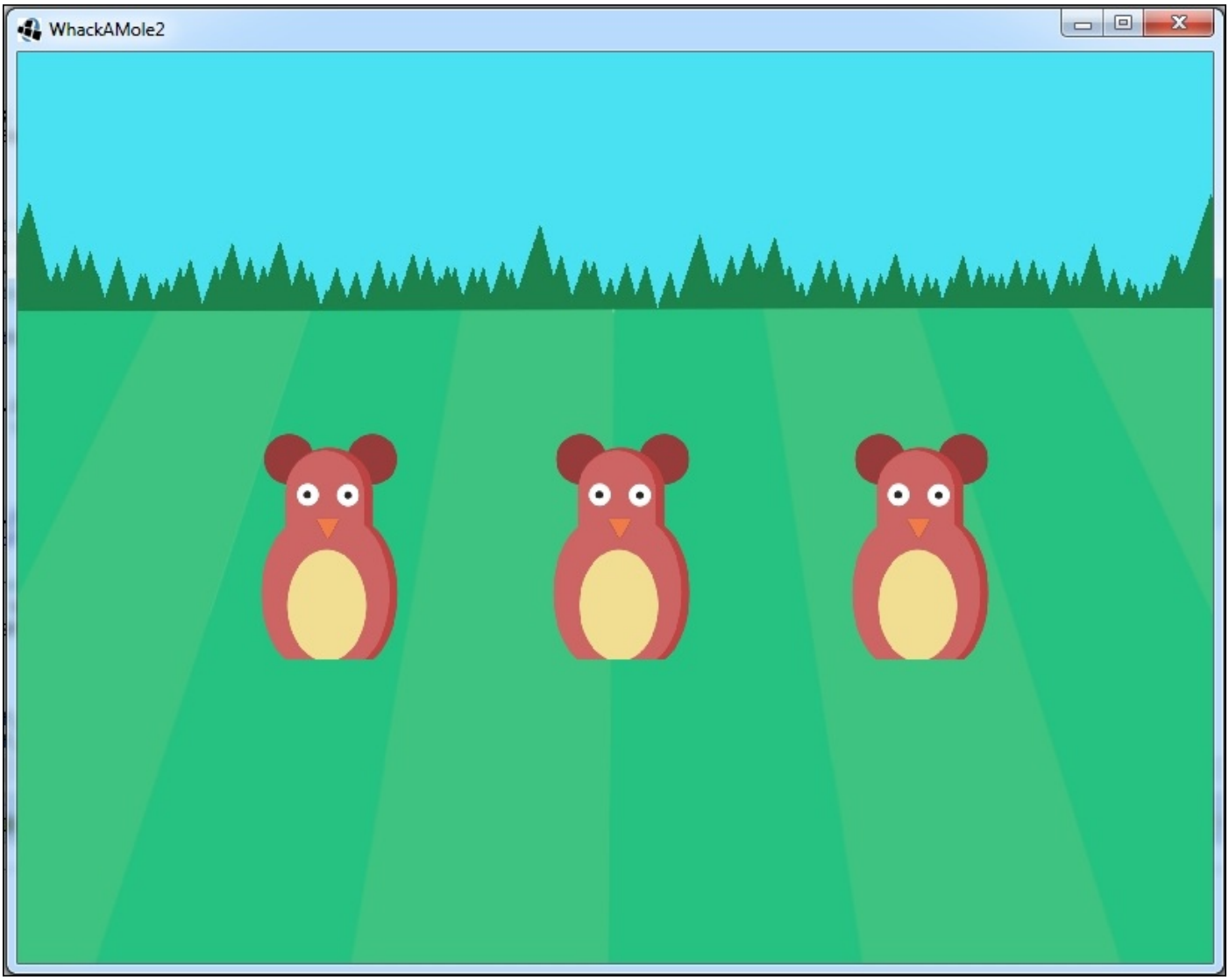
Draw the sprite in the `renderGame()` method before drawing the mole:

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

Then, dispose of the texture in the `dispose()` method:

```
backgroundTexture.dispose();
```

It will look like this:


WhackAMole2

Implementing the holes

In the background, we just added a ground texture. We will now add the holes from where the mole comes out. We will just use a single texture for the hole and draw it nine times on the screen. So, let's declare the array and the texture for the holes in the `GameManager` class:

```
static Texture holeTexture; // texture image for background
static Array<Sprite> holeSprites; // array of hole sprites
private static float HOLE_RESIZE_FACTOR = 1100f;
```

We'll initialize them in the `initialize()` method like this:

```
holeTexture = new Texture(Gdx.files.internal("data/hole.png"));
holeSprites = new Array<Sprite>();
for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        Sprite sprite = new Sprite(holeTexture);
        // resize the holes
        sprite.setSize(sprite.getWidth()*(width/HOLE_RESIZE_FACTOR),
sprite.getHeight()*(width/HOLE_RESIZE_FACTOR ));
        // position the holes so they are in the center of the ground
        sprite.setPosition(width*(j+1)/4f - sprite.getWidth()/2, height*
(i+1)/4.4f  - sprite.getHeight());
        holeSprites.add(sprite);
    }
}
```

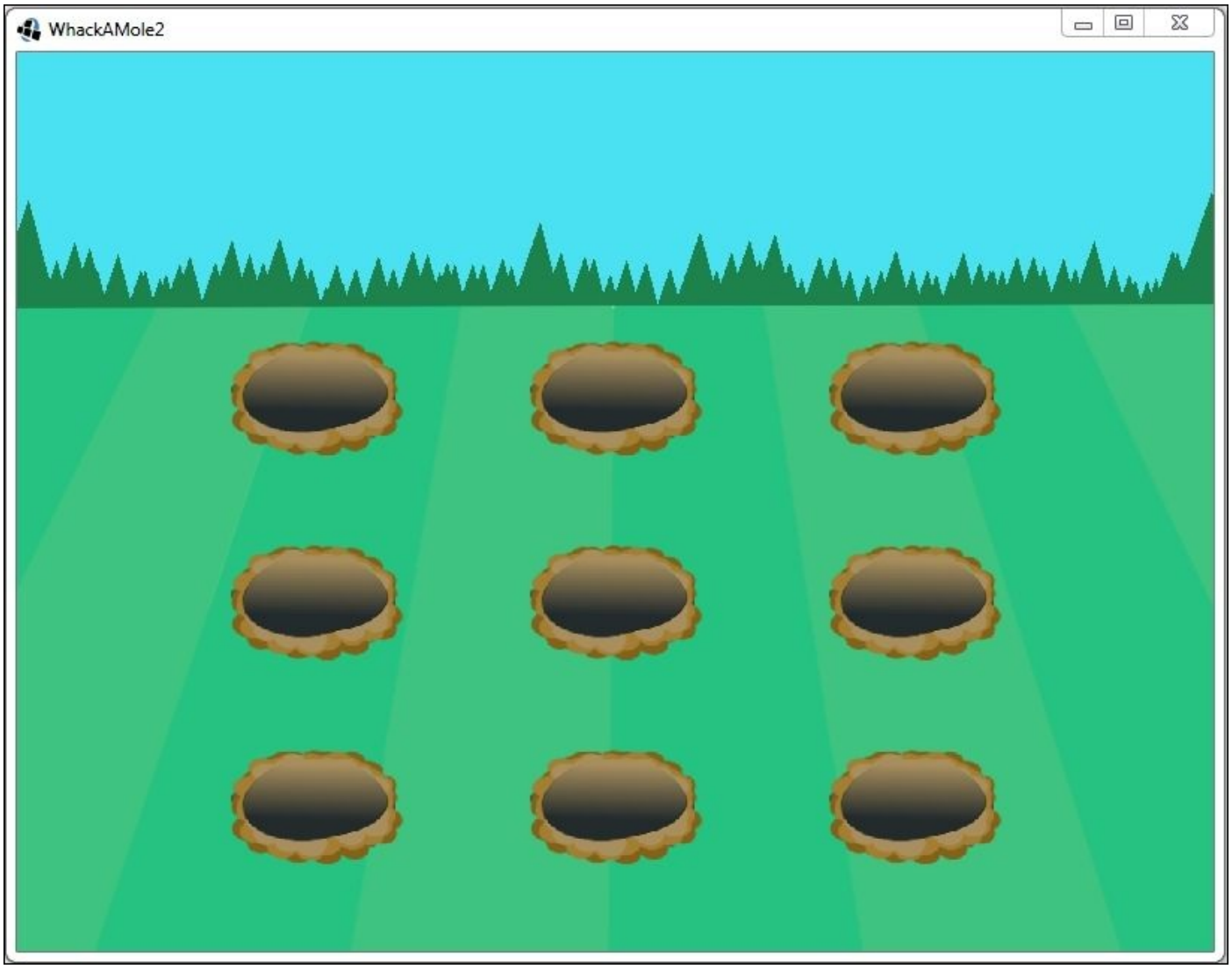
Although we are using a single array to hold nine instances of holes, they are laid out in a 3 x 3 grid on the screen. We position them appropriately and resize them so that they can be displayed in the center. We can display them in the `renderGame()` method between the background and the mole, as follows:

```
//render each hole
for(Sprite sprite : holeSprites)
    sprite.draw(batch);
```

Now, dispose of the hole texture in the `dispose()` method as usual:

```
holeTexture.dispose();
```

Comment out the mole's rendering code and run the game to see something like this:


WhackAMole2

Adding moles in holes

Now, we are going to display the moles coming out of the holes. Every mole needs to be displayed on top of a hole and positioned appropriately. Let's increase the number of moles to nine in the `initialize()` method of `GameManager`:

```
// instantiate new moles and add it to the array
for(int i=0;i<9;i++){
    moles.add(new Mole());
}
```

Let's position the moles. But before that, we need to add one more variable to our `Mole` class:

```
public float scaleFactor; // scaling factor for the mole
```

Since we are scaling the mole according to our screen dimensions, we will preserve the scaling factor for our mole. This information will be required later. Update the code in the `initialize()` method, where we positioned them, and set the size and delete the three lines where we positioned them (`moles.get(0).position`):

```
// set the mole's display position
for(int i=0;i<9;i++){

    Mole mole = moles.get(i);
    Sprite sprite = holeSprites.get(i);

    // instantiate sprite for the mole with the texture of it
    mole.moleSprite = new Sprite(moleTexture);

    //set mole's dimensions
    float scaleFactor = width/4000f;
    mole.scaleFactor=scaleFactor;
    mole.width = mole.moleSprite.getWidth()*(scaleFactor);
    mole.height = mole.moleSprite.getHeight()*(scaleFactor);
    mole.moleSprite.setSize(mole.width, mole.height);

    //set mole's position
    mole.position.x=(((2*sprite.getX() + sprite.getWidth())/2) -
(mole.moleSprite.getWidth()/2));
    mole.position.y=(sprite.getY() + sprite.getHeight()/5f);

    mole.moleSprite.setPosition(mole.position.x, mole.position.y);

}
```

Here, we set the *x* and *y* coordinates of each mole so that they are centered and placed on top of each hole. Let's talk about the *x* coordinate. Both the holes and the corresponding centers should be aligned. First, we calculate the center of a hole horizontally. This will be the midpoint of the *x* coordinate of the starting and ending point of the sprite. This comes out to the following:

```
(sprite.getX() + [sprite.getX() + sprite.getWidth()]  ) /2
```

Or this:

```
(2*sprite.getX()+ sprite.getWidth()  ) /2
```

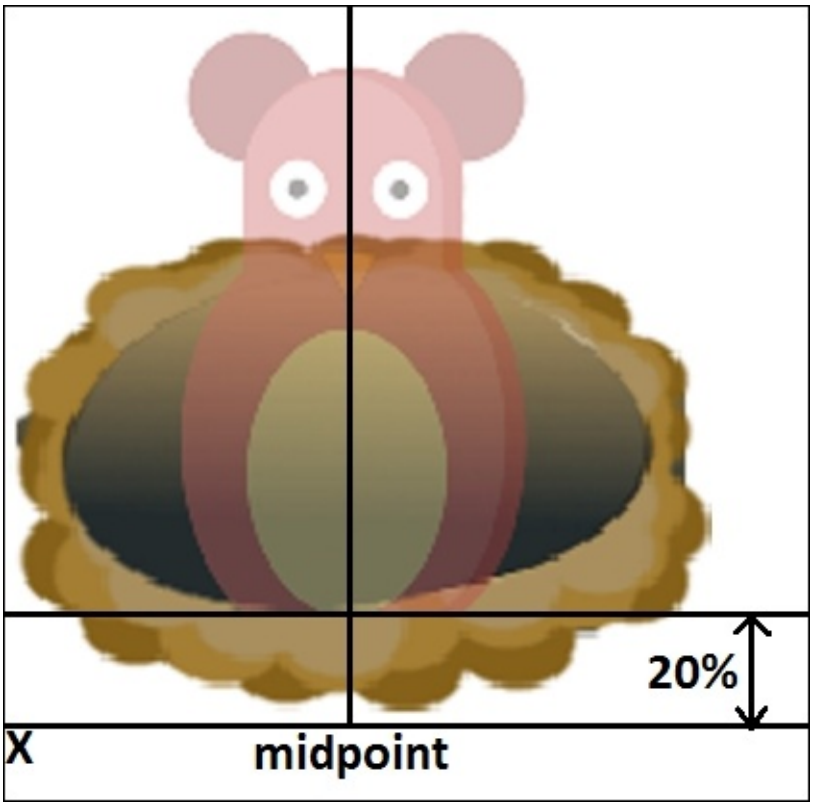
Then, we offset that location with half of the mole's width so that when we draw, the centers are aligned:

```
=(((2*sprite.getX() + sprite.getWidth())/2) -
(mole.moleSprite.getWidth()/2)
```

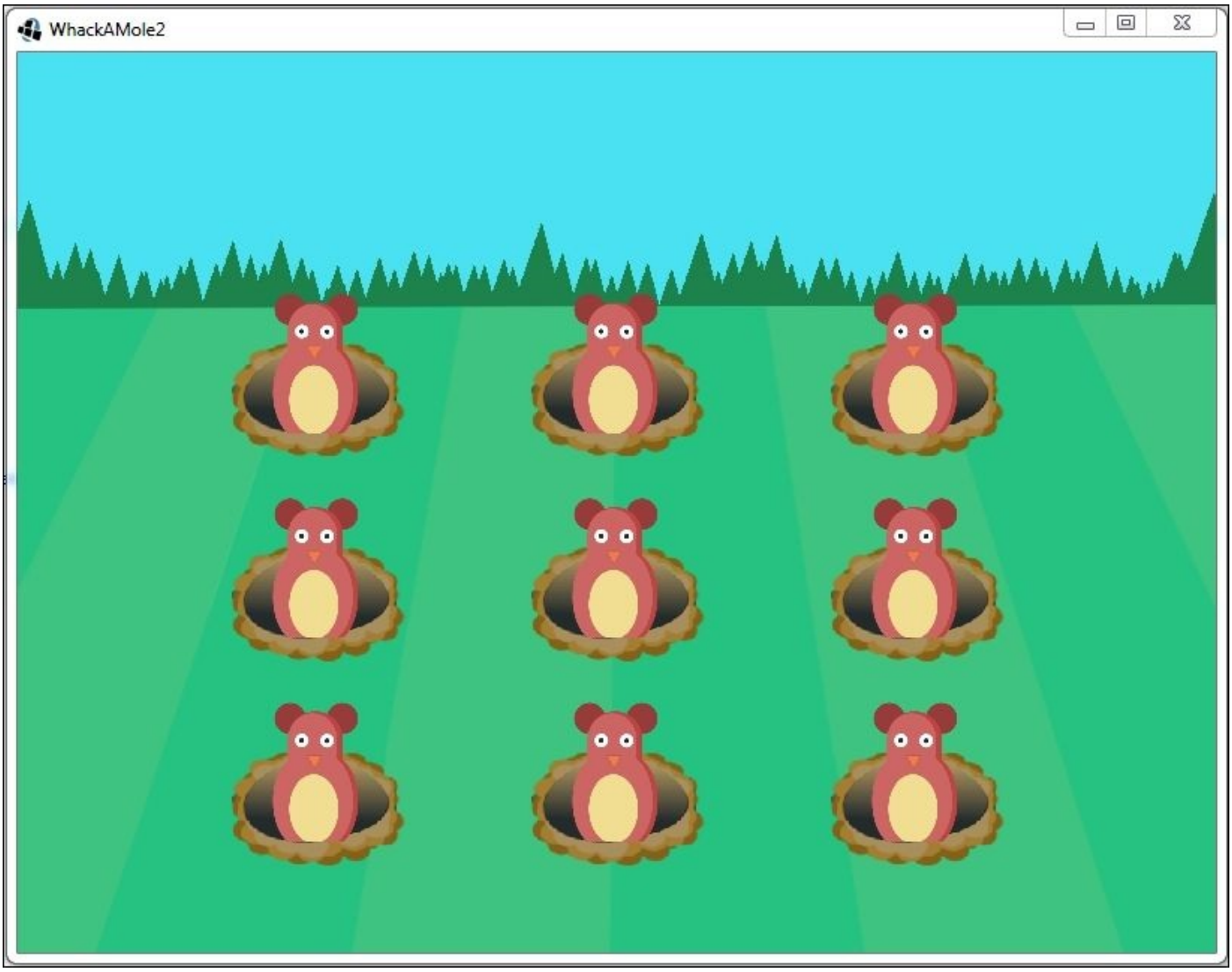
For the *y* coordinate, we don't want to align the centers, so we draw at the starting location of the hole plus a 20% offset:

```
(sprite.getY() + sprite.getHeight()/5f)
```

This is shown here:



Also, you will need to uncomment the drawing code for moles in the `renderGame()` method of the `GameManager` class. The screen will look like this:


WhackAMole2

Animating the mole

We will see how to animate our mole in this section.

Jumping up and down

We will animate the mole such that it comes out of the hole and goes back in again. This process continues repeatedly. Let's define some variables in our `Mole` class:

```
public enum State {GOINGUP,GOINGDOWN}; // define mole's states
public State state=State.GOINGUP;  // variable describing mole's current
state
public float currentHeight = 0.0f; // current height of the mole above
ground
public float speed =2f; // speed of the mole as it goes up and down
```

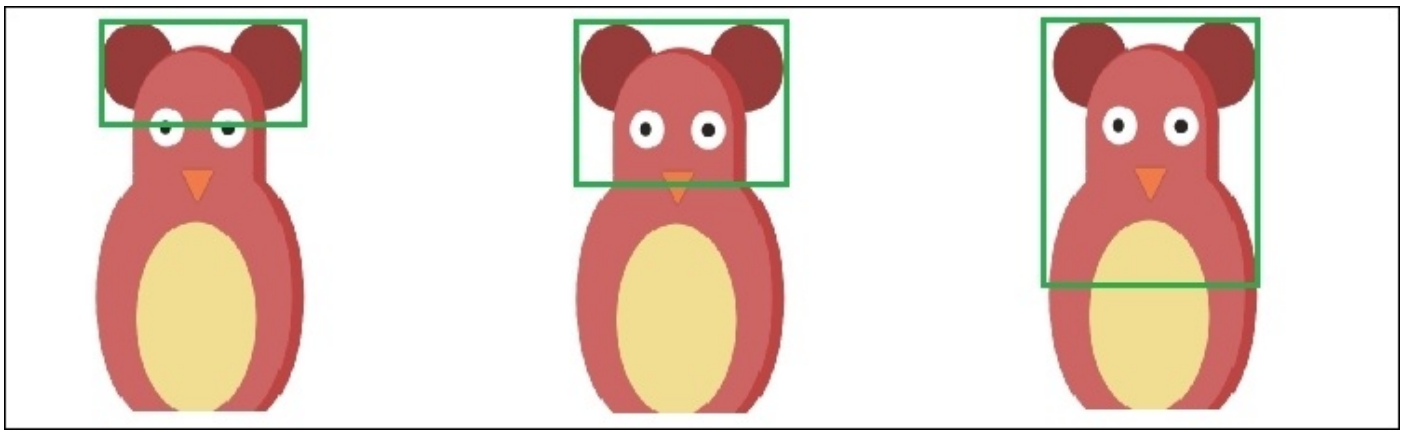
We have defined two states for the mole. One for going up and the other for going down. The `state` variable is used to hold the current value of the state. This is initialized to go up. The `currentHeight` variable is used to determine the current height of the mole above the ground. The `speed` variable denotes how fast the mole moves up or down.

We will add a new method to the same class called `update()`. This will contain the logic for the animation and is called in every frame:

```
public void update(){
    switch(state){

        // here increase the height till it reaches max, once it reaches,
change the state
        case GOINGUP:
            currentHeight+=speed;
            if(currentHeight>=height){
                currentHeight=height;
                state=State.GOINGDOWN;
            }
            break;
        // here decrease the height till it reaches min(0), once it
reaches, change the state
        case GOINGDOWN:
            currentHeight-=speed;
            if(currentHeight<=0.0){
                currentHeight=0.0f;
                state=State.GOINGUP;
            }
            break;

    }

    // draw only some portion of the mole image, depending on height
    moleSprite.setRegion(0, 0, (int)(width/scaleFactor), (int)
(currentHeight/scaleFactor));
    moleSprite.setSize(moleSprite.getWidth(), currentHeight);
}
```

The logic is as follows. If the mole is in the `GOINGUP` state, we only draw the part of the mole that is above the ground. The `currentHeight` variable determines this. We increment it every time using the `speed` variable. So, when it is going up, the mole's height goes on increasing until it reaches its actual height. After this point is reached, we switch the state

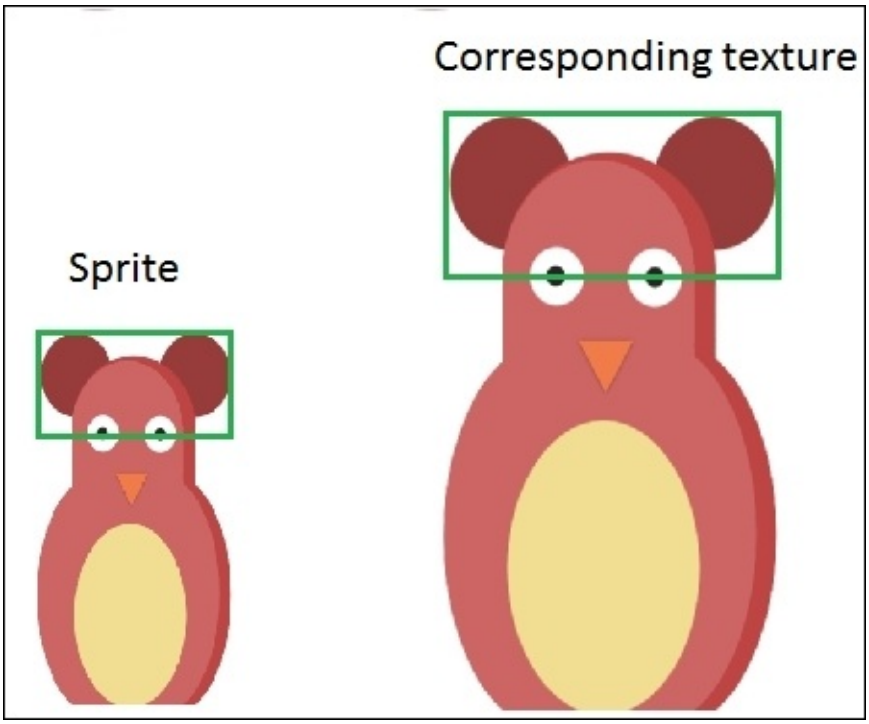
to `GOINGDOWN` where the mole goes down. This part of the logic is similar, except that now we decrease the height of the mole at every frame:



To draw the mole partially, we use the following function:

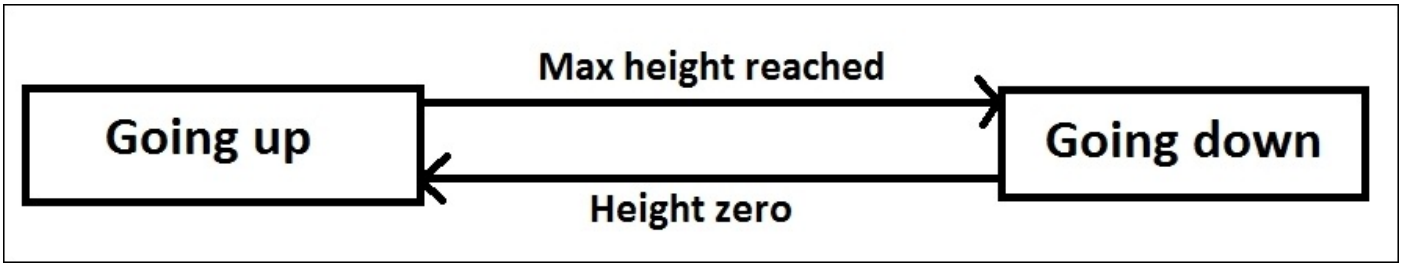
```
moleSprite.setRegion();
```

This function allows us to set a rectangular region inside a texture that we want to draw. It takes four int types as arguments. The first two arguments specify the *x* and *y* coordinates of the rectangle within the texture (from the top-left corner). The next two arguments specify the width and height of the rectangle. All these values correspond to the texture. Since we are scaling down our sprite, to get the width and height corresponding to the texture, we reverse-scale them up by `scaleFactor`:



We then set the height of our sprite to match that of the selected rectangle with the `moleSprite.setSize()` method.

The mole states look like this:



Remember to call the `update()` method in the `GameManager` class' `renderGame()` method before rendering each mole:

```
//render each mole
for(Mole mole : moles){
    mole.update();
    mole.render(batch);

}
```

Once the mole is completely down, it starts going up again and so on. If you run the game now, you will get a nice animation of the mole going up and down the hole.

Waiting underground

Now, when the moles go down, they immediately start going up as soon as they reach underground. Let’s make them wait underground for some time before they start coming up. Let’s first add some variables to our `Mole` class to achieve this effect:

```
public enum State {GOINGUP,GOINGDOWN,UNDERGROUND}; // define mole's states
public float timeUnderGround= 0.0f; // time since the mole is underground
public float maxTimeUnderGround= 0.8f; // max time allowed for the mole to
stay underground
```

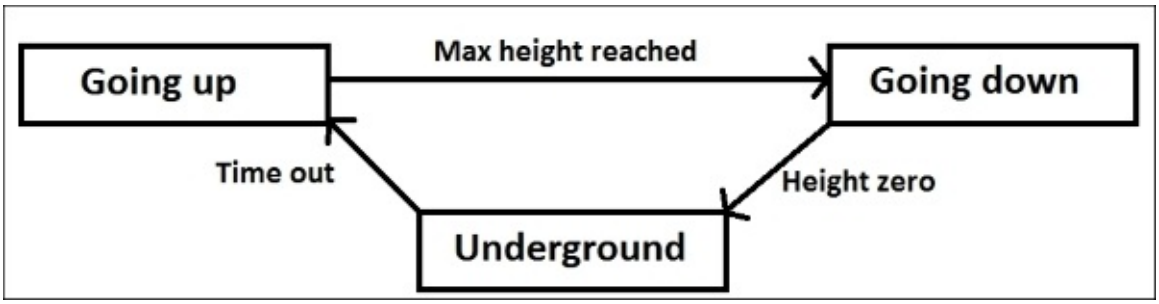
We add a new state called `UNDERGROUND` to determine whether the mole is underground. The `timeUnderGround` variable is used to tell how much time the mole has been underground since it went under the ground. The `maxTimeUnderGround` variable is used to tell how much time the mole has to wait underground before it starts coming up. Let’s make some changes to our `update()` method to simulate this effect:

```
public void update(){

    switch(state){

        case UNDERGROUND:
            if(timeUnderGround>=maxTimeUnderGround){
                state=State.GOINGUP;
                timeUnderGround=0.0f;
            }
            else{
                timeUnderGround+=Gdx.graphics.getDeltaTime();
            }
        break;
        // here increase the height till it reaches max, once it reaches,
change the state
        case GOINGUP:
            currentHeight+=speed;
            if(currentHeight>=height){
                currentHeight=height;
                state=State.GOINGDOWN;
            }
        break;

        // here decrease the height till it reaches min(0), once it
reaches, change the state
        case GOINGDOWN:
            currentHeight-=speed;
            if(currentHeight<=0.0){
                currentHeight=0.0f;
                state=State.UNDERGROUND;
            }
        break;

    }

// draw only some portion of the mole image, depending on height
moleSprite.setRegion(0, 0, (int)(width/scaleFactor), (int)
```

```
(currentHeight/scaleFactor));
moleSprite.setSize(moleSprite.getWidth(), currentHeight);

}
```

First, we check whether the mole is underground. If it is, then we check whether the time taken by our mole to be underground has exceeded the `maxTimeUnderGround` value. If yes, it is time for the mole to stop waiting and start going up. We set the state to `GOINGUP` and reset our underground time counter to 0. If the waiting time has not exceeded `maxTimeUnderGround`, we accumulate the wait time. We set the state to `UNDERGROUND` when the mole is going down and its height reaches 0.

For accumulating the time, we use the `Gdx.graphics.getDeltaTime()` function. This function gives us the time difference in seconds when the `WhackAMole` class' `render()` method is called the last time:



Run the game now to see our moles waiting for some time underground before coming up.

Adding randomness and taking input

Here, we will see how to add some randomness to the moles’ actions and respond to the user’s actions.

Randomizing wait times

Our system is predictable. The moles come up and go down with each other in perfect sync. Their cycle is fixed and we don’t want that. Let’s add some randomness to it in order to make our game interesting. We will randomize the wait times of our moles so that each one waits for a different period of time before coming up.

We will add a new function to our `Mole` class called `randomizeWaitTime()`:

```
public void randomizeWaitTime(){
    maxTimeUnderGround =(float)  Math.random()*2f;
}
```

This function will generate a number between 0 and 2 seconds.

We will call this method in the `initialize()` method of `GameManager`, where we initialize our moles:

```
mole.position.x=(((2*sprite.getX() + sprite.getWidth())/2) -
(mole.moleSprite.getWidth()/2));
mole.position.y=(((2*sprite.getY() + sprite.getHeight())/2));

mole.moleSprite.setPosition(mole.position.x, mole.position.y);
mole.randomizeWaitTime();
```

Taking input

In this section, we will talk about accepting the touch/click input from the user and processing it. Our system is simple. When the user taps on a mole, we send it underground. Let's implement a method called `handleTouch()` in the `Mole` class:

```
public boolean handleTouch(float touchX,float touchY){
    if((touchX>=position.x) &&touchX<=(position.x+width) &&
(touchY>=position.y) &&touchY<=(position.y+currentHeight) ){

        state = State.UNDERGROUND; // change the state to underground
        currentHeight=0.0f; // change the current height to 0

        moleSprite.setRegion(0, 0, (int)(width/scaleFactor), (int)
(currentHeight/scaleFactor));
        moleSprite.setSize(moleSprite.getWidth(), currentHeight);
        // reset the underground timer
        timeUnderGround=0.0f;
        randomizeWaitTime();
        return true;
    }
    return false;
}
```

This method accepts two parameters: `touchX` and `touchY`. These are the input coordinates of the point on the screen where the user has touched/clicked. We check whether the user has touched the mole, which is similar to how we did in the previous chapter. The only difference is how the vertical bounds checking is done. Since the height is constantly changing, we use `currentHeight` for detection.

Once we come to know that the user has tapped on a particular mole, we send it underground by setting the state to `UNDERGROUND`, setting `currentHeight` to `0`, resetting the `timeUnderGround` variable, and then randomizing the wait time. We return `true` if the touch is detected on a particular mole and `false` otherwise.

We will implement a new class called `InputManager` to call this method and pass touch coordinates. In the `com.packtpub.whackamole.managers` package, add a new class by this name:

```
package com.packtpub.whackamole.managers;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.math.Vector3;
import com.packtpub.whackamole.gameobjects.Mole;

public class InputManager {
    static Vector3 temp = new Vector3();

    public static void handleInput(OrthographicCamera camera){
        // Check if the screen is touched
        if(Gdx.input.justTouched()){
            // Get input touch coordinates and set the temp vector with
```

```
these values
            temp.set(Gdx.input.getX(),Gdx.input.getY(), 0);
            //get the touch coordinates with respect to the camera's
viewport
            camera.unproject(temp);

            float touchX = temp.x;
            float touchY= temp.y;

            // iterate the moles array and check if we
tapped/touched/clicked on any mole
            for(int i=0;i<GameManager.moles.size;i++){
                Mole mole  = GameManager.moles.get(i);

                if(mole.handleTouch(touchX, touchY)){
                    break;
                }

            }

        }

    }
```

This class needs no explanation as it is very similar to how we implemented it in the previous chapter. We will call the `InputManager` class' `handleInput()` method in the `render()` method of the `WhackAMole` class:

```
Gdx.gl.glClearColor(1, 1, 1, 1);
Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
InputManager.handleInput(camera);
```

Adding more effects

We will see how to add a stun effect to the mole in this section.

Stunning the mole

Let’s add an effect where the mole is stunned for some time when we tap on it. We will need some more variables for this in the `Mole` class:

```
public enum State {GOINGUP,GOINGDOWN,UNDERGROUND,STUNNED}; // define mole's
states
public float stunTime =0.1f; // The amount of time the mole would be
stunned
public float stunCounter=0.0f; // The amount of time the mole is currently
stunned
```

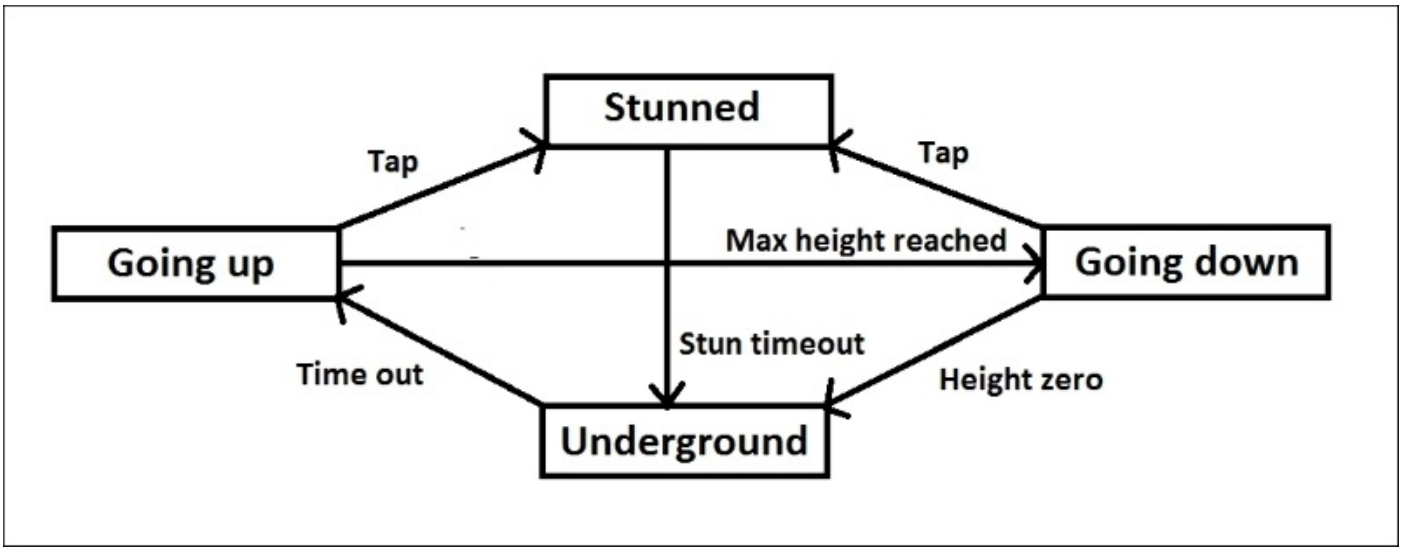
We add the `STUNNED` state to denote whether the mole is stunned or not. The `stunTime` variable denotes the amount of time the mole is to be kept in the `STUNNED` state, and the `stunCounter` variable is used to keep track of that time. As we are not sending the mole underground immediately, edit the `handleTouch()` method as follows:

```
public boolean handleTouch(float touchX,float touchY){
    if((touchX>=position.x) &&touchX<=(position.x+width) &&
(touchY>=position.y) &&touchY<=(position.y+currentHeight) ){

        state = State.STUNNED; // change the state to stunned

        return true;
    }
    return false;
}
```

Here, we just set the mole to the `STUNNED` state when a touch/click event is detected on it. We will edit the `update()` method to add a case to handle the stun effect:

```
case STUNNED:
    if(stunCounter>=stunTime){
        // send the mole underground
        state= State.UNDERGROUND;
        stunCounter=0.0f;
        currentHeight=0.0f;
        randomizeWaitTime();
    }
    else{
        stunCounter+=Gdx.graphics.getDeltaTime();
    }
break;
```

In the `STUNNED` state, if the `stunCounter` variable exceeds the maximum time we had set, we change the state to `UNDERGROUND`. We reset `stunCounter`, change the current height to 0, and then call `randomizeWaitTime()` to randomize the maximum time the mole stays underground. If `stunCounter` has not exceeded the maximum stun time, we keep accumulating `stunCounter`. The following diagram describes this:



One more thing, we don’t want to stun the mole again if he is already stunned, so we will update the `handleInput()` method in the `InputManager` class to reflect this change:

```
if(mole.state!= Mole.State.STUNNED && mole.handleTouch(touchX, touchY)){
    break;
}
```

Adding the stun sign

Now, we'll add a sign to our moles when they are stunned. I'm using a picture of a star. Let's add a new sprite to it in our `Mole` class:

```
public Sprite stunSprite; // sprite to display stun image
```

In the `GameManager` class, we will add a texture to our sprite:

```
static Texture stunTexture; // texture for stun image
```

We will load the texture with our stun image in the `GameManager` class' `initialize()` method:

```
stunTexture = new Texture(Gdx.files.internal("data/stun.png"));
```

Then, we will initialize the `stunSprite` in the loop where we initialized the moles:

```
mole.stunSprite = new Sprite(stunTexture);

float scaleFactor = width/4000f;
mole.scaleFactor=scaleFactor;
mole.width = mole.moleSprite.getWidth()*(scaleFactor);
mole.height = mole.moleSprite.getHeight()*(scaleFactor);
mole.moleSprite.setSize(mole.width, mole.height);

//set mole's position
mole.position.x=(((2*sprite.getX() + sprite.getWidth())/2) -
(mole.moleSprite.getWidth()/2));
mole.position.y=(sprite.getY() + sprite.getHeight()/5f);

mole.moleSprite.setPosition(mole.position.x, mole.position.y);

mole.stunSprite.setSize(mole.width/2f, mole.height/2f);
```

When we tap on the mole, we want to show the stun image near the mole's head. In the `handleTouch()` method of the `Mole` class, we do this by positioning our `stunSprite` in such a way that its center is aligned with the mole's top-right corner:

```
stunSprite.setPosition(position.x+width-(stunSprite.getWidth()/2),
position.y+currentHeight -(stunSprite.getHeight()/2));

state = State.STUNNED; // change the state to underground
```

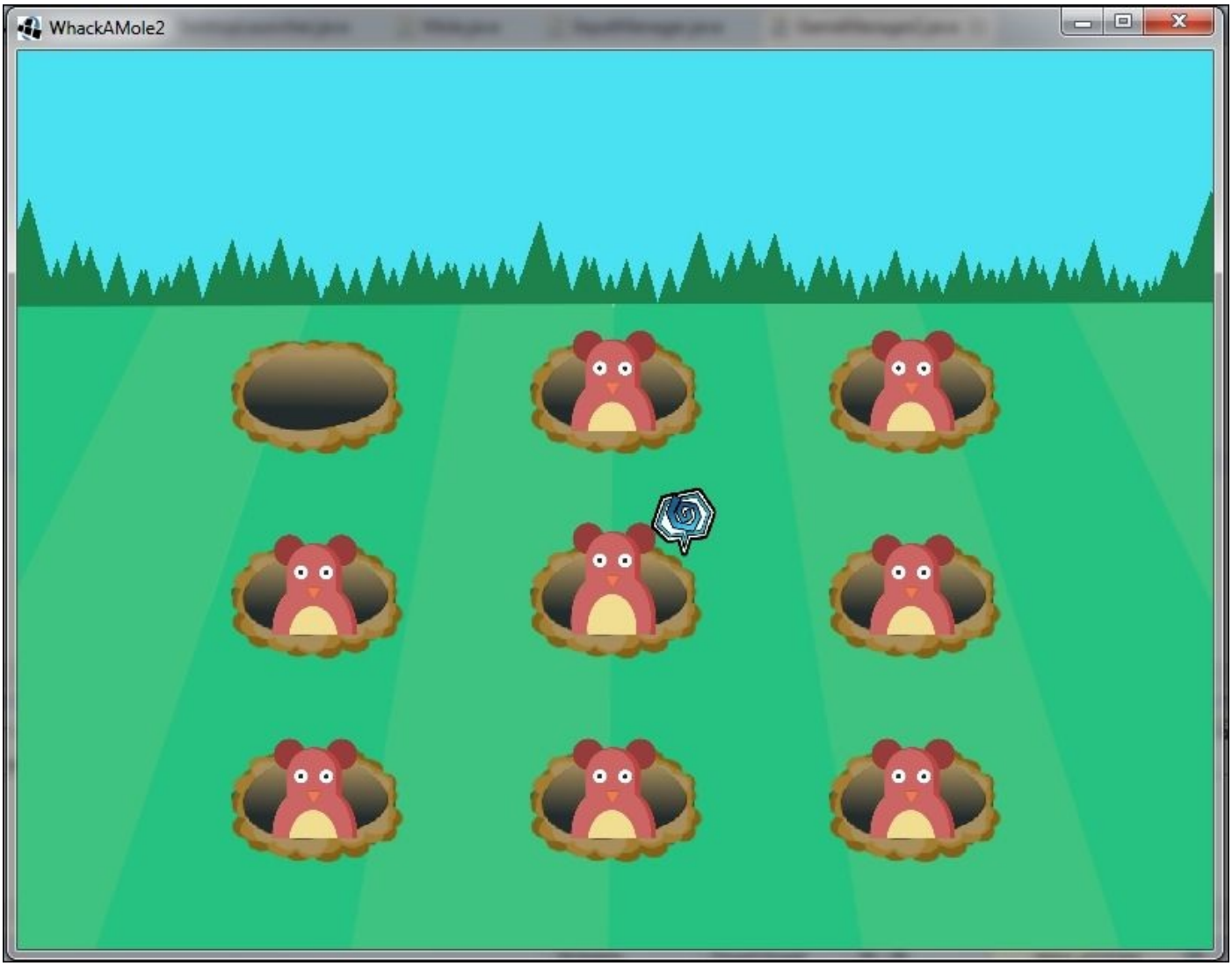
We display the image only when the mole is stunned in the `render()` method:

```
moleSprite.draw(batch);
if(state==State.STUNNED){
    stunSprite.draw(batch);
}
```

Finally, we dispose of the `stunTexture` in `GameManager` class' `dispose()` method:

```
stunTexture.dispose();
```

This is how the screen looks now:


WhackAMole2

Keeping scores and adding sounds

We will take a look at how to keep scores and add sound effects to the game in this section.

Keeping scores

We want to keep track of how many times the user has whacked a mole and show it to the user. For this, we add a new variable to the `GameManager` class called `score`:

```
public static int score; // score counter
```

Let's initialize it with `0` in the `initialize()` method:

```
score=0;
```

To display the score, we are going to add a new class called `TextManager` to the `com.packtpub.whackamole.managers` package, which is similar to what we did previously:

```
package com.packtpub.whackamole.managers;

import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

public class TextManager {
    static BitmapFont font ; // we draw the text to the screen using this
variable

    // viewport width and height
    static float width,height;

    public static void initialize(float width,float height){
        font = new BitmapFont();
        TextManager.width = width;
        TextManager.height= height;
        //set the font color to red
        font.setColor(Color.RED);
        //scale the font size according to screen width
        font.scale(width/1600f);
    }

    public static void displayMessage(SpriteBatch batch){
        float fontWidth =  font.getBounds( "Score:
"+GameManager.score).width; // get the width of the text being displayed

        //show the score display at top right corner
        font.draw(batch, "Score: "+GameManager.score, width - fontWidth -
width/15f,height*0.95f);

    }

}
```

We display the text at the top-right corner of the screen. Horizontally, we offset the drawing position by the text width and value of `screenWidth` divided by 15 from the right edge of the screen. Vertically, we just offset it by 5% from the top of the screen.

Initialize the `TextManager` class in the `GameManager` class:

```
TextManager.initialize(width, height);
```

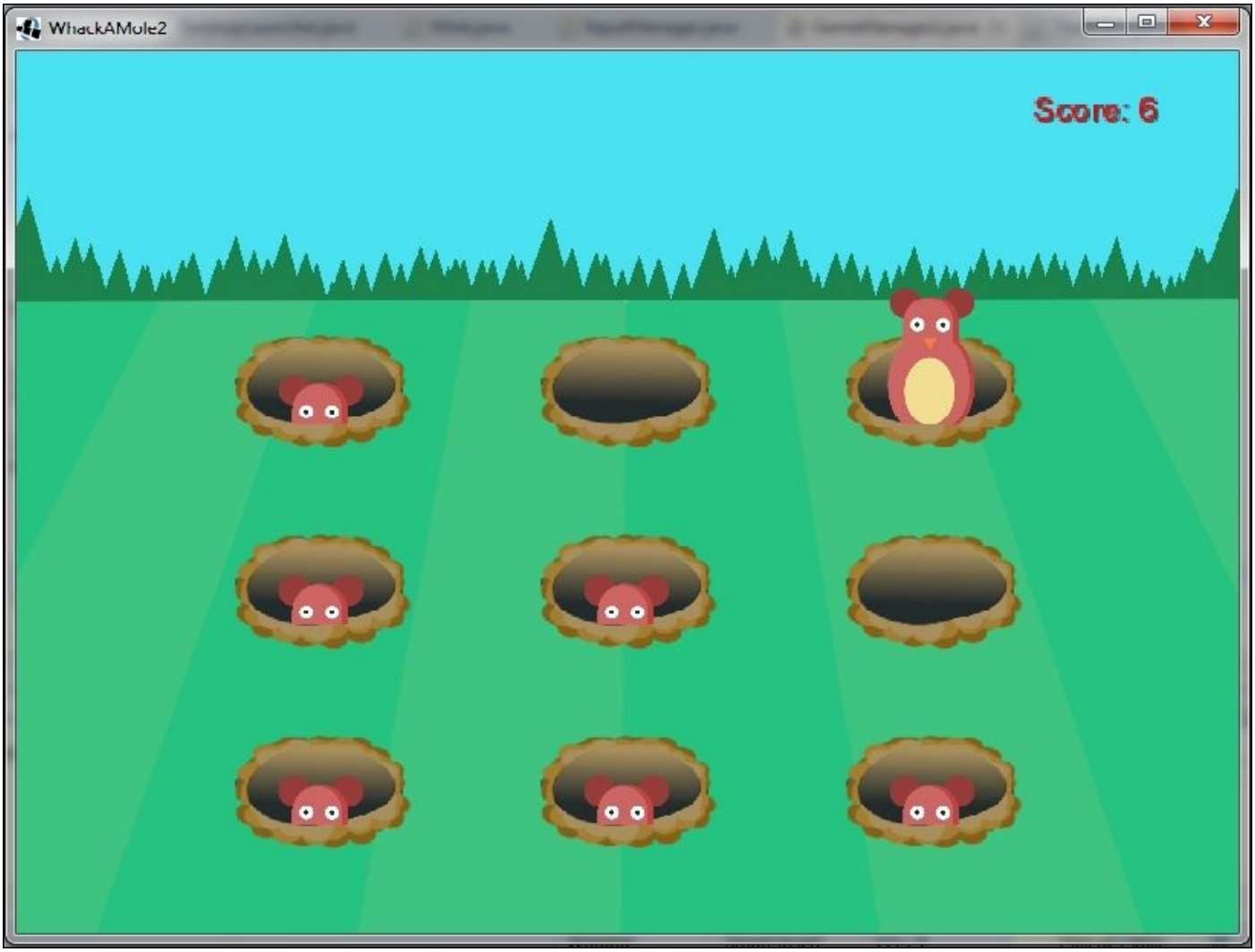
Call the `displayMessage()` method in the `renderGame()` method:

```
TextManager.displayMessage(batch);
```

Finally, in the `InputManager` class' `handleInput()` method, we increase the score when we hit the mole:

```
if(mole.state!= Mole.State.STUNNED&& mole.handleTouch(touchX, touchY)){
    GameManager.score++;  //increase the score by one
    break;
}
```

The screen now looks like this:



Adding sound effects

Let's add some sound when our player hits the moles. Sound effects are usually short audio clips suited especially for these kind of tasks. LibGDX supports the following three kinds of audio formats:

- **WAV**: This format is not proprietary but takes a large amount of storage space
- **MP3**: This format has a small size but is proprietary and may need licensing for distribution
- **OGG**: This format is not proprietary and has a small size, but it doesn't work on iOS

Note

On Android, the size of the file used for sound effects cannot be over 1 MB in size.

Each format has its own pros and cons, but we will use WAV here. I'm using the `hit.wav` sound clip taken from the http://opengameart.org/ website. We will make a new folder in our `assets` folder called `sounds` and copy the file to it.

Let's add a variable that holds our sound instance called `hitSound` in the `GameManager` class:

```
public static Sound hitSound;
```

Since we are using only one type of sound effect for the hit sound, we are using only one instance. Let's initialize the instance in the `initialize()` method:

```
hitSound = Gdx.audio.newSound(Gdx.files.internal("sounds/hit.wav"));
```

In the `Mole` class' `handleTouch()` method, we will play the sound when the mole is hit:

```
GameManager.hitSound.play();
```

Finally, we dispose of the sound in the `GameManager` class' `dispose()` method:

```
hitSound.dispose();
```

You can set the volume by passing a float variable to the `play` method. It should be in the range 0 to 1, with 0 being mute and 1 being full volume:

```
GameManager.hitSound.play(volume);
```

If you want more control over a specific instance of a hit sound for a particular mole, the `play` method also returns a long value:

```
long id = GameManager.hitSound.play();
```

This value serves as an ID to that instance and you can manipulate it. To manipulate the sound instance, you can call the following code:

```
GameManager.hitSound.stop(id); // to stop the sound instance
GameManager.hitSound.setLooping(id,true); // to keep the sound looping
GameManager.hitSound.setPitch(id,2); // set the pitch twice as much as
original
```

Summary

In this chapter, we learned some more concepts along with a game of Whack-A-Mole. These include the following:

- Animating sprites
- Implementing wait times
- Randomizing wait times
- Adding stun effects
- Adding sound effects

In the next chapter, we will make a game called Catch the Ball, where we will learn more concepts, such as making custom fonts, saving high scores, game physics, game menus, and many more.

Chapter 3. Catch the Ball

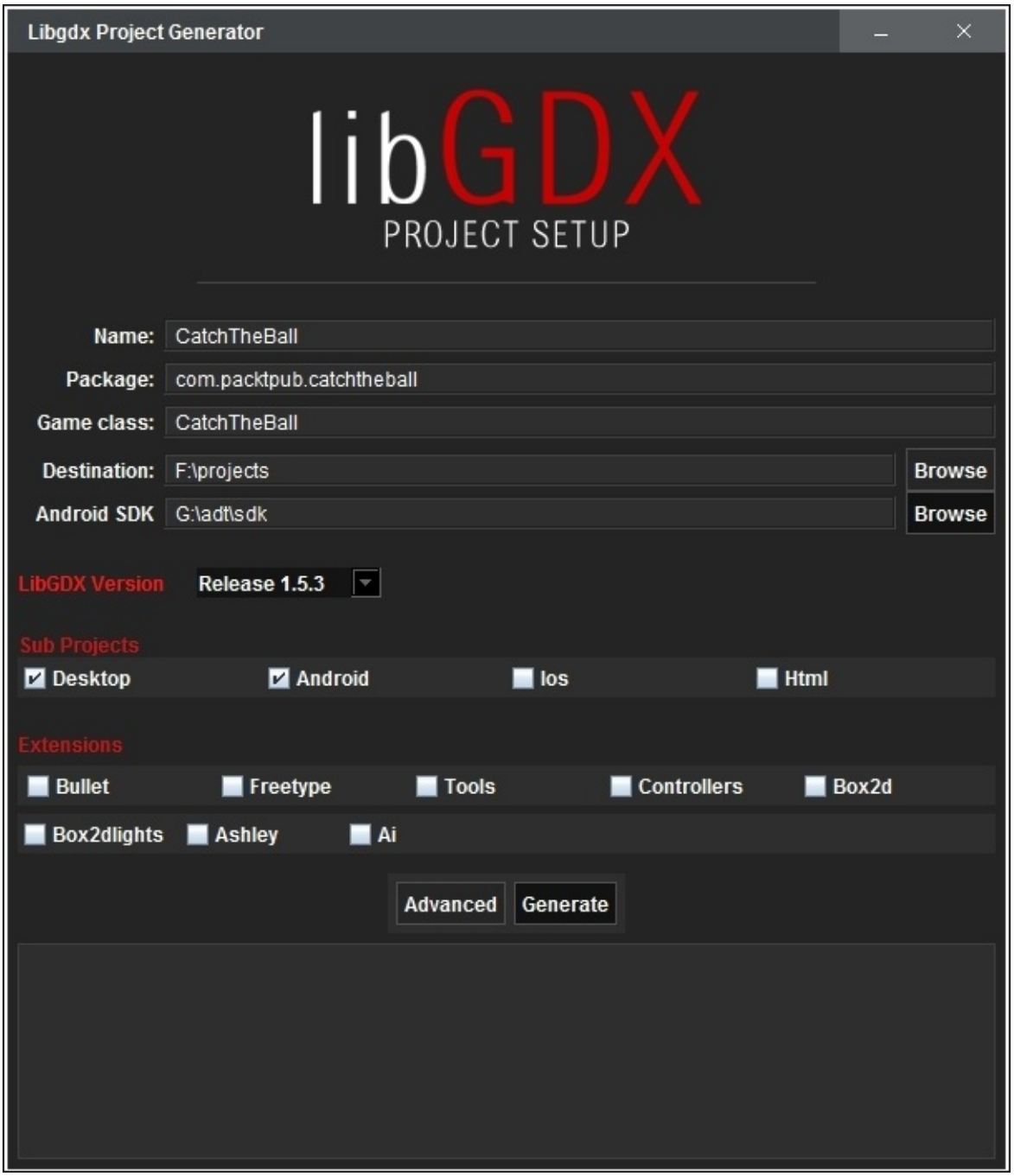
In this chapter, we will learn how to make a game called Catch the Ball. The user has to catch a ball thrown from a height in a basket. The ball will be randomly thrown from above. The user would be given a point from where he needs to catch the ball. We will display the score and also the highest score for the game.

The following topics will be covered in this chapter:

- Making a moving basket
- Throwing the ball
- Detecting collisions
- Throwing multiple balls
- Keeping score and saving the high score
- Implementing screens
- Adding sound effects and music

Making a moving basket

Set up a project similar to the one I have, as shown here:



We will make a basic game screen that has a basket that can be controlled with touch.

Implementing the Basket class

Let’s make a class to represent a basket. Create a new package in the core projects and name it `com.packtpub.catchtheball.gameobjects`. Create a new Java class in this package and name it `Basket`.

Type the following code in the file:

```
package com.packtpub.catchtheball.gameobjects;

import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

public class Basket {
    public Sprite basketSprite; //sprite to display the basket

    public void render(SpriteBatch batch){
      basketSprite.draw(batch);
    }

    public void setPosition(float x,float y){
      basketSprite.setPosition(x, y);
    }
}
```

You will find that the code is pretty self-explanatory. It’s nothing new from what we have learned in earlier chapters.

Implementing the GameManager class

Create a new package called `com.packtpub.catchtheball.managers`. Create a new `GameManager.java` file in this package. Type the following content:

```
package com.packtpub.catchtheball.managers;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.catchtheball.gameobjects.Basket;

public class GameManager {
    public static Basket basket; // basket instance
    static Texture basketTexture; // texture image for the basket
    public static Sprite backgroundSprite; // background sprite
    public static Texture backgroundTexture; // texture image for the
background

    private static float BASKET_RESIZE_FACTOR = 3000f;

    public static void initialize(float width,float height){

        basket = new Basket();
        basketTexture = new Texture(Gdx.files.internal("data/basket.png"));
        basket.basketSprite = new Sprite(basketTexture);
        basket.basketSprite.setSize(basket.basketSprite.getWidth()*
(width/BASKET_RESIZE_FACTOR), basket.basketSprite.getHeight()*
(width/BASKET_RESIZE_FACTOR));
        // set the position of the basket to bottom - left corner
        basket.setPosition(0, 0);

        backgroundTexture = new
Texture(Gdx.files.internal("data/background.jpg"));
        backgroundSprite= new Sprite(backgroundTexture);
        // set the background to completely fill the screen
        backgroundSprite.setSize(width, height);
    }

    public static void renderGame(SpriteBatch batch){
        backgroundSprite.draw(batch);
        basket.render(batch);
    }

    public static void dispose() {
        backgroundTexture.dispose();
        basketTexture.dispose();
    }
}
```

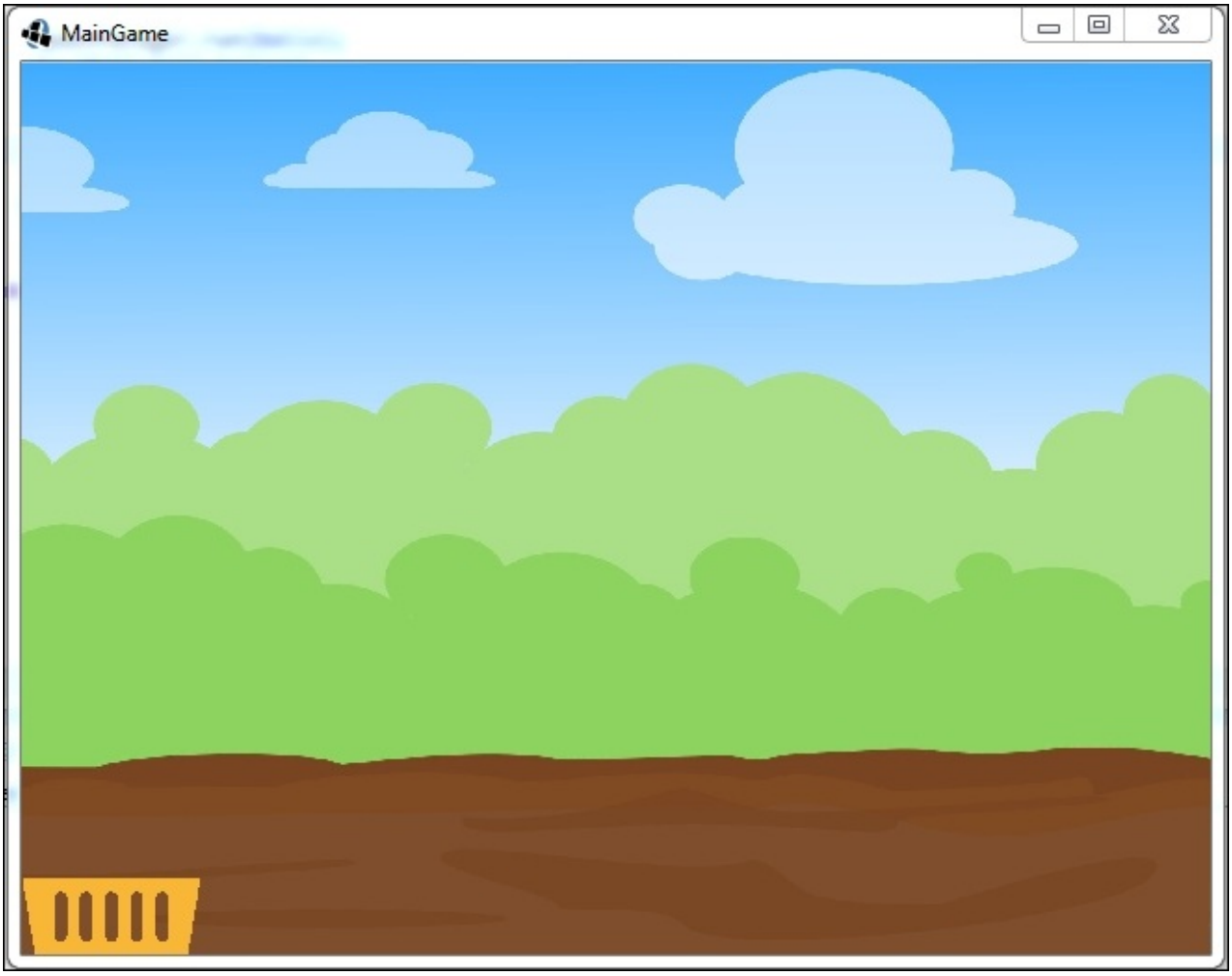
Implementing the CatchTheBall class

Update the following code in the `CatchTheBall.java` file in the `com.packtpub.catchtheball` package:

```
package com.packtpub.catchtheball;

import com.badlogic.gdx.ApplicationAdapter;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.catchtheball.managers.GameManager;

public class CatchTheBall extends ApplicationAdapter {
    SpriteBatch batch; // spritebatch for drawing
    OrthographicCamera camera;
    @Override
    public void create () {
        // get window dimensions and set our viewport dimensions
        float height= Gdx.graphics.getHeight();
        float width = Gdx.graphics.getWidth();
        // set our camera viewport to window dimensions
        camera = new OrthographicCamera(width,height);
        // center the camera at w/2,h/2
        camera.setToOrtho(false);

        batch = new SpriteBatch();
        //initialize the game
        GameManager.initialize(width, height);
    }

    @Override
    public void dispose() {
        super.dispose();
        //dispose the batch and the textures
        batch.dispose();
        GameManager.dispose();
    }

    @Override
    public void render () {
        // Clear the screen
        Gdx.gl.glClearColor(1, 1, 1, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

        // set the spritebatch's drawing view to the camera's view
        batch.setProjectionMatrix(camera.combined);

        // render the game objects
        batch.begin();
        GameManager.renderGame(batch);
        batch.end();

    }
```

```
}
```

Now, if you run the game, it should look something like this:



Moving the basket

We will add a method to our `Basket` class to handle the input:

```
public void handleTouch(float x,float y){
    if(x-(basketSprite.getWidth()/2)>0.0){
        setPosition(x-(basketSprite.getWidth()/2), 0);
    }
    else{
        setPosition(0,0);
    }
}
```

This method will set the basket's *x* coordinate to wherever the user has touched/clicked on the screen. We will set the position in such a way that the basket's center coincides with the touch coordinate. But if the user touches too close to the left end of the screen, the basket will be drawn outside the visible area. In that case, we just set the basket's position to (0, 0).

Let's make a new class called `InputManager`, which will handle the touch/click input in our game. We will use a different strategy this time to handle the input. We have used a strategy called **polling** previously. What we used to do is that at every frame, we polled/queried the processor whether the user had touched the screen. This wastes some processing time.

The strategy we are going to use now is called **event handling**. Basically, we set up some callback methods for different types of inputs, which are automatically called by the framework when they are triggered.

In the `com.packtpub.catchtheball.managers` package, add a new class named `InputManager`:

```
package com.packtpub.catchtheball.managers;

import com.badlogic.gdx.InputAdapter;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.math.Vector3;

public class InputManager extends InputAdapter {

    OrthographicCamera camera;
    static Vector3 temp = new Vector3();

    public InputManager(OrthographicCamera camera) {
        this.camera = camera;
    }

    @Override
    public boolean touchUp(int screenX, int screenY, int pointer, int
button) {

        temp.set(screenX,screenY, 0);
        //get the touch coordinates with respect to the camera's viewport
```

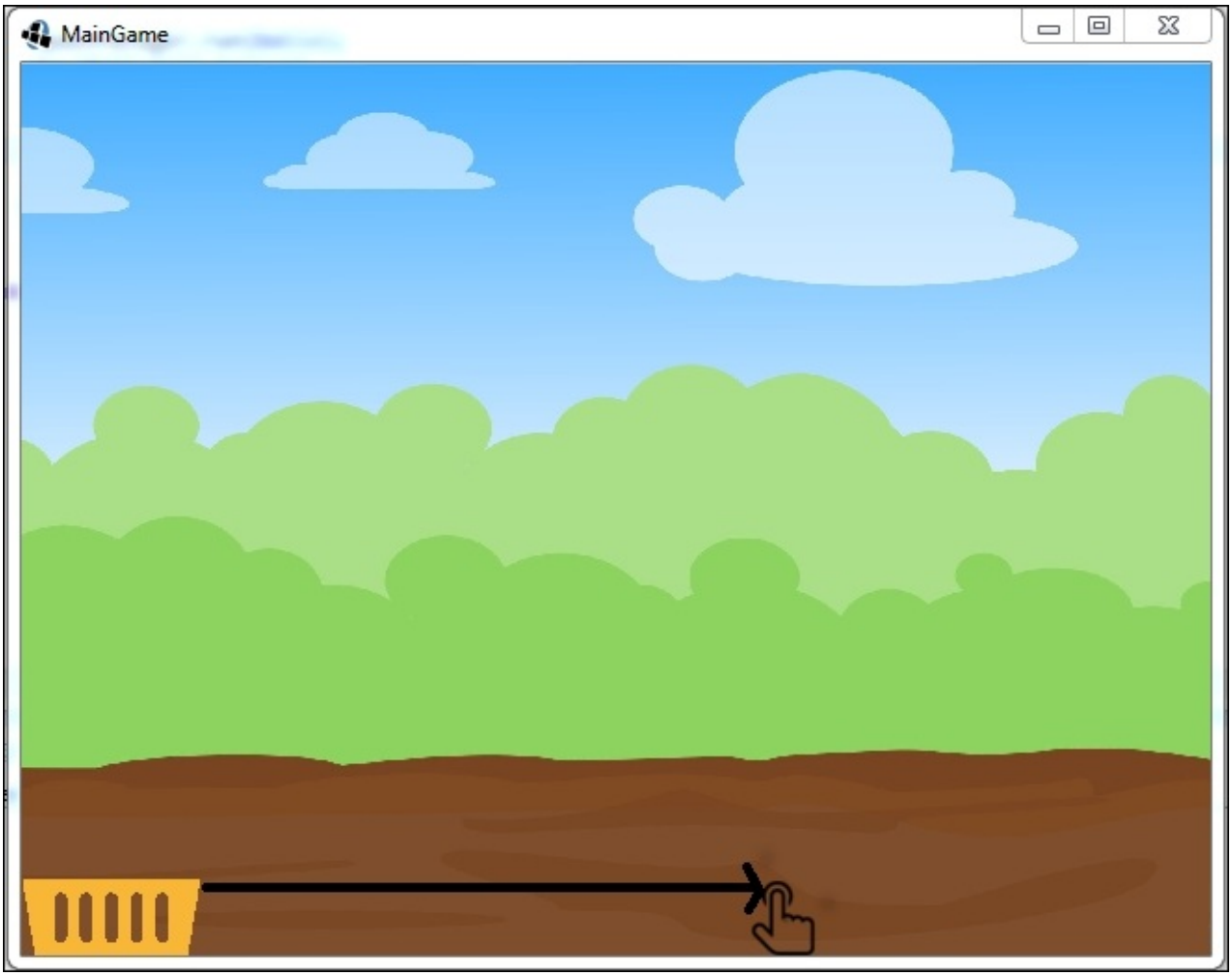
```
        camera.unproject(temp);

        float touchX = temp.x;
        float touchY = temp.y;

        GameManager.basket.handleTouch(touchX, touchY);
        return false;
    }
}
```

This class extends the `InputAdapter` class of LibGDX, which implements the callback methods to handle the input. We override a method called `touchup()`, which is a callback method that is called when the user taps/clicks on the screen. It takes four arguments, out of which the first two are the *x* and the *y* coordinates of the touch. The third one is the `pointer` ID, which is used for multi-touch handling. The last one identifies the button that was clicked on the desktop mouse.

The constructor receives the `camera` instance as an argument, which is saved in its instance variable. This is used to get the correct touch/click coordinates of the viewport. After we get them in the `touchUp()` method, we pass them to the basket's `handleTouch()` method to handle its movement. To enable receiving input events in our class, add the following line to the `CatchTheBall` class' constructor:

```
Gdx.input.setInputProcessor(new InputManager(camera));// enable
InputManager to receive input events
```

Take a look at the following screenshot:


MainGame

Throwing the ball

We will now see how to display a ball and throw it on the ground from above.

Making the ball

Let's make a new class called `Ball` to represent a ball. Under the `com.packtpub.catchtheball.gameobjects` package, create a new class called `Ball` and type in the following code:

```
package com.packtpub.catchtheball.gameobjects;

import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

public class Ball {
    public Sprite ballSprite; //sprite to represent a ball
    public void render(SpriteBatch batch){
        ballSprite.draw(batch);
    }
}
```

In the `GameManager` class, we will instantiate and initialize the ball, as we did for the basket. Let's add some new variables and constants:

```
static Ball ball; // ball instance
static Texture ballTexture; // texture image for the ball

private static final float BALL_RESIZE_FACTOR = 2500f;
```

We will initialize the ball in the `initialize()` method of the `GameManager` class:

```
ball = new Ball();
ballTexture = new Texture(Gdx.files.internal("data/ball.png"));
ball.ballSprite = new Sprite(ballTexture);

ball.ballSprite.setSize(ball.ballSprite.getWidth()*
(width/BALL_RESIZE_FACTOR), ball.ballSprite.getHeight()*
(width/BALL_RESIZE_FACTOR));
ball.ballSprite.setPosition(0.0f, height-ball.ballSprite.getHeight());
```

We will draw the ball in the `render()` method:

```
ball.render(batch);
```

We will dispose of the texture in the `dispose()` method:

```
ballTexture.dispose();
```

Adding movement

Let's add two more variables to our `Ball` class:

```
public Vector2 position = new Vector2(); // vector to represent the
position
public Vector2 velocity = new Vector2(); // vector to represent the
velocity
```

The `position` variable represents the current *x* and *y* coordinates of the ball. Velocity is defined as the rate of change of displacement. It indicates how fast the ball is moving. You can think of it as speed. If the velocity of a car is 100 km/hr, the car will travel 100 kilometers in one hour. Similarly, if we define the velocity of the ball as 10 units per second, then the ball will move 10 units in the game world in one second.

Let's add an `update()` method that will be called in every frame. The position changes every time with velocity. So, we will add the velocity component to the position in this method:

```
public void update() {
    position.add(velocity);// update the position w.r.t velocity
    ballSprite.setPosition(position.x, position.y); // set the position of
the sprite
}
```

Since we are dropping the ball from above, let's set the velocity to -5 units/frame (since it will be added to every frame) in the –ve *y* direction. We will do this when we initialize the ball in the `GameManager` class' `initialize()` method:

```
ball = new Ball();
ballTexture = new Texture(Gdx.files.internal("data/ball.png"));
ball.ballSprite = new Sprite(ballTexture);
ball.ballSprite.setSize(ball.ballSprite.getWidth()*
(width/BALL_RESIZE_FACTOR), ball.ballSprite.getHeight()*
(width/BALL_RESIZE_FACTOR));
ball.position.set(0.0f, height-ball.ballSprite.getHeight());
ball.velocity.set(0, -5);
```

We will call the `update()` method of the ball in the `renderGame()` method just before drawing it:

```
ball.update();
ball.render(batch);
```

Now if you run the game, you should see the ball falling from above.

Adding gravity

To have a more realistic simulation of the ball falling down, we need to factor in gravity. Here, gravity means acceleration due to gravity. Acceleration is defined as the rate of change of velocity. It tells us how much the velocity changes over time. Let’s define a variable for gravity in the `Ball` class:

```
public final Vector2 gravity = new Vector2(0,-0.4f); // vector to represent the acceleration due to gravity
```

Since gravity is constantly acting on the ball, it will constantly change its velocity. Edit the `update()` method to add gravity to the ball’s velocity:

```
velocity.add(gravity); // update the velocity with gravity
position.add(velocity);// update the position w.r.t velocity

// Update the initial velocity to 0 in the GameManager's initialize() method

ball.velocity.set(0, 0);
```

When you run the game now, you should see the ball accelerating toward the ground as it falls.

Detecting collisions

If you run the game, you will notice that the ball falls right off the screen. In this topic, we are going to check for collisions between the ball and the ground and between the ball and the basket.

Colliding with the ground

Checking for collision with the ground is actually pretty simple. We need to check whether the ball has hit the base of our game screen. Let's add a new function to the `Ball` class to check for collisions. We will call the function, `checkCollisions()`:

```
public boolean checkCollisions(){
    // check if the ball hit the ground
    if(position.y<=0.0){
        return true;
    }
    return false;
}
```

The only way to know whether the ball has hit the ground is by checking the *y* coordinate. If it falls below zero, it means that the ball has touched the ground. We call this method in the `update()` method, and we can display a simple text if the ball goes below the ground:

```
if(checkCollisions()){
    System.out.println("Collided with ground"); // just to check. can
remove later
}
velocity.add(gravity); // update the velocity with gravity
```

Colliding with the basket

To detect collisions with the basket, we are going to take a different approach. To make the detection easier, we are going to assume that the basket is rectangular, irrespective of its shape. LibGDX has utility methods to detect a collision between a rectangle (basket) and a circle (ball).

Let's add a member variable to the `Ball` class of the `circle` type:

```
public Circle ballCircle; // collision circle for the ball
```

Now, in order to correctly detect collisions, the circle's radius needs to be at the center of the ball sprite and the radius should be height/2. We set the radius and center of the circle in the `initialize()` method of the `GameManager` class. The `Circle` constructor takes the first argument as the center and the next argument as the radius:

```
ball.velocity.set(0, 0);

Vector2 center = new Vector2();
//set the center at the center of ball sprite
center.x=ball.position.x + (ball.ballSprite.getWidth()/2);
center.y=ball.position.y + (ball.ballSprite.getHeight()/2);

ball.ballCircle = new Circle(center, (ball.ballSprite.getHeight()/2));
```

We will have to update the position of the rectangle in every frame in the `update()` method of the `Ball` class:

```
ballSprite.setPosition(position.x, position.y); // set the position of the
sprite
ballCircle.setPosition(position.x+ (ballSprite.getWidth()/2),(position.y+
ballSprite.getHeight()/2));
```

We will follow similar steps for the basket. In the `Basket` class, add the following line of code:

```
public Rectangle basketRectangle = new Rectangle(); // collision rectangle
for the basket
```

In the `setPosition()` method, we set the rectangle's position, as follows:

```
public void setPosition(float x,float y){
    basketSprite.setPosition(x, y);
    basketRectangle.setPosition(x, y);
}
```

Finally, in the `GameManager` class, we set the rectangle's size:

```
basket.setPosition(0, 0);
// set the size of the basket's bounding rectangle
basket.basketRectangle.setSize(basket.basketSprite.getWidth(),
basket.basketSprite.getHeight());
```

We are going to separate the logic of detecting collisions with the ground into two functions in the `Ball` class. The first one is `detectCollisionwithGround()`:

```
public boolean checkCollisionsWithGround(){
    // check if the ball hits the ground
    if(position.y<=0.0){
        System.out.println("Collided with ground");
        return true;
    }
    return false;
}
```

It's the same as what we did earlier. We just change the name of the function and print the output if a collision takes place. Secondly, we will create a function named `checkCollisionsWithBasket()` to detect collisions with the basket:

```
public boolean checkCollisionsWithBasket(){
    // check if the ball collided with the basket
    if(Intersector.overlaps(ballCircle,
GameManager.basket.basketRectangle)){
        System.out.println("Collided with basket");
        return true;
    }
    return false;
}
```

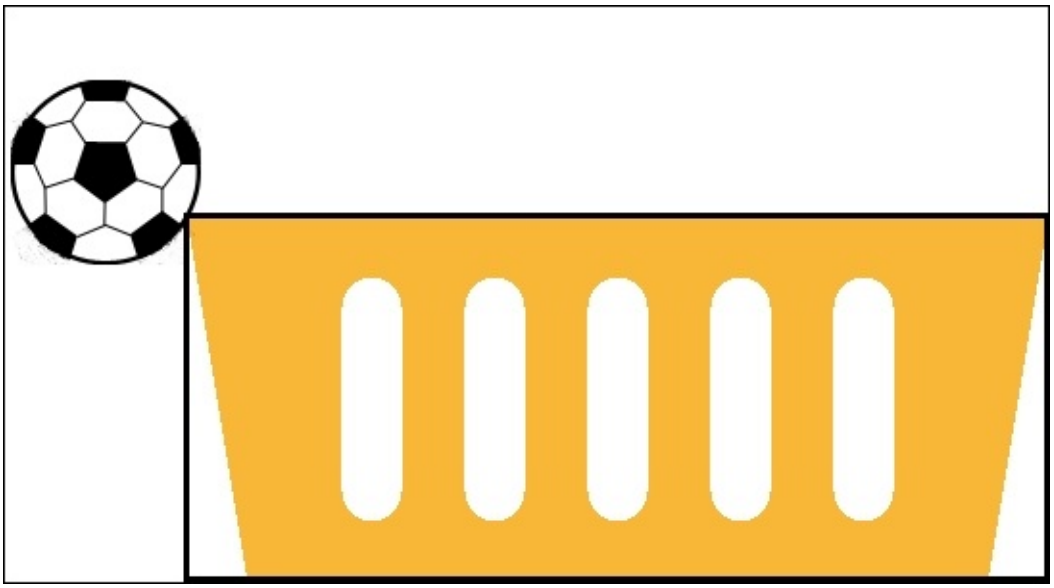
LibGDX has a utility class called `Intersector` to detect intersections between different shapes. We use its `overlaps()` method to check for collisions between a circle and a rectangle. We will call these two functions in the new `checkCollisions()` method:

```
public void checkCollisions(){
    checkCollisionsWithGround();
    checkCollisionsWithBasket();
}
```

We will call the `checkCollisions()` function in our `update()` method:

```
ballRectangle.setPosition(position); // set the position of the ball
rectangle
checkCollisions();
```

Let's take a look at the following diagram:



Throwing multiple balls

In this section, we will learn how to throw multiple balls from the air. We will also learn how to optimize our logic.

Throwing the balls after specific intervals

Before we do anything else, we need to add a flag to our `Ball` class to check whether the ball is alive or not:

```
public boolean isAlive; // flag to indicate if the ball is alive or not
```

We will set the flag to `false` if it collides with either the basket or the ground. In the `checkCollisionsWithBasket()` method, add the following lines of code:

```
if(Intersector.overlaps(ballCircle, GameManager.basket.basketRectangle)){
    isAlive=false;
    return true;
}
```

In the `checkCollisionsWithGround()` method, add the following lines of code:

```
public boolean checkCollisionsWithGround(){
    // check if the ball hit the ground
    if(position.y<=0.0){
        isAlive=false;
        return true;
    }
    return false;
}
```

In the `GameManager` class, we will set the ball to be alive at the start:

```
ball.velocity.set(0, 0);
// set the ball as alive
ball.isAlive=true;
```

We will only update and display the ball if it is alive. This will save some CPU cycles and make the game faster. In the `renderGame()` method, add the following lines of code:

```
if(ball.isAlive){
    ball.update();
    //Render(draw) the ball
    ball.render(batch);
}
```

Now, as we want to throw multiple balls, let's make an array called `balls` in our `GameManager` class to represent this:

```
public static Array<Ball> balls = new Array<Ball>(); // array of ball
objects
```

We will create a new class called `SpawnManager` that handles the creation and deletion of new `Ball` objects based on the interval:

```
package com.packtpub.catchtheball.managers;

import com.badlogic.gdx.graphics.Texture;

public class SpawnManager {
```

```
    static float delayTime = 0.8f; // delay between two throwing two balls
    static float delayCounter=0.0f; // counter to keep track of delay

    static float width,height; //viewport width and height

    static Texture ballTexture; // texture image for the ball

    public static void initialize(float width,float height,Texture
ballTexture){
        SpawnManager.width=width;
        SpawnManager.height=height;
        SpawnManager.ballTexture=ballTexture;
        delayCounter=0.0f;// reset delay counter
    }
}
```

Here, we declare a `delayTime` variable to indicate the delay between the creation of the two balls. The `delayCounter` variable keeps track of the time elapsed since the creation of the previous ball. We will instantiate and initialize the balls in this class. That is why we declare the viewport dimensions and the texture of the ball. We initialize these values that are passed from `GameManager` in the `initialize()` method. Next, we define the `createNewBall()` method in the same class. We will use a similar initialization logic for the ball as in `GameManager`. Also, we move the `BALL_RESIZE_FACTOR` constant to this class from `GameManager`:

```
public static Ball createNewBall(){
    Ball ball = new Ball();
    ball.ballSprite = new Sprite(ballTexture);
    ball.ballSprite.setSize(ball.ballSprite.getWidth()*
(width/BALL_RESIZE_FACTOR), ball.ballSprite.getHeight()*
(width/BALL_RESIZE_FACTOR));
    ball.position.set(0.0f, height-ball.ballSprite.getHeight());
    ball.velocity.set(0, 0);
    ball.isAlive=true;

    Vector2 center = new Vector2();
    //set the center at the center of ball sprite
    center.x=ball.position.x + (ball.ballSprite.getWidth()/2);
    center.y=ball.position.y + (ball.ballSprite.getHeight()/2);

    ball.ballCircle = new Circle(center, (ball.ballSprite.getHeight()/2));
    return ball;
}
```

This method is called when we want to spawn a new ball. We create and initialize a new ball and return it. Along with this, we also need to remove the balls, which are not alive. Let’s declare a variable to capture the indices of the balls, which are not alive:

```
static List<Integer> removeIndices = new ArrayList<Integer>(); // holds
indices of the balls to remove
```

To remove these `Ball` objects, we will write a `cleanup()`function:

```
public static void cleanup(Array<Ball> balls){
    removeIndices.clear(); // empty the indices list
```

```
    for(int i=balls.size-1;i>=0;i--){
        if(!balls.get(i).isAlive){
            removeIndices.add(i); // get the indices of ball objects which
are not alive/not active
        }
    }
    // Remove the ball objects from the array corresponding to the indices
    for (int i =0 ;i< removeIndices.size;i++)
        balls.removeIndex(i);

    }
```

Here, we iterate the `balls` array to see which objects are not alive or not active. We record the indices of these objects in the `removeIndices` list. Note that we start from the top end of the array as we want the indices in descending order. This will ensure proper deletion of the elements. Next, we will define the `run()` method that will implement the timing logic and creation of ball objects:

```
public static void run(Array<Ball> balls){
    // delaycounter has exceeded delay time
    if(delayCounter>=delayTime){
        balls.add(createNewBall()); //  create new ball
        delayCounter=0.0f;// reset delay counter
    }
    else{
        delayCounter+=Gdx.graphics.getDeltaTime(); // otherwise accumulate
the delay counter
    }
}
```

Here, we check whether the delay counter exceeds the delay time. If it exceeds, then we spawn a new ball object. We then add it to the `balls` array. Otherwise, we accumulate the delay counter with the delta time.

With all of this in place, in the `GameManager` class, we need to make some modifications. First of all, we need to remove the single ball instance and the initialization code for it. Keep the texture initialization code though. Next, we need to add the initialization method call of the `SpawnManager` class in the `initialize()` method:

```
SpawnManager.initialize(width, height, ballTexture);
```

Finally, we need to remove the `update()` and `render()` methods of the ball and replace them with the following code:

```
SpawnManager.run(balls);
for(Ball ball:balls){

    if(ball.isAlive){
        ball.update();
        ball.render(batch);
    }
}
SpawnManager.cleanup(balls);
```

If you run the game, you will see balls falling after a set delay time.

Randomizing and optimizing

In our game, the balls always fall from the same location, so let's add some logic that would make them fall from different places every time. For this, we will first need to add an instance of the `random` class to our `SpawnManager` class:

```
static Random random = new Random(); // object of random class to generate
random numbers
```

In our `createNewBall()` method, we set the *x* coordinate to 0 for the ball. Replace this line with the following:

```
ball.position.set(random.nextInt((int) (width -
ball.ballSprite.getWidth())), height-ball.ballSprite.getHeight());
```

The `nextInt()` method is a method in the `random` class, which takes an integer argument. It gives a random number between 0 and that integer. If we call it `random.nextInt(5)`, then it will return a random number between 0 and 5. We call it `width - ball.ballSprite.getWidth()` as we want to drop the ball between the left end of the screen (0) and the right end without the ball going out of the screen (`width - ball.ballSprite.getWidth()`).

To optimize our code, we are going to follow a strategy called pooling. In our code, we will create and delete objects from time to time. In the long run, this might cause memory issues or performance issues, especially on mobile devices as they have less memory and CPU speed than desktops. The key concept here is reuse.

To understand how pooling is implemented, think of a bag full of footballs. Whenever a child needs a ball to play, he takes one out of the bag. When he is done playing with the ball, he puts it back. The next child then does the same. This is exactly what we are doing here. In our scenario, we call this bag a pool. Whenever we need to display a ball in the game, we request the pool for a ball. The pool then gives us the ball from its collection.

In the event where there are no free balls in the pool, it just creates a new ball object and gives it back to us. Once we are done with the ball object, we release it back to the pool. This increases our game's performance to a good amount, as we are not creating new objects and thereby allocating memory every time. LibGDX provides a class for object pooling called `Pool`. Copy the following code to the `SpawnManager` class:

```
private final static Pool<Ball> ballPool = new Pool<Ball>() {
    // this method runs when a new instance of the ball object needs to be
created (pool is empty and an object has been requested)
    @Override
    protected Ball newObject() {
        Ball ball = new Ball();

        // instantiate basket sprite
        ball.ballSprite = new Sprite(ballTexture);

        return ball;
    }
};
```

The `ballPool` variable is our object pool. This will create a new ball object when it is empty and return the recycled ones from its collection when it's not. We override the `newObject()` method that is called when somebody requests an object from the pool and it is empty. Therefore, a new object has to be created and returned to the caller. Here, we instantiate the `Ball` class and the sprite within it and return it. We need to replace the `createNewBall()`and `resetBall()` methods and paste them in the following code:

```
public static Ball resetBall(Ball ball){
    ball.ballSprite.setSize(ball.ballSprite.getTexture().getWidth()*
(width/BALL_RESIZE_FACTOR),ball.ballSprite.getTexture().getHeight()*
(width/BALL_RESIZE_FACTOR));
    ball.position.set(random.nextInt((int) (width -
ball.ballSprite.getWidth())), height-ball.ballSprite.getHeight());
    ball.velocity.set(0, 0);
    ball.isAlive=true;

    Vector2 center = new Vector2();
    //set the center at the center of ball sprite
    center.x=ball.position.x + (ball.ballSprite.getWidth()/2);
    center.y=ball.position.y + (ball.ballSprite.getHeight()/2);

    ball.ballCircle = new Circle(center, (ball.ballSprite.getHeight()/2));
    return ball;
}
```

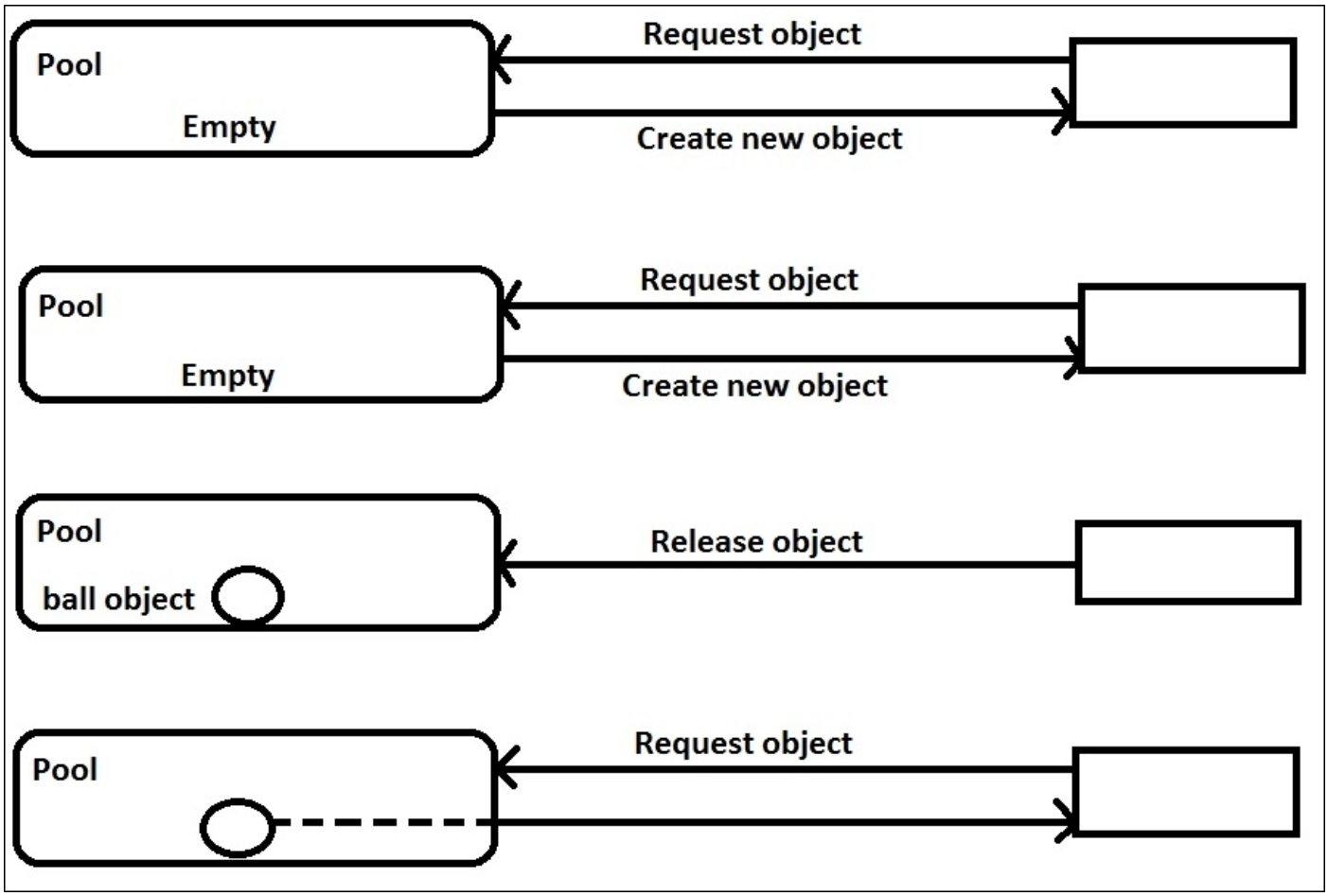
As we can get recycled ball objects, the state is unknown. We will reset the ball's properties in this method. We set the size of the ball with respect to the texture, as it stays the same every time. In the `run()` method, we need to replace the code where we created the new ball:

```
if(delayCounter>=delayTime){
    Ball ball= ballPool.obtain(); // get a ball from the ball pool
    resetBall(ball); // reinitialize the ball
    balls.add(ball); //  add the ball to our list
    delayCounter=0.0f;// reset delay counter
}
```

We also need to free the ball object pool in the `initialize()` method:

```
ballPool.clear(); // clear the object pool
```

When it is time to spawn the ball, we request a `ball` object from the ball pool, reinitialize it, and add it to our active ball list. In our `cleanup()` method, instead of just removing the `ball` objects, we return them to the pool with the `free()` method:



The code for this is as follows:

```
for (int i =0 ;i< removeIndices.size;i++){
    Ball ball= balls.removeIndex(i);
    ballPool.free(ball);// return the ball back to the pool
}
```

If you want to test how many new `ball` objects have been created, add a print statement inside the `newObject()` method.

Keeping the score and maintaining the high score

In this topic, we will learn how to display the game score and save the high score. We will also see how to use custom fonts to display text on the screen.

Keeping the score

We want to keep track of how many times the user has collected the ball and show it to him. So, we add a new variable to the `GameManager` class called `score`:

```
public static int score;
```

Let's initialize it to `0` in the `initialize()` method:

```
score=0;
```

To display the score, we are going to add a new class called `TextManager` to the `com.packtpub.catchtheball.managers` package, similar to what we did previously:

```
package com.packtpub.catchtheball.managers;

import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

public class TextManager {
    static BitmapFont font; // we draw the text to the screen using this
variable

    // viewport width and height
    static float width,height;

    public static void initialize(float width,float height){
        font = new BitmapFont();
        TextManager.width = width;
        TextManager.height= height;
        //set the font color to red
        font.setColor(Color.RED);
        //scale the font size according to screen width
        font.setScale(width/500f);
    }

    public static void displayMessage(SpriteBatch batch){
        float fontWidth =  font.getBounds( "Score:
"+GameManager.score).width; // get the width of the text being displayed

        //top the score display at top right corner
        font.draw(batch, "Score: "+GameManager.score, width - fontWidth -
width/15f,height*0.95f);
    }
}
```

Initialize the `TextManager` class in `GameManager` class' `initialize()` method:

```
TextManager.initialize(width, height);
```

Call the `displayMessage()` method in the `renderGame()` method:

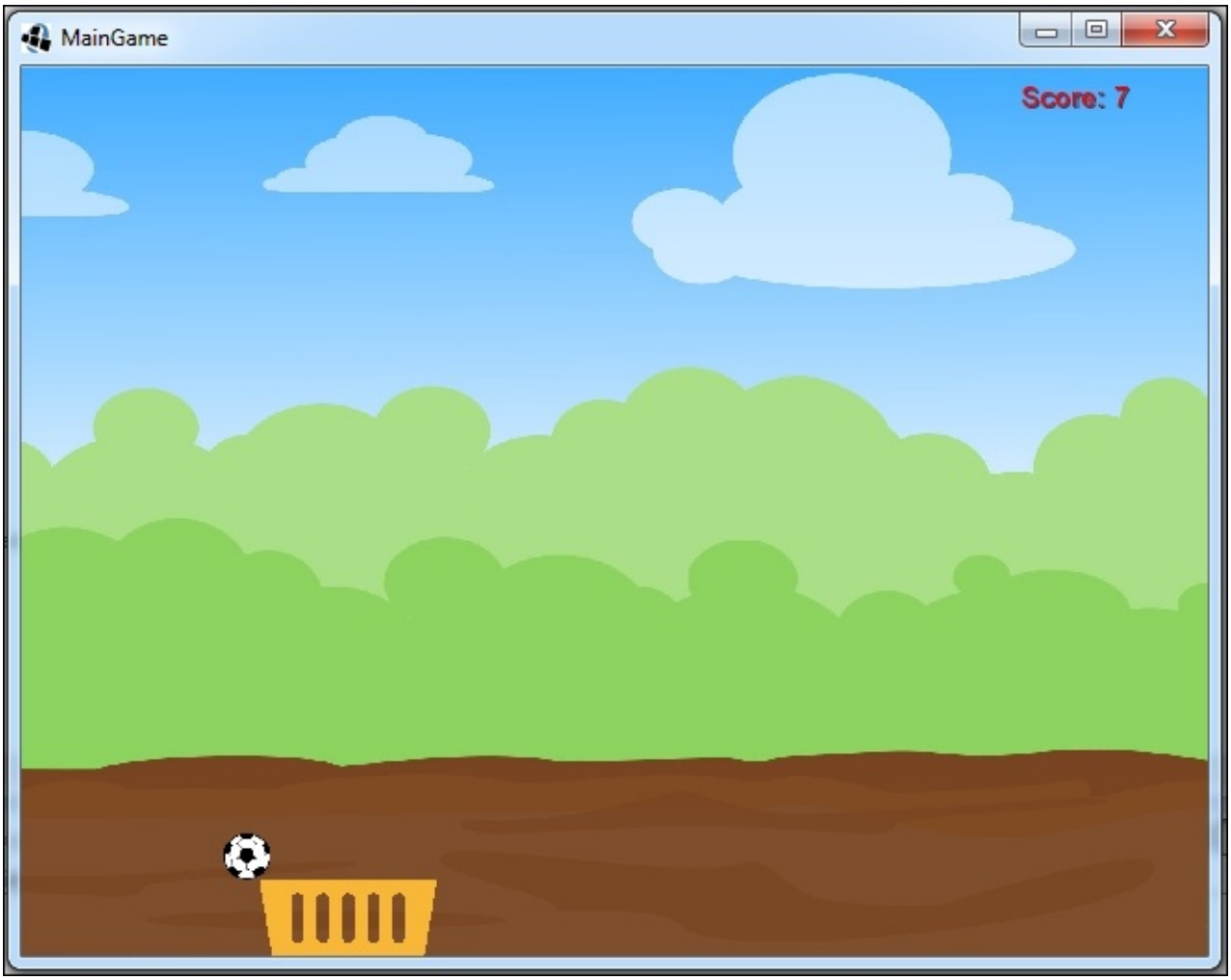
```
TextManager.displayMessage(batch);
```

Finally, in the `Ball` class' `checkCollisionsWithBasket()` method, we increase the score

when we catch the ball with the basket:

```
if(Intersector.overlaps(ballCircle, GameManager.basket.basketRectangle)){
    GameManager.score++;
    isAlive=false;
    return true;
}
```

If you run the game now, you can see the score increasing when we catch the balls:

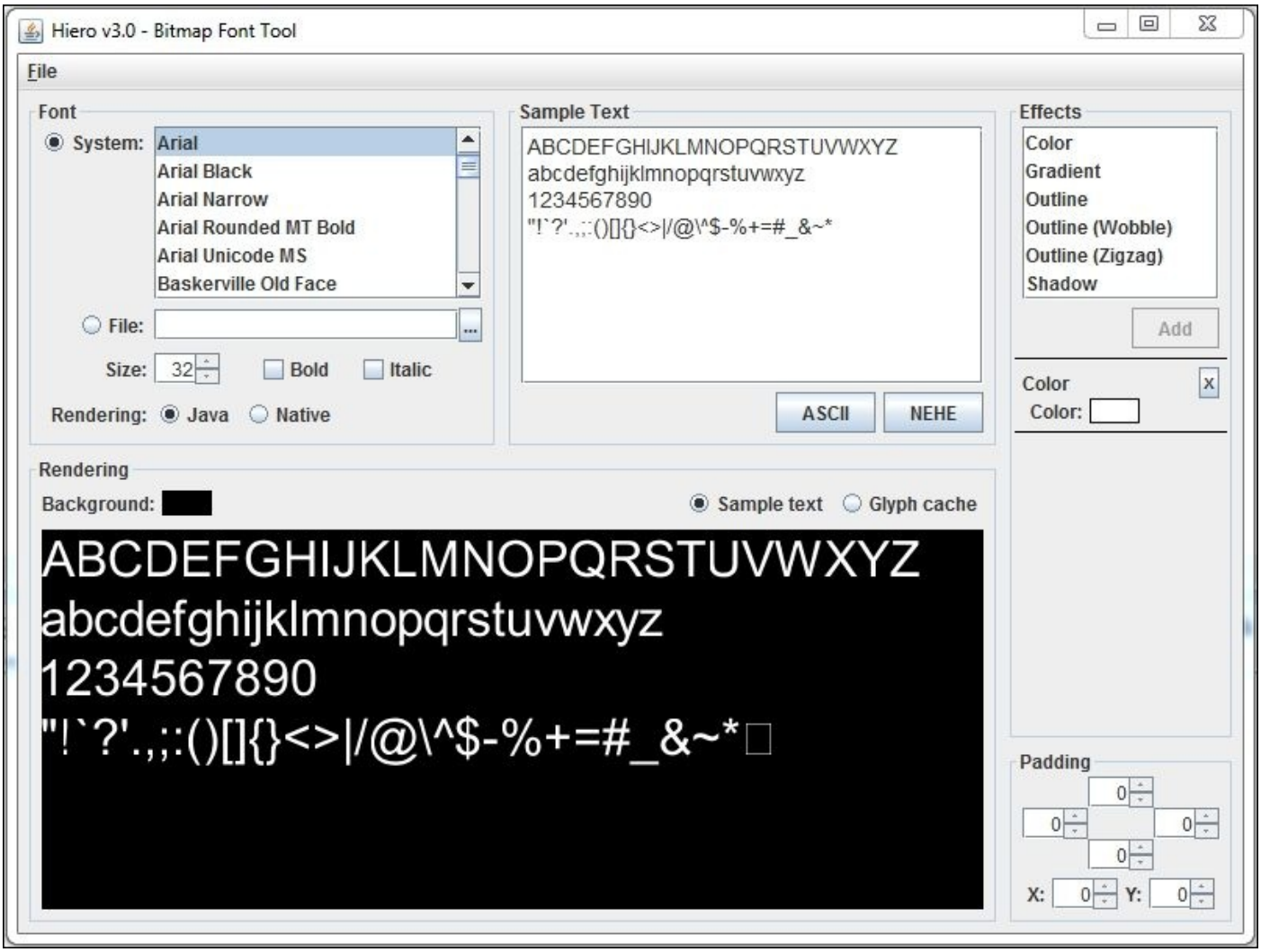


Custom fonts

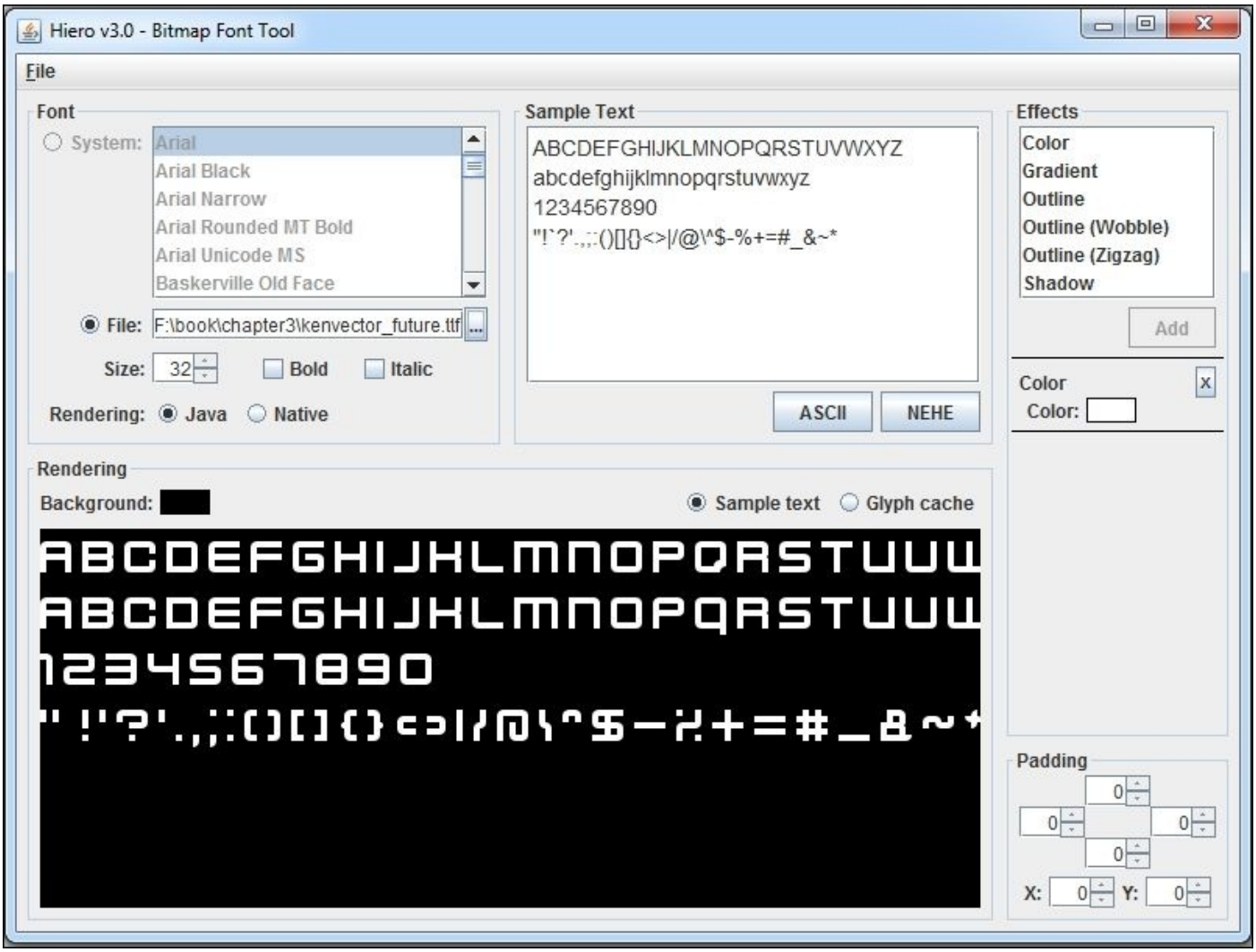
Let's see how to use custom fonts in a game. LibGDX allows you to specify the font file to use within the font's constructor. We cannot use the TrueType font or the `.ttf` file as LibGDX requires the bitmap font format.

The `BitmapFont` file format stores each character as an image. This is very easy and efficient to render instead of the TTF format. So, we need to convert our font file from TTF to the bitmap format. Fortunately, there is a tool called **Hiero** which can do this for us.

You can download Hiero from https://libgdx.googlecode.com/files/hiero.jar. You will get a JAR file, which you can double-click to open:



In the **Font** section, there is a file input area where you can select the TTF file. Once you select it, you can see how the font looks in the rendering section. To keep it simple, we will not add any extra effects:



Save the font by navigating to **File** | **Save BMFont files (text)**. Give the file a `.fnt` extension and save it. Hiero creates one more file with the `.png` extension. You can actually open the image in any image viewer/editor to see how the font characters are stored. To load the font in our game, create a new folder called `fonts` in the `assets/data` directory. Copy the font file and the image to this folder.

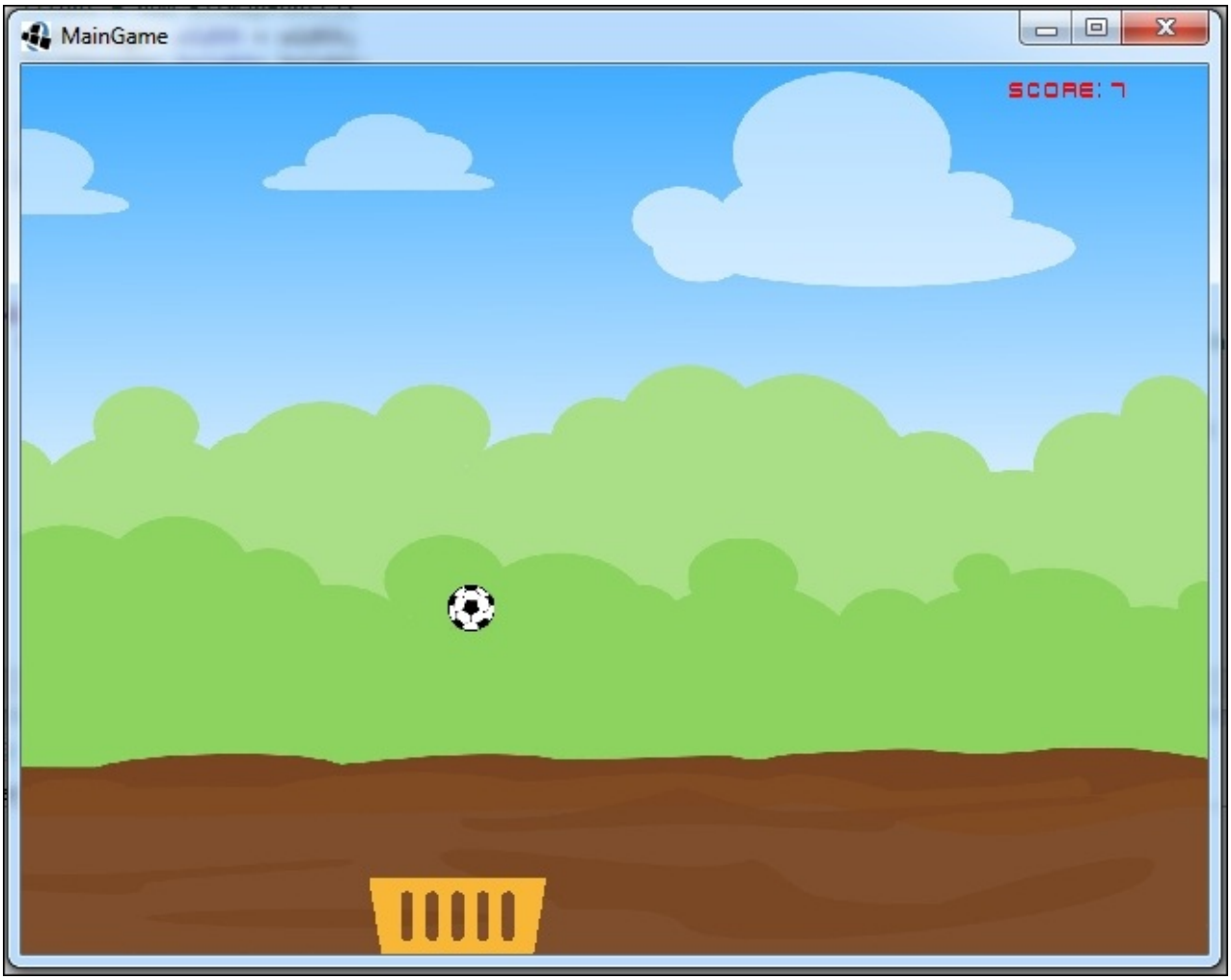
In the code where we instantiated `BitmapFont`, replace the line with the following code:

```
// load the font from the font file
font = new BitmapFont(Gdx.files.internal("data/fonts/[fontname].fnt"));
```

Since we set the font size to 32, we need to resize the font to look better. Next, we set the scale:

```
font.setScale(width/1400f);
```

That’s it. You can now see the score text in your custom font:



Tip

Don’t ship system fonts with your game. You might not have a license for this. You can use royalty-free fonts from the Internet.

Saving high scores

In LibGDX, you can save persistent data, such as a high score, using preferences. Preferences are a way to store the kind of data that will persist after an app relaunch. On desktop OSes, they are stored as files in user directories. On mobile devices, they are stored using native APIs on the devices.

First, let's declare a variable for the high score in the `GameManager` class:

```
public static int highScore; // high score
```

Next, let's declare the variable for preferences:

```
static Preferences prefs; // preferences instance
```

In the `initialize()` method, add the following two lines:

```
prefs = Gdx.app.getPreferences("My Preferences"); // get the preferences
highScore = prefs.getInteger("highscore"); // get current high score
```

We get the preferences and then we get the current high score from them. In the `Ball` class' `checkCollisionsWithBasket()` method, we set the current score to the high score if it exceeds the current high score:

```
GameManager.score++;
if(GameManager.score>GameManager.highScore){
    GameManager.highScore=GameManager.score;
}
```

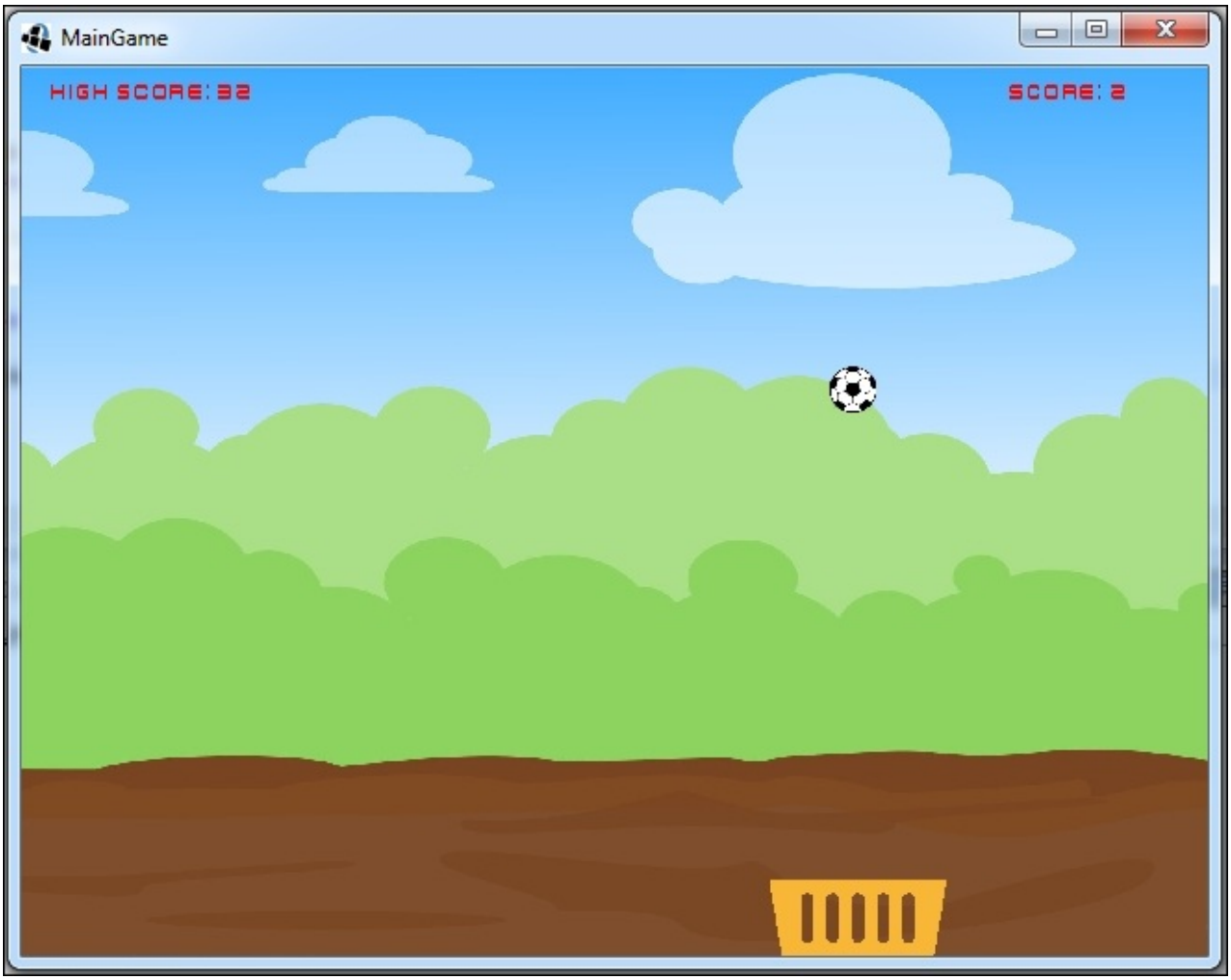
In the `dispose()` method of `GameManager`, when we close our game, we will save the high score:

```
prefs.putInteger("highscore", score);
prefs.flush();
```

To display the high score, add this line to the `TextManager` class' `displayMessage()` method:

```
font.draw(batch, "High Score: "+GameManager.highScore,
width/40f,height*0.95f);
```

This is similar to what we did for the score, except that here we will display the high score text in the top-left corner of our screen:


MainGame
HIGH SCORE: 32
SCORE: 2

Implementing screens

In this section, we will learn how to implement a menu screen for our game and how to transition between it and the game screen.

Implementing the menu screen

Let’s implement a menu screen for our game. The game will start with the menu screen. We will add two buttons to this screen: Start and Exit and a background. On pressing the Start button, the user will be directed to the game screen. On pressing the Exit button, the application quits.

To create the menu screen, create a new class in the `com.packtpub.catchtheball` package called `MenuScreen` and type the following code:

```
package com.packtpub.catchtheball;

import com.badlogic.gdx.Screen;
public class MenuScreen implements Screen {

    @Override
    public void show() {

    }

    @Override
    public void render(float delta) {

    }

    @Override
    public void resize(int width, int height) {

    }

    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void hide() {

    }

    @Override
    public void dispose() {

    }

}
```

To implement a screen in LibGDX, we have to implement the `Screen` interface. As it is an interface, we will have to implement all the methods from it. These methods are similar to

the `AppicationListener` interface, which we saw earlier. It adds the two `show()` and `hide()`methods. These methods are called when the screen is being shown (active) and when the screen is hidden (deactivated).

Let's declare some variables in this class:

```
SpriteBatch batch; // spritebatch for drawing
OrthographicCamera camera;

Texture startButtonTexture;
Texture exitButtonTexture;
Texture backGroundTexture;
Sprite startButtonSprite;
Sprite exitButtonSprite;
Sprite backGroundSprite;

private static float BUTTON_RESIZE_FACTOR = 800f;
private static float START_VERT_POSITION_FACTOR = 2.7f;
private static float EXIT_VERT_POSITION_FACTOR = 4.2f;
```

We declare textures and sprites for the Start and Exit buttons. As there is no `create()` method, we will initialize the variables in the constructor. Let's first initialize the camera and the batch:

```
public MenuScreen(){

    // get window dimensions and set our viewport dimensions
    float height= Gdx.graphics.getHeight();
    float width = Gdx.graphics.getWidth();

    // set our camera viewport to window dimensions
    camera = new OrthographicCamera(width,height);

    // center the camera at w/2,h/2
    camera.setToOrtho(false);

    batch = new SpriteBatch();
}
```

Next, we will initialize our textures and the sprites for the buttons in the same method:

```
//initialize button textures and sprites
startButtonTexture = new
Texture(Gdx.files.internal("data/start_button.png"));
exitButtonTexture = new
Texture(Gdx.files.internal("data/exit_button.png"));
backGroundTexture = new
Texture(Gdx.files.internal("data/menubackground.jpg"));

startButtonSprite = new Sprite(startButtonTexture);
exitButtonSprite = new Sprite(exitButtonTexture);
backGroundSprite = new Sprite(backGroundTexture);

// set the size and positions
startButtonSprite.setSize(startButtonSprite.getWidth()*
(width/BUTTON_RESIZE_FACTOR), startButtonSprite.getHeight()*
```

```
(width/BUTTON_RESIZE_FACTOR));
exitButtonSprite.setSize(exitButtonSprite.getWidth()*
(width/BUTTON_RESIZE_FACTOR), exitButtonSprite.getHeight()*
(width/BUTTON_RESIZE_FACTOR));
backGroundSprite.setSize(width,height);

startButtonSprite.setPosition((width/2f -startButtonSprite.getWidth()/2) ,
width/START_VERT_POSITION_FACTOR);
exitButtonSprite.setPosition((width/2f -exitButtonSprite.getWidth()/2) ,
width/EXIT_VERT_POSITION_FACTOR);

// set the transparency for the background
backGroundSprite.setAlpha(0.2f);
```

The `Sprite` class has a method called `setAlpha()` where you can set the transparency. The values range from 0 to 1. The 0 value makes it completely transparent and 1 makes it completely opaque.

Now, render the objects in the `render()` method:

```
// Clear the screen
Gdx.gl.glClearColor(1, 1, 1, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

// set the spritebatch's drawing view to the camera's view
batch.setProjectionMatrix(camera.combined);

// render the game objects
batch.begin();
backGroundSprite.draw(batch);
startButtonSprite.draw(batch);
exitButtonSprite.draw(batch);
batch.end();
```

Finally, dispose of the objects in the `dispose()` method:

```
startButtonTexture.dispose();
exitButtonTexture.dispose();
batch.dispose();
```

Implementing screen transitions

We created the menu screen, but we haven’t displayed it or handled the screen transitions. Let’s do that. We cannot call this class from the launcher. We need a new class that we can call from the launcher and do screen transitions. This class has to extend the `Game` class from the LibGDX APIs. Create a new class in the `com.packtpub.catchtheball` package called `MainGame` and paste the following code:

```
package com.packtpub.catchtheball;

import com.badlogic.gdx.Game;

public class MainGame extends Game {

    @Override
    public void create() {
        setScreen(new MenuScreen());
    }

}
```

In the `create()` method, we call the `setScreen()` method to change our currently displayed screen to `MenuScreen` by passing an instance of it. In the launcher classes, pass the instance of this class. For example, in the desktop launcher, it is implemented as follows:

```
new LwjglApplication(new MainGame(), config);
```

If you run the game now, you can see the menu screen:

Let’s now implement the transition of the game screens. First, let’s edit the `CatchTheBall` class so that we can call this from the menu screen:

```
public class CatchTheBall implements Screen {
```

Now, instead of extending `ApplicationAdapter`, we will implement the `Screen` interface. Add the unimplemented methods using Eclipse’s assistance, remove any super calls in the implemented methods, and replace the `create()` method with the constructor. You will have to change the signature of the `render()` method as well:

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

In the `MenuScreen` class, let’s declare an instance of the `MainGame` class. We will need this to call its `setScreen()` method to transition between the screens. Parameterize the constructor of the `MenuScreen` class to set this instance:

```
MainGame game; // instance of the main game, to call setScreen methods

MenuScreen(MainGame game){
    this.game= game;
```

We will also need to modify the line in the `create()` method of the `MainGame` class where we set the menu screen to pass its instance:

```
setScreen(new MenuScreen(this));
```

Let's now handle the click/touch input for the buttons. Declare a temporary vector in the `MenuScreen` class to store the input coordinates:

```
Vector3 temp = new Vector3(); //temporary vector to capture input
coordinates
```

We will add a new method called `handleTouch()` to our `MenuScreen` class to handle the touch input:

```
void handleTouch(){
    // Check if the screen is touched
    if(Gdx.input.justTouched()) {
        // Get input touch coordinates and set the temp vector with these
values
        temp.set(Gdx.input.getX(),Gdx.input.getY(), 0);
        //get the touch coordinates with respect to the camera's viewport
        camera.unproject(temp);

        float touchX = temp.x;
        float touchY= temp.y;

        // handle touch input on the start button
        if((touchX>=startButtonSprite.getX()) && touchX<=
(startButtonSprite.getX()+startButtonSprite.getWidth()) &&
(touchY>=startButtonSprite.getY()) && touchY<=
(startButtonSprite.getY()+startButtonSprite.getHeight()) ){
            game.setScreen(new CatchTheBall()); // Bring the game screen to
front

        }

        // handle touch input on the exit button
        else if((touchX>=exitButtonSprite.getX()) && touchX<=
(exitButtonSprite.getX()+exitButtonSprite.getWidth()) &&
(touchY>=exitButtonSprite.getY()) && touchY<=
(exitButtonSprite.getY()+exitButtonSprite.getHeight()) ){
            Gdx.app.exit(); // Quit the application

        }

    }
}
```

In this method, after capturing the input coordinates, we first check whether the user has touched the Start button. If he has touched it, then we bring the game screen to the front by calling the `setScreen()` method of the game. If the user has touched the Exit button, then we quit the application. We call this method in the `render()` method:

```
batch.end();

handleTouch();
```

We call the `dispose()` method to free resources in the `hide()` method, as it is not called by the framework automatically this time. When we switch from one screen to another, the

`hide()` method is called for the first screen:

```
public void hide() {
    dispose();
}
```

Implementing the Back button

We can go from the menu screen to the game screen, but we can't go back. Let's add this functionality with the help of a Back button. First, we will save a reference to the `MainGame` object in the `CatchTheBall` class so that we can switch screens:

```
public static MainGame game; // instance of the main game, to call
setScreen methods

CatchTheBall (MainGame game) {
    CatchTheBall.game = game;
```

We will pass the reference from the `MenuScreen` class:

```
game.setScreen(new CatchTheBall(game)); // Bring the game screen to front
```

Now, we'll declare the texture and the sprite for the Back button in the `GameManager` class:

```
public static Sprite backButtonSprite; // back button sprite
public static Texture backButtonTexture; // texture image for the back
button
```

We need to initialize them in the `initialize()` method:

```
//load back button texture
backButtonTexture = new Texture(Gdx.files.internal("data/backbutton.png"));
//set back button sprite with the texture
backButtonSprite= new Sprite(backButtonTexture);
```

Set the Back button's dimensions and position it on the top center of the screen:

```
backButtonSprite.setSize(backButtonSprite.getWidth()*
(width/BACK_BTN_RESIZE_FACTOR), backButtonSprite.getHeight()*
(width/BACK_BTN_RESIZE_FACTOR));
// set the button's position to top center
backButtonSprite.setPosition(width/2- backButtonSprite.getWidth()/2,
height*0.935f);
```

Render it in the `renderGame()` method:

```
//draw the back  button
backButtonSprite.draw(batch);
```

Finally, dispose of the texture when it is no longer needed using the `dispose()` method:

```
backButtonTexture.dispose();
```

Now, we need to handle touch/tap events on the Back button so that we can go back to the menu screen. We will add a method named `handleBackButton()` to the `InputManager` class. This will check whether the Back button has been touched and set the screen back to the menu screen:

```
public void handleBackButton(float touchX,float touchY){
    // handle touch input on the back button
    if((touchX>=GameManager.backButtonSprite.getX()) && touchX<=
(GameManager.backButtonSprite.getX()+GameManager.backButtonSprite.getWidth(
```

```
)) && (touchy>=GameManager.
      backButtonSprite.getY()) && touchY<=
(GameManager.backButtonSprite.getY()+GameManager.backButtonSprite.getHeight
()) ){
        CatchTheBall.game.setScreen(new MenuScreen(CatchTheBall.game)); //
Bring the menu screen to front
    }
}
```

We will call this method in the `touchup()` method:

```
GameManager.basket.handleTouch(touchX, touchY);
handleBackButton(touchX, touchY);
```

We will call the `dispose()` method in the `hide()` method of the `CatchTheBall` class:

```
@Override
public void hide() {
    dispose();
}
```

The screen will now look like this:

Catching the Back button

In Android, when the user presses the Back button, he is taken out of our application. This is the default behavior of the OS. We need our application to go back to the menu screen after the Back button is pressed. For this to happen, the OS needs to pass the key event to our application so that we can override the default behavior. In the `GameManager` class' `initialize()` method, just add this line of code:

```
Gdx.input.setCatchBackKey(true); // catch back key press event
```

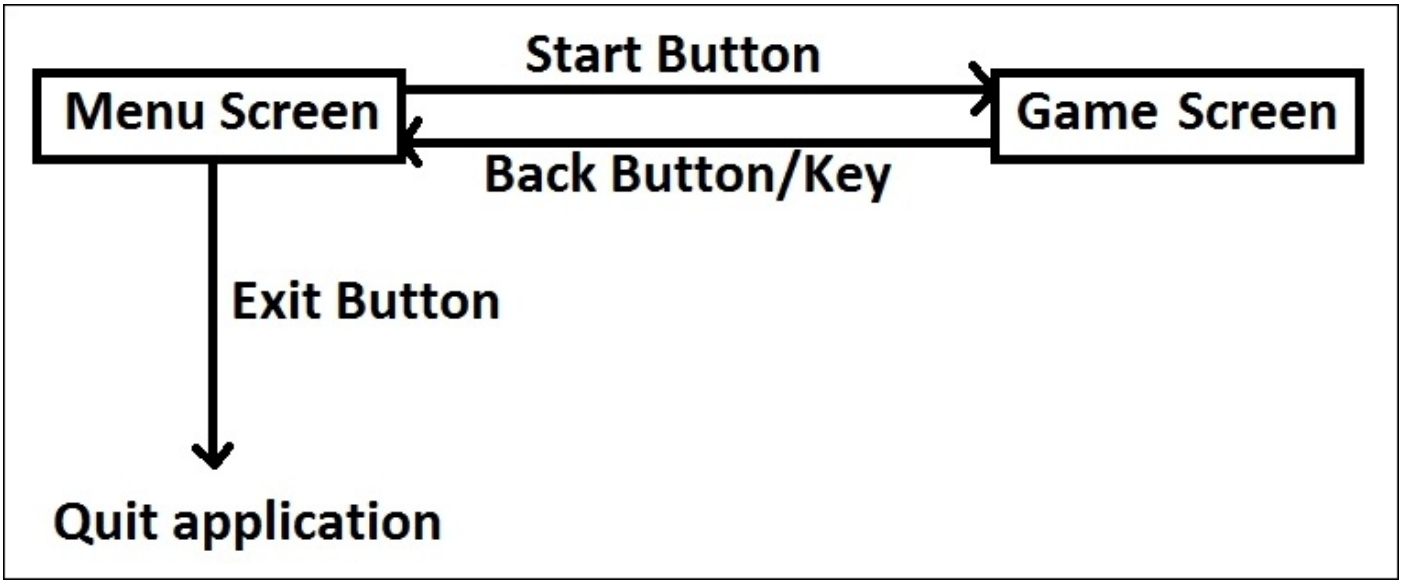
Now, our `InputManager` will receive the Back keypress event. We need to handle this by implementing the `keyUp()` method:

```
@Override
public boolean keyUp(int keycode) {
    if(keycode==Keys.BACK){
        CatchTheBall.game.setScreen(new MenuScreen(CatchTheBall.game)); //
Bring the menu screen to front
    }
    return false;
}
```

This method receives the key code as an argument. We then check whether the key pressed was the Back button, and if it is, then we set the current screen to the menu screen. We can even handle the *Esc* key on the desktop and cause the transition from the game screen to the menu screen as well, as shown in the following code:

```
if(keycode==Keys.BACK || keycode==Keys.ESCAPE)
```

Let's take a look at the following diagram:



Adding sound effects and music

In this section, we will add collision sound effects and background music to our game.

Adding sound effects

We will play a different sound effect when the ball is colliding with the ground and when it is collected by the basket. Let's add the variables that hold our sound instances in the `GameManager` class:

```
public static Sound groundHitSound; // instance of sound to play when the
ball hits the ground
public static Sound basketHitSound; // instance of sound to play when the
ball is collected by the basket
```

Make a new folder called `sounds` in the Android project's `assets/data` directory and copy the two files for the effects in it. Let's initialize the instance in the `initialize()` method:

```
//load the sound effects from file
groundHitSound =
Gdx.audio.newSound(Gdx.files.internal("data/sounds/groundHit.wav"));
basketHitSound =
Gdx.audio.newSound(Gdx.files.internal("data/sounds/basketHit.wav"));
```

In the `Ball` class' `checkCollisionsWithGround()` method, we will play `groundHitSound` when it collides with the ground:

```
public boolean checkCollisionsWithGround(){
    // check if the ball hit the ground
    if(position.y<=0.0){
        GameManager.groundHitSound.play();
```

In the `checkCollisionsWithBasket()` method, we will play `basketHitSound` when it is collected by the basket:

```
if(Intersector.overlaps(ballCircle, GameManager.basket.basketRectangle)){
    GameManager.groundHitSound.play();
```

Finally, we will dispose of the sound instances when they are not needed using the `dispose()` method:

```
//dispose the sound instances
groundHitSound.dispose();
basketHitSound.dispose();
```

Adding background music

To play background music, we will use the `Music` interface. Music files are usually longer in length than sound effects. This is why they are streamed from the disk rather than loaded in the memory.

Let's add a music instance to our `GameManager` class:

```
public static Music  backgroundMusic; // instance of background music
```

Copy the music file to the `sounds` folder. We will load the music in the `initialize()` method:

```
backgroundMusic =
Gdx.audio.newMusic(Gdx.files.internal("data/sounds/backmusic.mp3")); //
load the background music from file
```

Let's set the music to looping, which will replay the music after it is over:

```
backgroundMusic.setLooping(true); // set the music to loop
```

We will play the music by calling the `play()` method on the instance:

```
backgroundMusic.play(); // play the music
```

Finally, we will dispose of the instance in the `dispose()` method if not needed:

```
backgroundMusic.dispose();
```

We need to stop the music instance when we dispose of the resources in the `CatchTheBall` class' `dispose()` method:

```
@Override
public void dispose() {
    //dispose the batch and the textures
    batch.dispose();
    GameManager.backgroundMusic.stop();
    GameManager.dispose();
}
```

To pause the music, we can call the `pause()` method on the instance, and to stop it, there is a `stop()` method as well.

Summary

In this chapter, we made a game called Catch the Ball and learned some concepts along the way. These include the following:

- Motion physics
- Collision detection
- Optimizing memory
- Using custom fonts
- Saving high scores
- Implementing different screens
- Adding music

In the next chapter, we will begin learning about creating a platformer game called Dungeon Bob. We will learn about character motion and animation.

Chapter 4. Dungeon Bob

Starting from this chapter, we are going to make a platform game called Dungeon Bob. The story is quite simple. Our main character, Bob, is stuck in a dungeon and has to find his way out to get home. The dungeon is full of hostile creatures who will attack him, and he needs to fight them. It also contains some environmental hazards, which he needs to evade. This game will have multiple levels and the player must complete all of them to beat the game.

We will cover the following topics in this chapter:

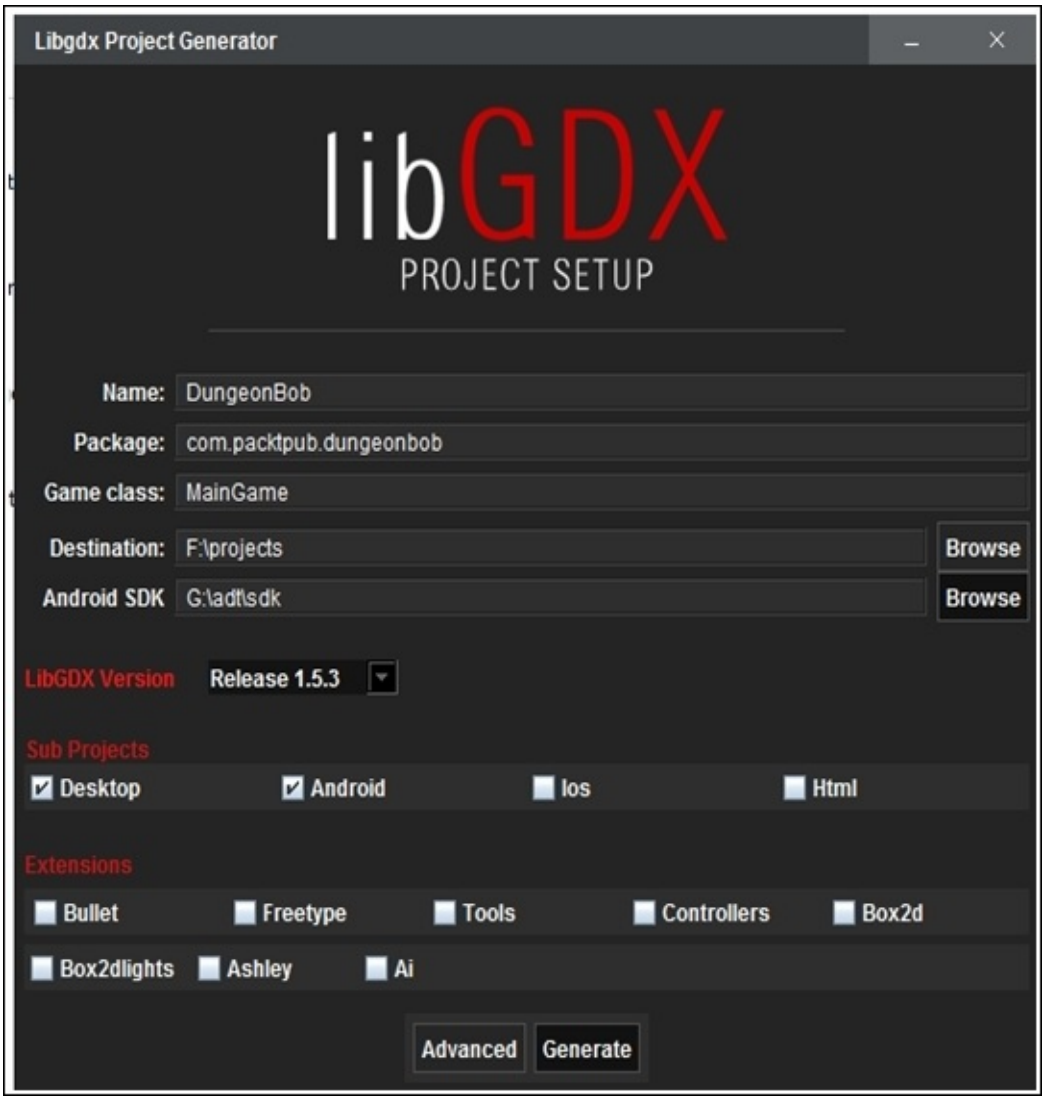
- Creating the player
- Moving the player
- Character animation

Creating the player

In this section, we will see how to create the player character and display it against a background.

Implementing the Bob class

First, we need to set up a new project. Run the project generator and follow on similar lines as in the previous chapters. The following is the screenshot of my project settings:



Import the project in Eclipse as usual. Let's make a class to represent our player, Bob. Create a new package in the core projects and name it `com.packtpub.dungeonbob.gameobjects`. Create a new Java class in this package and name it `Bob`. Type the following code in the file:

```
package com.packtpub.dungeonbob.gameobjects;

import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

public class Bob {
    public Sprite bobSprite; //sprite to display Bob

    public void render(SpriteBatch batch){
        bobSprite.draw(batch);
    }
```

```
    public void setPosition(float x,float y){
        bobSprite.setPosition(x, y);
    }
}
```

Implementing the GameManager class

Create a new package called `com.packtpub.dungeonbob.managers`. Create a new `GameManager.java` file in this package. Type the following content:

```
package com.packtpub.dungeonbob.managers;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.dungeonbob.gameobjects.Bob;

public class GameManager {
    static Bob bob; // bob instance
    static Texture bobTexture; // texture image for the bob
    public static Sprite backgroundSprite; // background sprite
    public static Texture backgroundTexture; // texture image for the
background
    public static final float BOB_RESIZE_FACTOR = 400f;

    public static void initialize(float width,float height){
        // instantiate the bob
        bob = new Bob();
        // load the bob texture with image from file
        bobTexture = new Texture(Gdx.files.internal("data/bob.png"));
        // instantiate bob sprite
        bob.bobSprite = new Sprite(bobTexture);
        //set the size of the bob
        bob.bobSprite.setSize(bob.bobSprite.getWidth()*
(width/BOB_RESIZE_FACTOR), bob.bobSprite.getHeight()*
(width/BOB_RESIZE_FACTOR)); // set the position of bob to bottom - center
        bob.setPosition(width/2f, 0);
        //load background texture
        backgroundTexture = new
Texture(Gdx.files.internal("data/background.jpg"));
        //set background sprite with the texture
        backgroundSprite= new Sprite(backgroundTexture);
        // set the background to completely fill the screen
        backgroundSprite.setSize(width, height);
    }

    public static void renderGame(SpriteBatch batch){
        // draw the background
        backgroundSprite.draw(batch);
        // Render(draw) the bob
        bob.render(batch);

    }
    public static void dispose() {
        //dispose the background texture
        backgroundTexture.dispose();
        // dispose of the bob texture to ensure no memory leaks
        bobTexture.dispose();
```

```
    }
}
```

Implementing the GameScreen class

Create a new class in `com.packtpub.dungeonbob` called `MainGame` and paste it in the following code:

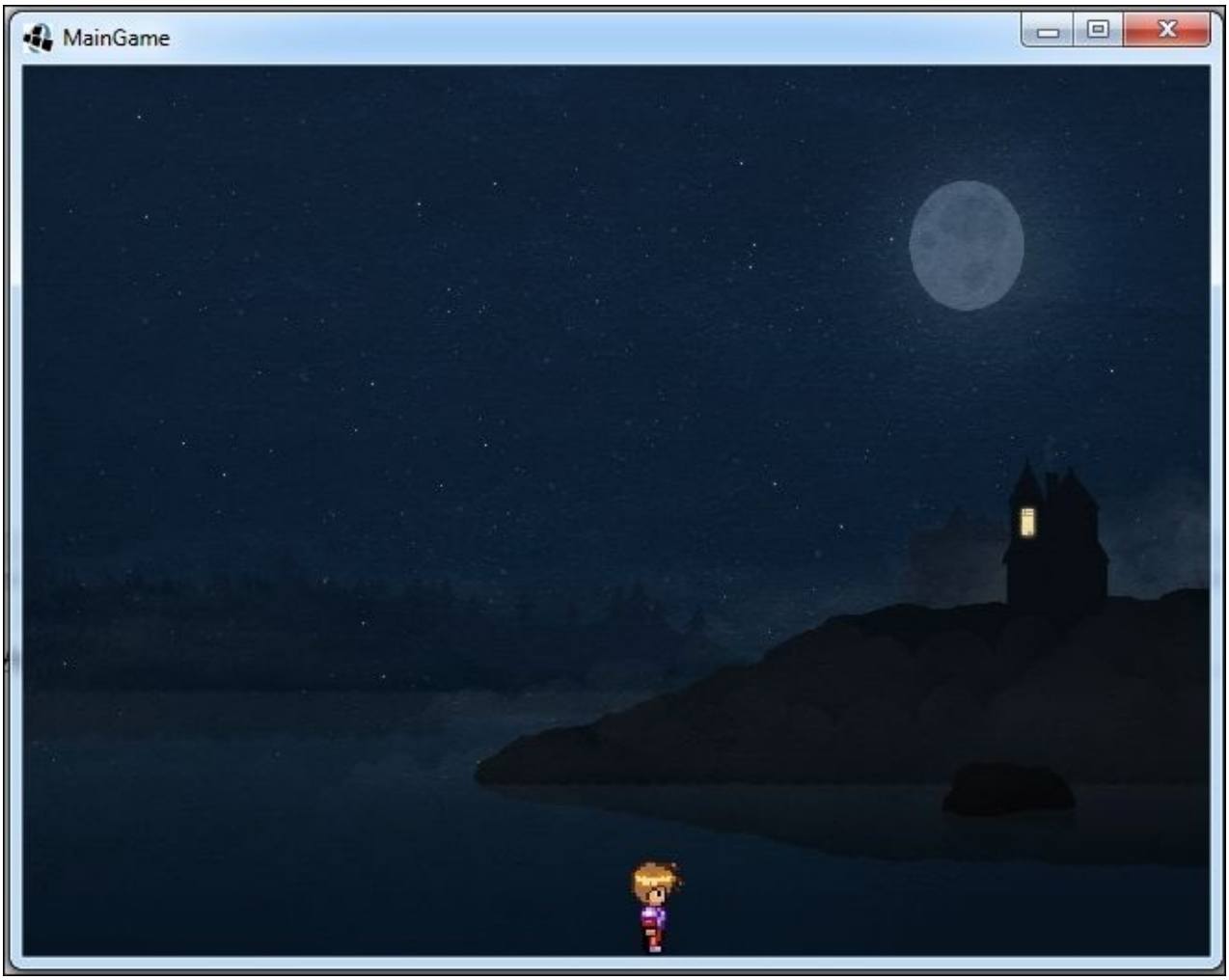
```
package com.packtpub.dungeonbob;
import com.badlogic.gdx.Game;

public class MainGame extends Game {

    @Override
    public void create() {
        setScreen(new GameScreen(this));
    }
}
```

This is our entry point in the game and it sets the current screen to `GameScreen`, which we are going to implement. Create a new class in `com.packtpub.dungeonbob` called `GameScreen` and paste it in the following code:

```
package com.packtpub.dungeonbob;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Screen;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.dungeonbob.managers.GameManager;

public class GameScreen implements Screen {

    MainGame game ;
    SpriteBatch batch; // spritebatch for drawing
    OrthographicCamera camera;

    public GameScreen (MainGame game){
        this.game=game;
        // get window dimensions and set our viewport dimensions
        float height= Gdx.graphics.getHeight();
        float width = Gdx.graphics.getWidth();
        // set our camera viewport to window dimensions
        camera = new OrthographicCamera(width,height);
        // center the camera at w/2,h/2
        camera.setToOrtho(false);

        batch = new SpriteBatch();
        //initialize the game
        GameManager.initialize(width, height);

    }

    @Override
    public void show() {

    }
```

```
    @Override
    public void render(float delta) {
        // Clear the screen
        Gdx.gl.glClearColor(1, 1, 1, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

        // set the spritebatch's drawing view to the camera's view
        batch.setProjectionMatrix(camera.combined);

        // render the game objects
        batch.begin();
        GameManager.renderGame(batch);
        batch.end();

    }

    @Override
    public void resize(int width, int height) {

    }

    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void hide() {

    }

    @Override
    public void dispose() {
        //dispose the batch and the textures
        batch.dispose();
        GameManager.dispose();
    }
}
```

The screen will look like this:


MainGame

Moving the player

This section handles the movement of the player, Bob, for both desktop and mobile platforms.

Bob’s movement on desktop

We are going to make Bob move by taking an input from the keyboard. First, let’s add a method to the `Bob` class called `move()`:

```
// move bob's with the specified amount
public void move (float x, float y){
    setPosition(bobSprite.getX()+x,bobSprite.getY()+y);
}
```

This method takes the distance Bob needs to move in the *x* and *y* directions and updates his position by that. Let’s add a new class to handle the input. Create a class named `InputManager` in the `com.packtpub.dungeonbob.managers` package and type the following code:

```
package com.packtpub.dungeonbob.managers;

import com.badlogic.gdx.InputAdapter;
import com.badlogic.gdx.Input.Keys;

public class InputManager extends InputAdapter{

    @Override
    public boolean keyDown(int keycode) {
        // move bob 3 units to left or right depending on the key presses
        if(keycode==Keys.LEFT){
            GameManager.bob.move(-3f, 0);
        }

        else if(keycode==Keys.RIGHT){
            GameManager.bob.move(3f, 0);
        }

        return false;
    }

}
```

Here, we move Bob 3 units to the left if the left key is pressed on the keyboard and 3 units to the right if the right key is pressed. In the `GameScreen` class’ constructor, add the following line so that `InputManager` can start receiving input events:

```
Gdx.input.setInputProcessor(new InputManager()); // enable InputManager to
receive input events
```

If you run the game now, you can move Bob by pressing left or right keys on the keyboard.

Continuous movement

You must have noticed that Bob does not move continuously if the left or right key is kept pressed. You have to tap the keys again and again for him to move. To enable a continuous movement, let's first add two variables to the `Bob` class and also a constant to signify horizontal movement units:

```
boolean isLeftPressed;// indicates if left key is pressed
boolean isRightPressed; // indicates if right key is pressed
private static final float X_MOVE_UNITS = 3f;// units bob will move in x
direction
```

We will keep track of whether the left or right key is pressed with the help of these two variables. We will need to add two functions that will set or unset the values of the variables on input:

```
public void setLeftPressed(boolean isPressed)
{
    // to have motion in only one direction if both are pressed
    if(isRightPressed && isPressed){
        isRightPressed = false;
    }

    isLeftPressed = isPressed;
}
public void setRightPressed(boolean isPressed)
{
    // to have motion in only one direction if both are pressed
    if(isLeftPressed && isPressed){
        isLeftPressed = false;
    }

    isRightPressed = isPressed;
}
```

In the `setLeftPressed()` method, we pass a `isPressed` value, which indicates the state of the left button. If the right button is already pressed, we mark it as not pressed. We then update the value of `isLeftPressed`. A similar logic applies to the `setRightPressed()` method. To have a continuous motion, we will also need an update method, which will be called continuously:

```
public void update(){
    // move specified units to left if left key is pressed
    if (isLeftPressed){
        move (-X_MOVE_UNITS ,0);
    }

    // move specified units to right if right key is pressed
    else if (isRightPressed){
        move (X_MOVE_UNITS,0);
    }
}
```

We move the player to the left by 3 units if the left key is pressed and to the right by 3

units if the right key is pressed. This method is called in the `GameManager` class' `renderGame()` method:

```
bob.update()
// Render(draw) the bob
bob.render(batch);
```

Our input handling logic in the `InputManager` class also needs to be changed:

```
@Override
public boolean keyDown(int keycode) {
    // set the left key status to pressed
    if(keycode==Keys.LEFT){
        GameManager.bob.setLeftPressed(true);
    }
    // set the right key status to pressed
    else if(keycode==Keys.RIGHT){
        GameManager.bob.setRightPressed(true);
    }

    return false;
}

@Override
public boolean keyUp(int keycode) {
    // set the left key status to pressed
    if(keycode==Keys.LEFT){
        GameManager.bob.setLeftPressed(false);
    }
    // set the right key status to pressed
    else if(keycode==Keys.RIGHT){
        GameManager.bob.setRightPressed(false);
    }

    return false;
}
```

We basically set the pressed status of the corresponding keys to `true` in the `keyDown()` function and to `false` in the `keyUp()` function. If you now run the game, you will be able to see Bob moving continuously to the left or the right when the corresponding buttons are kept pressed.

Bob’s movement on mobile

We designed the movement for Bob keeping desktops in mind, which have a keyboard. Most of the mobile devices don’t have a keyboard nowadays, so we need to make up for this fact. What we are going to do is display a paddle on the screen that upon touch will handle the movement for Bob.

So, let’s add a left and right paddle to our game. We declare two textures and sprites in the `GameManager` class, one for each paddle:

```
static Texture leftPaddleTexture;
static Texture rightPaddleTexture;
static Sprite leftPaddleSprite;
static Sprite rightPaddleSprite;
public static final float PADDLE_RESIZE_FACTOR = 700f;
public static final float PADDLE_ALPHA = 0.25f;
public static final float PADDLE_HORIZ_POSITION_FACTOR = 0.02f;
public static final float PADDLE_VERT_POSITION_FACTOR = 0.01f;
```

Let’s create a method called `initializeLeftPaddle()` to instantiate and initialize the left paddle:

```
public static void initializeLeftPaddle(float width,float height){
    //load background texture
    leftPaddleTexture = new
Texture(Gdx.files.internal("data/paddleLeft.png"));
    //set left paddle sprite with the texture
    leftPaddleSprite= new Sprite(leftPaddleTexture);
    // resize the sprite
    leftPaddleSprite.setSize(leftPaddleSprite.getWidth()*width/
PADDLE_RESIZE_FACTOR, leftPaddleSprite.getHeight()*width/
PADDLE_RESIZE_FACTOR);
    // set the position to bottom left corner with offset
    leftPaddleSprite.setPosition(width* PADDLE_HORIZ_POSITION_FACTOR,
height* PADDLE_VERT_POSITION_FACTOR);
    // make the paddle semi transparent
    leftPaddleSprite.setAlpha(PADDLE_ALPHA);
}
```

We make the paddle semitransparent so that the game world can also be partially seen behind. We do the same thing for the right paddle:

```
public static void initializeRightPaddle(float width,float height){
    //load background texture
    rightPaddleTexture = new
Texture(Gdx.files.internal("data/paddleRight.png"));
    //set right paddle sprite with the texture
    rightPaddleSprite= new Sprite(rightPaddleTexture);
    // resize the sprite
    rightPaddleSprite.setSize(rightPaddleSprite.getWidth()*width/
PADDLE_RESIZE_FACTOR, rightPaddleSprite.getHeight()*width/
PADDLE_RESIZE_FACTOR);
    // set the position to bottom left corner with offset
    rightPaddleSprite.setPosition(leftPaddleSprite.getX()+
leftPaddleSprite.getWidth()+ width*PADDLE_HORIZ_POSITION_FACTOR,
```

```
height*PADDLE_VERT_POSITION_FACTOR);
    // make the paddle semi transparent
    rightPaddleSprite.setAlpha(PADDLE_ALPHA);
}
```

We call these functions in the `GameManager` class' `initialize()` method:

```
initializeLeftPaddle(width,height);
initializeRightPaddle(width,height);
```

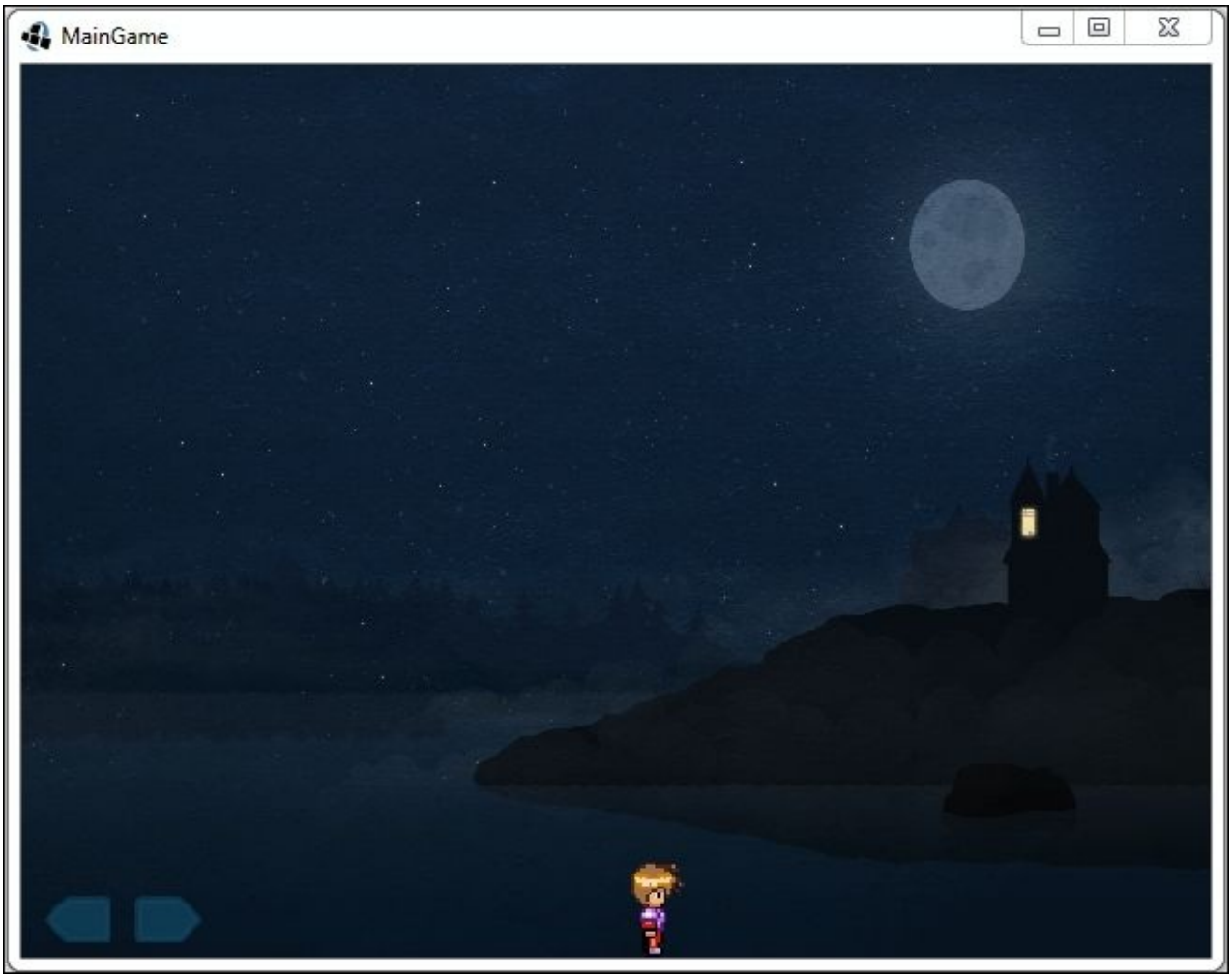
We display the paddles in the `renderGame()` method:

```
leftPaddleSprite.draw(batch);
rightPaddleSprite.draw(batch);
```

Finally, we dispose the textures in the `dispose()` method:

```
leftPaddleTexture.dispose();
rightPaddleTexture.dispose();
```

The screen now looks like this:



To detect the touch input, we will add two new methods to the `InputManager` class. First, we will add the `isLeftPaddleTouched()` method to handle the touch input on the left paddle:

```
boolean isLeftPaddleTouched(float touchX, float touchY){
    // handle touch input on the left paddle
    if((touchX>=GameManager.leftPaddleSprite.getX()) && touchX<=
(GameManager.leftPaddleSprite.getX()+GameManager.leftPaddleSprite.getWidth(
)) && (touchY>=GameManager.leftPaddleSprite.getY()) && touchY<=
(GameManager.leftPaddleSprite.getY()+GameManager.leftPaddleSprite.getHeight
()) ){
        return true;
    }
    return false;
}
```

We have a similar function for the right paddle:

```
boolean isRightPaddleTouched(float touchX, float touchY){
    // handle touch input on the right paddle
    if((touchX>=GameManager.rightPaddleSprite.getX()) && touchX<=
(GameManager.rightPaddleSprite.getX()+GameManager.rightPaddleSprite.getWidt
h()) && (touchy>=GameManager.rightPaddleSprite.getY()) && touchY<=
(GameManager.rightPaddleSprite.getY()+GameManager.rightPaddleSprite.getHeig
ht()) ){
        return true;
    }
    return false;
}
```

In the `Bob` class, we need to add two variables that check whether the corresponding paddles are touched or not:

```
boolean isLeftPaddleTouched;  // indicates if left paddle is touched
boolean isRightPaddleTouched; // indicates if right paddle is touched
```

Similar to what we did previously, we need to add two methods to set the touched status for both the paddles:

```
public void setLeftPaddleTouched(boolean isTouched)
{
    // to restrict motion in only one direction if both are touched
    if(isRightPaddleTouched && isTouched){
        isRightPaddleTouched = false;
    }

    isLeftPaddleTouched = isTouched;
}
public void setRightPaddleTouched(boolean isTouched)
{
    // to restrict motion if both are touched
    if(isLeftPaddleTouched && isTouched){
        isLeftPaddleTouched = false;
    }

    isRightPaddleTouched = isTouched;
}
```

In the `InputManager` class, let's add a constructor to save the `camera` instance, which is required to get the correct touch coordinates. First, let's declare it and a temporary vector:

```
OrthographicCamera camera;
static Vector3 temp = new Vector3(); // temporary vector
public InputManager(OrthographicCamera camera) {

    this.camera = camera;
}
```

Then, edit the corresponding line in `GameScreen`:

```
Gdx.input.setInputProcessor(new InputManager(camera)); // enable
InputManager to receive input events
```

We will override the `onTouchUp()` and `onTouchDown()`callback methods in `InputManager`. The `onTouchDown()` method is called when the user touches the screen. The `onTouchUp()` method is called when the user lifts their finger from the screen:

```
@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button)
{
    temp.set(screenX,screenY, 0);
    //get the touch co-ordinates with respect to the camera's viewport
    camera.unproject(temp);

    float touchX = temp.x;
    float touchY = temp.y;

    if(isLeftPaddleTouched(touchX,touchY)){
        GameManager.bob.setLeftPaddleTouched(true);
    }
    else if(isRightPaddleTouched(touchX,touchY)){
        GameManager.bob.setRightPaddleTouched(true);
    }
    return false;

}

@Override
public boolean touchUp(int screenX, int screenY, int pointer, int button) {
    temp.set(screenX,screenY, 0);
    //get the touch co-ordinates with respect to the camera's viewport
    camera.unproject(temp);

    float touchX = temp.x;
    float touchY = temp.y;

    if(isLeftPaddleTouched(touchX,touchY)){
        GameManager.bob.setLeftPaddleTouched(false);
    }
    else if(isRightPaddleTouched(touchX,touchY)){
        GameManager.bob.setRightPaddleTouched(false);
    }
    return false;
}
```

The logic is similar to how we handle the keyboard input. Now, for the actual movement, add the following lines to the `Bob` class' `update()` method:

```
// move specified units to left if left paddle is touched
if (isLeftPaddleTouched){
    move (-X_MOVE_UNITS,0);
}

// move specified units to right if right paddle is touched
else if (isRightPaddleTouched){
    move (X_MOVE_UNITS,0);
}
```

Character animation

In this section, we will learn how to animate our character Bob.

Walking Bob 1

You might have noticed that when we move Bob, he does not appear to be walking. He just slides on the screen. To create a realistic walking effect, we need to animate our character. Animation is just a sequence of images shown one after the other at regular intervals. This creates an illusion of a continuous motion.

Alright, so let's get started. First, we will need the sequence of images. We will use a single image that consists of all these images. This is called a sprite sheet. I have already acquired the sprite sheet for our character, which looks similar to this:

The image contains the different walk stages of Bob. Each stage or each image of Bob within this sheet is called an animation frame. The interval after which we switch the animation frames is called an animation time. LibGDX has an `Animation` class, which helps us with this. First, let's declare some variables in the `Bob` class:

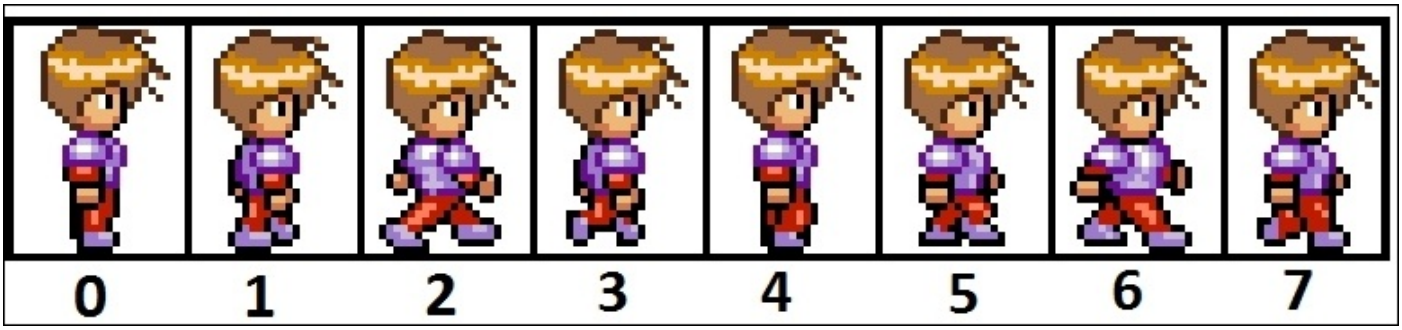
```
Animation walkAnimation;          // animation instance
Texture walkSheet;                // sprite sheet
TextureRegion currentFrame;       // current animation frame
float stateTime;                  // elapsed time

private static int ANIMATION_FRAME_SIZE =8; // this specifies the number of
frames(images) that we are using for animation

private static float ANIMATION_TIME_PERIOD =0.08f;// this specifies the
time between two consecutive frames of animation
```

The `walkAnimation` instance is the animation instance, which will help us animate the player. The `walkSheet` instance is the texture image for Bob's sprite sheet. The `currentFrame` instance is the current walk image from the sprite sheet. The `stateTime` instance is used to keep track of the time if and when Bob moves. Let's create a method called `initialize()` in the `Bob` class to initialize the properties:

```
public void initialize(float width,float height,Texture walkSheet){
    this.walkSheet = walkSheet; // save the sprite sheet
    //split the sprite sheet into different textures
    TextureRegion[][] tmp = TextureRegion.split(walkSheet,
walkSheet.getWidth()/ANIMATION_FRAME_SIZE, walkSheet.getHeight());
    // convert 2D array to 1D
    TextureRegion[] walkFrames = tmp[0];
    // create a new animation sequence with the walk frames and time period
of specified seconds
    walkAnimation = new Animation(ANIMATION_TIME_PERIOD, walkFrames);
}
```

This method takes the screen width, height, and walksheet (sprite sheet) as arguments, which will be passed by the `GameManager` class. We need to split the walksheet into separate frames, which is done by the `TextureRegion.split()` method. We get a 2D array

of `TextureRegion`, out of which we just use the first row to represent a 1D array. We will create the animation with these images/frames with the time period of 0.08 seconds. This means that the time gap between two successive frames is 0.08 seconds:



This concept is explained in the following screenshot:

Let's move some of Bob's initialization lines along with the corresponding constants from `GameManager` to this function with slight changes:

```
// instantiate bob sprite
bobSprite = new Sprite();
//set the size of the bob
bobSprite.setSize((walkSheet.getWidth()/ANIMATION_FRAME_SIZE)*
(width/BOB_RESIZE_FACTOR),walkSheet.getHeight()*(width/BOB_RESIZE_FACTOR));
// set the position of the bob to bottom - center
setPosition(width/2f, 0);
```

As we are not going to use Bob's static image here, replace `Texture` with the sprite sheet:

```
static Texture bobSpriteSheet; // texture sprite sheet for the bob
```

In the `initialize()` method, add the following code:

```
// instantiate the bob
bob = new Bob();
// load the bob texture with image from file
bobSpriteSheet= new
Texture(Gdx.files.internal("data/bob_spritesheet.png"));
// initialize Bob
bob.initialize(width,height,bobSpriteSheet);
```

In the `dispose()` method, add the following code:

```
// dispose of the bob spritesheet texture to ensure no memory leaks
bobSpriteSheet.dispose();
```

In the `initialize()` method of the `Bob` class, add these two lines to the end of the code:

```
// set the animation to loop
walkAnimation.setPlayMode(PlayMode.LOOP);
// get initial frame
currentFrame = walkAnimation.getKeyFrame(stateTime, true);
```

We set the animation mode to loop that will cycle the animation frames. When the last frame is drawn, it will start from the beginning and so on. We set the first frame for our player to draw the next line. The way `Animation` class works is that it asks for the state time and based on the time period that we set, it gives us an image from the array of frames. This image or current frame is then drawn by us on the screen.

Walking Bob 2

We keep updating the state time of Bob only when he is moving. It doesn't make any sense to keep updating the animation state time when Bob is not moving. This will also result in an erroneous animation. Let's add a variable to the `Bob` class to implement this:

```
boolean updateAnimationStateTime = false; // keep track of when to update
Bob's state time
```

We will change the `update()` method as follows:

```
public void update(){
    updateAnimationStateTime=false;

    // move specified units to left if left key is pressed
    if (isLeftPressed){
        updateAnimationStateTime=true;
        move (-X_MOVE_UNITS,0);
    }

    // move specified units to right if right key is pressed
    else if (isRightPressed){
        updateAnimationStateTime=true;
        move (X_MOVE_UNITS,0);
    }

    // move specified units to left if left paddle is touched
    if (isLeftPaddleTouched){
        updateAnimationStateTime=true;
        move (-X_MOVE_UNITS,0);
    }

    // move specified units to right if right paddle is touched
    else if (isRightPaddleTouched){
        updateAnimationStateTime=true;
        move (X_MOVE_UNITS,0);
    }

    //If Bob is moving, only then update his state time

    if(updateAnimationStateTime){
        stateTime += Gdx.graphics.getDeltaTime();
        currentFrame = walkAnimation.getKeyFrame(stateTime, true);
    }

}
```

Finally, in the `render()` method of the `Bob` class, we will set the sprite's texture to the one we get from the animation:

```
bobSprite.setRegion(currentFrame); // set the bob sprite's texture to the
current frame
bobSprite.draw(batch);
```

If you now run the game, you should be able to see the walking animation for Bob when

he moves. There is one thing remaining here. When the player moves to the left, he walks backward. We want the player to look toward the left when he moves to the left. To enable this, we will first add an enum to our `Bob` class:

```
enum Direction{LEFT,RIGHT};
Direction direction = Direction.RIGHT; //denotes player's direction.
defaulted to right
```

Next, we will set the player's face direction appropriately while he moves. In the `update()` method, add the following code:

```
public void update(){
    updateAnimationStateTime=false;
    // move specified units to left if left key is pressed
    if (isLeftPressed){
        updateAnimationStateTime=true;
        direction=Direction.LEFT;
        move (-X_MOVE_UNITS,0);
    }

    // move specified units to right if right key is pressed
    else if (isRightPressed){
        updateAnimationStateTime=true;
        direction=Direction.RIGHT;
        move (X_MOVE_UNITS,0);
    }

    // move specified units to left if left paddle is touched
    if (isLeftPaddleTouched){
        updateAnimationStateTime=true;
        direction=Direction.LEFT;
        move (-X_MOVE_UNITS,0);
    }

    // move specified units to right if right paddle is touched
    else if (isRightPaddleTouched){
        updateAnimationStateTime=true;
        direction=Direction.RIGHT;
        move (X_MOVE_UNITS,0);
    }

    //If Bob is moving, only then update his state time
    if(updateAnimationStateTime){
        stateTime += Gdx.graphics.getDeltaTime();
        currentFrame = walkAnimation.getKeyFrame(stateTime, true);
    }

}
```

To actually draw the player with the changed direction, we will update the `update()` method as follows:

```
public void render(SpriteBatch batch){

    bobSprite.setRegion(currentFrame); // set the bob sprite's texture to
```

```
the current frame
    if(direction==Direction.LEFT){
        bobSprite.setFlip(true, false);
    }
    else{
        bobSprite.setFlip(false, false);
    }
    bobSprite.draw(batch);
}
```

The `setFlip()` methods of the sprite class take two arguments that allow us to reverse the sprite's direction horizontally or vertically or both. If the direction is left, we flip Bob's image so that he looks to the left; otherwise, we remove the flip:

If you run the game now, you can see the player looking left and right as we move him.

Summary

In this chapter, we introduced a platform game called Dungeon Bob and learned the following concepts:

- Character motion
- Handling the keyboard input
- A continuous motion
- Facilitating the touch input for mobile devices using navigational paddles
- Character animation

In the next chapter, we will learn about a tool called Tiled. This tool will be used to make levels inside our game.

Chapter 5. Using the Tiled Map Editor

LibGDX is a game framework and not a game engine. This is why it doesn’t have an editor to place game objects or to make levels. Imagine a game, such as *Super Mario*, where you have a lot of levels. It would be cumbersome to place all the objects from the code. Tiled is a 2D level/map editor most suited for this purpose. In this chapter, we will learn how to use this editor and its features and also learn how to use it to visually make levels for your game. LibGDX provides excellent support for reading and rendering maps/levels made through Tiled.

We will cover the following topics in this chapter:

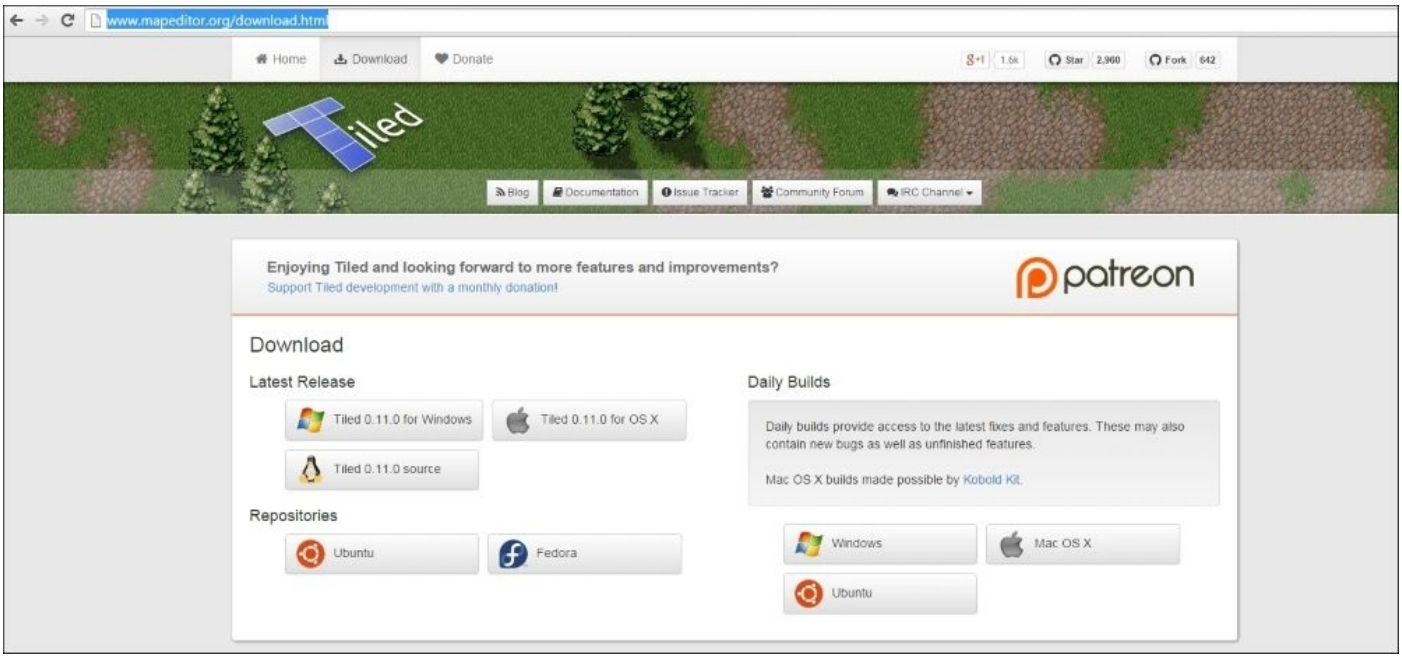
- Installation and basics
- Miscellaneous

Installation and basics

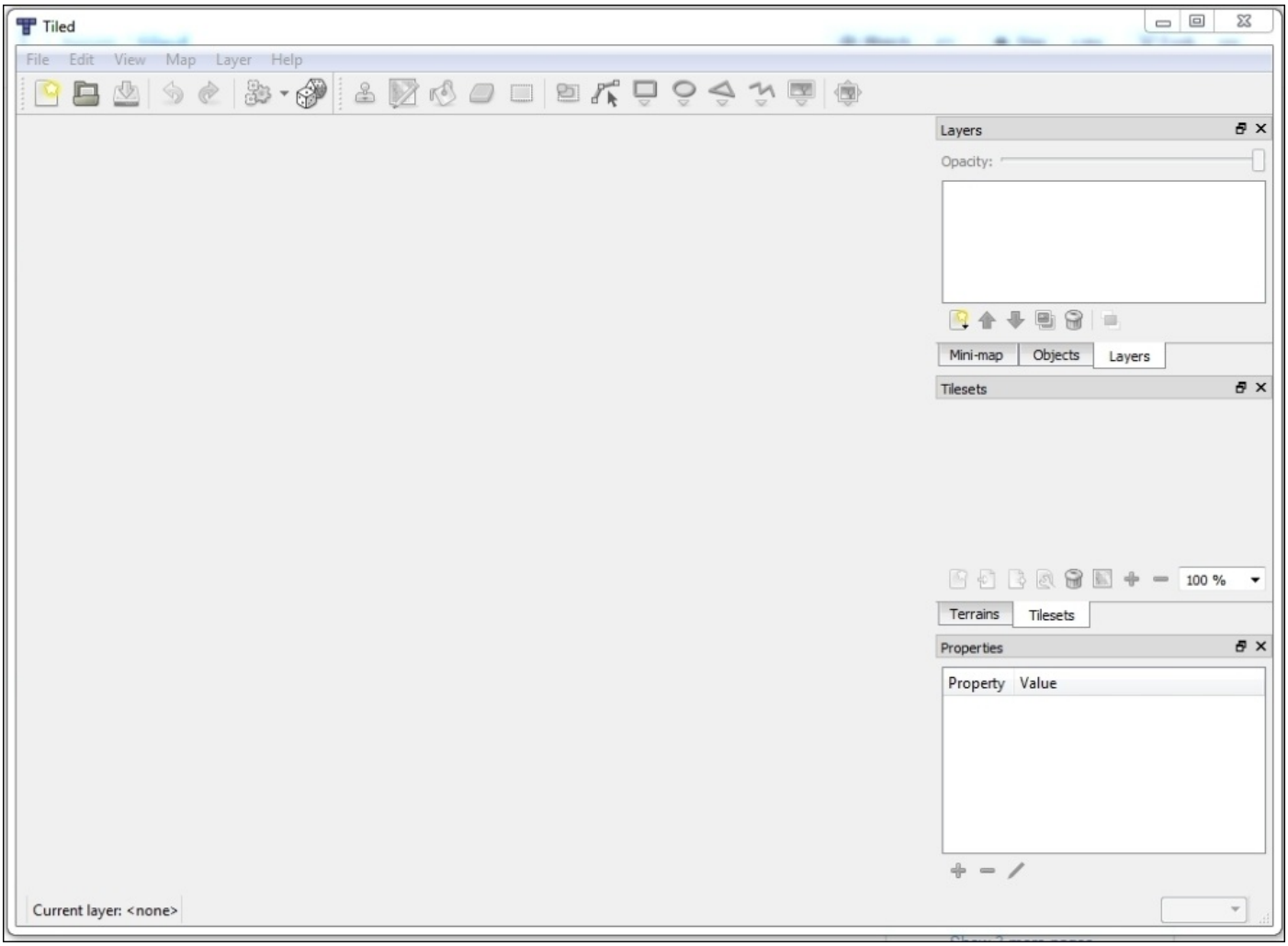
This section will cover how to install and set up the editor. We will also learn about map layers and the basic drawing of maps.

Installing and setting up Tiled

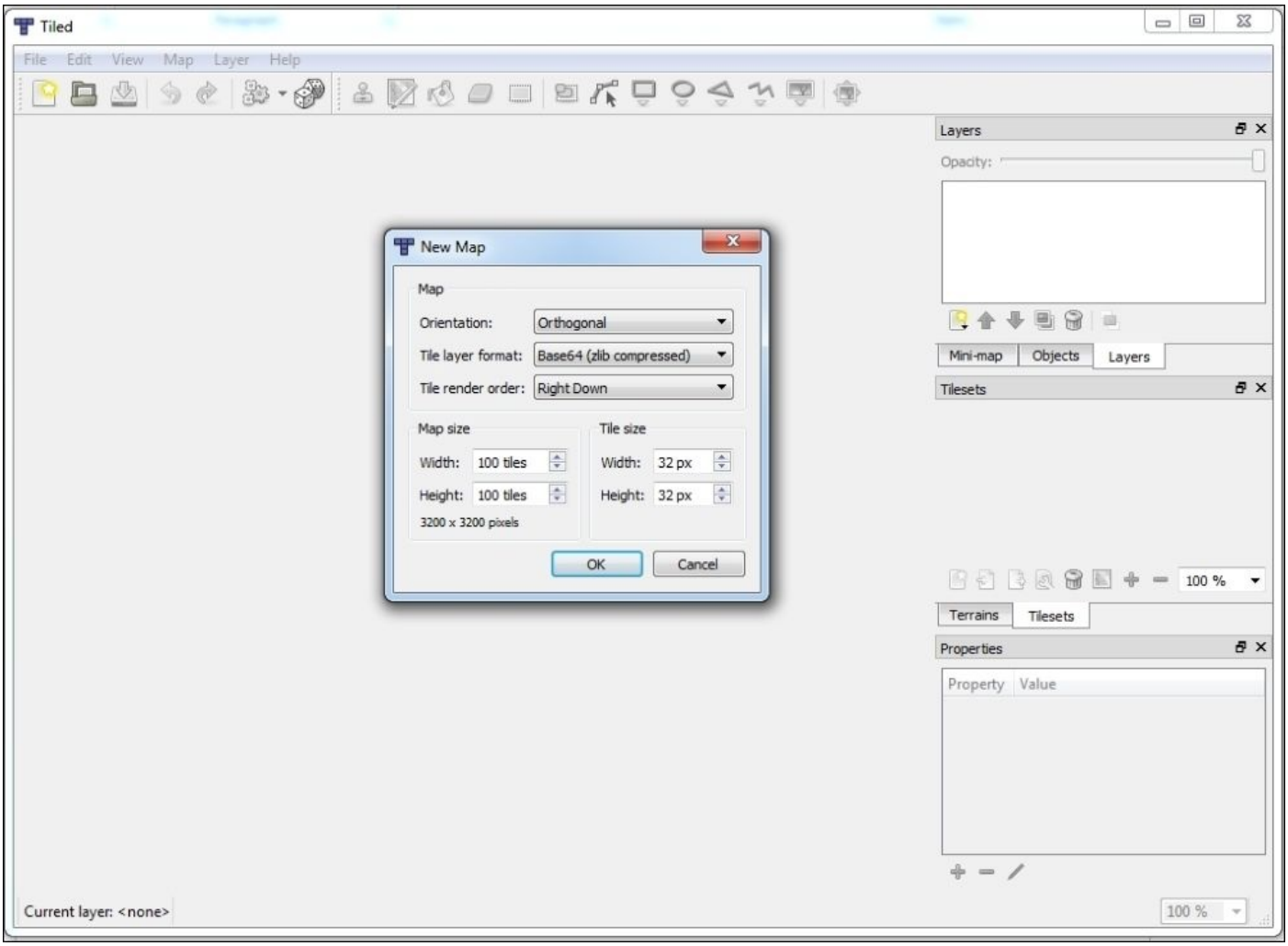
To install the Tiled map editor, navigate to http://www.mapeditor.org/download.html and download the version of Tiled according to your OS:



Install the software and open it. At the time of writing, I am using the version 0.11.0. When you open it, you will be presented with an interface similar to this:



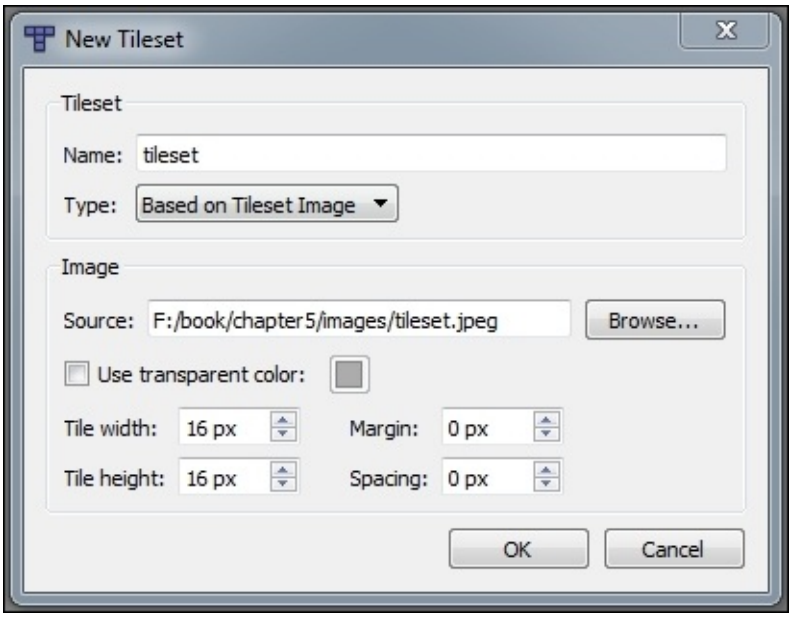
To create a new map, go to the **File** | **New** option. You will be presented with a menu to set some of the basic map properties, as shown in the following screenshot:



In the **Orientation** option, select **Orthogonal**, as this is what we are going to use throughout this book and is mostly used for 2D games. You can consider it a rectangular grid view. The other options for this are **Isometric** (for a 3D-like view), **Isometric (Staggered)**, and **Hexagonal (Staggered)**. Keep the default values for the **Tile layer format** and **Tile render order** options.

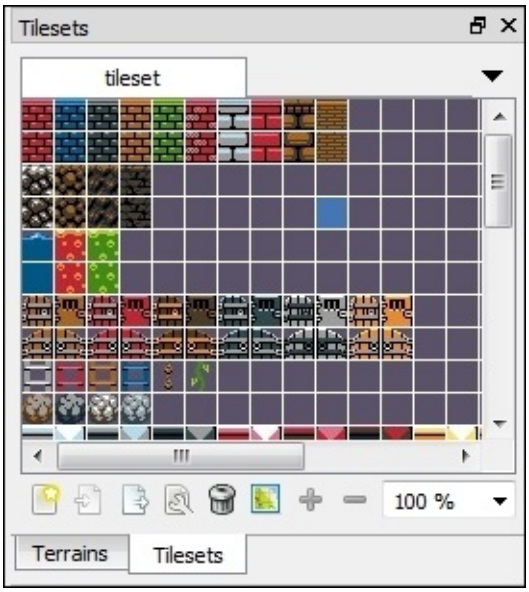
Coming to the **Tile size** section; you can specify the height and width for an individual tile (unit) for your map in pixels. Keep this 16 x 16 for now. The **Map size** section specifies how big your map is in terms of tiles. By multiplying this with the tile size, you will get the actual size of your map in pixels. We will keep the width to 50 tiles and the height to 20 tiles.

Click on **OK**. The next thing that we need is a tileset. A tileset is an image that is made up of different images called tiles. This is what you need in order to create the maps. Click on **Map** in the menu and then click on **New Tileset**:



Name it `tileset` and select the image for which you want to use it. We will use a tileset, which looks like the following screenshot:

Keep the other options as they are and click on the **OK** button. The tilesets will be loaded in the editor after this step. You can see them in the middle pane, also called the **Tilesets** pane, on the right-hand side of the editor:

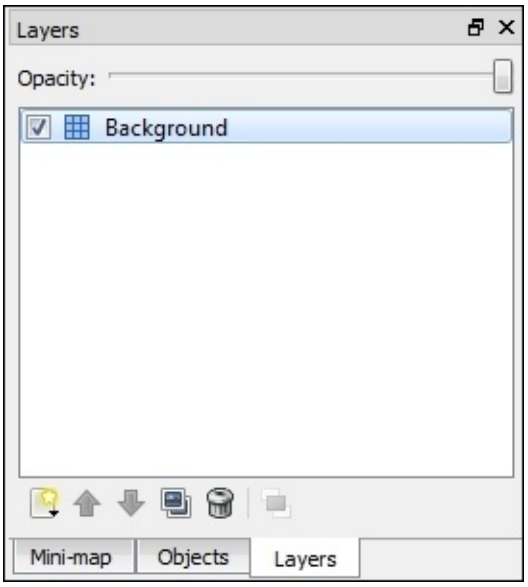


Tiled automatically divides the tiles from the tileset image according to the configuration. As we set the size to 16 x 16, the image is divided into individual tile images of sizes 16 x 16. You can add multiple tilesets as well.

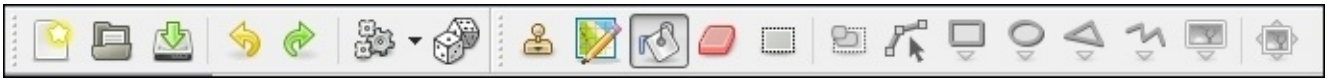
Map layers and drawing

Before we draw anything, let’s first go to map layers. In Tiled, you can create a map with multiple layers, which are stacked on top of one another. For example, the layers can be backgrounds, walls, collectibles, and so on.

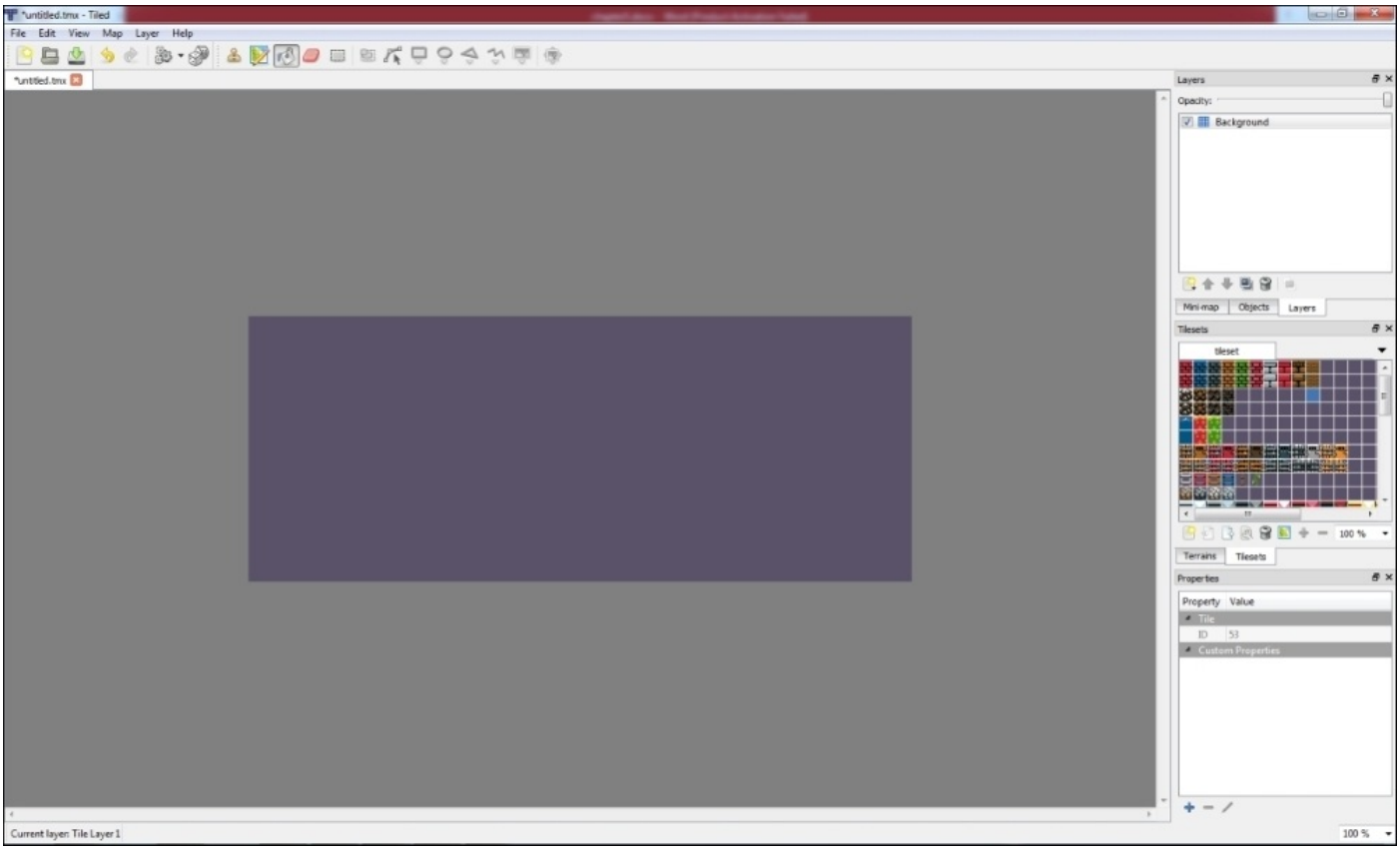
Structuring the map in this way makes it easier to parse and apply rules. For example, while parsing the map, you can ignore the collision between the player and the background layer. You can also handle the collectibles’ layer separately if you want. By default, a layer is created when you create a new map. Double-click on **Tile Layer 1** to rename it. Change the name to `Background`:



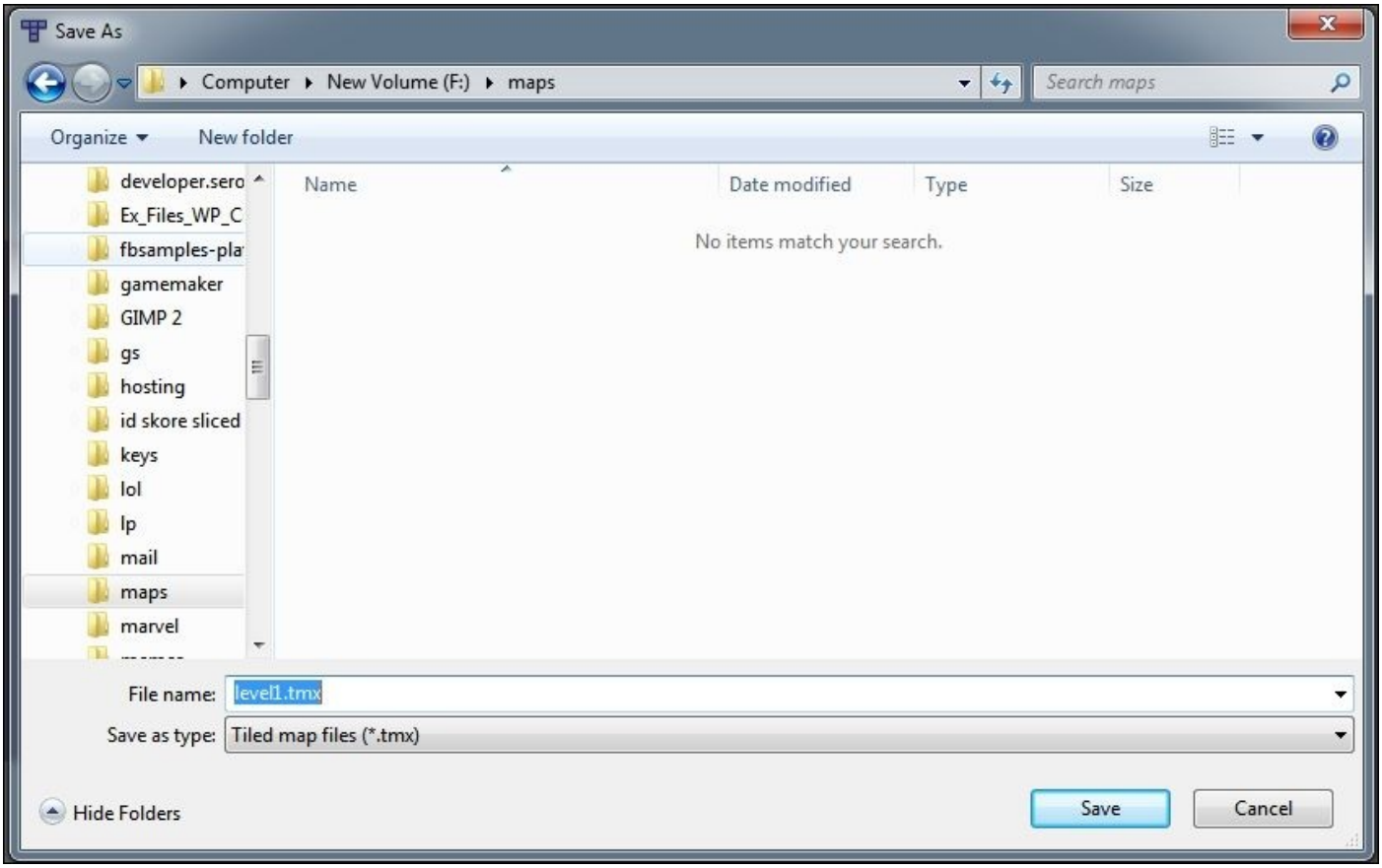
Let’s now fill the **Background** layer with the background tile. Select the purple tile from the tileset and select the bucket icon from the tool menu:



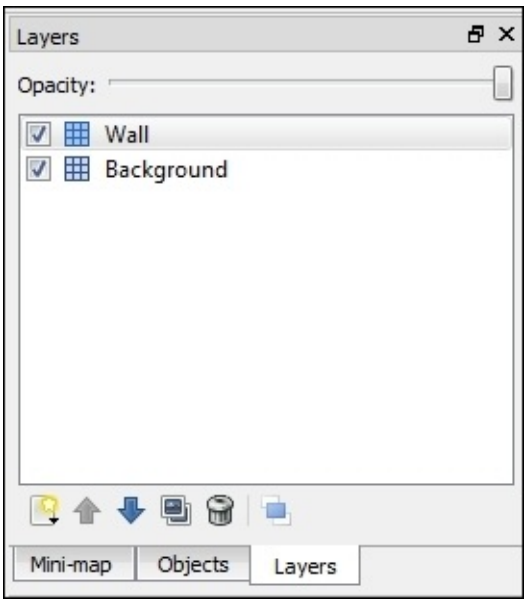
This is used to fill the map completely with one tile or used to fill an enclosed area with the specified tile. When you move the cursor to the map area, you can see the map being highlighted. Click there to apply the tile to the whole map:



Save the map by going to **File | Save**. In the **Save As** window, give a name to the file you want and don't forget to save the map in the same folder as that of your tileset:

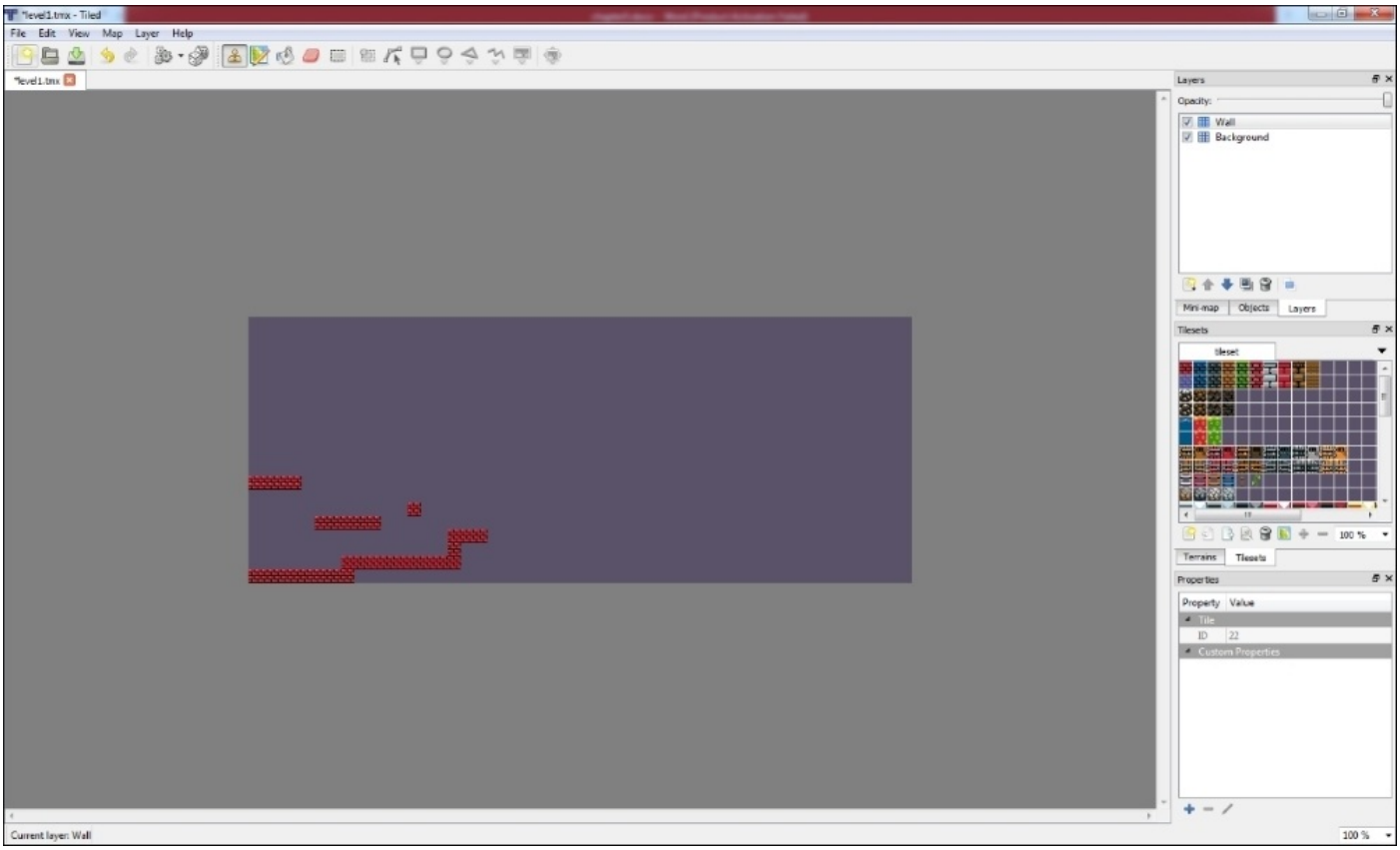


Next, we will draw some walls. Create a new layer by navigating to **Layer** | **Add Tile Layer** in the menu bar. You will notice that a new layer called **Tile layer 2** has been added to the **Layers** pane. Change the name to `Wall`. You can change the order of the layers by using the arrow symbols below the **Layers** pane:

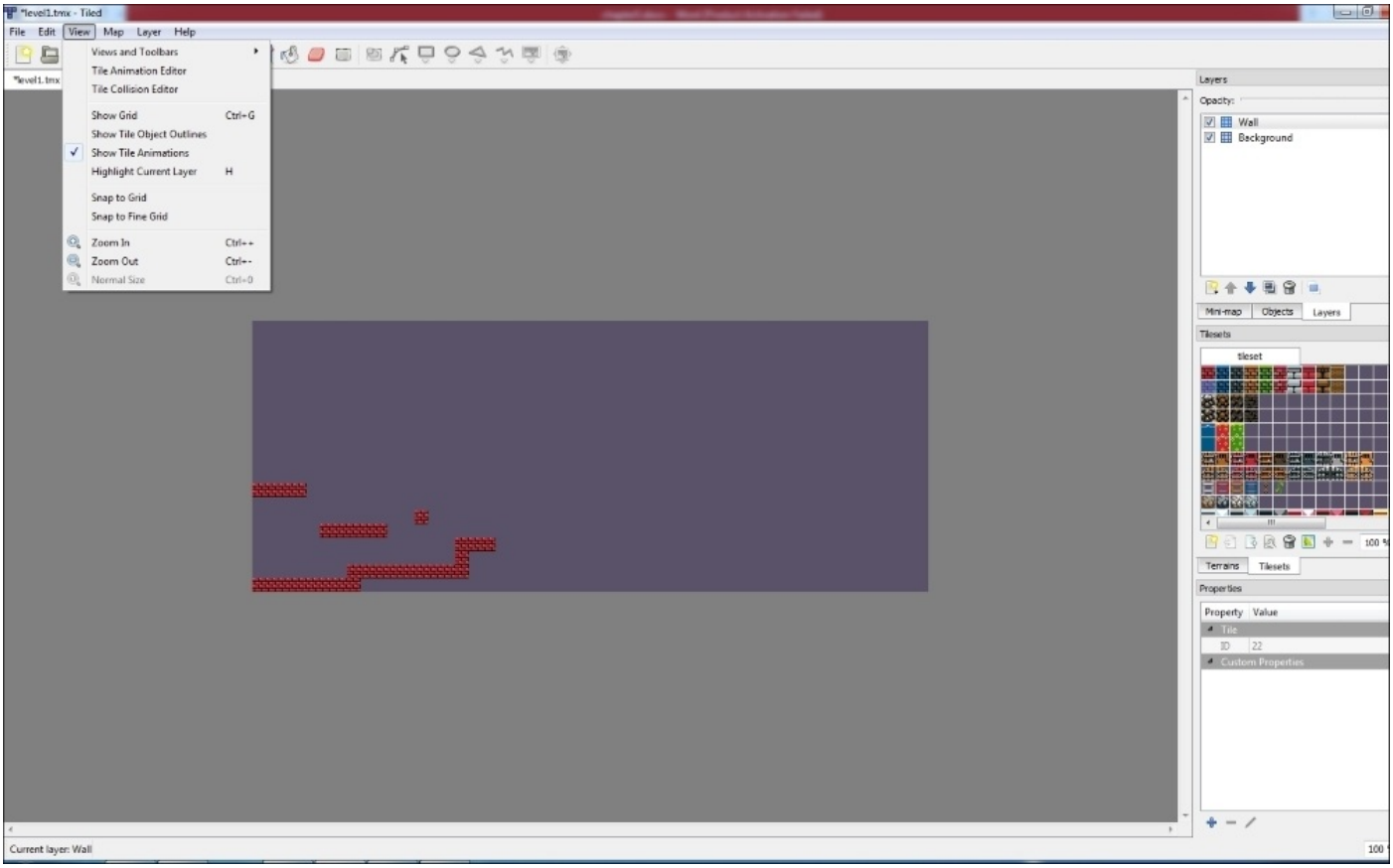


Select a wall tile from the **Tilesets** pane and select the stamp brush icon from the top menu:

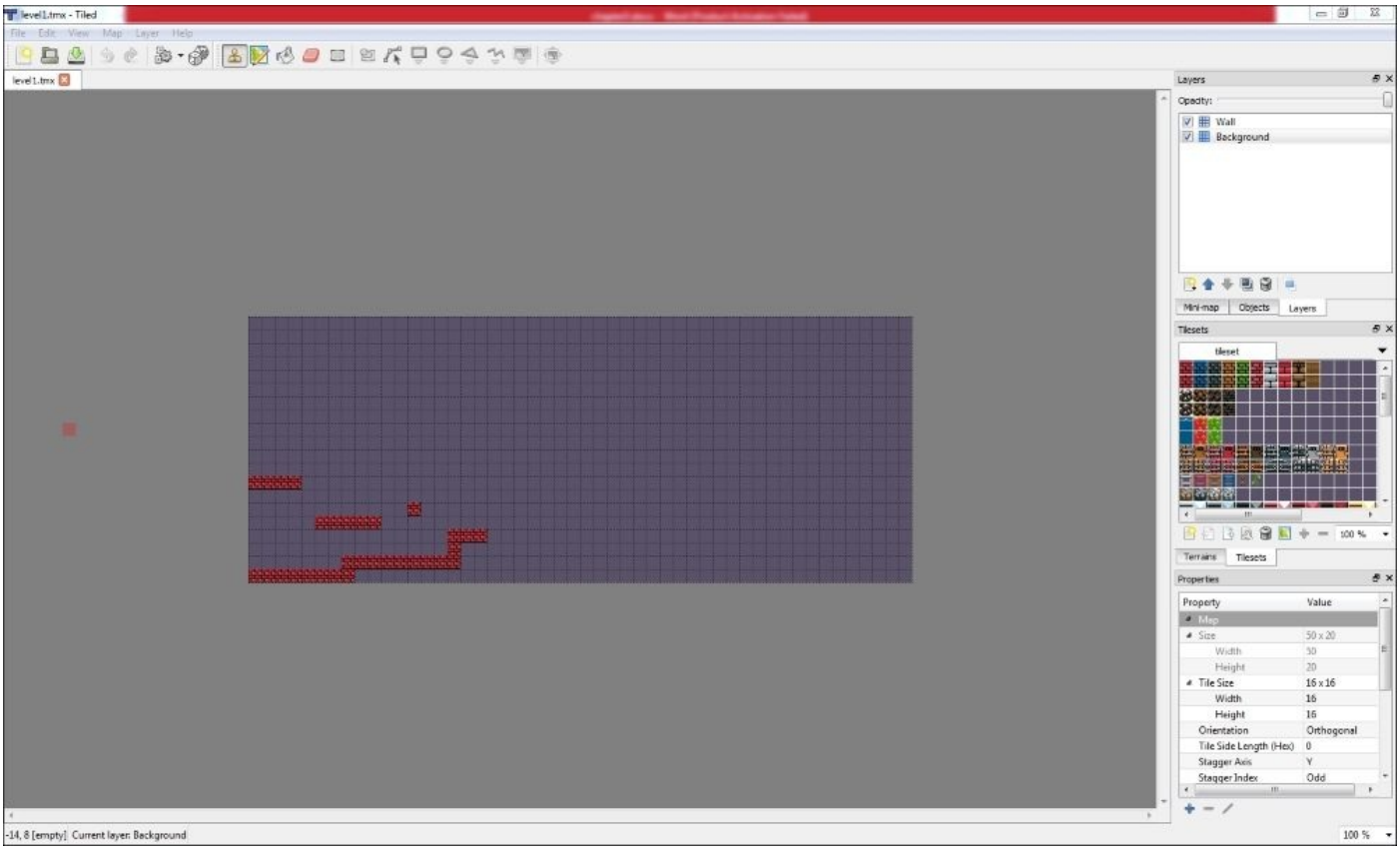
The stamp tool allows you to paint the map per tile or use it as a set of tiles. You can now paint the map with the walls you want:



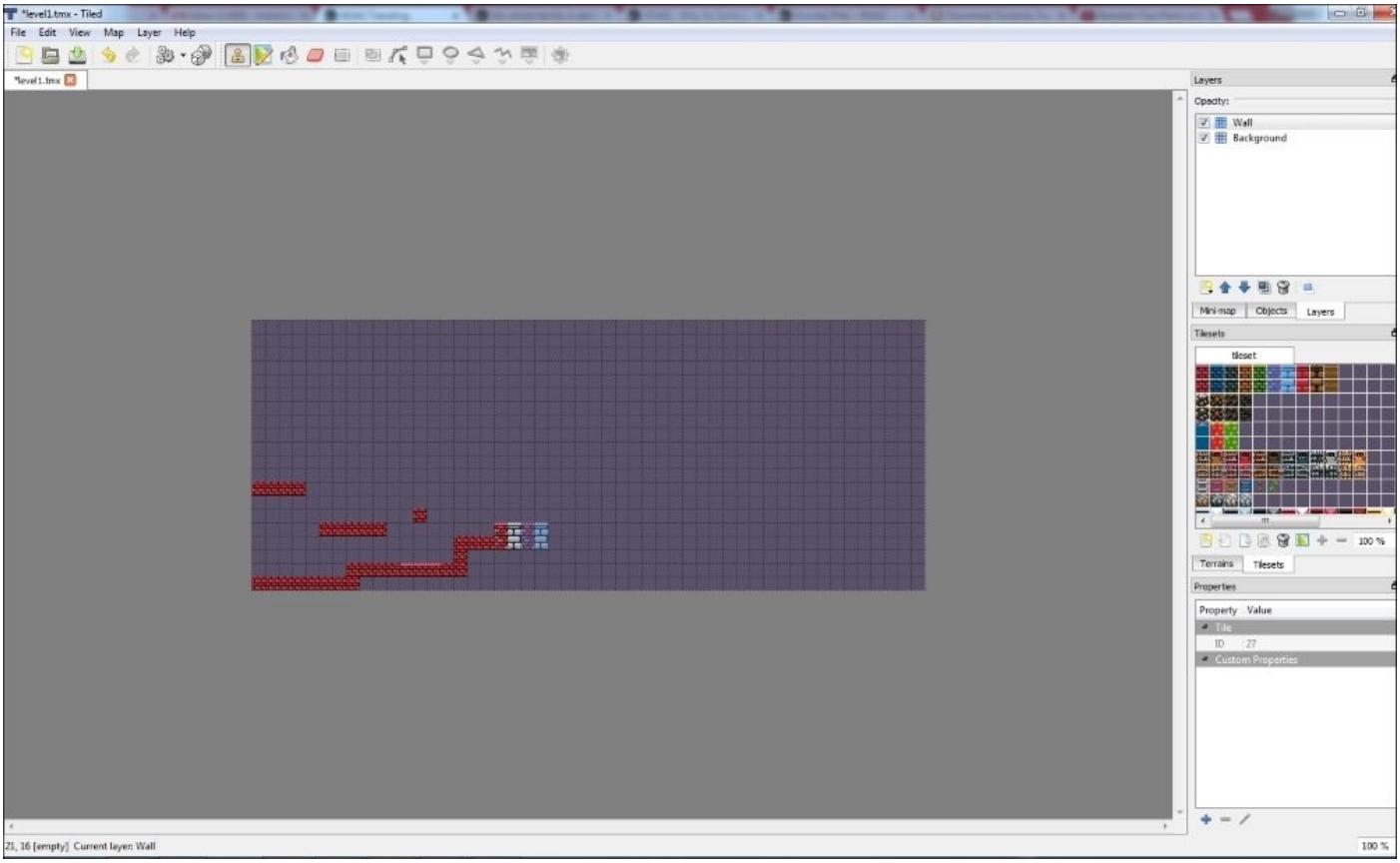
To paint it more precisely, you can turn the grid on. It will show the tile boundaries on the map. Go to **View** and select the **Show Grid** option:



After selecting **View** and the **Show Grid** option, you will get the following view:

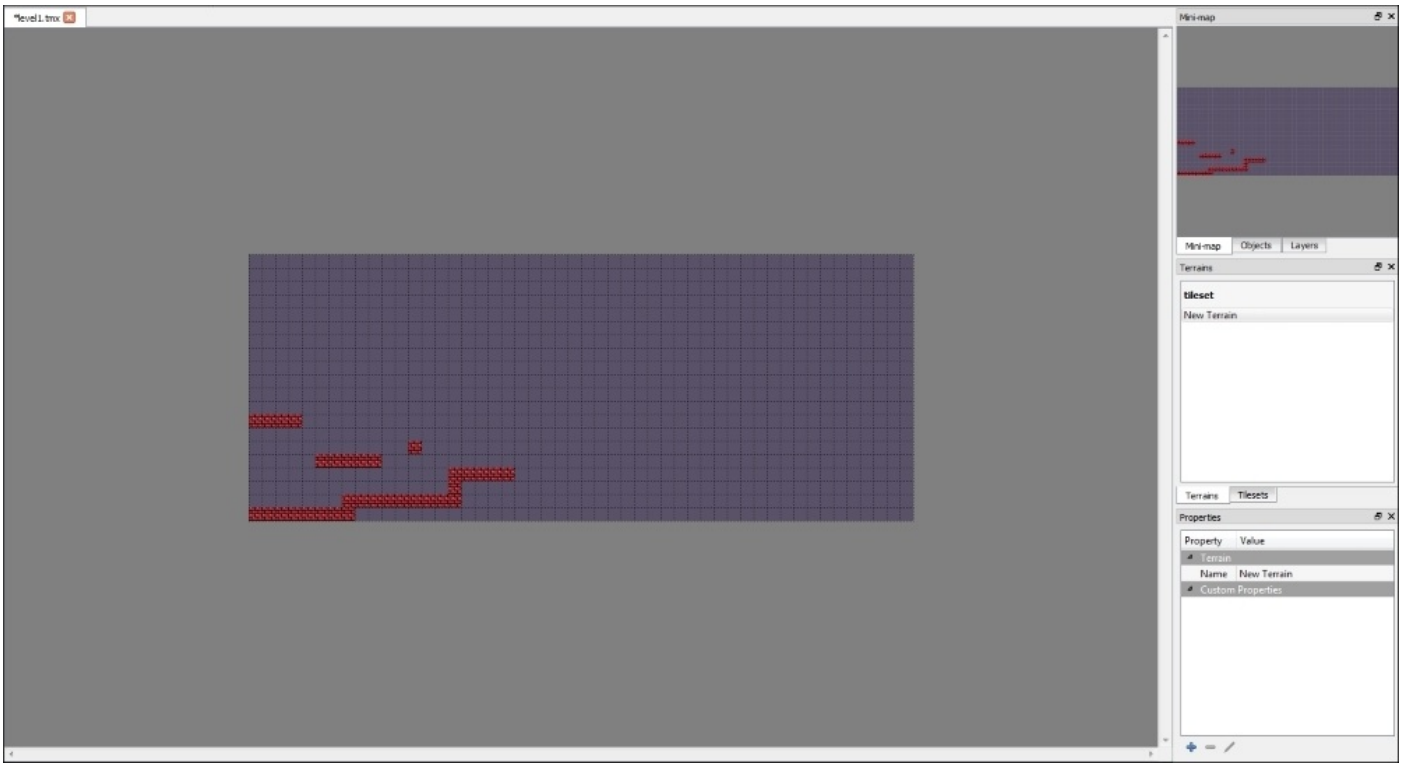


To select multiple tiles, hold the *Ctrl* key on your keyboard and then select the tiles. You can then paint the map with them:

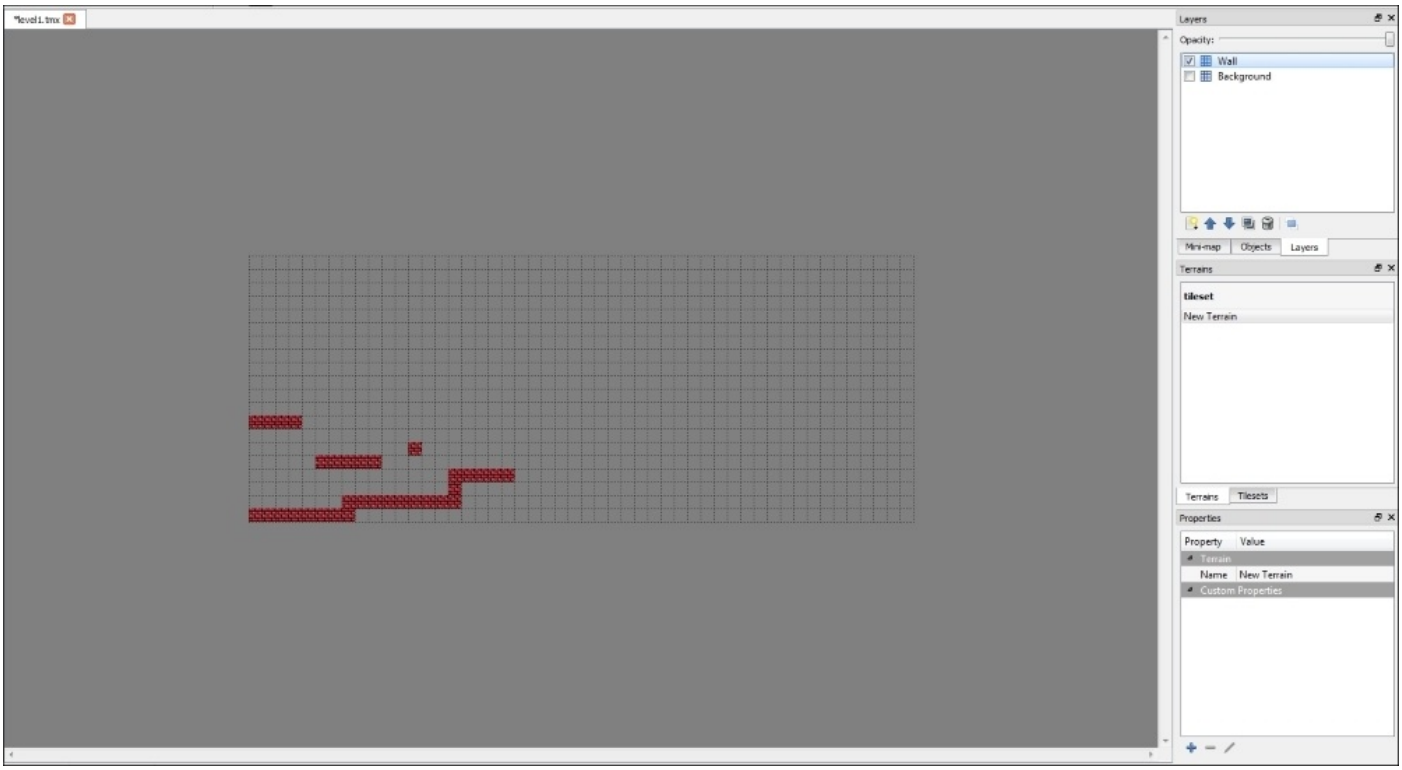


To erase anything that you painted on the map, select the eraser icon from the toolbar menu:

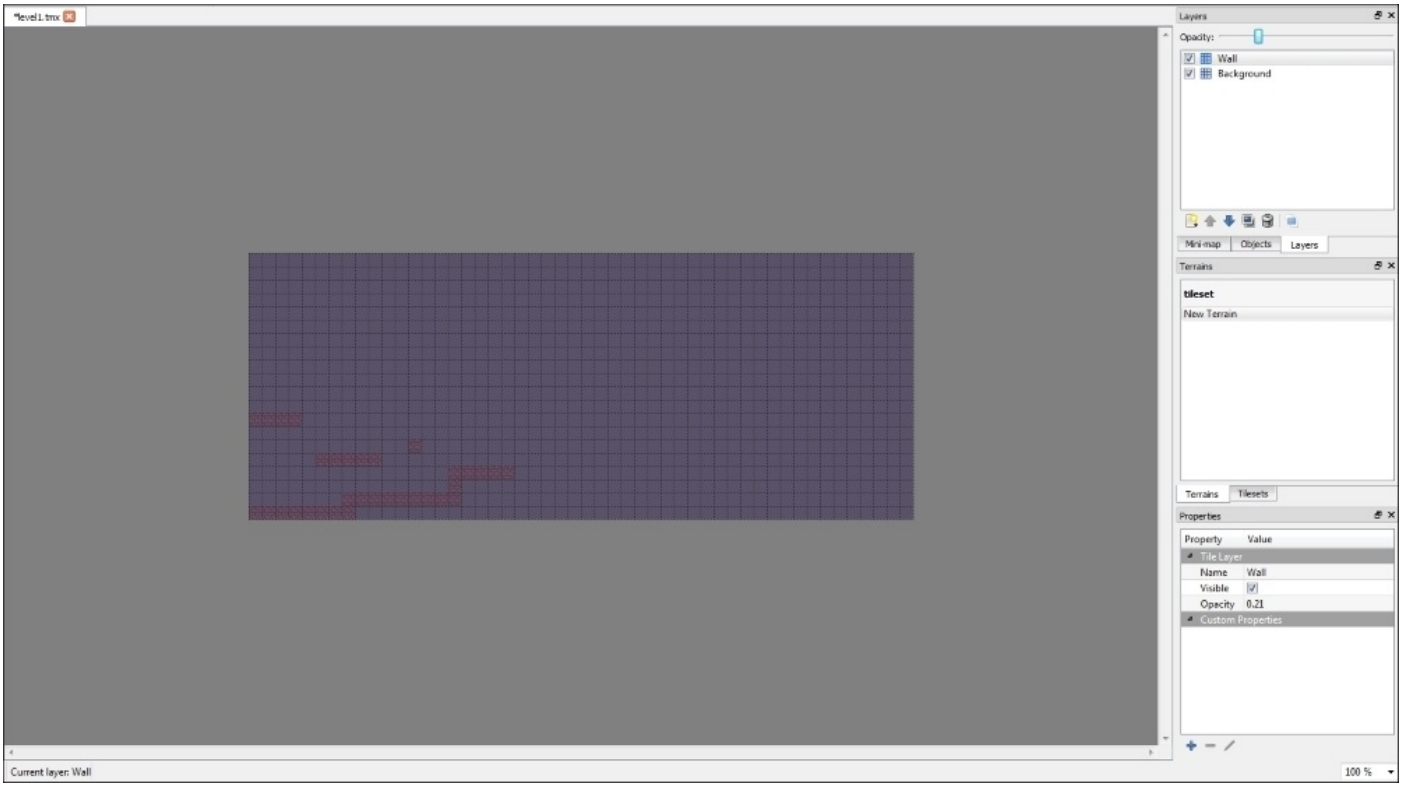
Using this, you can remove the tiles that you have painted. If your map is huge and if you want to see a miniature version of it, there is a minimap feature available. To view it, click on the **Mini-map** tab below the **Layers** window:



To show/hide layers, click on the checkbox next to their names:



You can also change the opacity of a layer by selecting it and moving the slider at the top of the **Layers** pane:

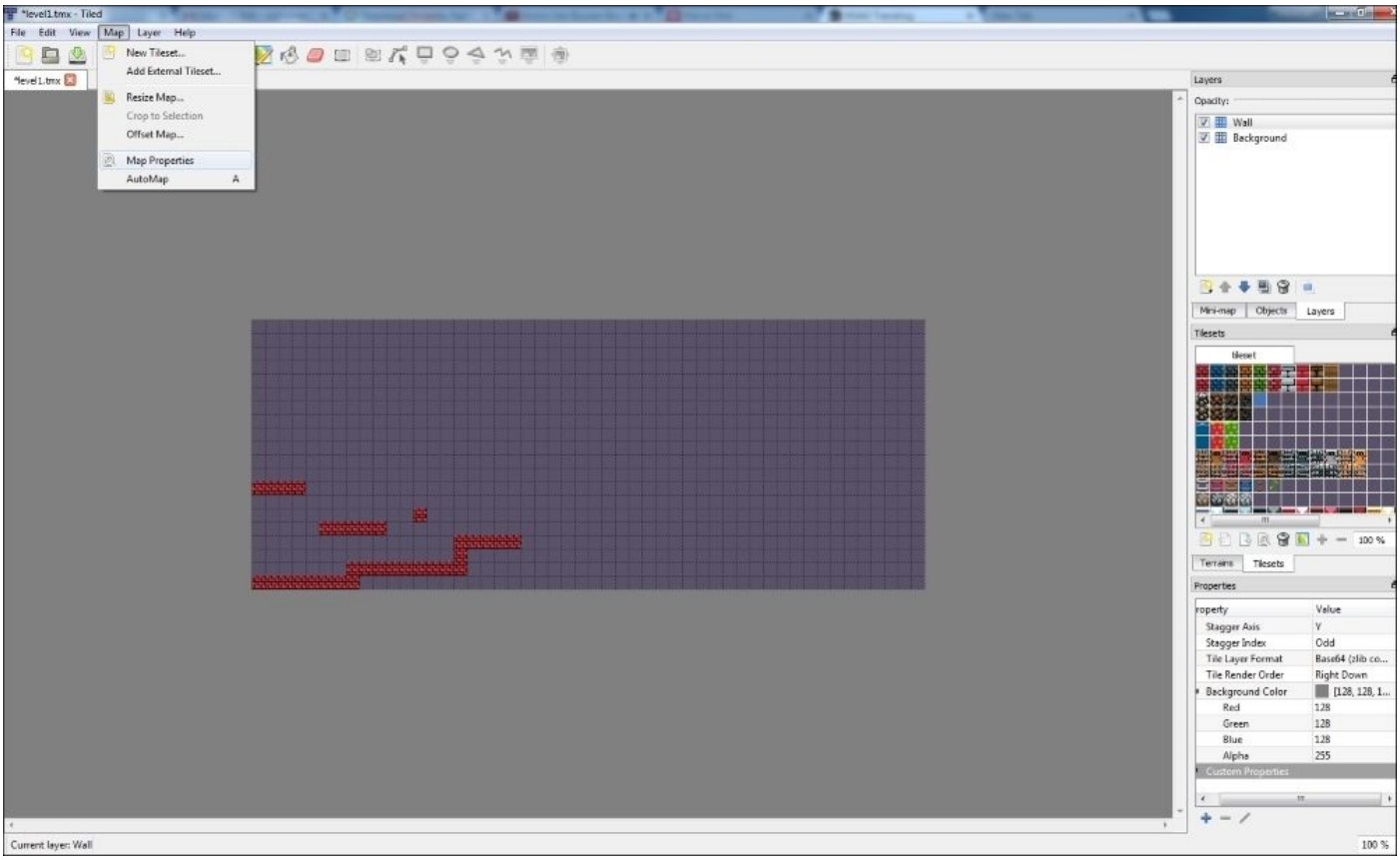


Miscellaneous

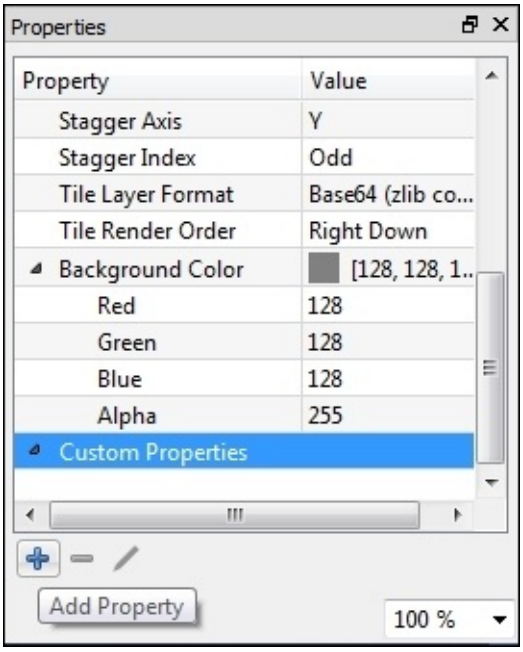
This topic will cover how to add properties to the map components and how to add objects and tile animations to the map.

Custom properties

You can add custom properties to your map, which you can then use in your game while rendering. To add them, go to **Map** | **Map Properties**:

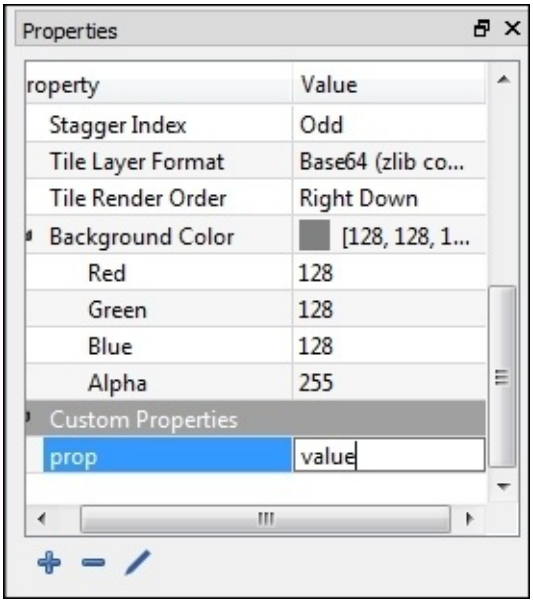


In the **Properties** pane, click on the + icon to add a new property:

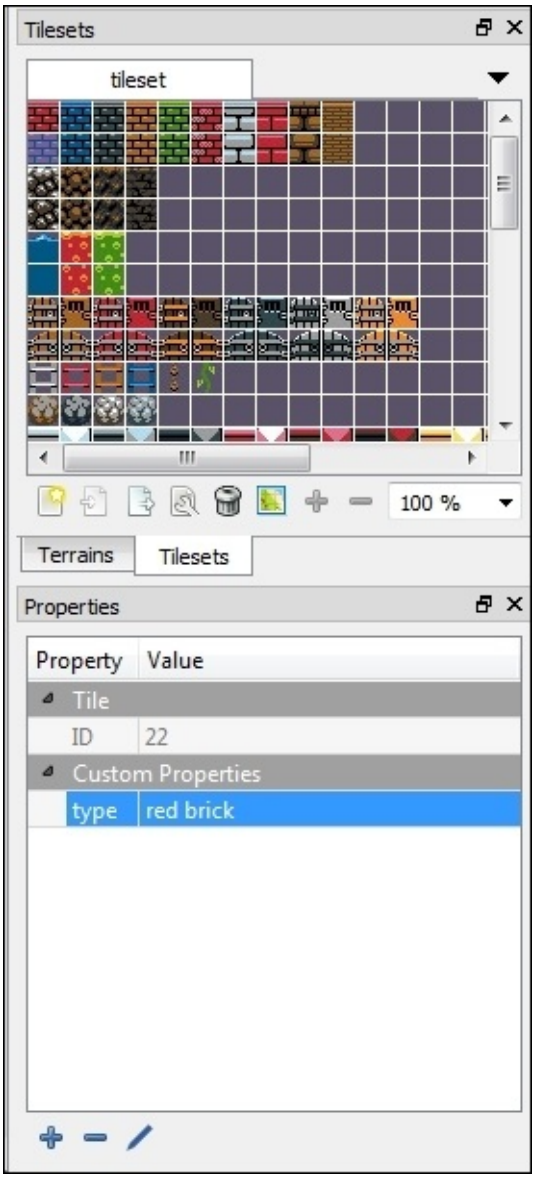


A new dialog box will open, where you can give the property name:

After you give the property name and click on **OK**, the property appears in the **Custom Properties** section of the **Properties** pane, where you can give its value:

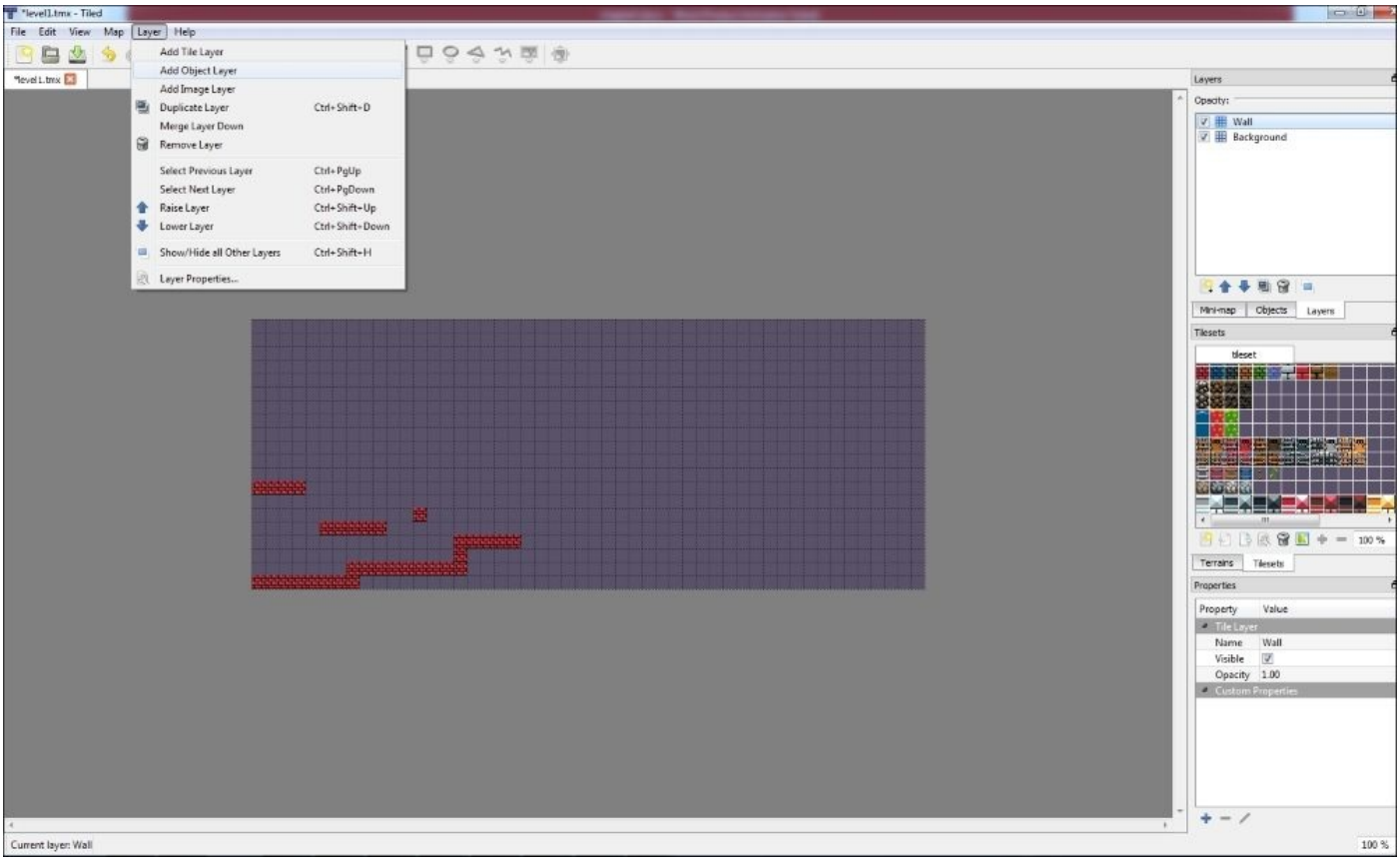


Similarly, you can add properties to an individual tile/layer. You need to select them to add a new property:

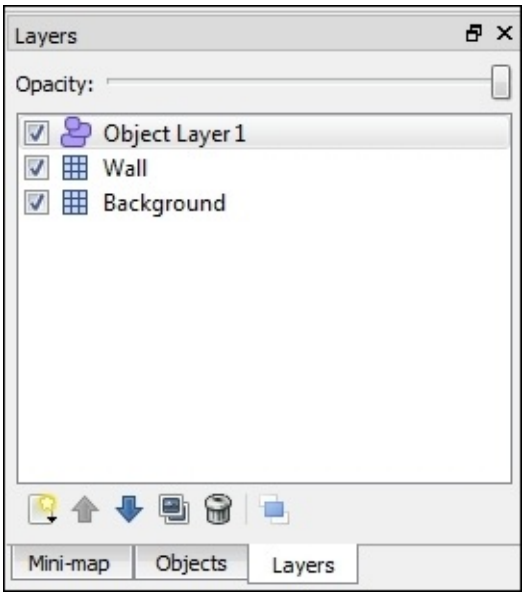

Tilesets
tileset
100 %
Terrains
Tilesets
Properties
Property
Value
Tile
ID
22
Custom Properties
type
red brick

Drawing objects

Sometimes, simple tiles may not satisfy your requirements. You might need to create objects with complex shapes. You can easily define these shape outlines in the editor. The first thing you need to do is create an object layer. Navigate to **Layer** | **Add Object Layer**:

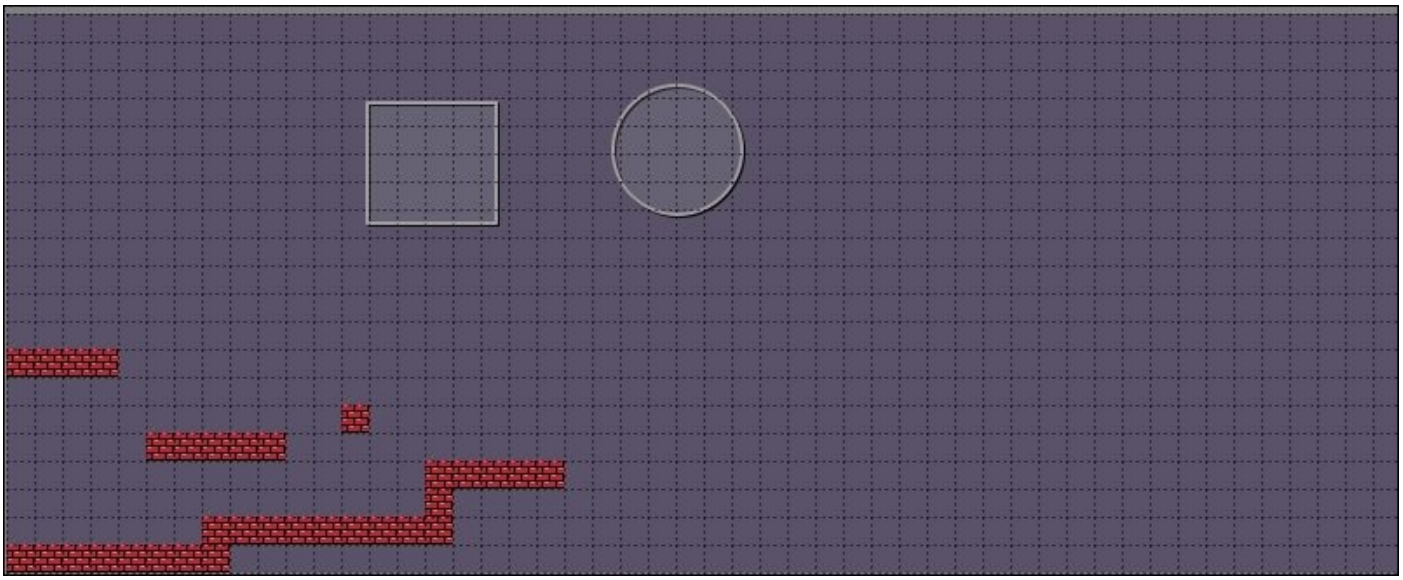


You will notice that a new layer has been added to the **Layers** pane called **Object Layer 1**. You can rename it if you like:

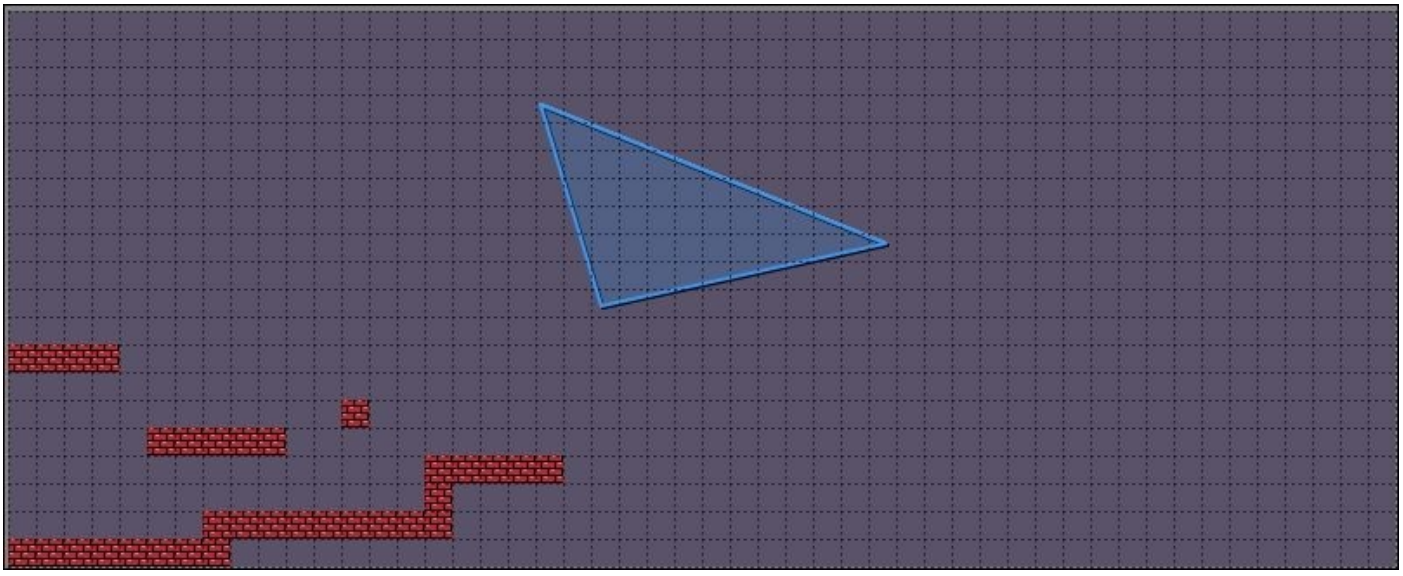


With this layer selected, you can see the object toolbar getting enabled:

You can draw basic shapes, such as a rectangle or an ellipse/circle:

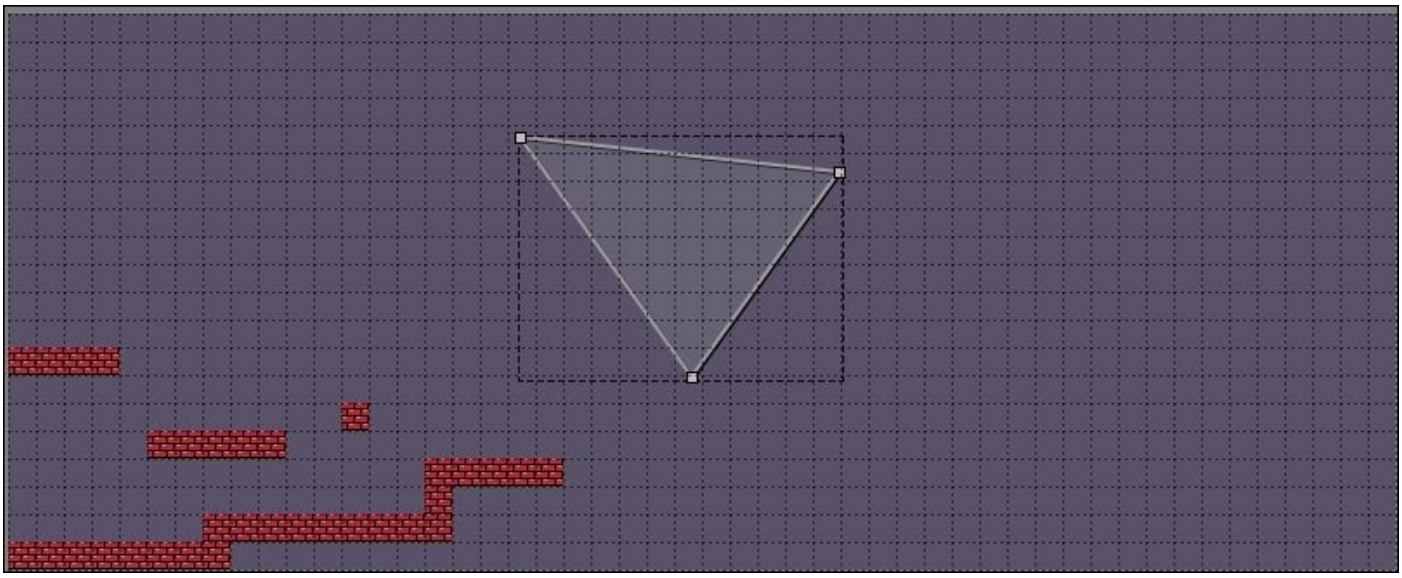


You can also draw a polygon and a polyline by selecting the appropriate options from the toolbar. Once you have added all the edges, click on the right mouse button to stop drawing the current object:

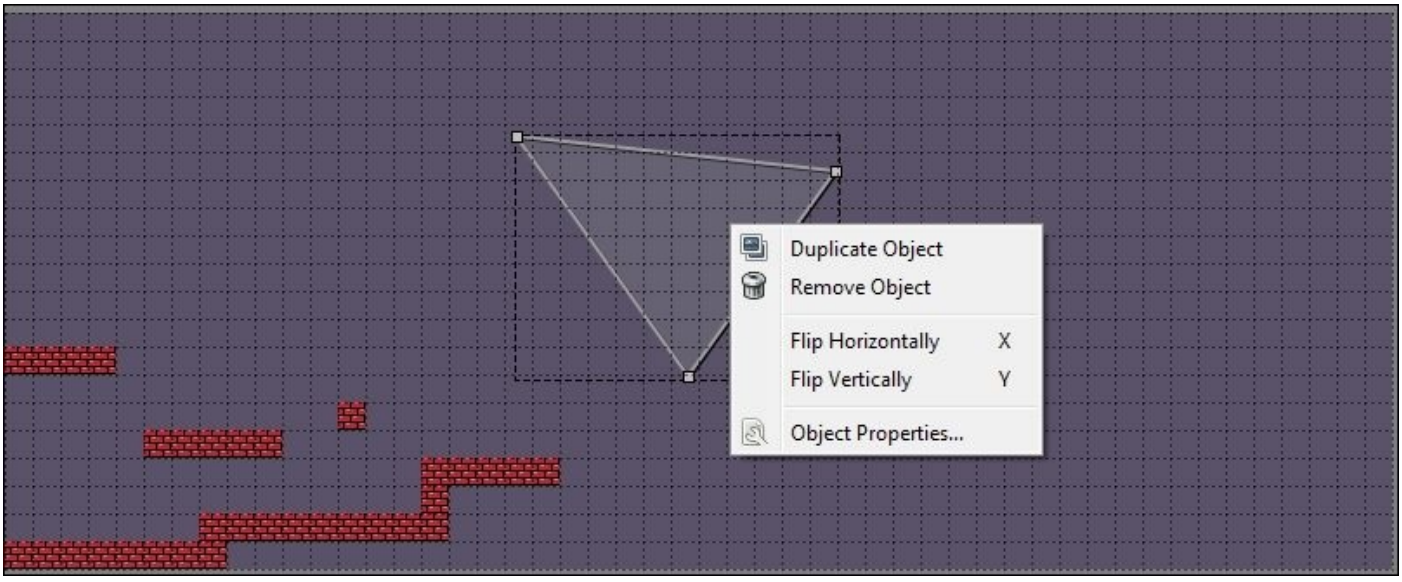


Once the polygon/polyline has been drawn, you can edit it by selecting the Edit Polygons option from the toolbar:

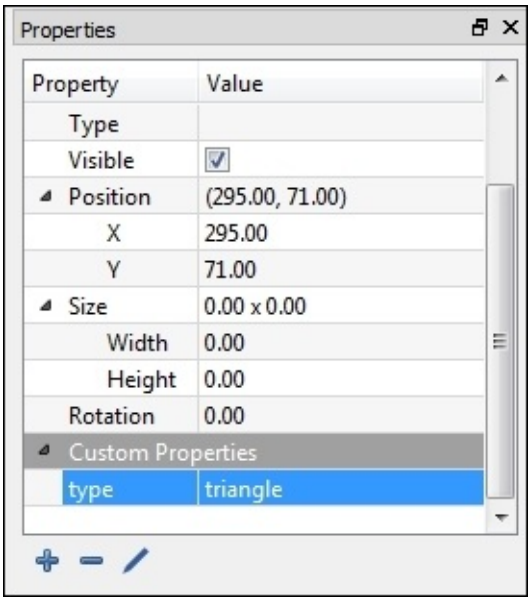
After this, select the area that encompasses your polygon in order to change it to the edit mode. You can edit your polygons/polylines now:



You can also add custom properties to your polygons by right-clicking on them and selecting **Object Properties**:

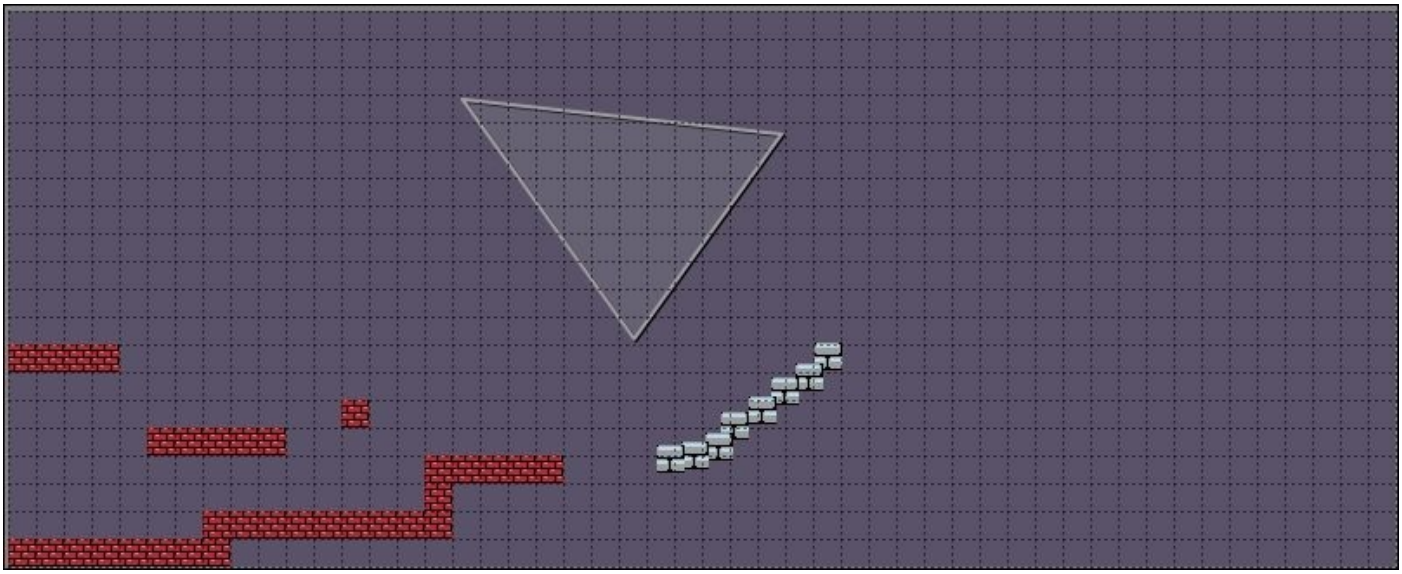


You can then add custom properties, as mentioned previously:



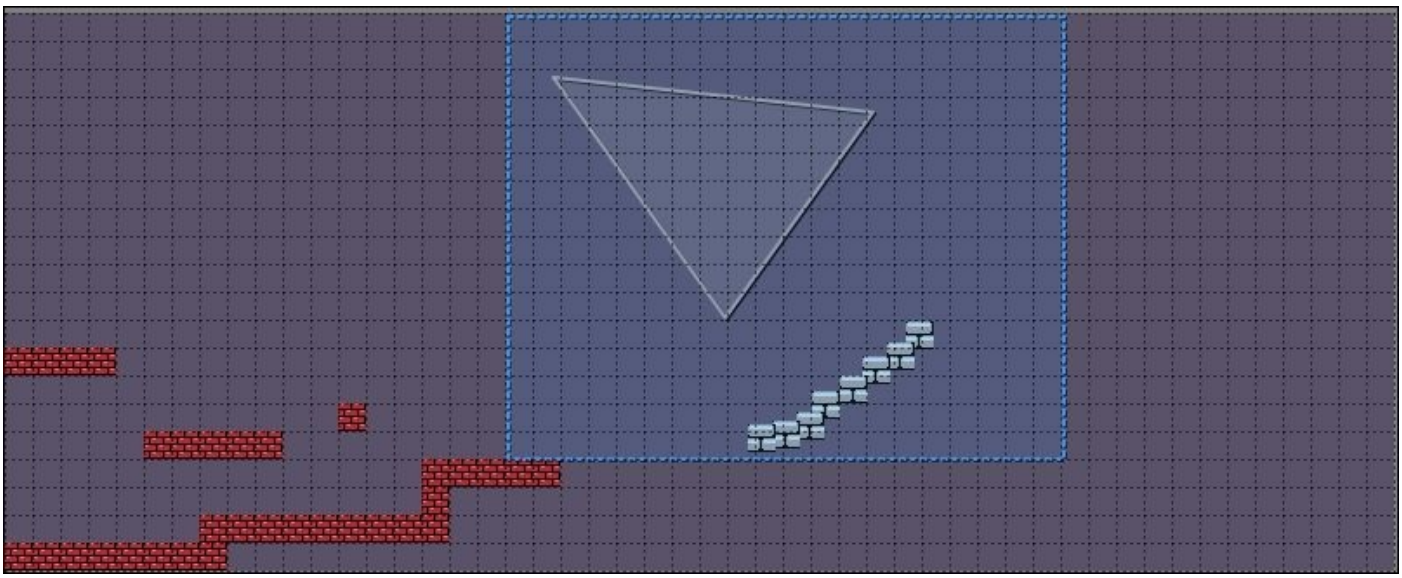
You can also add tiles as an object. Click on the Insert Tile icon in the toolbar:

Once you have selected this icon, you can insert tiles as objects into the map. You will observe that the tiles can be placed anywhere now, irrespective of the grid boundaries:

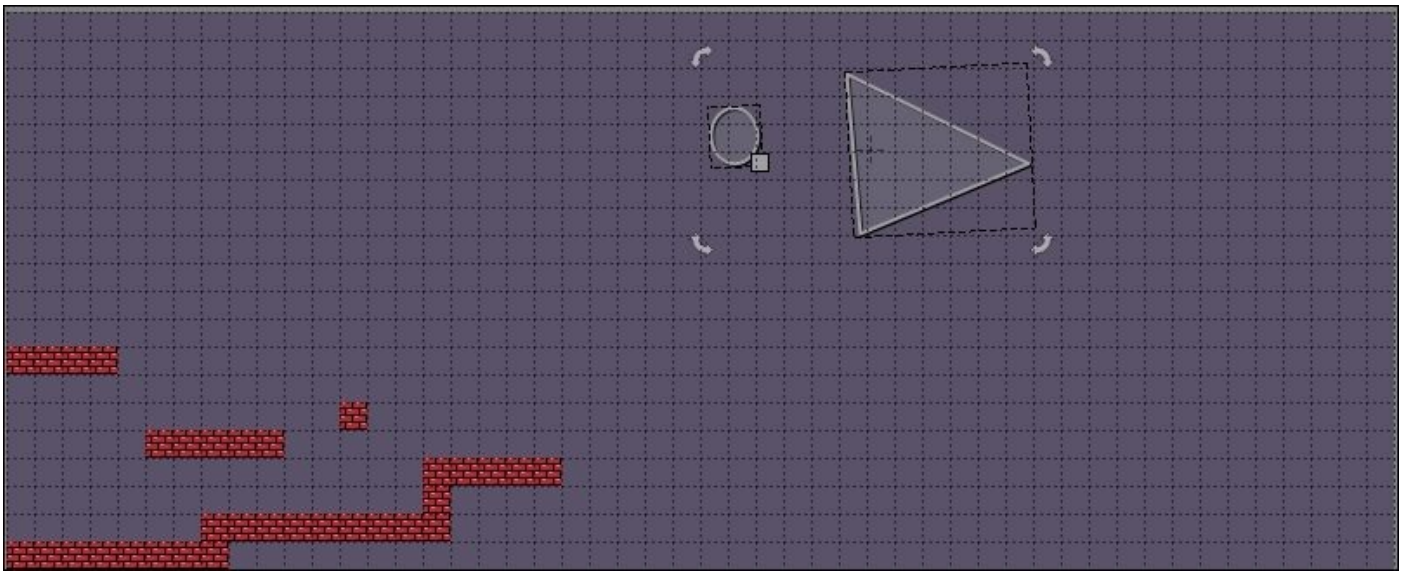


To select and move multiple objects, you can select the Select Objects option from the toolbar:

You can then select the area that encompasses the objects. Once they have been selected, you can move them by dragging them with your mouse cursor:



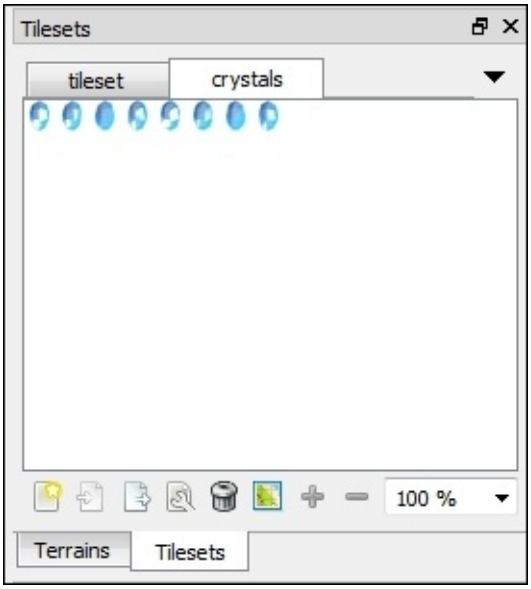
You can also rotate the object by dragging the indicators at the corners after they have been selected:



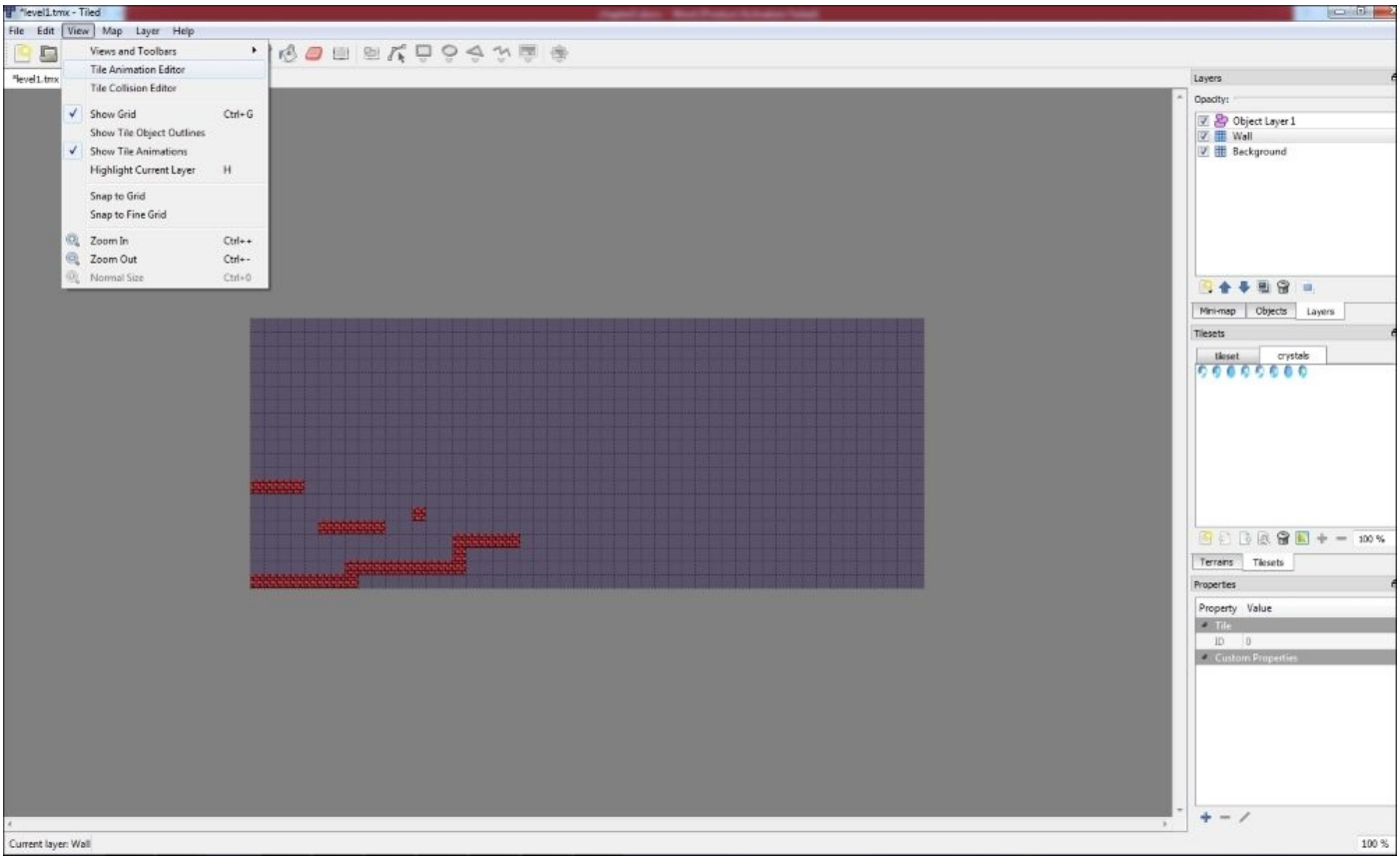
Tile animations and images

Tiled allows you to create animations in the editor. Let’s make an animated shining crystal. First, we will need an animation sheet of the crystal. I am using this one, which is 16 x 16 pixels per crystal:

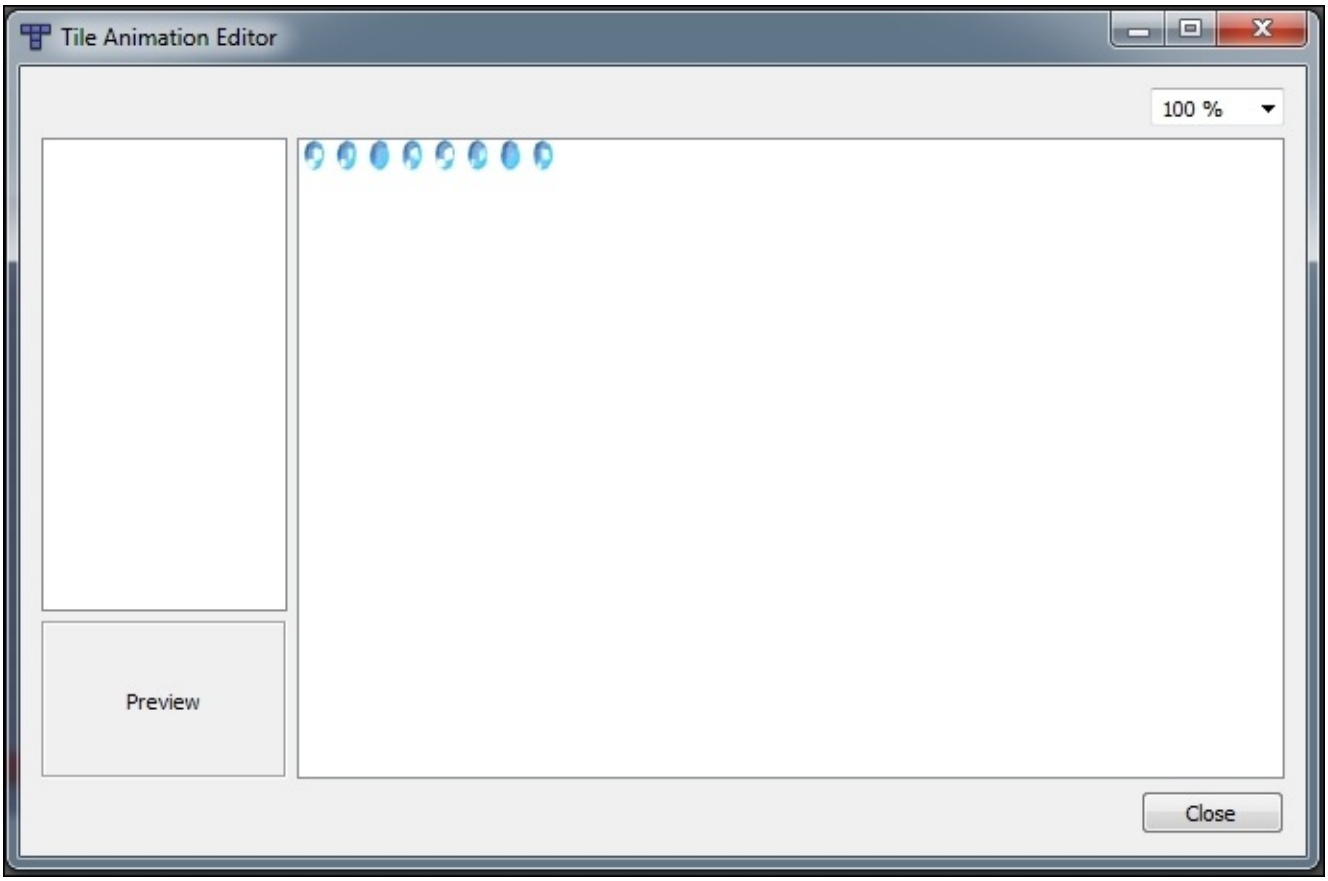
The next thing we need to do is add this sheet as a tileset to the editor and name it `crystals`. To do this, follow the same steps, as mentioned earlier. After you add the tileset, you will see a new tab in the **Tilesets** pane:



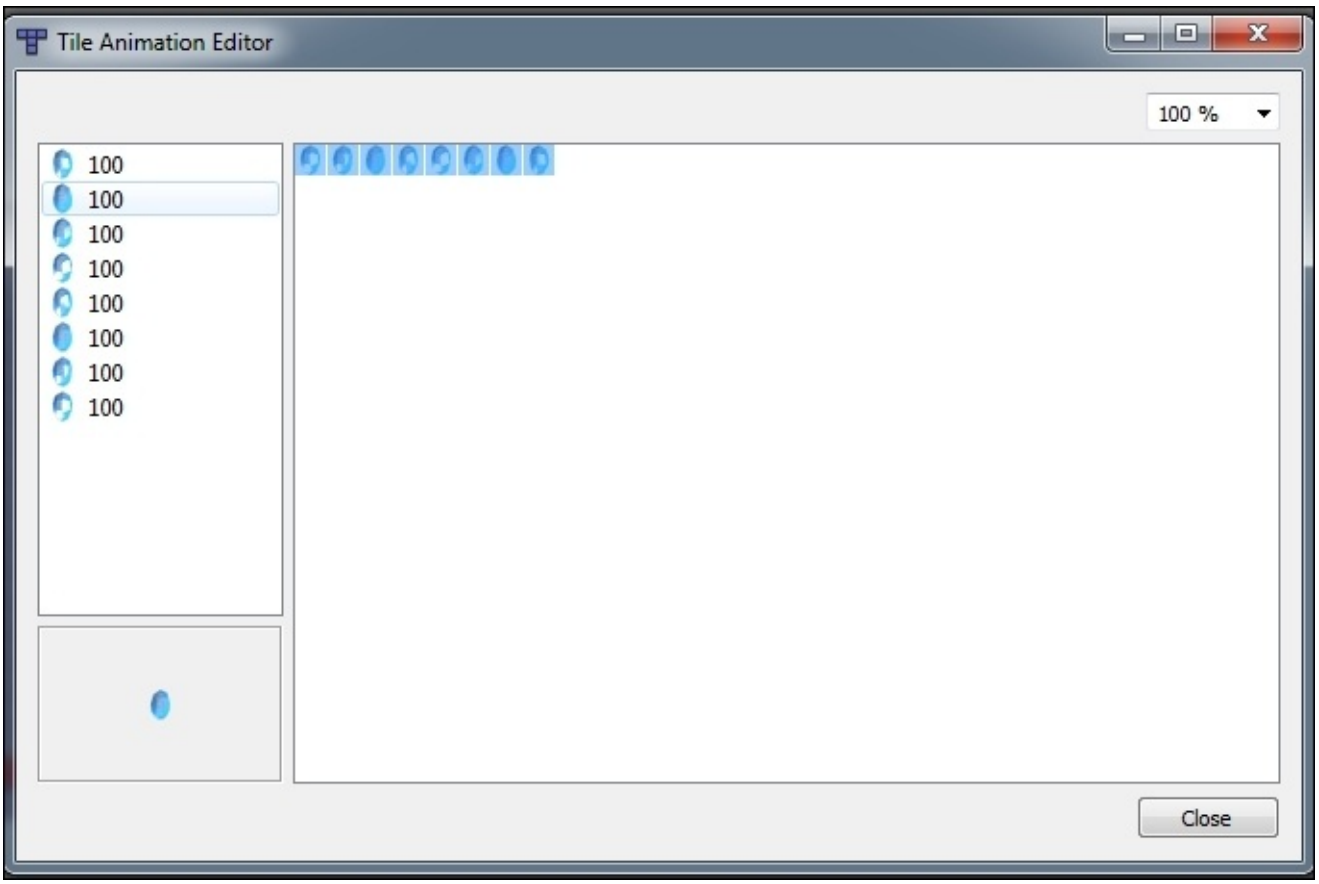
Navigate to **View** | **Tile Animation Editor** to open the animation editor:



A new window will open that will allow you to edit the animations:

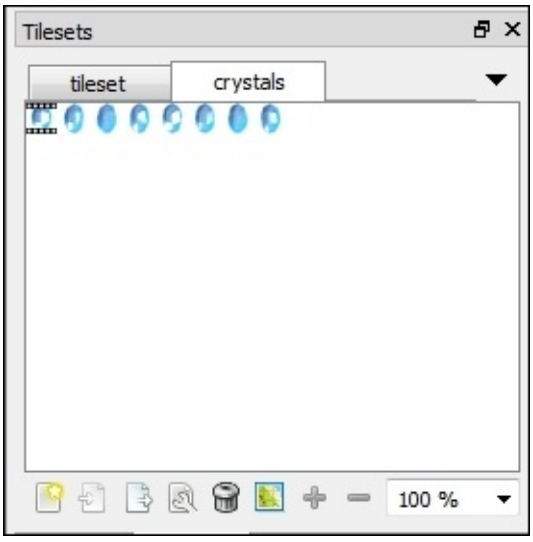


On the right-hand side of the window, you will see the individual animation frames that make up the animation. This is the animation tileset, which we added. Hold the *Ctrl* key on your keyboard and select all of them with your mouse. Then, drag them to the left-hand side of the window:

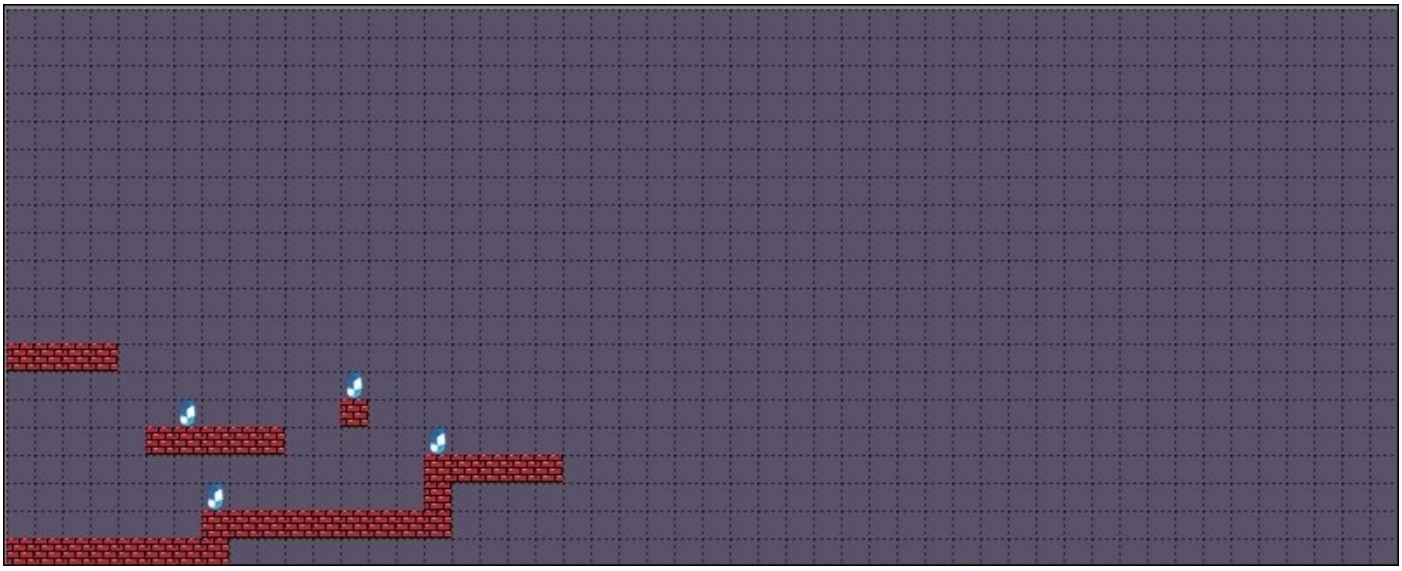


The numbers beside the images indicate the amount of time each image will be displayed in milliseconds. The images are displayed in this order and repeated continuously. In this example, every image will be shown for 100 ms or one-tenth of a second. In the bottom-left corner, you can preview the animation you just created. Click on the **Close** button.

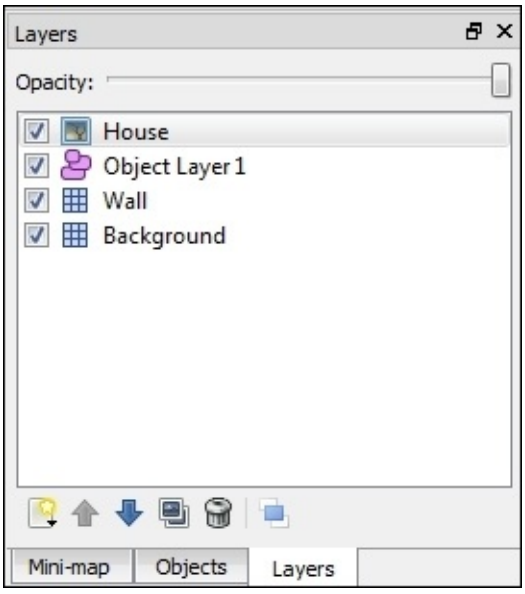
You can now see something like this in the **Tilesets** pane:



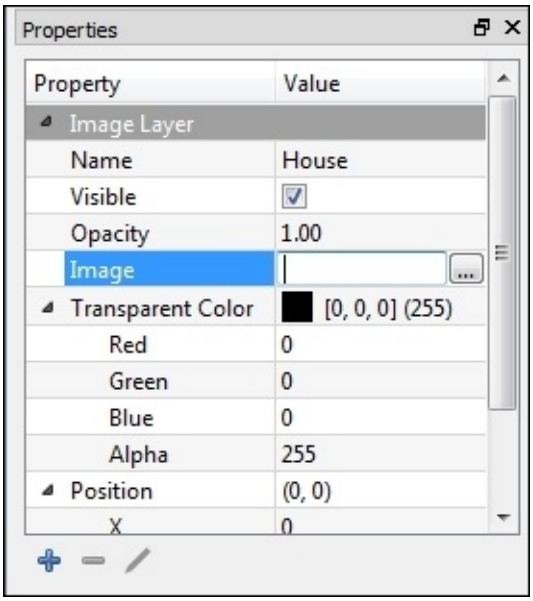
The first tile represents the animation that we just created. Select it and you can draw the animation anywhere in the map. You can see the animation playing within the map:



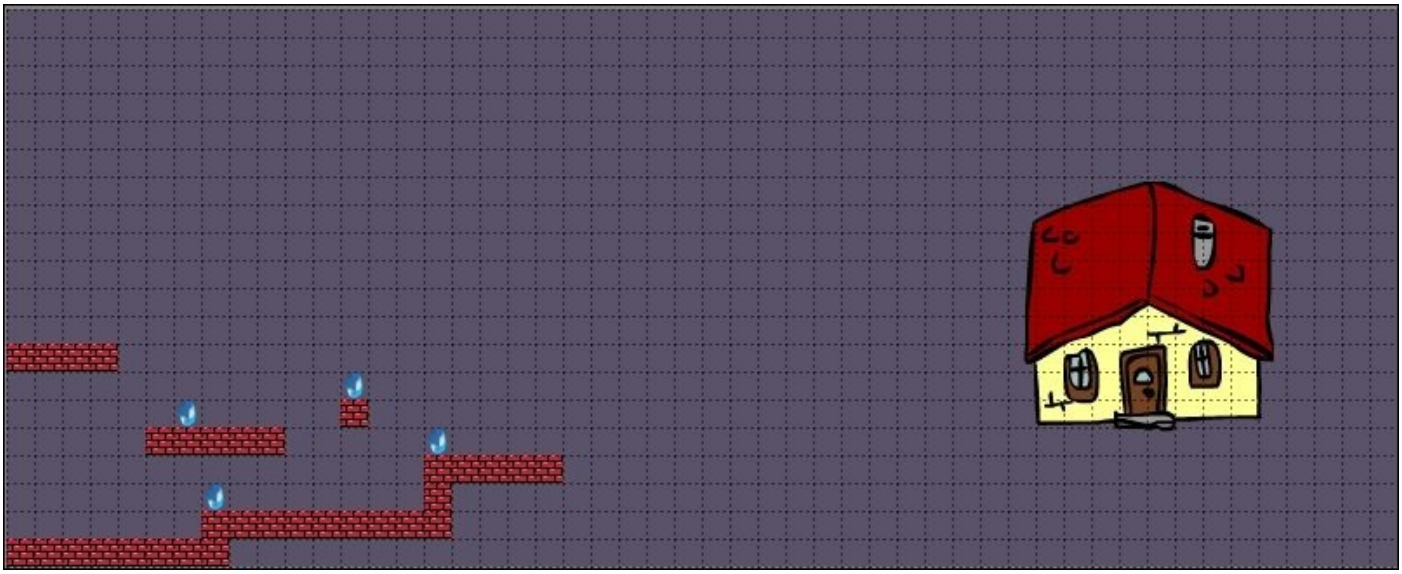
Lastly, we can also add images to our map. To use them, we need to add an image layer to our map. Navigate to **Layer** | **Add Image Layer**. You will notice that a new layer has been added to the **Layers** pane. Rename it `House`:



To use an image, we need to set the image's path as a property for this layer. In the **Properties** pane, you will find a property called **Image**. There is a file picker next to it where you can select the image you want:



Once you have set the image, you can use it to draw on the map:



Summary

In this chapter, we learned about a tool called Tiled that is used to design 2D levels/maps. We learned the following topics:

- Adding tilesets
- Using map layers
- Drawing tools, such as stamp and fill tools
- Adding custom properties to the map
- Drawing various objects
- Making tile animations and adding images

In the next chapter, we will see how to read the maps and render them in the game.

Chapter 6. Drawing Tiled Maps

In the previous chapter, we created a game level in the editor. In this chapter, we will learn how to display the level in the game. We will also learn about a technique used to improve the game performance called **texture packing**. Along with this, we will learn how to effectively manage our game assets.

We will cover the following topics in this chapter:

- Asset management
- Map rendering

Asset management

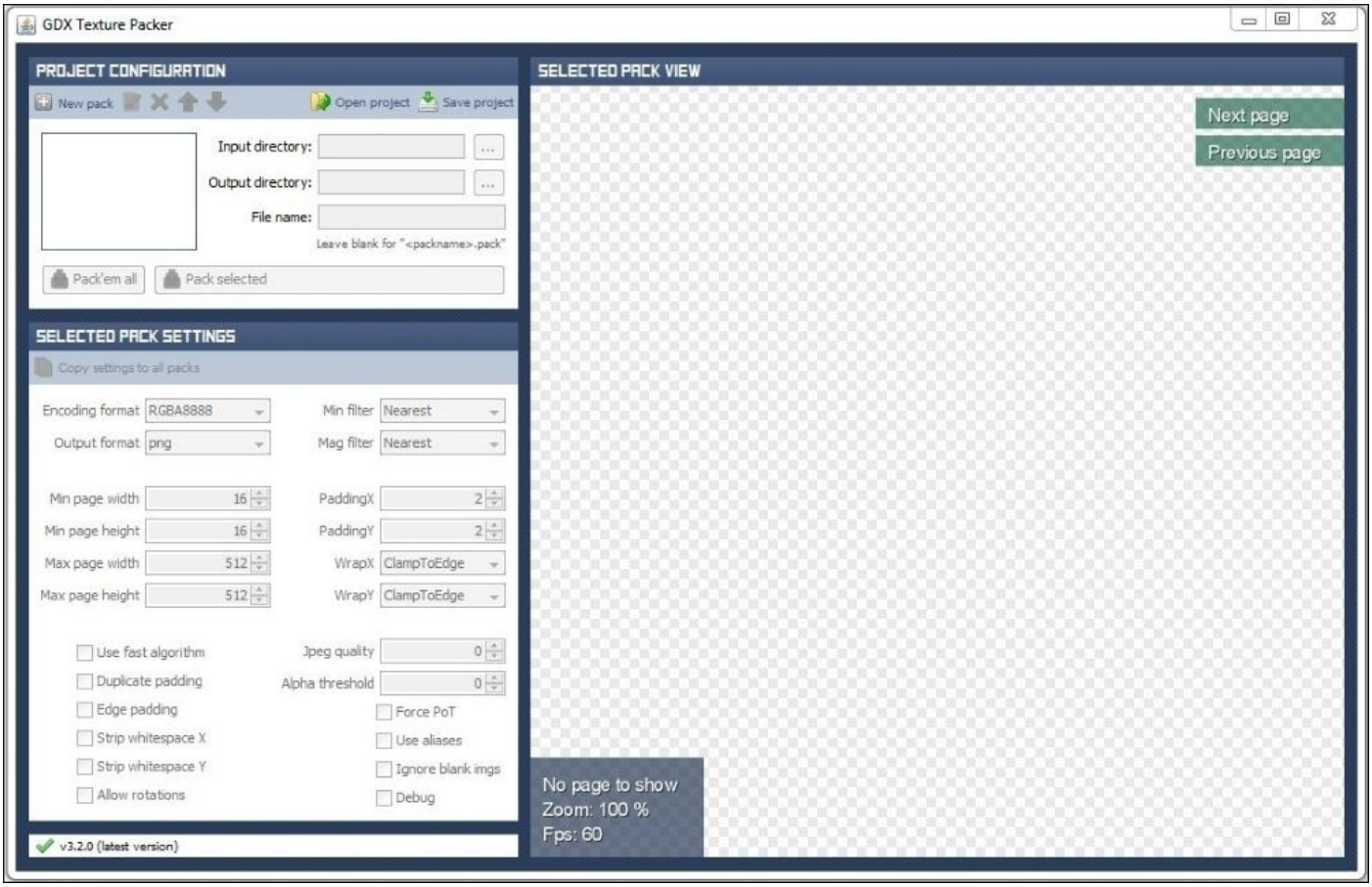
This topic will teach you how to optimize and manage assets in your game. This is necessary as the manual management of game assets is quite cumbersome as the code and game size increases.

Texture packer

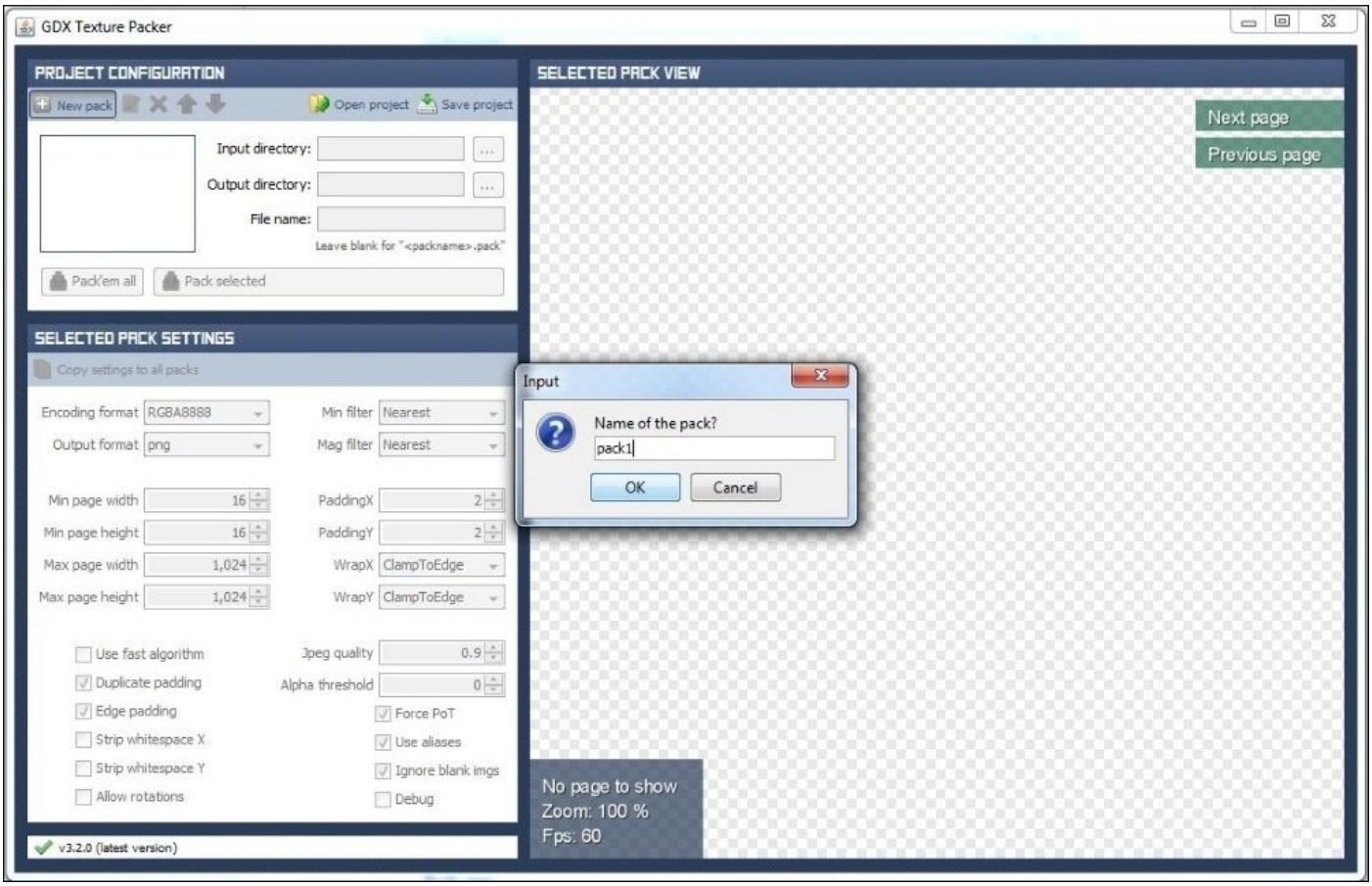
So far, we have been using separate images or textures in our games for different game objects, such as a player, mole, door, and so on. However, this method is slow in performance due to the way in which it is carried out in the backend. The GPU needs to switch textures each and every time we draw different game objects, which is a costly operation.

To avoid this issue, we pack different images into a single large image. This image is called a **texture atlas**. Doing so avoids the GPU from having to switch textures, increasing the performance of our game. Now, you don't have to do image packing manually in Photoshop or any image editing software. There is a readymade tool called **Texturepacker-GUI** which does that.

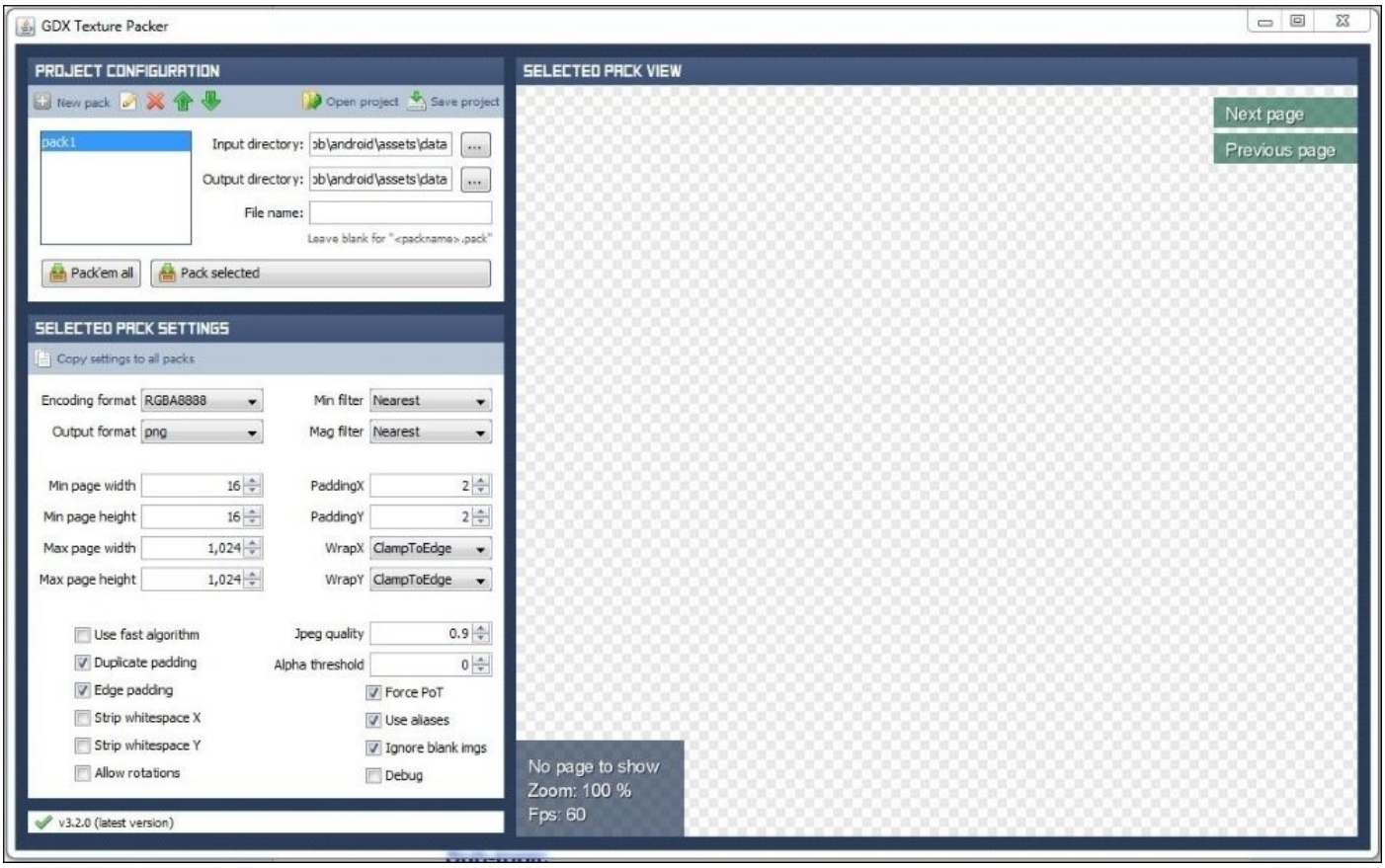
To download the tool, navigate to https://code.google.com/p/libgdx-texturepacker-gui/downloads/list. You can download the latest version from there. I am using version 3.2.0. After you extract the file, double-click on the `gdx-texturepacker.jar` file to run the app:



Click on the **New pack** button, which will open a dialog box that asks for a name. Give the name you want for the pack file. Here, I am giving `pack1`:

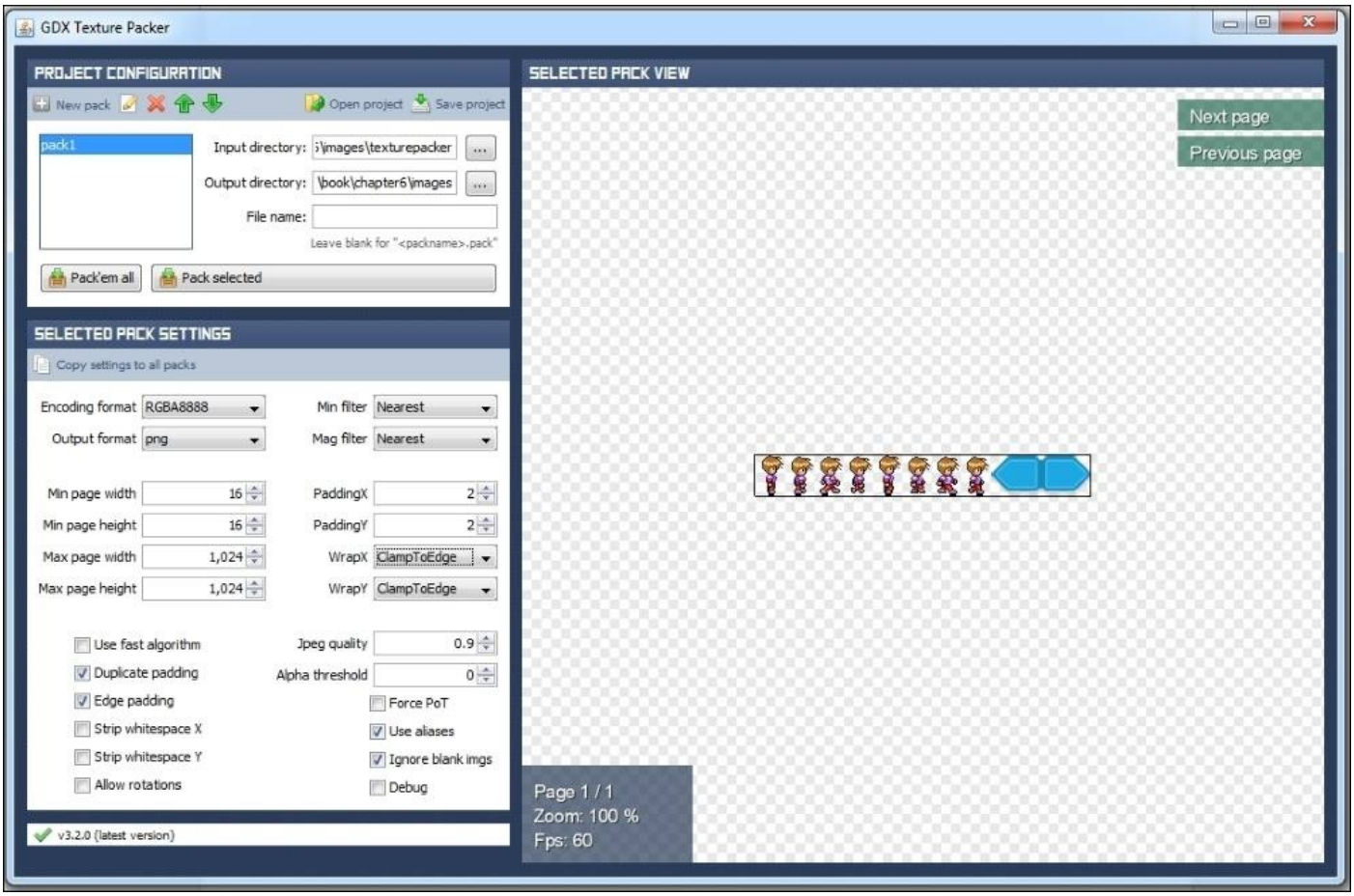


In the **Input directory** option, select the folder where you have kept the separate images for your game. In our case, let's select the directory where we have kept the images for Dungeon Bob. The **Output directory** is the folder where you want the packed image to be generated. Do not keep the background image in this directory, as we would need to handle it separately:

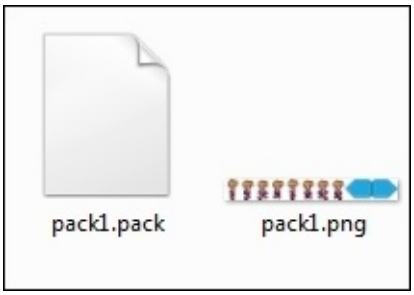


In the pack settings section, the **Min page width** and **Min page height** options represent the size of the packed image/texture; 1024 x 1024 should be fine for us as older devices can handle a texture with this resolution. The only constraint is that this setting should be higher than any of the input image you are using. If you want to pack large textures, you can increase this setting.

Uncheck the **Force PoT** option and click on **Pack selected**. Now, you should be able to see something like this:



The view on the right-hand side of the screen shows the packed image that comprises your images. If the combined size of the images exceed the maximum page size, multiple pages of packed textures are generated. You can navigate through them using the **Next page** and **Previous page** buttons in the top-right corner of the screen. The output directory will contain the packed images and `.pack` files that contain information of where each image is located in the packed image. Copy the pack file and the image to the `data` folder of your Android project:



The AssetManager class

The `AssetManager` class is a utility class in LibGDX that allows you to manage your game assets in an easy and uniform way. With this class, you can load game resources asynchronously, which makes it easy to display a loading screen. Let's refactor our game to make use of both `TextureAtlas` (packed textures) and `AssetManager`. Let's first create a class to keep track of the configuration information, such as filenames, level names, and so on. Create a new class in the `com.packtpub.dungeonbob` package called `GameConstants` and paste in the following code:

```
package com.packtpub.dungeonbob;

public class GameConstants {

    public static final String bobSpriteSheet = "bob_spritesheet";

    public static final String backGroundImage = "data/background.jpg";

    public static final String leftPaddleImage  = "paddleLeft";

    public static final String rightPaddleImage = "paddleRight";

    public static final String texturePack = "data/pack1.pack";
}
```

Right now, we have set the paths and names for the image assets in our game. In the `GameManager` class, add a reference to `AssetManager`:

```
static AssetManager assetManager;
```

We will instantiate it in the `initialize()` method:

```
assetManager = new AssetManager();
```

Now, we'll create a method to load the assets into the same class:

```
public static void loadAssets(){
    // queue the assets for loading
    assetManager.load(GameConstants.backGroundImage, Texture.class);

    assetManager.load(GameConstants.texturePack, TextureAtlas.class);

    //blocking method to load all assets
    assetManager.finishLoading();
}
```

To queue any assets for loading, we call the `load()` method. The asset's path is passed as the first argument and the class as the second one. The assets are not available for use after the `load()` method. To make them available, we call the `finishLoading()` method. This is a blocking function that makes all the assets called using the `load()` method available for use. This is not an asynchronous loading of the assets. We will learn about that in the subsequent chapters.

We call this function after instantiating `assetManager` in the `initialize()` method:

```
assetManager = new AssetManager();
loadAssets();
```

To use the background texture now, replace the line where we instantiated it with the following line of code:

```
backgroundTexture = assetManager.get(GameConstants.backGroundImage);
```

The `get()` method of `AssetManager` retrieves the asset specified. As it already knows which type of the asset it is (we gave this information while calling the `load()` method), we don't need any cast. Now, to dispose of the assets, we use the `unload()` method of `AssetManager`, which takes the filename of the asset as an argument. In the `dispose()` method of our `GameManager` class, replace the `backgroundTexture.dispose();` line with the following code:

```
assetManager.unload(GameConstants.backGroundImage);
```

To dispose of all the managed assets at once, you can use the following code:

```
assetManager.clear();
```

Tip

You should not dispose of managed assets manually.

We need to make some changes to the `Bob` class for it to work correctly with the Texturepacker-GUI tool. Since the animation sheet for Bob (or walksheet) is not a complete texture anymore (the whole texture is being packed into one), we need to change the type of `walkSheet` to `TextureRegion`:

```
TextureRegion walkSheet;   // sprite sheet
```

To accommodate these changes, we need to change the `initialize()` method as follows:

```
public void initialize(float width,float height,TextureRegion walkSheet){
    this.walkSheet=walkSheet; // save the sprite-sheet

    //split the sprite-sheet into different textures
    TextureRegion[][] tmp = walkSheet.split(
walkSheet.getRegionWidth()/ANIMATION_FRAME_SIZE,
walkSheet.getRegionHeight());
    // convert 2D array to 1D
    TextureRegion[] walkFrames = tmp[0];

    // create a new animation sequence with the walk frames and time period
of specified seconds
    walkAnimation = new Animation(0.08f, walkFrames);

    // instantiate bob sprite
    bobSprite = new Sprite();
    //set the size of bob
    bobSprite.setSize((walkSheet.getRegionWidth()/ANIMATION_FRAME_SIZE)*
(width/BOB_RESIZE_FACTOR),walkSheet.getRegionHeight()*
(width/BOB_RESIZE_FACTOR));

    // set the position of the bob to bottom - center
```

```
    setPosition(width/2f, 0);

    // set the animation to loop
    walkAnimation.setPlayMode(PlayMode.LOOP);
    // get initial frame
    currentFrame = walkAnimation.getKeyFrame(stateTime, true);

}
```

In the `GameManager` class, we need to change the type of `bobSpriteSheet` to `TextureRegion` as well:

```
static TextureRegion bobSpriteSheet; // texture spriteSheet for the bob.
```

We need to add a new variable to the texture atlas:

```
static TextureAtlas texturePack ; // packed texture.
```

In the `initialize()` method, we need to get the texture atlas/texture pack from `assetManager`:

```
texturePack = assetManager.get(GameConstants.texturePack); // get the
packed texture from asset manager
```

To get the texture region that corresponds to Bob's spritesheet from the packed image/texture pack, use its `findRegion()` method:

```
// load the bob sprite sheet from the packed image
bobSpriteSheet = texturePack.findRegion(GameConstants.bobSpriteSheet);
```

If you have used the `assetManager.clear()` method, then you don't need to dispose of these assets separately.

Similar steps are required for the paddles:

```
static TextureRegion leftPaddleTexture;
static TextureRegion rightPaddleTexture;
```

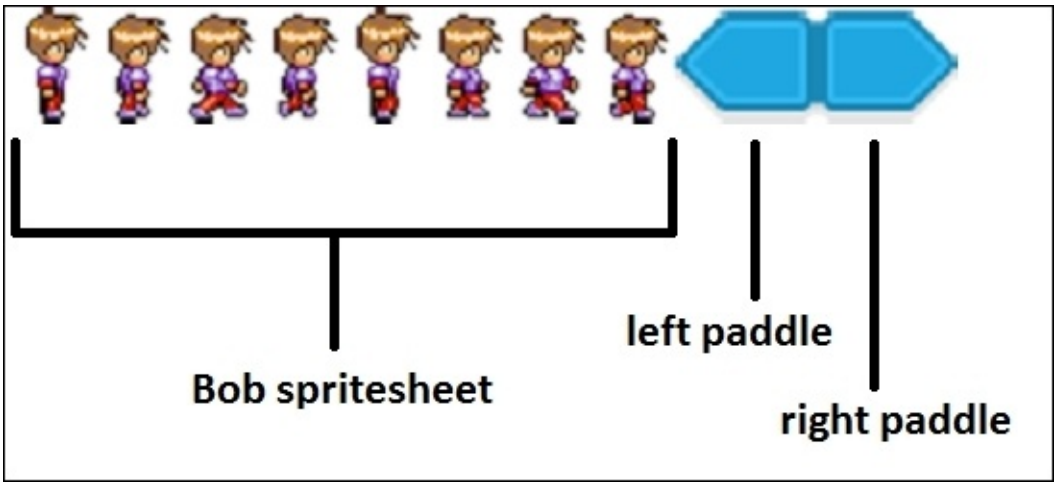
In the `initializeLeftPaddle()` method, add the following code:

```
//load left paddle texture region
leftPaddleTexture = texturePack.findRegion(GameConstants.leftPaddleImage);
```

In the `initializeRightPaddle()` method, add the following code:

```
//load right paddle texture region
rightPaddleTexture =
texturePack.findRegion(GameConstants.rightPaddleImage);
```

Let's take a look at the following screenshot:



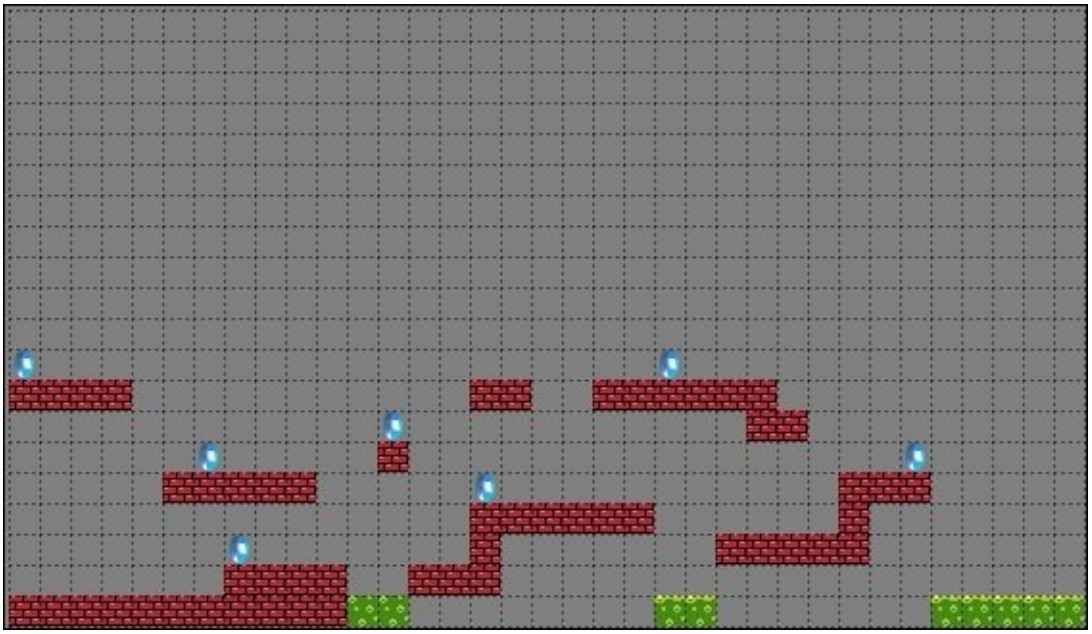
Also, don’t forget to remove the `dispose()` calls for these two methods in the `dispose()` method. If you run the game now, it should work in the same way, as mentioned earlier, except that now our assets are better managed and show better performance.

Rendering maps

This section will cover how to display the Tiled map in the game. We will also learn about the LibGDX APIs that can parse the Tiled map format and display them in the game.

Basic map rendering

Let’s deal with rendering the maps made with Tiled in our game. For this, I’ve made a new small map in Tiled, which looks something like this. We will continue with the same project from Chapter 4, *Dungeon Bob*:



This map is 35 x 20 units in size. I have used two layers. One for the crystals and the other for things such as walls and hazardous liquids. Keep the `.tmx` file along with the corresponding assets in the `data/maps` folder of your Android project.

First of all, let’s disable the drawing of our game objects for the time being in order to concentrate on the map. We’ll later integrate them. Comment out the `draw()` calls to Bob, paddles, and backgrounds. We will need a `map` instance to work with our map. In the `GameManager` class, add this line of code:

```
static TiledMap map;
```

Let’s define the path for the map file in the `GameConstants` class:

```
public static final String level1 = "data/maps/level1.tmx";
```

To queue the map for loading, add these lines of code to the `loadAssets()` method of `GameManager` before the `finishedLoading()` method’s call:

```
// set the tiled map loader for the assetmanager
assetManager.setLoader(TiledMap.class, new TmxMapLoader(new
InternalFileHandleResolver()));
//load the tiled map
assetManager.load(GameConstants.level1, TiledMap.class);
//blocking method to load all assets
assetManager.finishLoading();
```

Now, you can get the `map` instance loaded into the `initialize()` method:

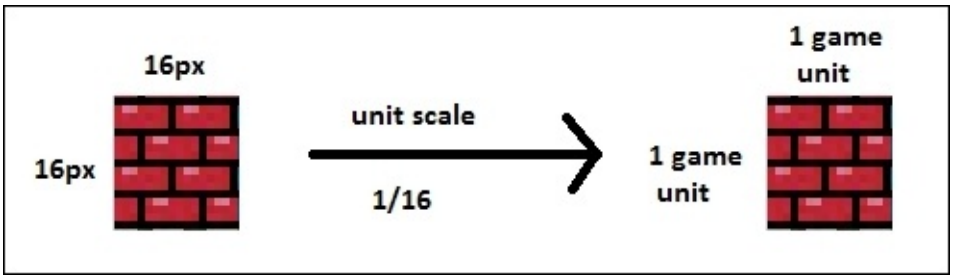
```
loadAssets();
// get the map instance loaded
map = assetManager.get(GameConstants.level1);
```

To render the map, we need a map renderer. As we are using an orthogonal map, the `OrthogonalTiledMapRenderer` from the Tiled package is suitable for this purpose. Declare its instance in the `GameManager` class:

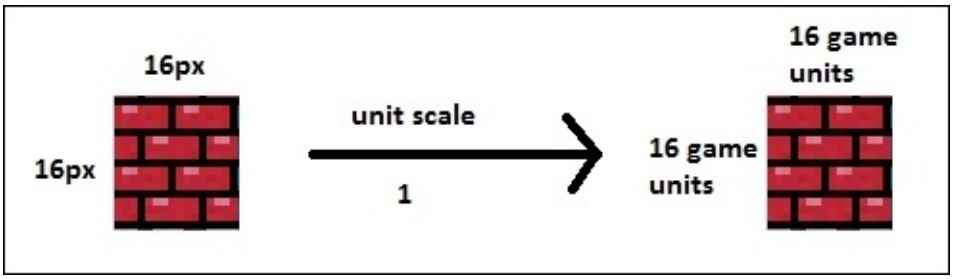
```
public static OrthogonalTiledMapRenderer renderer; // map renderer
```

Now, we will instantiate it in the `initialize()` method. But first, we need to define the unit scale. The unit scale indicates the pixel to game-unit ratio. If it is one, then one map pixel will be equal to one game unit. If your tiles are 16 x 16 in a map and you want to map them as 1 x 1 squares, then you need to define the unit scale as 1/16.

With the unit scale as 1/16, we get the following diagram:



With the unit scale as 1, we get the following diagram:



Let's define it as `1/16f` for now in the `GameConstants` class:

```
public static final float unitScale = 1/16f;
```

To instantiate the map renderer, add the following code after we get the map in the `GameManager` class:

```
// get the map instance loaded
map = assetManager.get(GameConstants.level1);
renderer = new OrthogonalTiledMapRenderer(map, GameConstants.unitScale);
```

It takes two parameters: the map and the unit scale. Next, we have to set the renderer's view to the main camera's view. As we can't access the camera from `GameManager`, let's first make it public and static in the `GameScreen` class:

```
public static OrthographicCamera camera;
```

Add the following line to the `initialize()` method of `GameManager`:

```
renderer = new OrthogonalTiledMapRenderer(map, GameConstants.unitScale);
// set the renderer's view to the game's main camera
renderer.setView(GameScreen.camera);
```

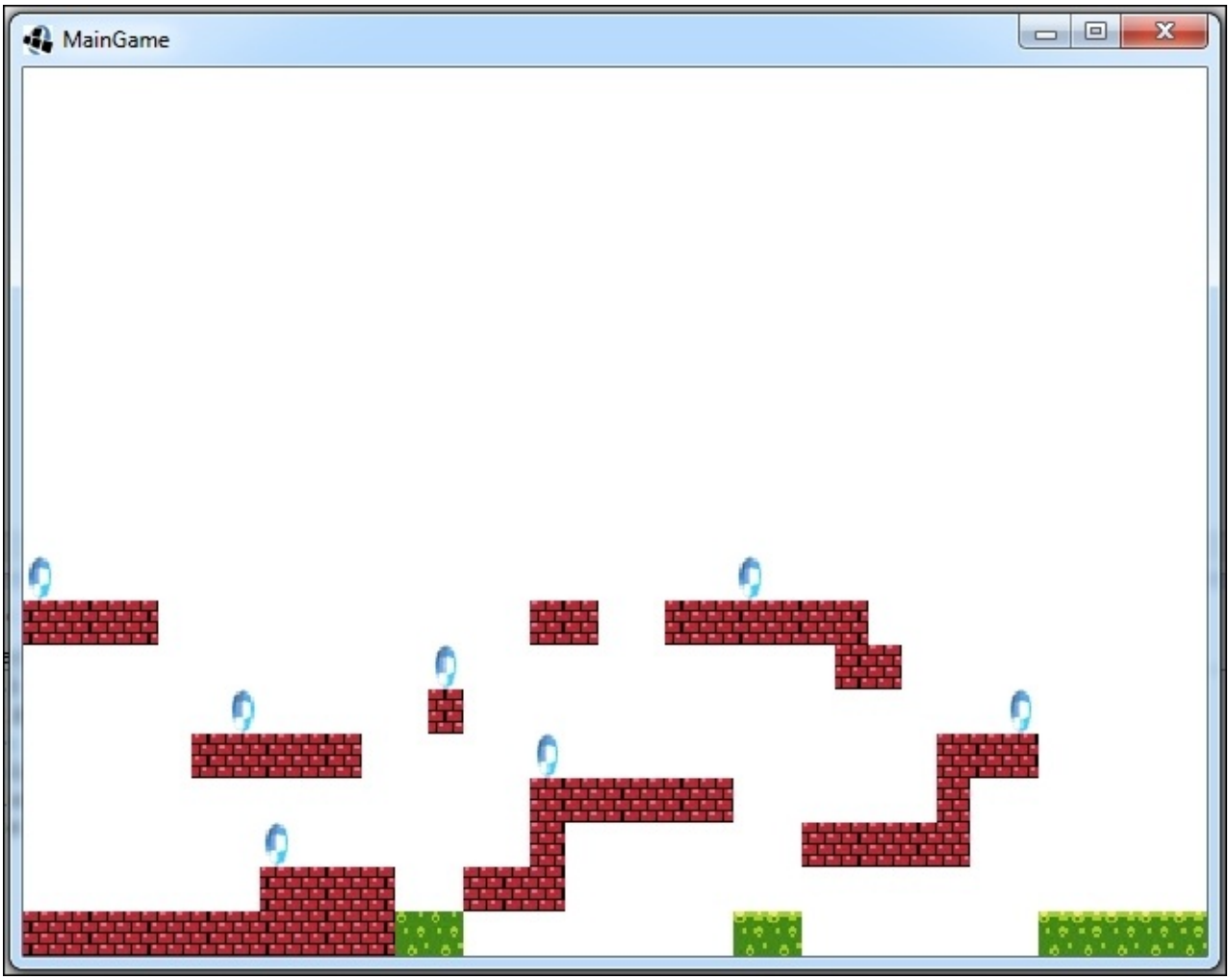
To draw the map, we will call the renderer's `render()` method. We won't call it in the `renderGame()` method as it will interfere with the sprite batch's rendering process. Instead, we will call it in the `GameScreen` class' `render()` method:

```
// render the game objects
batch.begin();
GameManager.renderGame(batch);
batch.end();

GameManager.renderer.render();
```

To show the complete map, we would need to change the camera's viewport to the map's width and height. Add the following lines of code after the renderer's instantiation in the `initialize()` method:

```
renderer = new OrthogonalTiledMapRenderer(map, GameConstants.unitScale);

GameScreen.camera.setToOrtho(false, 35,20); // show 35x20 map tiles on screen
GameScreen.camera.update();
// set the renderer's view to the game's main camera
renderer.setView(GameScreen.camera);
```

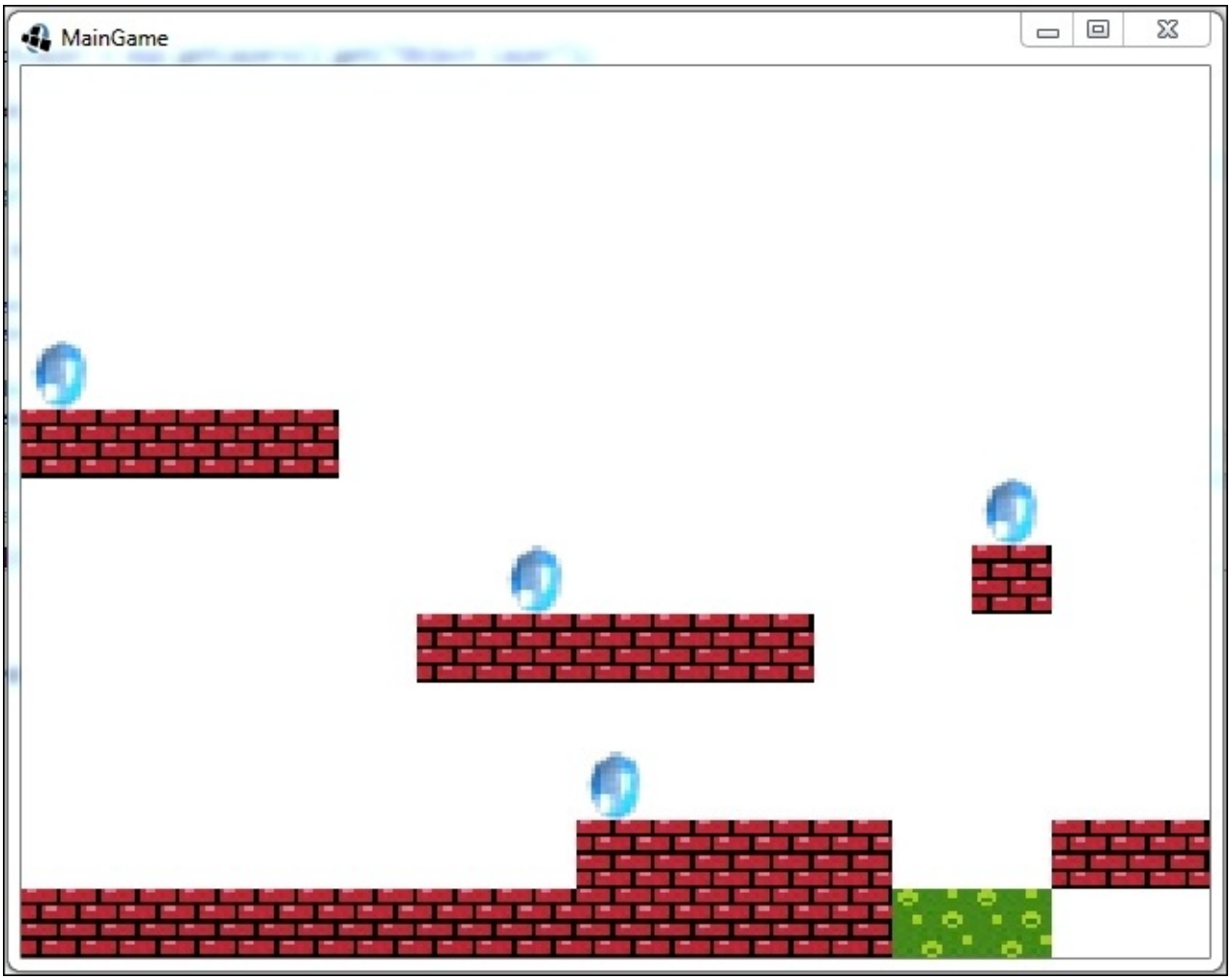
If you run the game now, you will notice that the complete map has been displayed on the screen with the crystal animations as well!



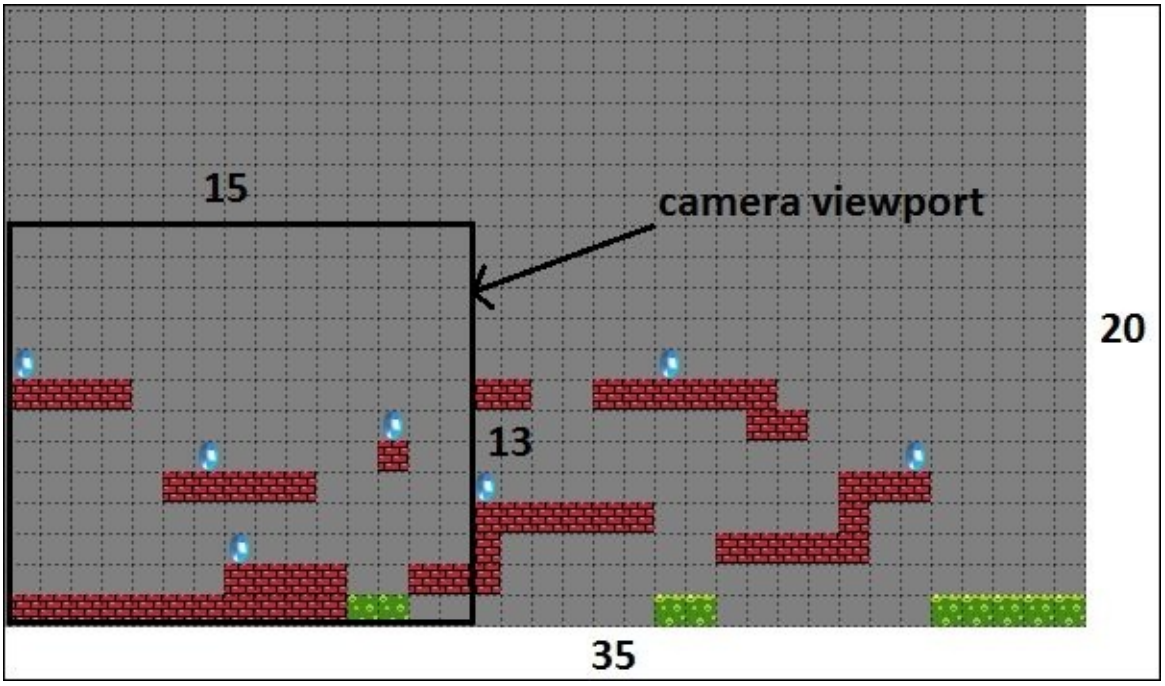
If your level is huge like most platformers, you would want to show only part of the map on the screen at a time. Let’s show 15 x 13 units at a time. Edit the appropriate line so that it looks like this:

```
GameScreen.camera.setToOrtho(false, 15,13); // show 15x13 map tiles on screen
```

The screen will now look like this:



The following screenshot shows the total map and viewport distinction:



Reading the map

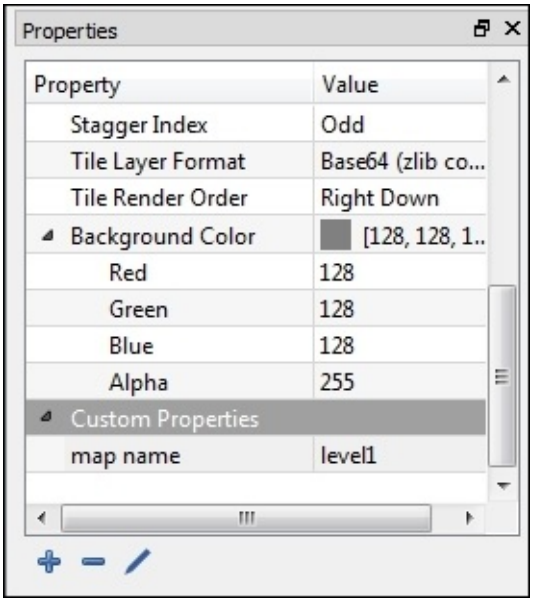
The map has properties. These are key-value pairs, which you can get in your game. To view them, you can iterate over the keys and get the corresponding properties. Reset the code to show 35 x 20 units and add the following lines of code after the map initialization to see the properties:

```
// get the map instance loaded
map = assetManager.get(GameConstants.level1);

Iterator<String> iterator = map.getProperties().getKeys();
while(iterator.hasNext()){
    String key =iterator.next();
    System.out.println("Name:" +key+" Value:
"+map.getProperties().get(key));
}
```

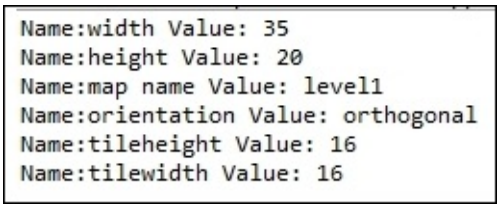
If you run the game now, you will see the following output in the console output window:

```
Name:width Value: 35
Name:height Value: 20
Name:orientation Value: orthogonal
Name:tileheight Value: 16
Name:tilewidth Value: 16
```

These are the map's default properties. As I've mentioned earlier, you can set custom properties for the map in the editor and get them through code:



The code output after reading the map's properties is as follows:



A map contains layers. To get these layers, you need to call `map.getLayers()`, which will give all the layers contained in the map. To get a specific layer, you can call the `GameManager` class' `initialize()` method after you get the `map` instance:

```
TiledMapTileLayer tiledLayer = (TiledMapTileLayer)map.getLayers().get(0);
```

This will give you the first map layer, starting from the bottom. Alternatively, you can get a layer by its name:

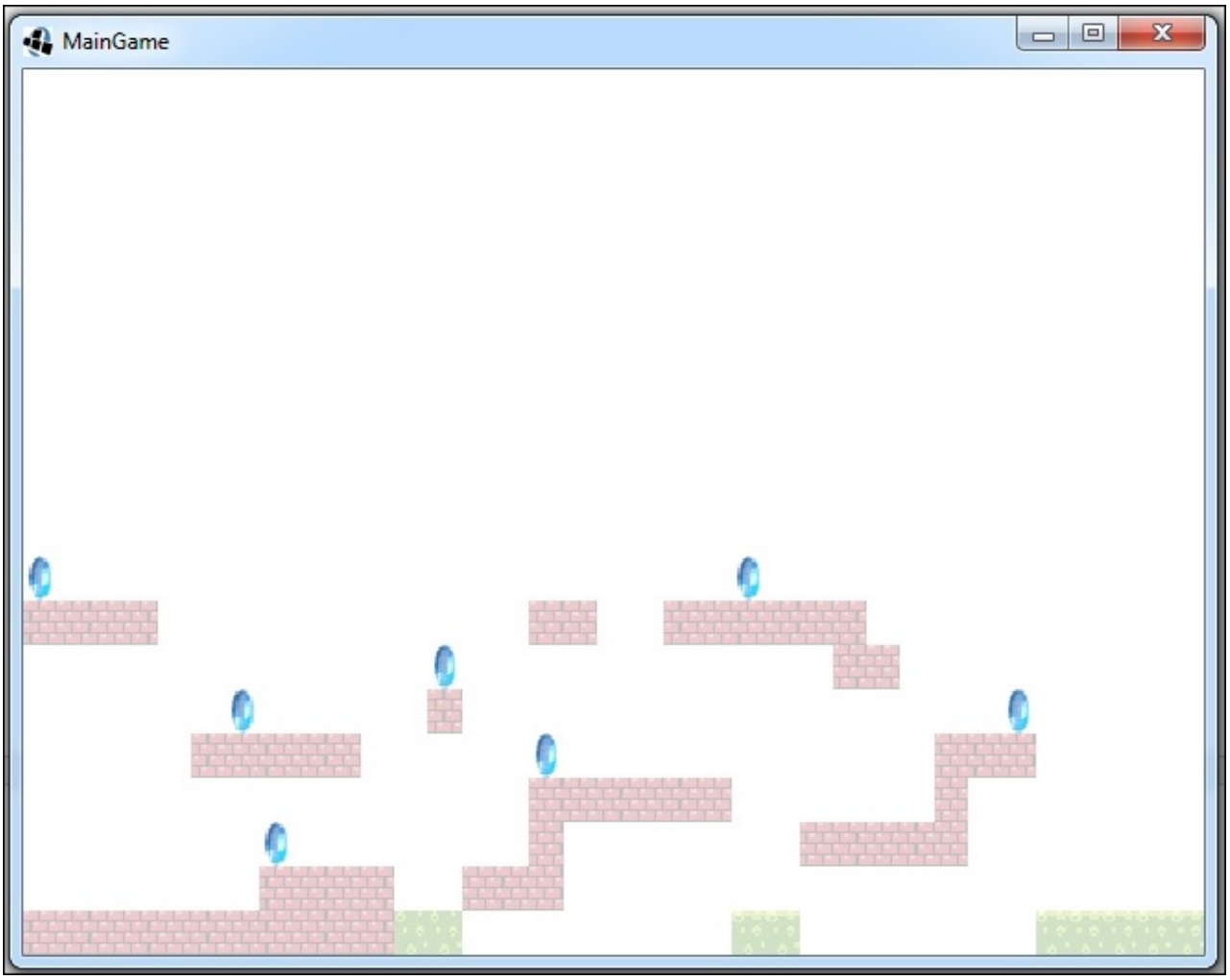
```
TiledMapTileLayer tiledLayer =
(TiledMapTileLayer)map.getLayers().get("Wall");
```

The preceding line will give you the layer named `Wall`. Once you get a layer, you can then get its opacity using `tiledLayer.getOpacity()` and check whether the layer is visible using `tiledLayer.isVisible()`. You can also set these properties at runtime.

Let's try to set the wall layer's opacity to `0.25`. First, get the layer from the map, as mentioned earlier, and call the `initialize()` method of `GameManager`:

```
tiledLayer.setOpacity(0.25f);
```

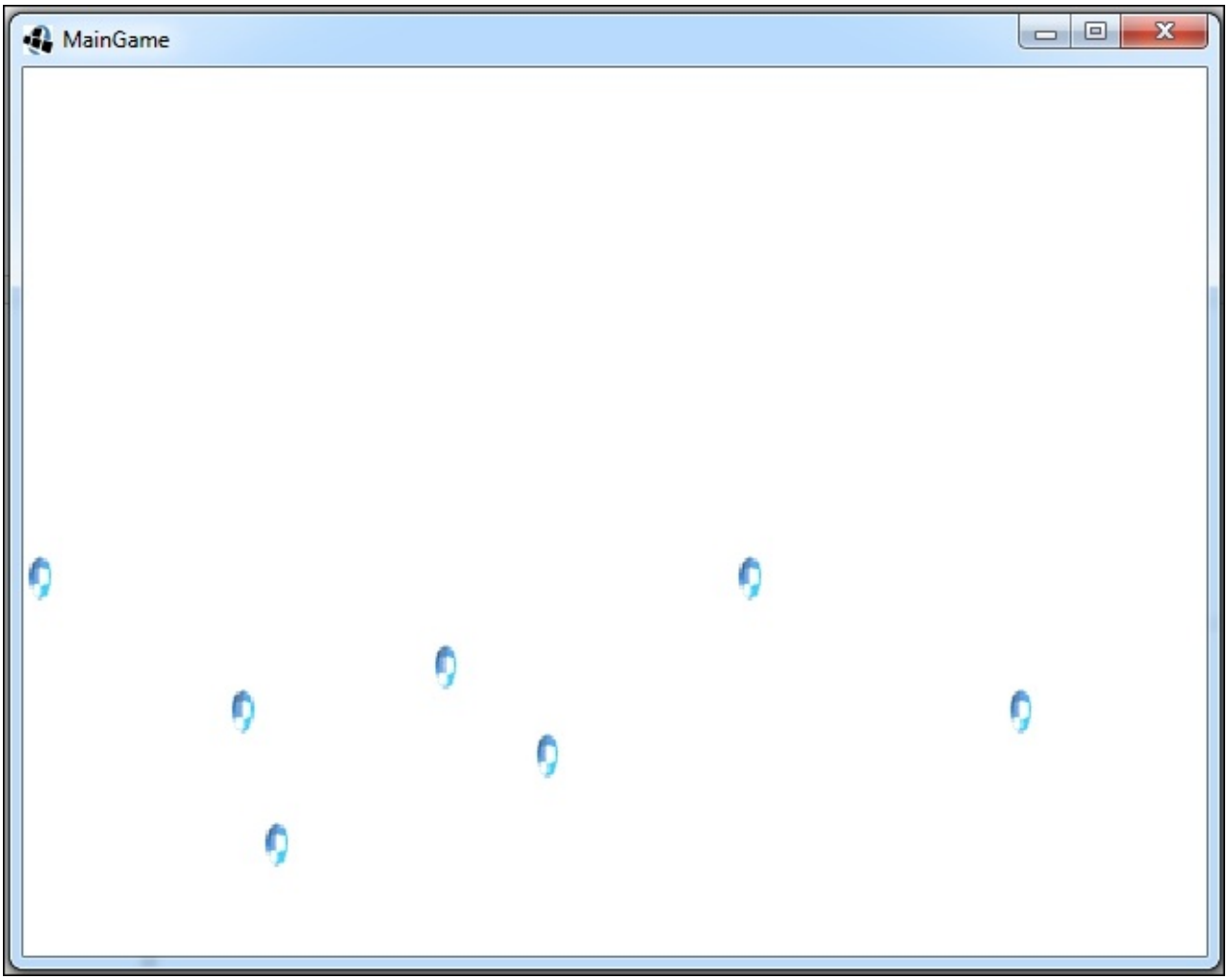
The screen will now look like this:



As you can see, the wall layer is now semitransparent, but the crystals are unaffected. To make the layer invisible, use the following line of code:

```
tiledLayer.setVisible(false);
```

The screen will now look like this:



We can only see the crystals as the wall layer is invisible. To get the properties of a layer, use `tiledLayer.getProperties()`. A Tiled map layer also contains a matrix of cells, which are the tiles that we paint in the editor. A cell is a container for these tiles and contains some additional attributes. For example, if we place a brick tile in a certain position in the layer, then that information can be queried from the layer and will be stored as a cell. To get a cell, use the following code:

```
Cell cell = tileLayer.getCell(column, row);
```

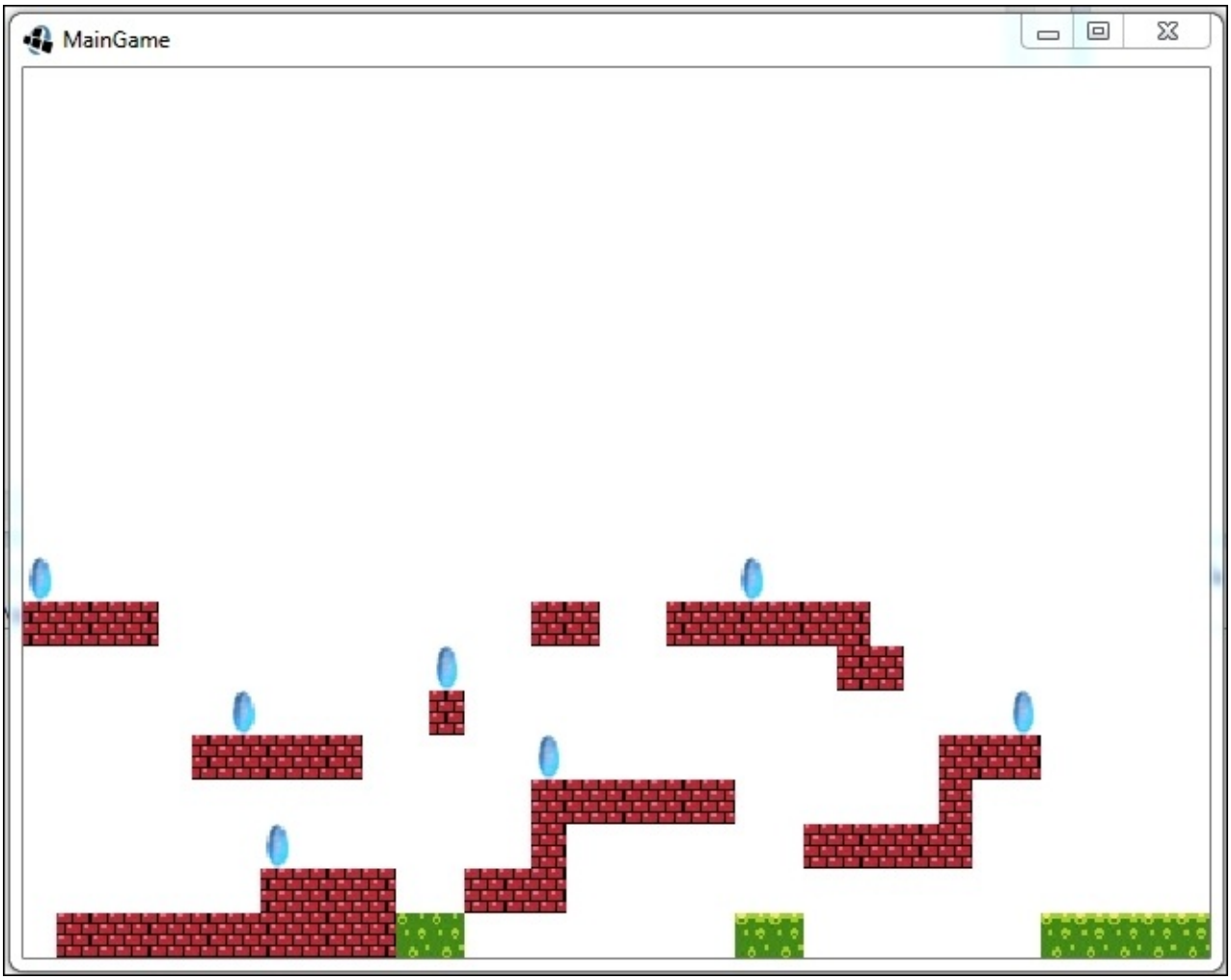
Tip

This is different from how we access 2D arrays (row,column).

The bottom-left cell of the map is located at (0, 0) and the top-right cell is located at (width-1, height-1). To remove a cell, you can use the `setCell()` method of the layer and set it to `null`. Let's try to remove the first cell. First, remove the calls to `setOpacity()` and `setVisible()`:

```
tiledLayer.setCell(0, 0, null);
```

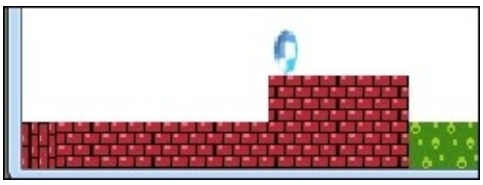
This is shown in the following screenshot:



As you can see, the cell in the bottom-left corner is removed. This technique can be used in a game such as *Super Mario* where the brick disappears when Mario touches it with his head. Once you get a cell, you can set/get its rotation or flip it horizontally/vertically. Let's rotate the tile at (0, 0) by 90 degrees. Replace the `setCell()` line with the following code:

```
tiledLayer.getCell(0, 0).setRotation(Cell.ROTATE_90);
```

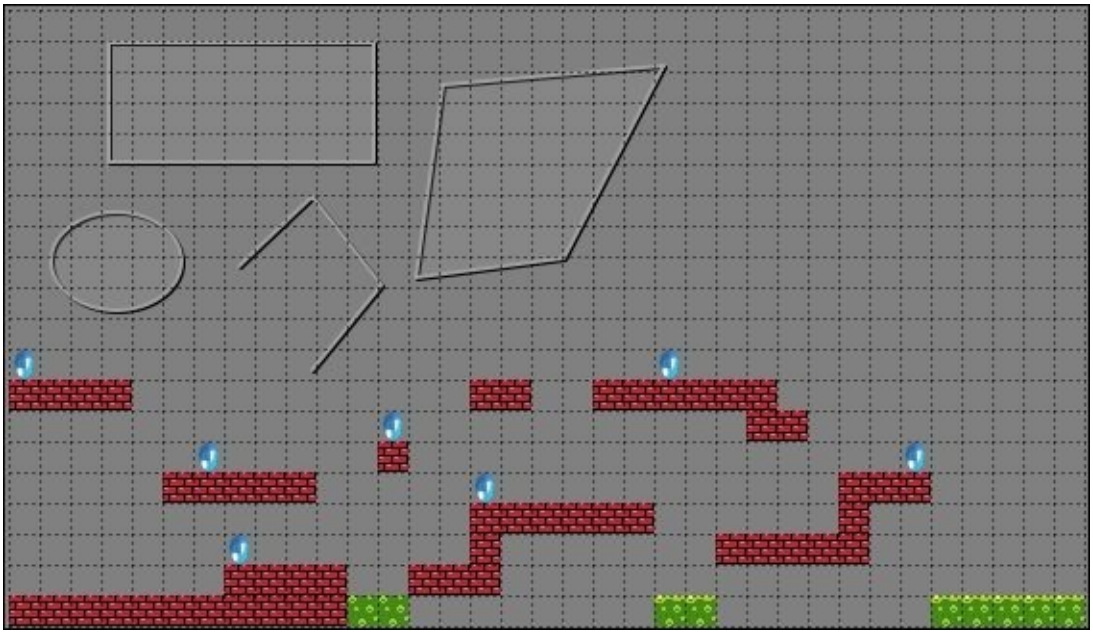
Add this line after you have got the layer from the map. If you run the game now, you will notice that the particular tile has been rotated by 90 degrees:



To flip the tile horizontally, use `cell.setFlipHorizontally(true)` and to flip it vertically, use `setFlipVertically(true)`.

Map objects

Let’s add some shapes to our map using the editor in the object layer:



I’ve added a rectangle, polygon, eclipse, and polyline to the map. To draw the map objects, let’s define a new method called `drawShapes()` in the `GameManager` class:

```
static void drawShapes(){
}
```

In this method, we will retrieve the objects from the map and draw them based on their types. To get the objects, we will first have to get the object layer. In the `initialize()` method of the `GameManager` class, add this line of code to get the object layer:

```
MapLayer objectLayer = map.getLayers().get("Object Layer");
```

We declare a variable to store the map objects in the class:

```
static MapObjects mapObjects;
```

To get all the map objects in the object layer, add the following line of code:

```
mapObjects = objectLayer.getObjects();
```

Now, we can proceed to retrieve the individual objects and draw them. We will use an iterator in the `drawShapes()` function to iterate over the objects. Add the following lines of code to the `drawShapes()` function:

```
Iterator<MapObject> mapObjectIterator = mapObjects.iterator();
while(mapObjectIterator.hasNext()){
    MapObject mapObject = mapObjectIterator.next();
}
```

There are different types of objects in our map. Let’s start off by displaying the rectangle map object. We need to check whether the map object retrieved in the current iteration is

of the rectangle type. To do this, add the following lines of code:

```
if(mapObject instanceof RectangleMapObject){
    Rectangle rectangle = ((RectangleMapObject)mapObject).getRectangle();
}
```

If the object is of the rectangle type, we cast it to `RectangleMapObject` and get a `Rectangle` instance from it. To actually draw the object on the screen, you need something called `ShapeRenderer`. The `ShapeRenderer` is a class in LibGDX that allows you to draw basic shapes. We declare a reference of `ShapeRenderer` in the `GameManager` class:

```
static ShapeRenderer shapeRenderer; // for drawing shapes
```

Next, we will instantiate the shape renderer in the `initialize()` method of `GameManager`:

```
shapeRenderer = new ShapeRenderer();
```

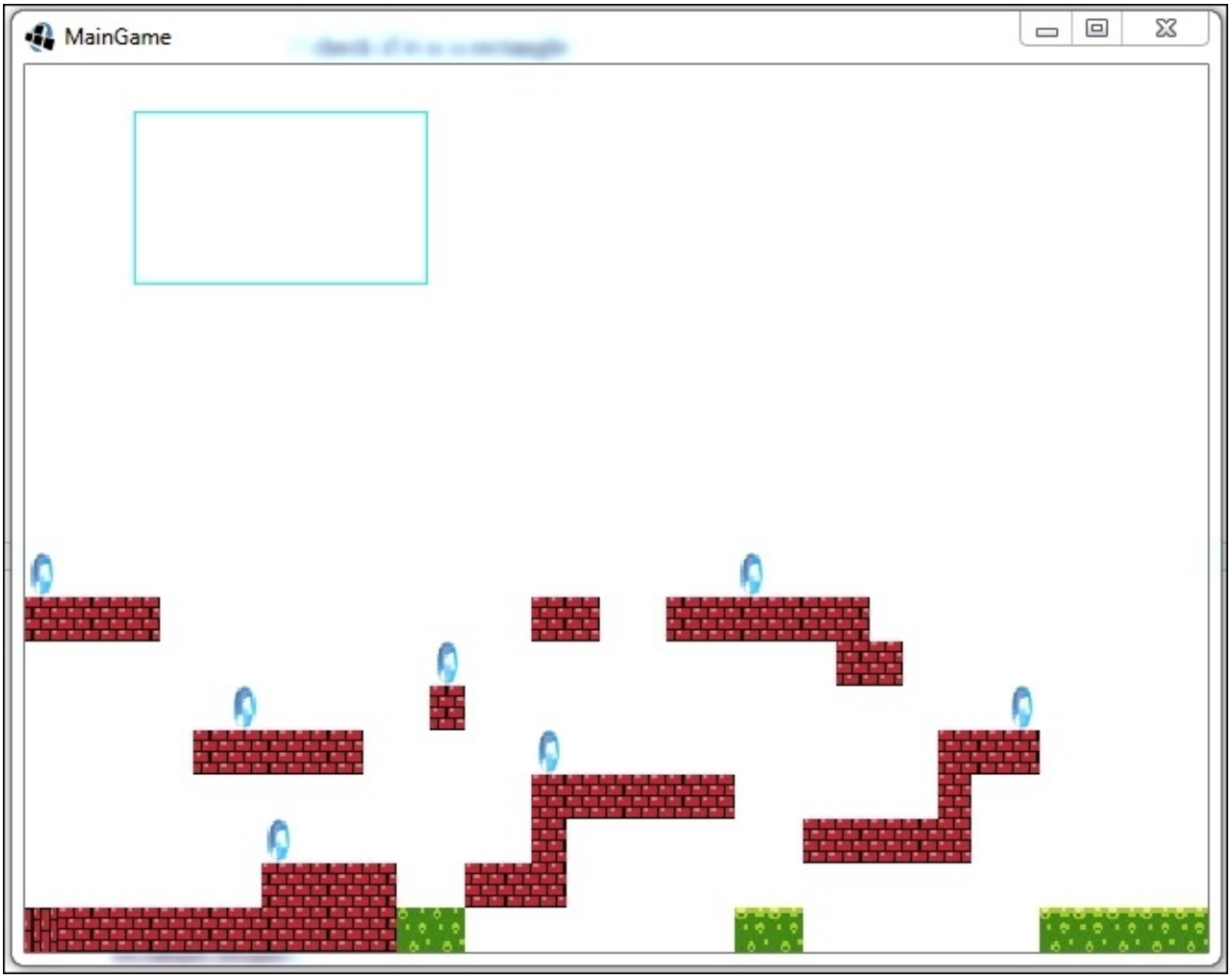
The `drawShapes()` method looks something like this after the rendering code is added to it and it is called by the `renderGame()` function of the same class:

```
static void drawShapes(){
    GameScreen.camera.update();
    // set the shaperenderer's view to camera's view

shapeRenderer.setProjectionMatrix(GameScreen.camera.combined.scl(GameConsta
nts.unitScale));

    // set the shape as completely filled
    shapeRenderer.begin(ShapeType.Line);
    //set the shape's color as blue
    shapeRenderer.setColor(0, 1, 1, 1);

    Iterator<MapObject> mapObjectIterator = mapObjects.iterator();
    while(mapObjectIterator.hasNext()){
        // get the map object from iterator
        MapObject mapObject = mapObjectIterator.next();

        // check if it is a rectangle
        if(mapObject instanceof RectangleMapObject){
            Rectangle rectangle =
((RectangleMapObject)mapObject).getRectangle();
            //draw rectangle shape on the screen
            shapeRenderer.rect(rectangle.x, rectangle.y, rectangle.width,
rectangle.height);
        }

    }
    shapeRenderer.end();
}
```

First, we set the `ShapeRenderer` class' view to the camera's view multiplied by the unit scale. This is needed as the coordinates of the objects obtained from the maps are absolute ones. To render them properly, we need to scale them as per our unit scale. The shapes are drawn between the `begin()` and `end()` methods of `ShapeRenderer`. The `begin()` method takes an argument that specifies whether the shape would be completely filled with the

color or just an outline.

We are using `ShapeType.Line`, which means that it would be a rectangle outline. Then, we set the color to blue with the RGBA values. Finally, we draw the rectangle shape using the `shapeRenderer.rect()` method and using the `rectangle` instance that we obtained, we end `ShapeRenderer`:

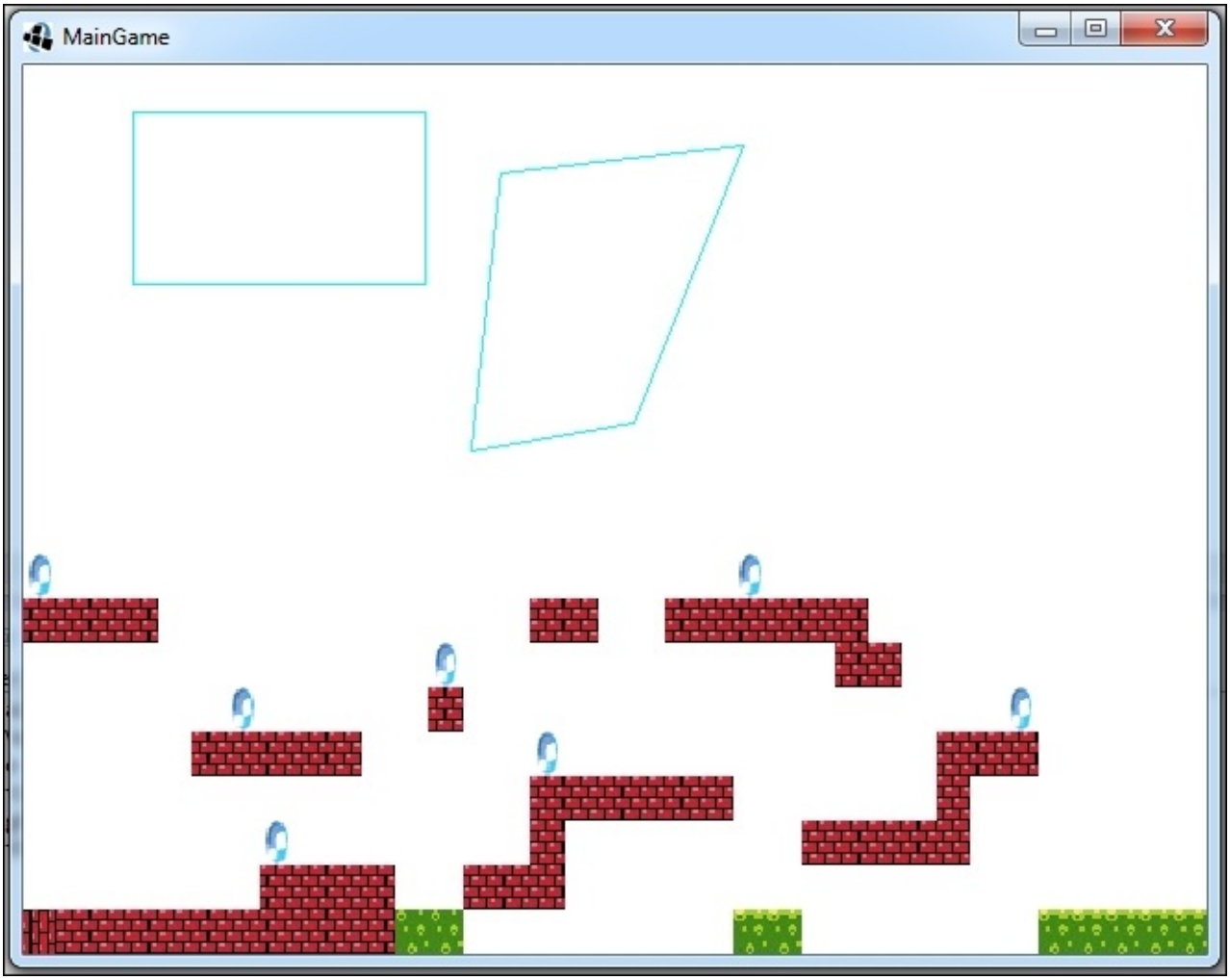


Let’s draw another object type. This one is a polygon object. Update the `drawShapes()` method like this:

```
// check if it is a rectangle
if(mapObject instanceof RectangleMapObject){
    Rectangle rectangle = ((RectangleMapObject)mapObject).getRectangle();
    //draw rectangle shape on the screen
    shapeRenderer.rect(rectangle.x, rectangle.y, rectangle.width,
rectangle.height);
}

// check if it is a polygon
else if(mapObject instanceof PolygonMapObject){
    Polygon polygon = ((PolygonMapObject)mapObject).getPolygon();
    shapeRenderer.polygon(polygon.getTransformedVertices());
}
```

Here, we check whether the object is of the polygon type. We then retrieve a `polygon`

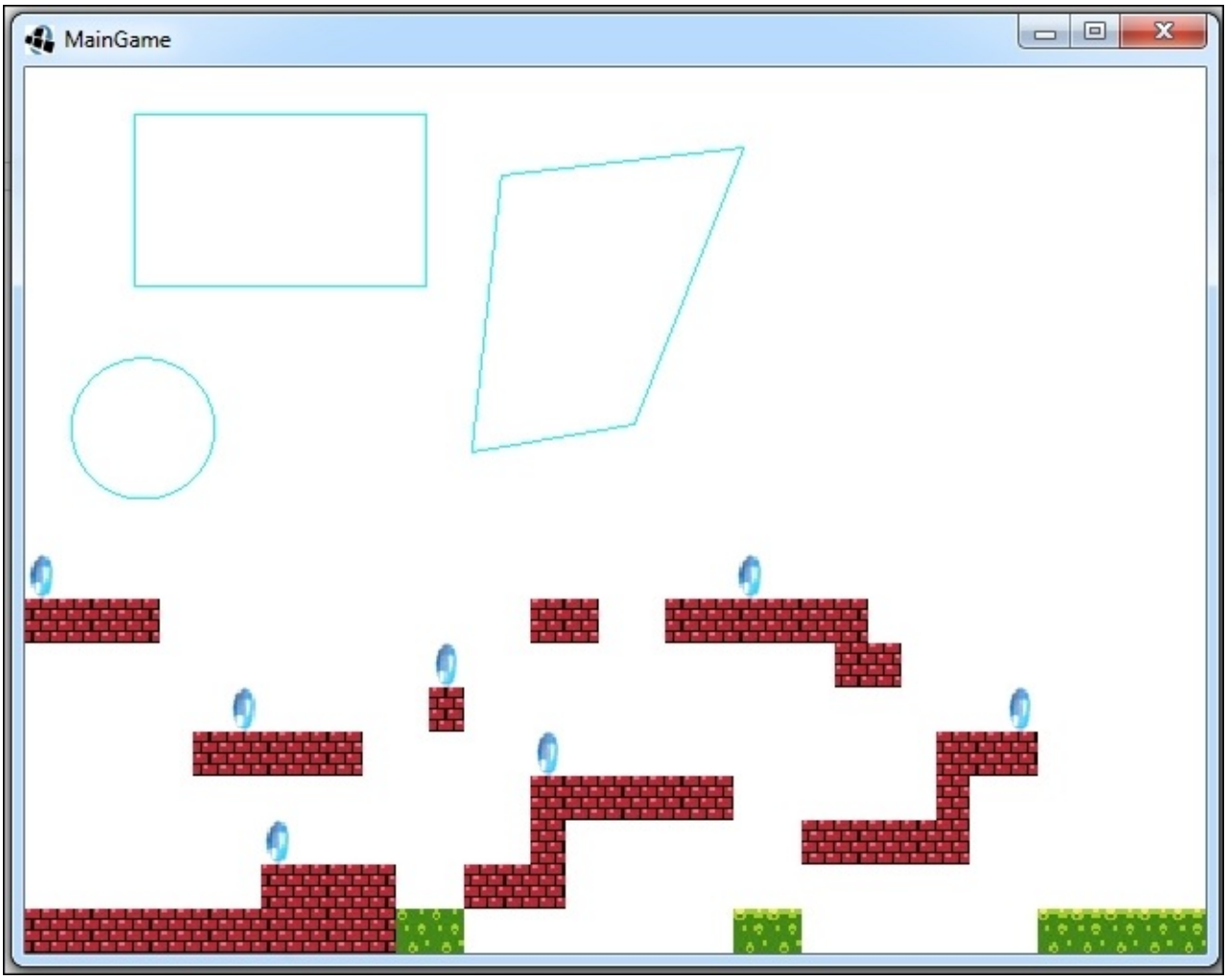
instance from it. The `ShapeRenderer` class' `polygon()` method is used to draw a polygon on the screen and expects an array of polygon vertices as an argument. We get this array from the `polygon` instance by calling `getTransformedVertices()` on it:



Update the `drawshapes()` method again for an ellipse this time:

```
// check if it is a polygon
else if(mapObject instanceof PolygonMapObject){
    Polygon polygon = ((PolygonMapObject)mapObject).getPolygon();

    shapeRenderer.polygon(polygon.getTransformedVertices());
}

//check if it an ellipse
else if(mapObject instanceof EllipseMapObject){
    Ellipse ellipse = ((EllipseMapObject)mapObject).getEllipse();

    shapeRenderer.ellipse(ellipse.x, ellipse.y,
ellipse.width,ellipse.height);
}
```

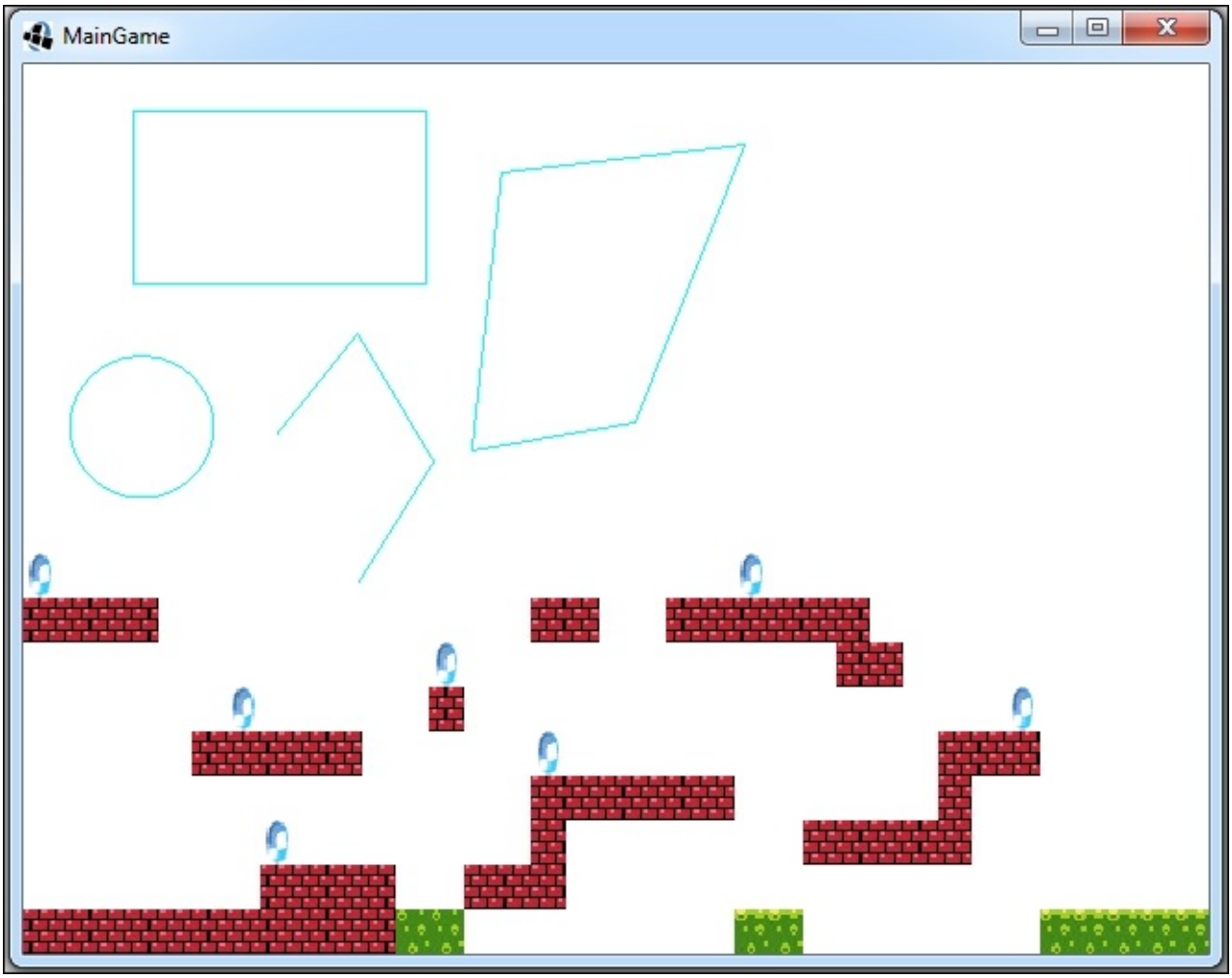
The screen will now look like this:



Finally, for a polyline, we add the following highlighted code:

```
//check if it an ellipse
else if(mapObject instanceof EllipseMapObject){
    Ellipse ellipse = ((EllipseMapObject)mapObject).getEllipse();

    shapeRenderer.ellipse(ellipse.x, ellipse.y,
ellipse.width,ellipse.height);
}

// check if it is a polyline
else if(mapObject instanceof PolylineMapObject){
    Polyline polyline = ((PolylineMapObject)mapObject).getPolyline();

    shapeRenderer.polyline(polyline.getTransformedVertices());
}
```

The screen will now look like the following screenshot:


MainGame

Summary

In this chapter, we learned the following topics:

- Packing textures for performance
- Managing assets using the `AssetManager` class
- Drawing the Tiled map on the game screen
- Reading map properties and layers, and updating them on the fly
- Reading objects from the map and displaying them

In the next chapter, we will cover game physics and collisions.

Chapter 7. Collision Detection

In this chapter, we will learn how to integrate Bob and other game objects with the Tiled map that we created. We will also learn how to control the camera and follow the player around. We will also learn how to add realistic physics and detect collisions between obstacles from the map.

In this chapter, we will cover the following topics:

- Integrating game objects
- Physics and collision

Scaling objects and adding a secondary camera

In this section, we will learn how to integrate the game objects with the Tiled map and camera control.

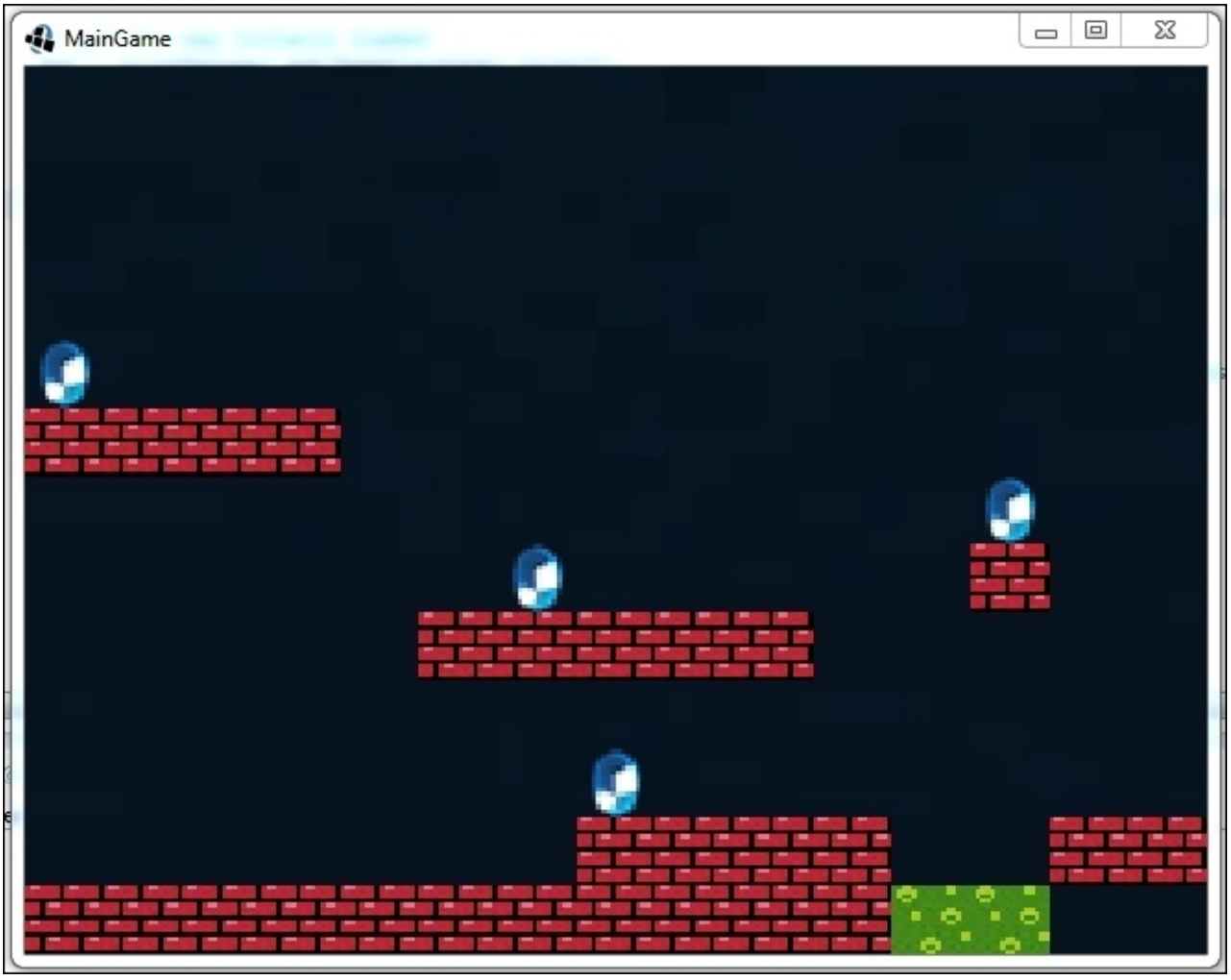
Integrating Bob

In the previous chapter, we rendered the map but other game objects, such as Bob, the paddles, and the background were out of picture. Let's first try to integrate them with the map. Remove the map iteration code (layers/tiles/objects). Remove the call to `drawshapes()` as well.

Reduce the viewport size to 15 x 13 in the `initialize()` method of the `GameManager` class:

```
GameScreen.camera.setToOrtho(false, 15,13);
```

Now, uncomment all the draw calls for the other game objects and run the game:



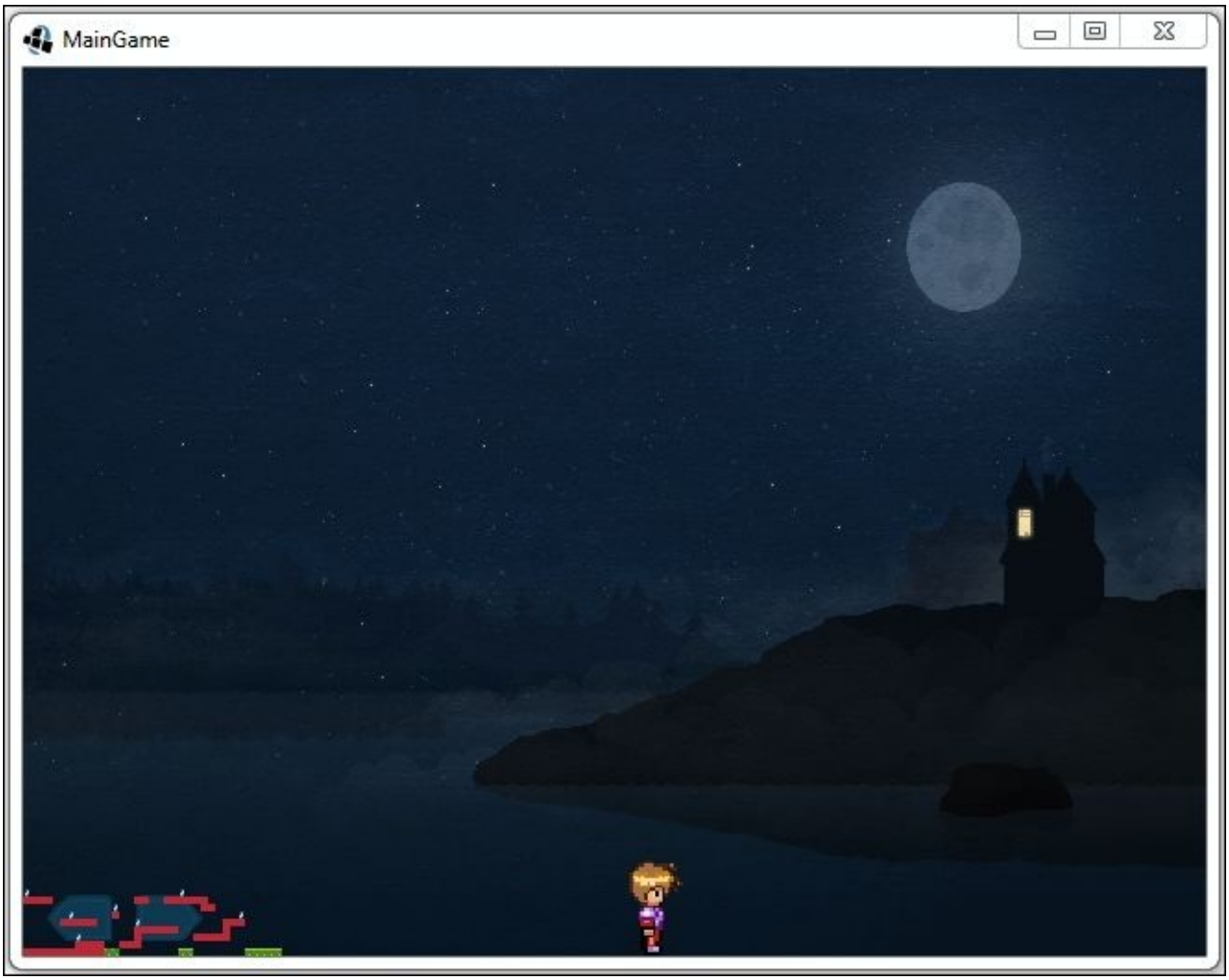
Okay, so what happened to the game objects? Why could we not see them? To understand this, change the unit scale to 1/4 in the `GameConstants` class:

```
public static final float unitScale = 1/4f;
```

Change the camera's viewport to `width` and `height` in the `GameManager` class' `initialize()` method:

```
GameScreen.camera.setToOrtho(false, width,height);
```

The screen will now look like this:

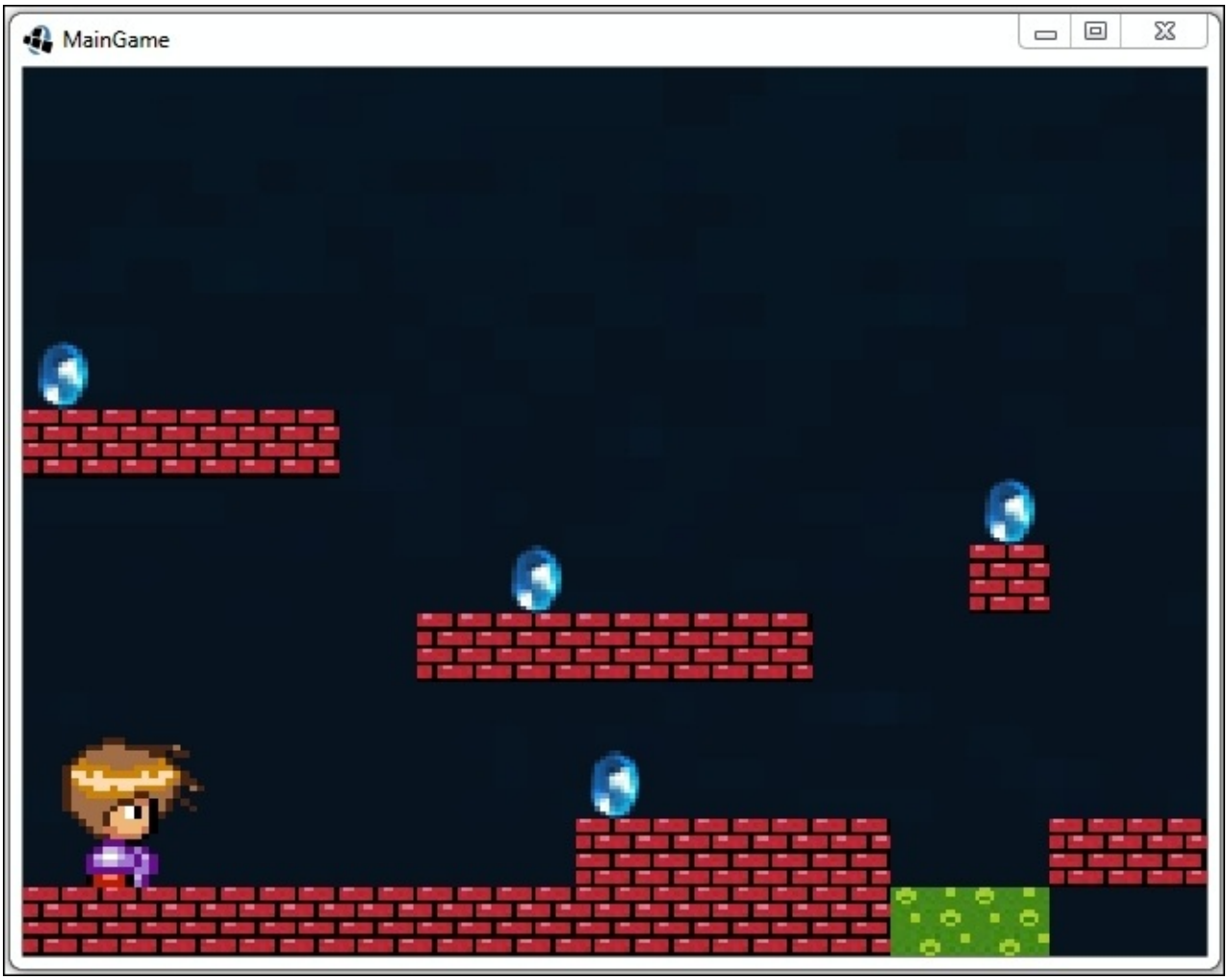


If you take a look at the bottom-left corner, that is where our map is. And that is the area the camera focuses on and zooms in on. As the other objects are too big or out of the view of the camera, we cannot see them. Revert to those changes (the unit scale and camera viewport), and then we will focus on how to display the game objects in proportion to the map, starting with Bob.

As the unit scale is 1/16, let's try to bring the size of Bob to that scale. In the `Bob` class' `initialize()` method, update the following code:

```
//set the size of the bob
bobSprite.setSize((walkSheet.getRegionWidth()/ANIMATION_FRAME_SIZE)*
(width/BOB_RESIZE_FACTOR),walkSheet.getRegionHeight()*
(width/BOB_RESIZE_FACTOR));
//scale bob's size w.r.t unit scale
bobSprite.setSize(bobSprite.getWidth()*GameConstants.unitScale,bobSprite.ge
tHeight()*GameConstants.unitScale);
// set the position of the bob to bottom - left
setPosition(0, 0);
```

The screen will now look like this:



Now, we are getting somewhere. Let's resize him a bit more and set the position to 2 units up. Update the resize factor in the `Bob` class:

```
public static final float BOB_RESIZE_FACTOR = 700f;
```

Also, set the position in the `Bob` class' `initialize()` method:

```
bobSprite.setSize(bobSprite.getWidth()*GameConstants.unitScale,bobSprite.ge
tHeight()*GameConstants.unitScale);
// set the position of the bob to bottom - left
setPosition(0, 2);
```

The screen will now look like this:

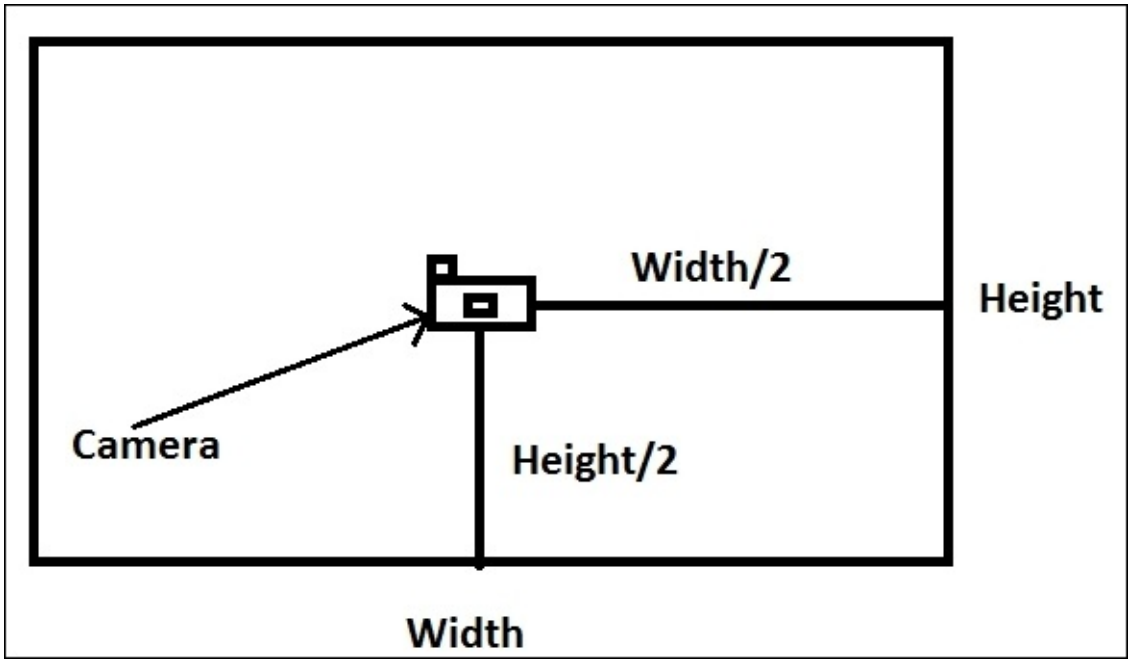
If you now try to make Bob walk, you will notice that he moves too fast and too far on the screen. Let’s try to normalize this. Update the `X_MOVE_UNITS` constant:

```
private static final float X_MOVE_UNITS = 0.1f; // units bob will move in x direction
```

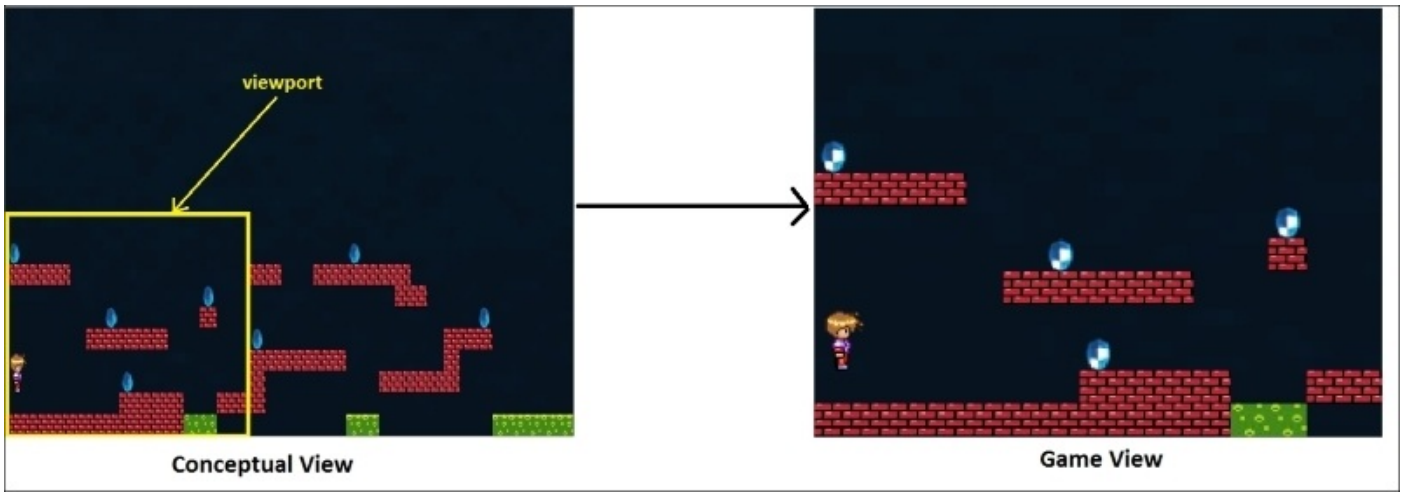
If you run the game now, you can see Bob moving at an appropriate pace.

Camera control

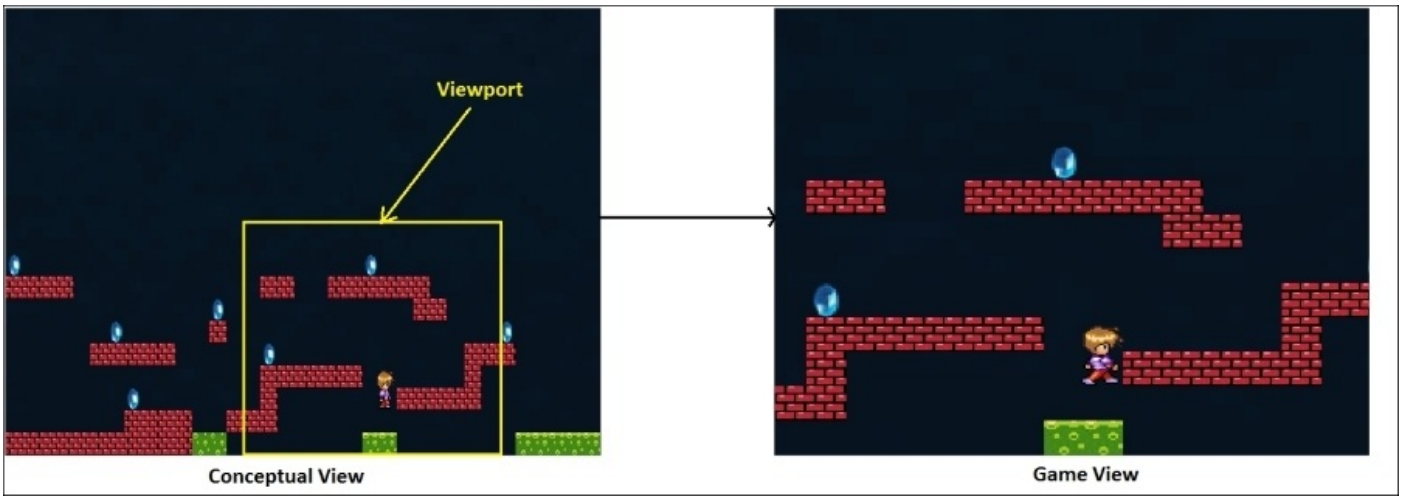
When we move the player, he goes off the screen when he is at the edge. We need a way to restrict this. We will also need a way to show different parts of the map when the player moves. To do this, we first need to understand how the camera is positioned:



As shown in the preceding diagram, the camera is positioned at the center of the viewport window. In order to make more areas of the map visible on the screen, we need to move the camera as well. This would move the viewport window along with it and we would get the desired effect:



When we move the player with the camera, this is how it looks:



First, let's declare the map's width and height as variables in the `GameManager` class:

```
public static int mapWidth;
public static int mapHeight;
```

Next, we will use the techniques to read map properties from the previous chapter to set the values of these variables. We will add a new method called `setMapDimensions()` to the `GameManager` class. This method will read the level's height and width and update the variables accordingly:

```
static void setMapDimensions(){
    MapProperties properties = map.getProperties();
    mapHeight = Integer.parseInt(properties.get("height").toString());
    mapWidth = Integer.parseInt(properties.get("width").toString());
}
```

This method will be called in the `GameManager` class' `initialize()` method:

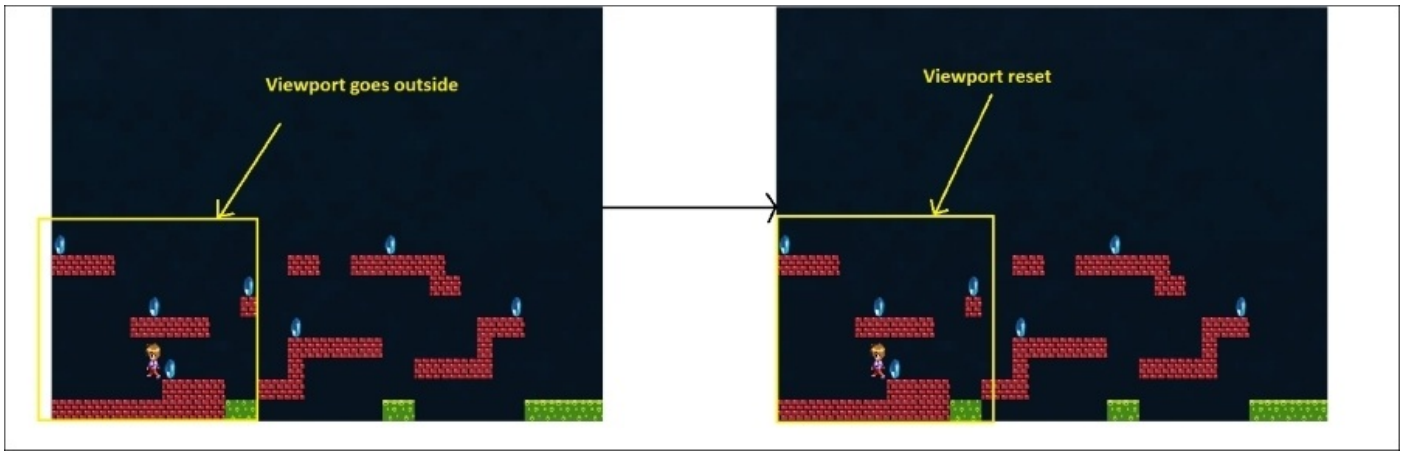
```
map = assetManager.get(GameConstants.level1);
setMapDimensions();
```

In the `renderGame()` method, add these lines of code so that the camera scrolls with the player:

```
bob.render(batch);
//update the camera's x position to Bob's x position
GameScreen.camera.position.x= bob.bobSprite.getX();
//if the viewport goes outside the map's dimensions update the camera's
position correctly
if(!((GameScreen.camera.position.x-GameScreen.camera.viewportWidth/2)>0)){
    GameScreen.camera.position.x = GameScreen.camera.viewportWidth/2;
}
else
if(((GameScreen.camera.position.x+GameScreen.camera.viewportWidth/2)>=mapWi
dth)){
    GameScreen.camera.position.x = mapWidth -
GameScreen.camera.viewportWidth/2;
}
```

```
renderer.setView(GameScreen.camera);
GameScreen.camera.update();
```

The first line of the code sets the camera's *x* coordinate to Bob's *x* coordinate. This gives an effect of the camera following the player in the *x* direction. The next `if else` statement checks whether the camera's viewport goes outside the map's bounds when Bob is at the edge. If it does, we just reset the camera's position so that it remains within those boundaries:



Run the game now to see the scrolling effect in action. Our player Bob still goes off the screen and there is no check to prevent this. Instead of a boundary check, we will use tile collisions to handle this, which will be shown later.

Integrating game objects

Let’s now try to integrate the paddles and background in the game. To display them, we can do the same thing that we did for Bob, that is, change their dimensions and coordinates. This would cause a problem when we move the camera and they would be out of view. We want them to be on the screen at all times. We can always move them with the camera, which would be a hassle. There is an easy way to avoid all this; we can use a secondary camera. This camera is just used to display the nonmoving objects on the screen.

We declare the secondary camera in the `GameScreen` class. We will call this `hudCamera`:

```
public static OrthographicCamera camera,hudCamera;
```

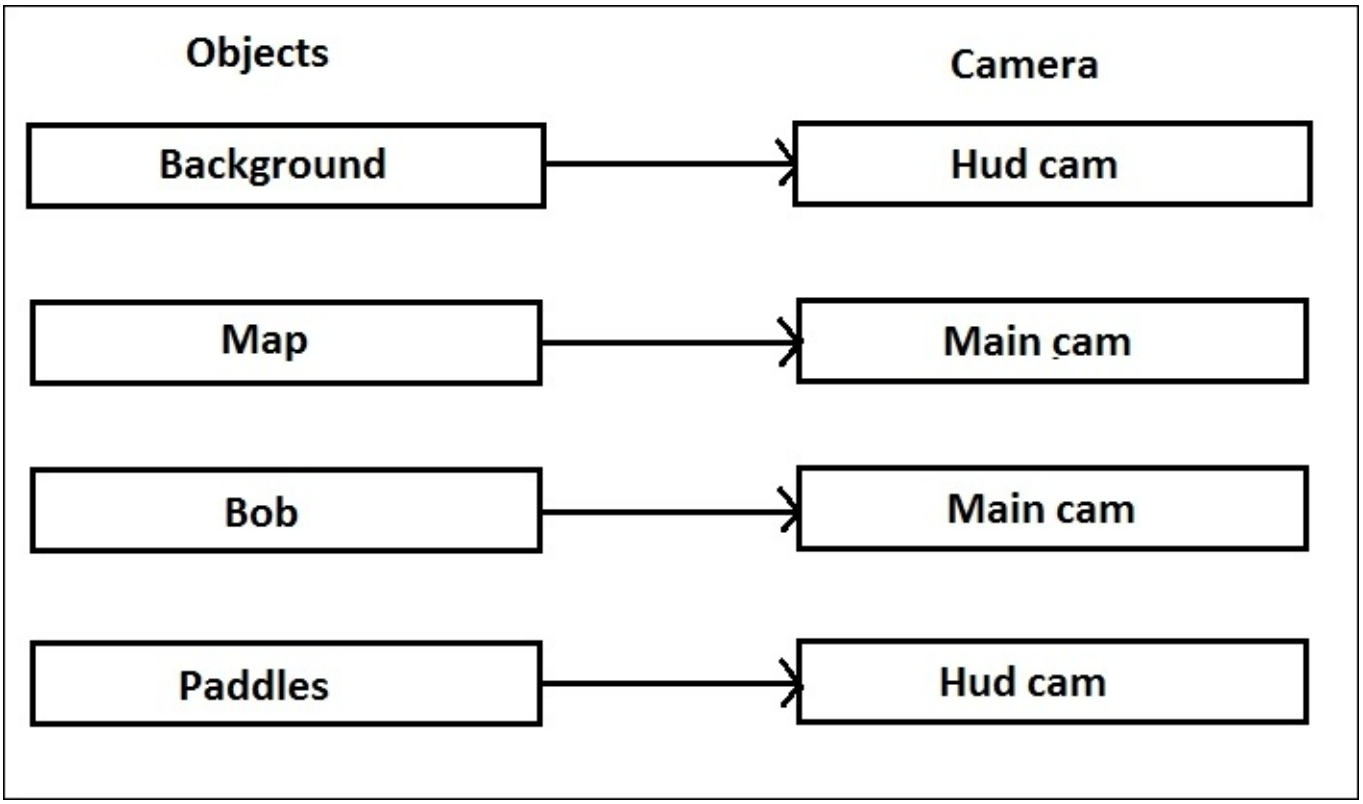
We then instantiate and initialize it in the constructor:

```
// set our hud camera's viewport to window dimensions
hudCamera = new OrthographicCamera(width,height);
// center the camera at w/2,h/2
hudCamera.setToOrtho(false);
```

Since we are using `hudCamera` for the paddles, we would want to couple our input manager with this camera instead of the main one:

```
Gdx.input.setInputProcessor(new InputManager(hudCamera)); // enable
InputManager to receive input events
```

The following diagram shows how we are going to display the objects and camera, which would be coupled with them:



To get this order, we first need to separate the drawing of the background into a separate method in the `GameManager` class:

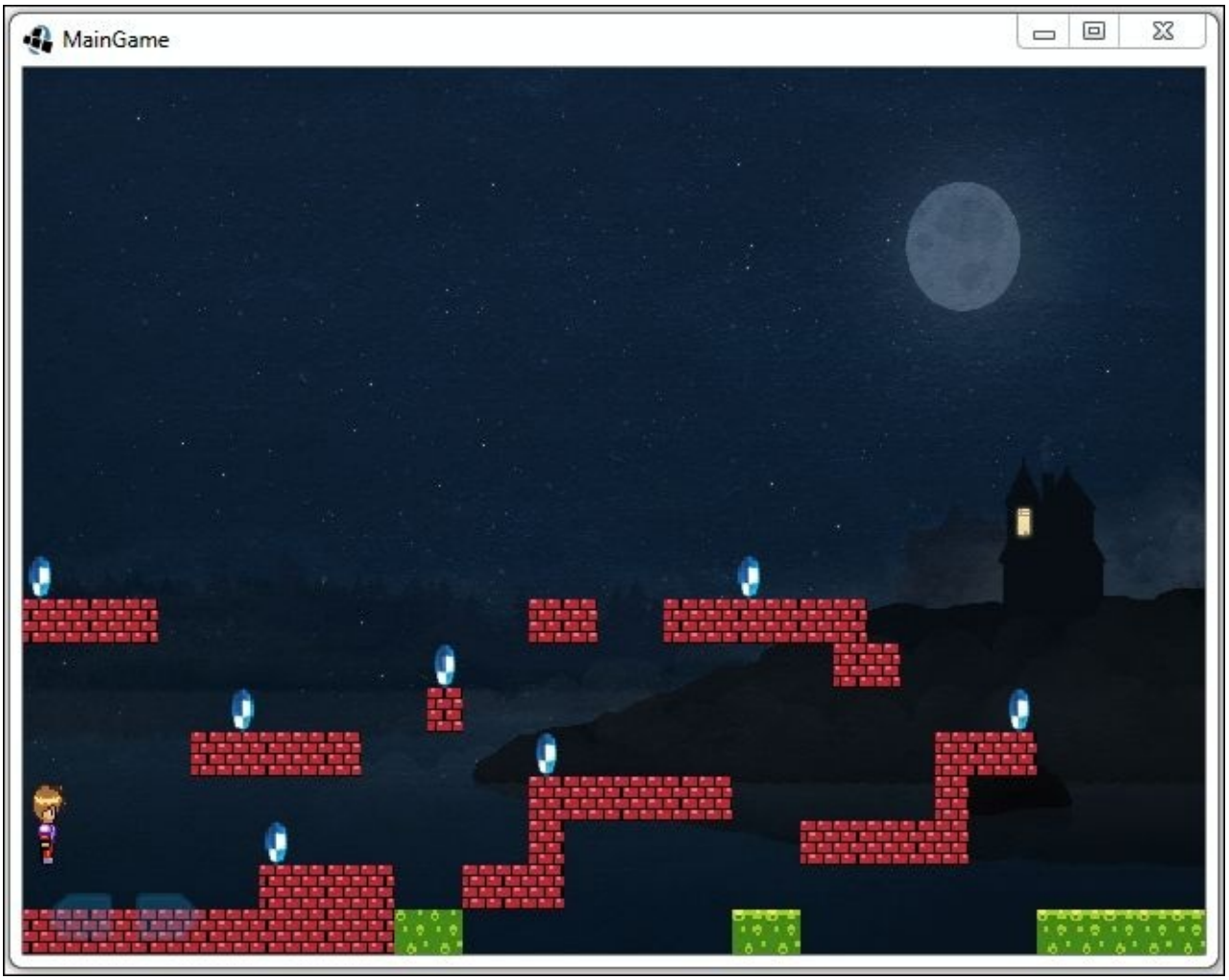
```
public static void renderBackground(SpriteBatch batch){
    // draw the background
    backgroundSprite.draw(batch);
}

public static void renderGame(SpriteBatch batch){
    //draw the Bob with respect to main cam
    batch.setProjectionMatrix(GameScreen.camera.combined);

    bob.update();
    // Render(draw) the bob
    bob.render(batch);
    //update the camera's x position to Bob's x position
    GameScreen.camera.position.x= bob.bobSprite.getX();
    //if the viewport goes outside the map's dimensions update the camera's
position correctly
    if(!((GameScreen.camera.position.x-
(GameScreen.camera.viewportWidth/2))>0)){
        GameScreen.camera.position.x = GameScreen.camera.viewportWidth/2;
    }
    else if(((GameScreen.camera.position.x+
(GameScreen.camera.viewportWidth/2))>=mapWidth)){
        GameScreen.camera.position.x = mapWidth -
GameScreen.camera.viewportWidth/2;
    }

    renderer.setView(GameScreen.camera);
    GameScreen.camera.update();

    //draw the paddles with respect to hud cam
    batch.setProjectionMatrix(GameScreen.hudCamera.combined);

    leftPaddleSprite.draw(batch);
    rightPaddleSprite.draw(batch);

}
```

Then, we update the `GameScreen` class' `render()` method, as shown in the following code:

```
public void render(float delta) {
    // Clear the screen
    Gdx.gl.glClearColor(1, 1, 1, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    // set the spritebatch's drawing view to the hud camera's view
    batch.setProjectionMatrix(hudCamera.combined);

    batch.begin();
    GameManager.renderBackground(batch);
    batch.end();

    GameManager.renderer.render();
```

```
        batch.begin();
        GameManager.renderGame(batch);
        batch.end();

}
```

What we basically do here is couple the appropriate camera with the batch and then draw the game objects. If you run the game now, you can see all the game objects:



Physics and collision

In this section, we will cover how to add realistic physics to the game and collision detection.

Adding physics

Let’s add more realistic physics to our game. We will need to keep track of Bob’s velocity. Add a `velocity` variable to the `Bob` class:

```
Vector2 velocity; // Bob's velocity
```

Instantiate and initialize it in the `initialize()` method:

```
velocity = new Vector2(0, 0);
```

Now, when we move the player, we set the velocity to a predefined value. Let’s call it `maxVelocity` and change the `X_MOVE_UNITS` constant to `maxVelocity`:

```
private static final float maxVelocity = 0.1f;
```

Now, let’s update the player’s velocity when we press the arrow keys/touch paddles in the `update()` method:

```
// move specified units to left if left key is pressed
if (isLeftPressed){
    direction=Direction.LEFT;
    velocity.x=-maxVelocity;
}

// move specified units to right if right key is pressed
else if (isRightPressed){
    direction=Direction.RIGHT;
    velocity.x=maxVelocity;
}

// move specified units to left if left paddle is touched
if (isLeftPaddleTouched){
    direction=Direction.LEFT;
    velocity.x=-maxVelocity;
}

// move specified units to right if right paddle is touched
else if (isRightPaddleTouched){
    direction=Direction.RIGHT;
    velocity.x=maxVelocity;
}
```

We removed the call to move here, as Bob would be moving according to the velocity he currently has. We also removed the setting of the `updateAnimationstateTime` variable, as it will be set according to the velocity. We update Bob’s animation `stateTime` value only when he is moving, that is, when his velocity is not 0. Add the following line of code to make that effect:

```
// if bob is not at rest, animate him
if(velocity.x!=0){
    updateAnimationStateTime=true;
}
move(velocity.x, velocity.y); // update Bob's position according to
velocity
```

If you run the game now and try to move the player, you will observe that he continues to walk even if you don't press the button. This is because once he is in motion, there is nothing to restrict him. We would need something such as friction or a damping effect so that he can eventually slow down and come to a halt. Let's add a constant named `damping` to denote this:

```
private static final float damping= 0.03f;
```

To take it into effect, add these lines of code to the `update()` method of the `Bob` class:

```
//reduce bob's velocity if he is not at rest by damping factor
if(velocity.x<0){
    velocity.x+=damping;
}
else if(velocity.x>0) {
    velocity.x-=damping;
}
// if bob is not at rest, animate him
if(velocity.x!=0){
    updateAnimationStateTime=true;
}
```

Here, we first check whether he is moving to the left; if he is, then we add a damping force to the right direction. If he is moving to the right, then we add it to the left direction. If you run the game now, you can indeed see Bob slowing after a short while. But he also moves in the backward direction. This is because his velocity never remains at 0 after damping as it continues unconditionally. To fix this, we will stop his motion as soon as his velocity becomes too low:

```
//if bob's velocity becomes too low make it 0 so that he stays at rest
if(direction==Direction.RIGHT && velocity.x<=0.02f){
    velocity.x=0.0f;
}

else if(direction==Direction.LEFT && velocity.x>=-0.02f){
    velocity.x=0.0f;
}
// if bob is not at rest, animate him
if(velocity.x!=0){
    updateAnimationStateTime=true;
}
```

If you run the game now, you can see the player moving properly on the screen. Now, let's add gravity to the game. Let's define the `gravity` value as a constant:

```
private static final Vector2 gravity = new Vector2(0, -0.02f);
```

As the gravity acts only in the downward *y* direction, we have given the value as (0, -0.2) to the `gravity` vector. Applying the gravity to Bob is very easy. You just need to add this line of code to the `Bob` class' `update()` method:

```
velocity.add(gravity); // factor gravity into Bob's velocity
move(velocity.x, velocity.y); // update Bob's position according to
velocity
```

The gravitational force affects Bob’s velocity. This is why we add both these vectors to every frame. If you run the game now, you can see Bob falling straight down. This is because we have not done any collision detection between Bob and the tiles.

Collision detection – 1

Let’s talk about collision detection. We need to first check whether the player collides with the walls. One strategy would be to check the collision between the player and all the wall tiles at every frame. This would be inefficient as a lot of computations would need to be performed at every frame, which would lower the game’s performance. Our strategy would be to only consider the tiles that are near the player for collision detection.

Let’s create a utility method to get tiles from the map lying within a given range. Create a new package named `com.packtpub.dungeonbob.utils` and add a new class named `MapUtils` to it. Type the following code in the class:

```
package com.packtpub.dungeonbob.utils;

import com.badlogic.gdx.maps.tiled.TiledMap;

public class MapUtils {

    public static TiledMap map;
    public static void initialize(TiledMap map){
        MapUtils.map= map;
    }

}
```

We declare a reference to the map; there is an `initialize()` method so that we can set the reference. Add the following lines of code to the class as well:

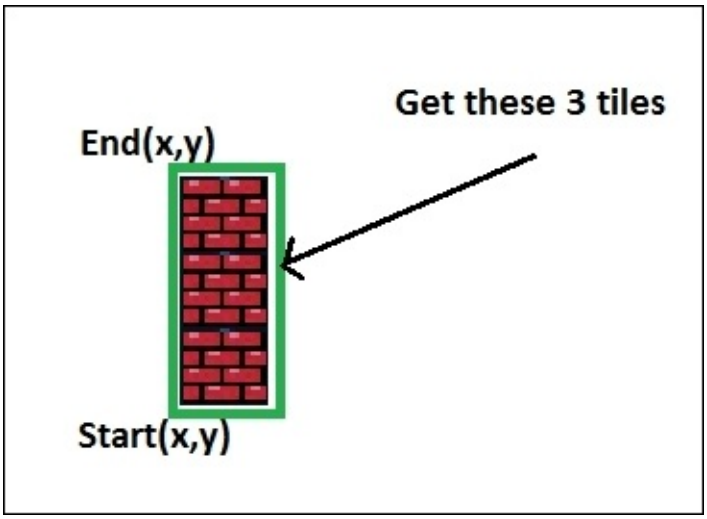
```
// create a pool of rectangle objects for collision detection
private static Pool<Rectangle> rectPool = new Pool<Rectangle>() {
    @Override
    protected Rectangle newObject () {
        return new Rectangle();
    }
};
// denotes a list of tiles which are likely to collide with the player
private static Array<Rectangle> tiles = new Array<Rectangle>();
```

Here, we create a pool of rectangle objects and a list of tiles, which are likely to collide with the player. Now, we’ll add a method to actually get the tiles:

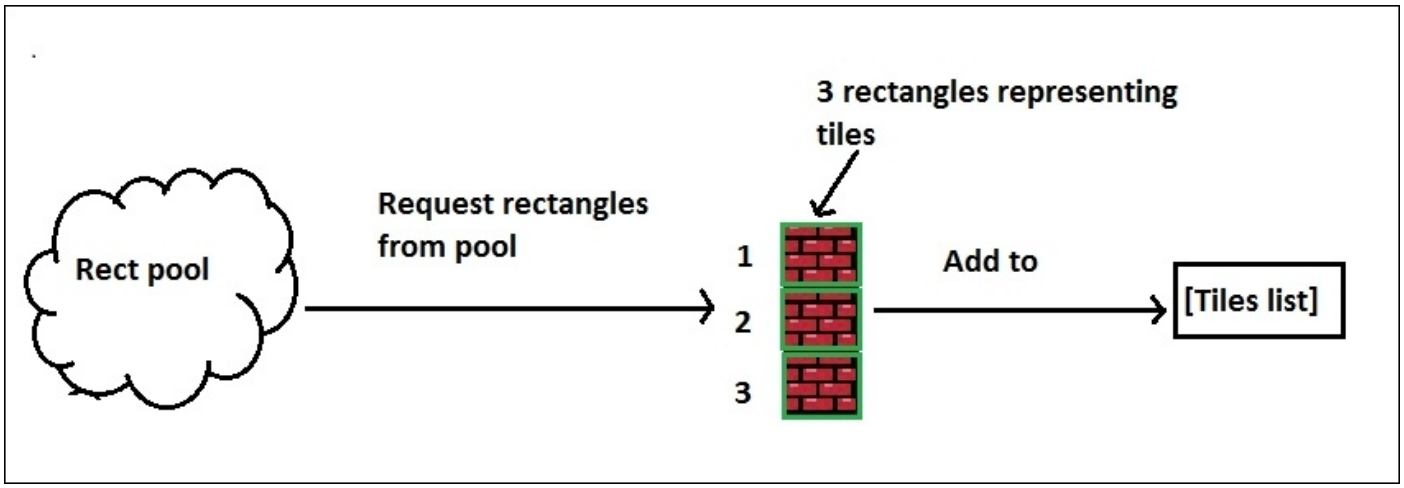
```
public static Array<Rectangle> getTiles (int startX, int startY, int endX,
int endY, String layerName) {

    TiledMapTileLayer layer =
(TiledMapTileLayer)map.getLayers().get(layerName);
    // return the rectangle objects to the pool from previous frame
    rectPool.freeAll(tiles);

    tiles.clear();

    for (int y = startY; y <= endY; y++) {
        for (int x = startX; x <= endX; x++) {
```

```
            Cell cell = layer.getCell(x, y);
            //if cell is present at a particular location in the map,
            if (cell != null) {
                //add a rectangle object representing its position and
dimensions to the tiles list
                Rectangle rect = rectPool.obtain();
                rect.set(x, y, 1, 1);
                tiles.add(rect);
            }
        }
    }
    return tiles;
}
```

This method accepts coordinates of start and end points as the first four parameters. We will get the tiles from the map between these points:



The method also accepts `layerName` as a parameter. As you might have items and collectibles on different layers, this method should also be able to get them for collision. At the start of this method, we release the rectangles, which were used previously, and clear the list of tiles to start fresh. Next, the `for` loop iterates the tiles between the start and the end point and finds the tiles between them. Once the tile is found, we request a rectangle object from the pool, set its position and dimensions to the same as that of the tile, and add it to the list of tiles:



Don’t forget to call the `initialize()` method in the `GameManager` class’ `initialize()` method:

```
initializeRightPaddle(width,height);

MapUtils.initialize(map);
```

Collision detection – 2

In our `Bob` class, we need a rectangle object to represent a bounding box around him. This will be required to detect collisions between tiles, so let's add this:

```
Rectangle bobRectangle; // represents collision box around Bob
```

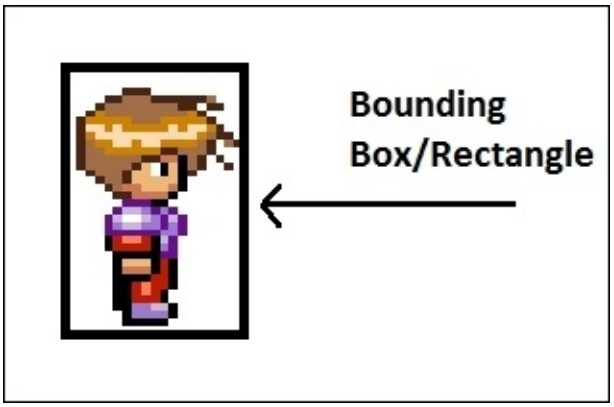
We instantiate it in the `initialize()` method:

```
bobRectangle = new Rectangle();
```

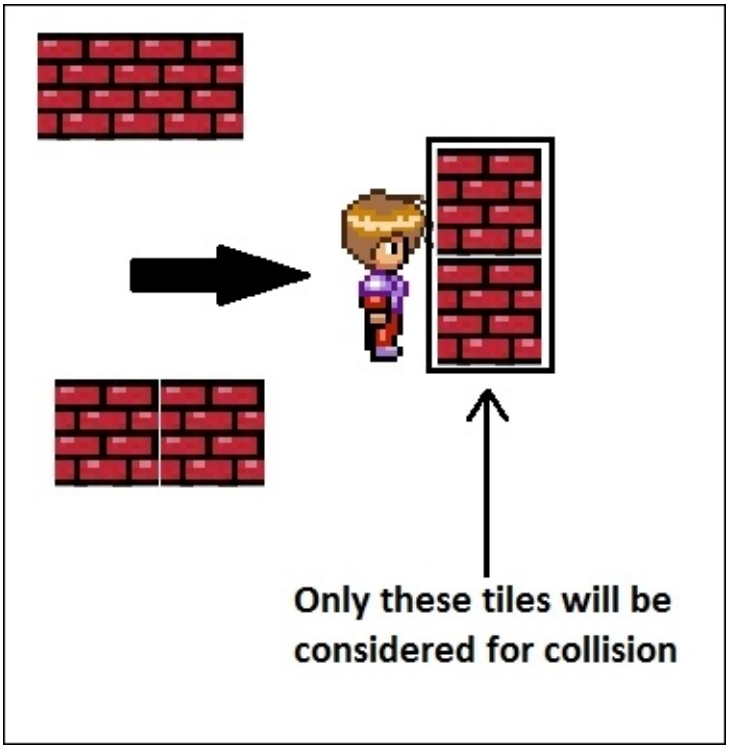
We'll create a new method to detect whether Bob has collided with the walls. Add a method to the `Bob` class with the `checkWallHit()`name and type the following code:

```
public void checkWallHit(){

    // set the bob's bounding rectangle to its position and dimensions
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
bobSprite.getWidth(), bobSprite.getHeight());
    int startX, startY, endX, endY;
    //if bob is moving right, get the tiles to his right side
    if (velocity.x > 0) {
        startX = endX = (int)(bobSprite.getX() +
bobSprite.getWidth()+velocity.x);
    }
    //if bob is moving left, get the tiles to his left side
    else {
        startX = endX = (int)(bobSprite.getX() + velocity.x);
    }
    startY = (int)(bobSprite.getY());
    endY = (int)(bobSprite.getY() + bobSprite.getHeight());

    // get the tiles from map utilities
    Array<Rectangle> tiles =  MapUtils.getTiles(startX, startY, endX,
endY,"Wall");

    // if bob collides with any tile while walking right, stop his
horizontal motion
    for (Rectangle tile : tiles) {
        if (bobRectangle.overlaps(tile)) {

            velocity.x = 0;

            break;
        }
    }
}
```

First of all, we set Bob's bounding rectangle according to its position and dimensions:



In this method, we are detecting the collisions on the horizontal and vertical axis separately. So far, we have written the code for the horizontal axis. Next, we check whether Bob is moving to the right by checking his velocity on the *x* axis. If he is, then we get the tiles to his right:

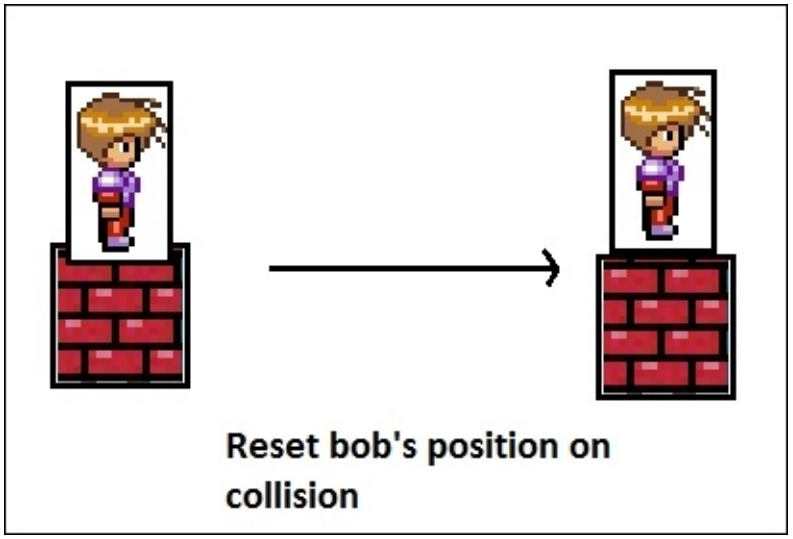


A similar logic is applied when he moves to the left. Once we set the start and end points from where to consider, we pass them to the `getTiles()` function of the `MapUtils` class in order to get their bounding rectangles for collision detection. Once we get them, we check whether any tile collide with Bob. If it does, then we stop his motion. To check for collisions on the vertical axis, add the following lines of code to the same function, as shown here:

```
bobRectangle.x = bobSprite.getX();

//if bob is moving up, get the tiles above him
if (velocity.y > 0) {
```

```
    startY = endY = (int)(bobSprite.getY() + bobSprite.getHeight() );

}
// if bob is moving down, get the tiles below him
else {
    startY = endY = (int)(bobSprite.getY() + velocity.y);
}
startX = (int)(bobSprite.getX());
endX = (int)(bobSprite.getX() + bobSprite.getWidth());
// get the tiles from map utilities
tiles=  MapUtils.getTiles(startX, startY, endX, endY,"Wall");

bobRectangle.y += velocity.y;
for (Rectangle tile : tiles) {
    if (bobRectangle.overlaps(tile)) {
        // we reset the Bob's y-position here
        // so it is just below/above the tile we collided with

        if (velocity.y > 0) {
            bobSprite.setY( tile.y - bobSprite.getHeight() );

        } else if(velocity.y < 0) {
            bobSprite.setY( tile.y + tile.height);

        }
        velocity.y = 0;
        break;
    }
}
```

Here, we follow a similar procedure to detect collisions, except that the `y` component of Bob's velocity is used here. Once we detect that a tile has collided with Bob, we reset his position so that he is just above or just below that tile:



We need to add this method just before we update Bob's position in the `update()` method:

```
checkWallHit();
move(velocity.x, velocity.y); // update Bob's position according to
```

`velocity`

If you run the game now, you can see that Bob is now unable to pass through walls.

Jumping

Let's now implement a logic to make Bob jump. We will add a new constant called `jumpVelocity` to the `Bob` class:

```
private static final float jumpVelocity = 0.35f;
```

This defines the extent of his jump.

We add a method called `jump` to the `Bob` class and type the following code:

```
public void jump(){
    velocity.y=jumpVelocity;
}
```

To trigger the jump action, we need to map it to a key. In the `InputManager` class' `keyDown()` method, add the following highlighted code:

```
// make bob jump
else if(keycode==Keys.SPACE){
    GameManager.bob.jump();
}

return false;
```

If you run the game, you can make Bob jump by pressing the Space key. We still need an onscreen button for the jump so that it can work on mobile devices. First, add the image for the jump in the packed image using the Texturepacker-GUI tool. Once this is done, add a constant to the `GameConstants` class, representing the name of the button image:

```
public static final String jumpImage = "buttonA";
```

We add the texture and sprite to the button in the `GameManager` class:

```
static TextureRegion jumpButtonTexture;
static Sprite jumpButtonSprite;
```

Next, we will need to write a separate function in the `GameManager` class to initialize it:

```
public static void initializeJumpButton(float width,float height){
    //load jump button texture region
    jumpButtonTexture =  texturePack.findRegion(GameConstants.jumpImage);
    //set jump button sprite with the texture
    jumpButtonSprite= new Sprite(jumpButtonTexture);
    // resize the sprite
    jumpButtonSprite.setSize(jumpButtonSprite.getWidth()*width/
PADDLE_RESIZE_FACTOR,
jumpButtonSprite.getHeight()*width/PADDLE_RESIZE_FACTOR);
    // set the position to bottom right corner with offset
    jumpButtonSprite.setPosition(width*0.9f, height*0.01f);
    // make the button semi transparent
    jumpButtonSprite.setAlpha(0.25f);
}
```

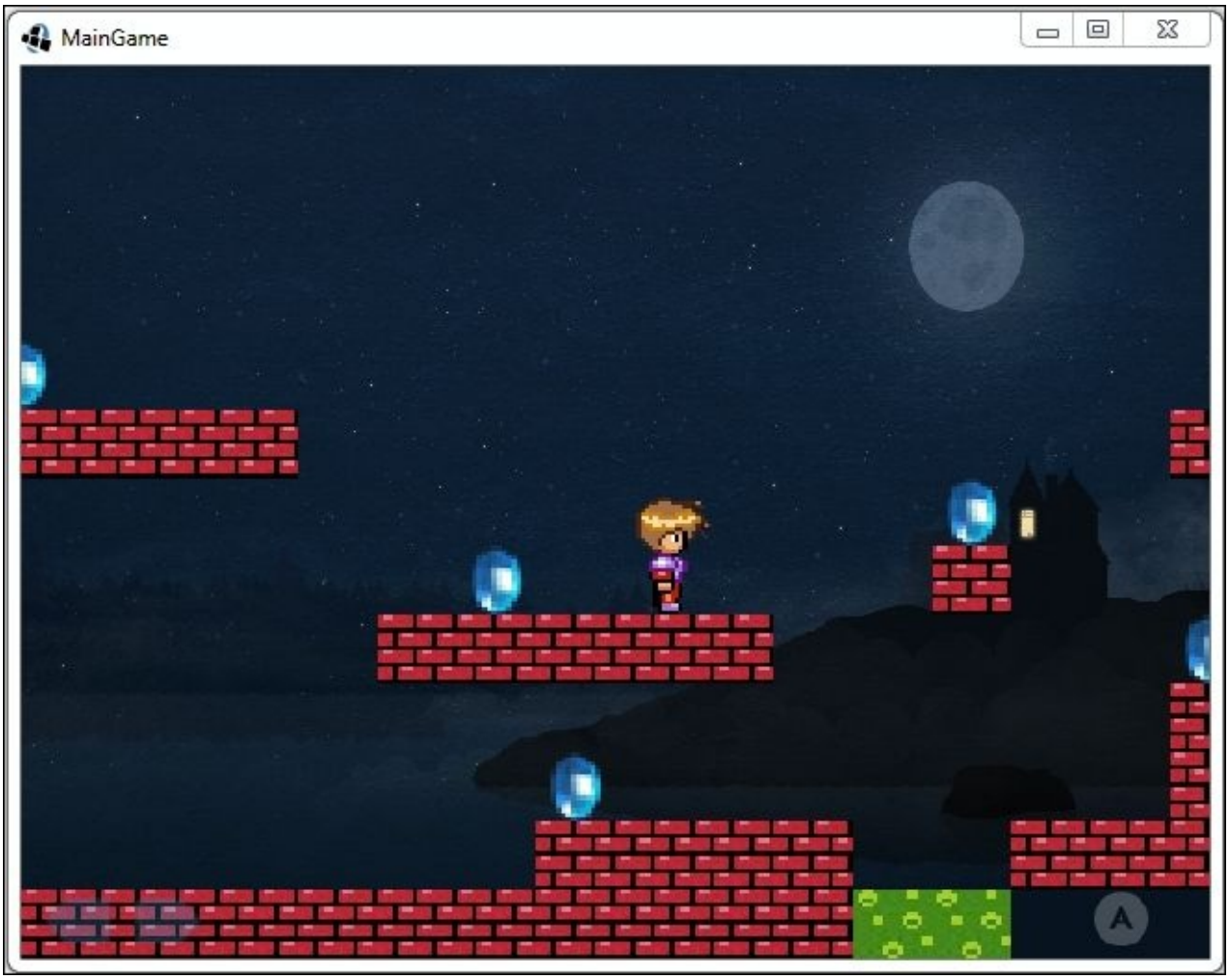
Now, we call it in the `initialize()` method of the same class:

```
initializeJumpButton(width, height);
```

To render it, call its `draw()` method in the `renderGame()` method:

```
rightPaddleSprite.draw(batch);
jumpButtonSprite.draw(batch);
```

If you run the game, you will be able to see the **A** button on the screen:



We still haven't mapped the jump functionality to the button, so let's do it. In the `InputManager` class, we need to add a function that will detect the touch input on the button:

```
boolean isjumpButtonTouched(float touchX, float touchY){
    // handle touch input on the jump button
    if((touchX>=GameManager.jumpButtonSprite.getX()) && touchX<=
(GameManager.jumpButtonSprite.getX()+GameManager.jumpButtonSprite.getWidth(
)) && (touchY>=GameManager.jumpButtonSprite.getY()) && touchY<=
(GameManager.jumpButtonSprite.getY()+GameManager.jumpButtonSprite.getHeight
()) ){
        return true;
    }
    return false;
}
```

In the `touchdown()` method, we need to handle the case when the user taps on the jump

button. Add the highlighted code:

```
else if(isRightPaddleTouched(touchX,touchY)){
    GameManager.bob.setRightPaddleTouched(true);
}
else if(isjumpButtonTouched(touchX, touchY)){
    GameManager.bob.jump();
}
return true;
```

Now, if you run the game, we can jump using the onscreen control too. If you notice, Bob can jump even when he is in the air. We want to restrict the player from jumping if he is not on the ground. To implement this, we need to add a variable called `isGrounded` to the `Bob` class to check whether he is on the ground:

```
boolean isGrounded = false; // denotes whether the player is on the ground
```

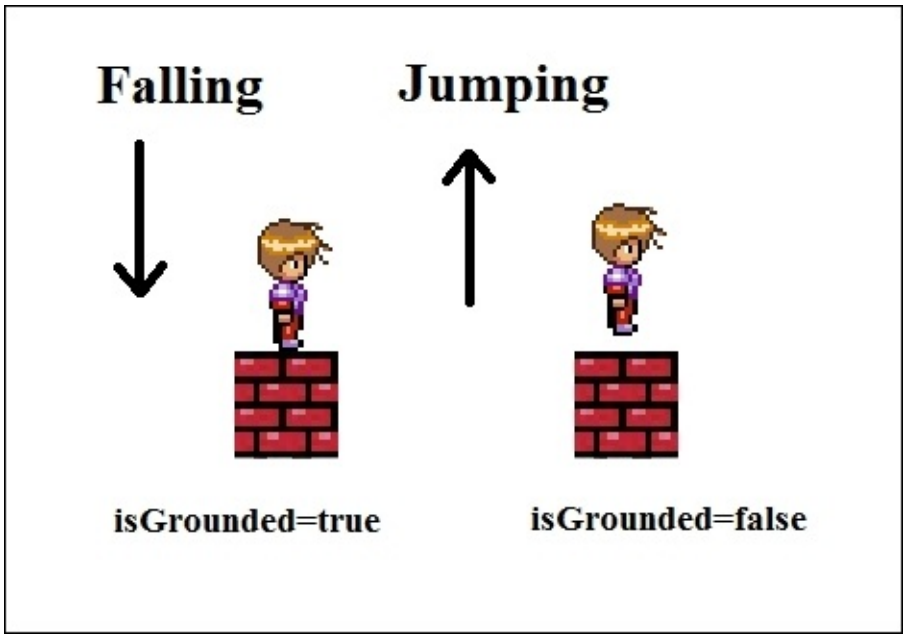
We will set the status to `true` when a collision is detected while falling down. In the `checkWallHit()` function, add the highlighted line to the appropriate section:

```
if (velocity.y > 0) {
    bobSprite.setY( tile.y - bobSprite.getHeight() );

}
else if(velocity.y < 0) {
    bobSprite.setY( tile.y + tile.height);
    isGrounded=true;
}
```

When the player is jumping, he is off the ground. That is, when the status is set to `false`. Add the highlighted line to the `jump()` function:

```
if(isGrounded){
    velocity.y=jumpVelocity;
    isGrounded=false;
}
```

Let’s take a look at the following diagram:



We run the game now to see the desired effect.

Summary

In this chapter, we learned the following topics:

- Integrating Bob and the game objects with the Tiled map
- Camera control and scrolling
- Adding realistic physics
- Collision detection between the walls and Bob
- Implementing the jumping effect

In the next chapter, we will learn how to keep scores and add collectibles and enemies to our game.

Chapter 8. Collectibles and Enemies

In this chapter, we will learn how to collect the items that we placed in the map. We will also see how to add hazardous locations and detect them. We will also learn how to add enemies to our game and make them move around the game world.

In this chapter, we will cover the following topics:

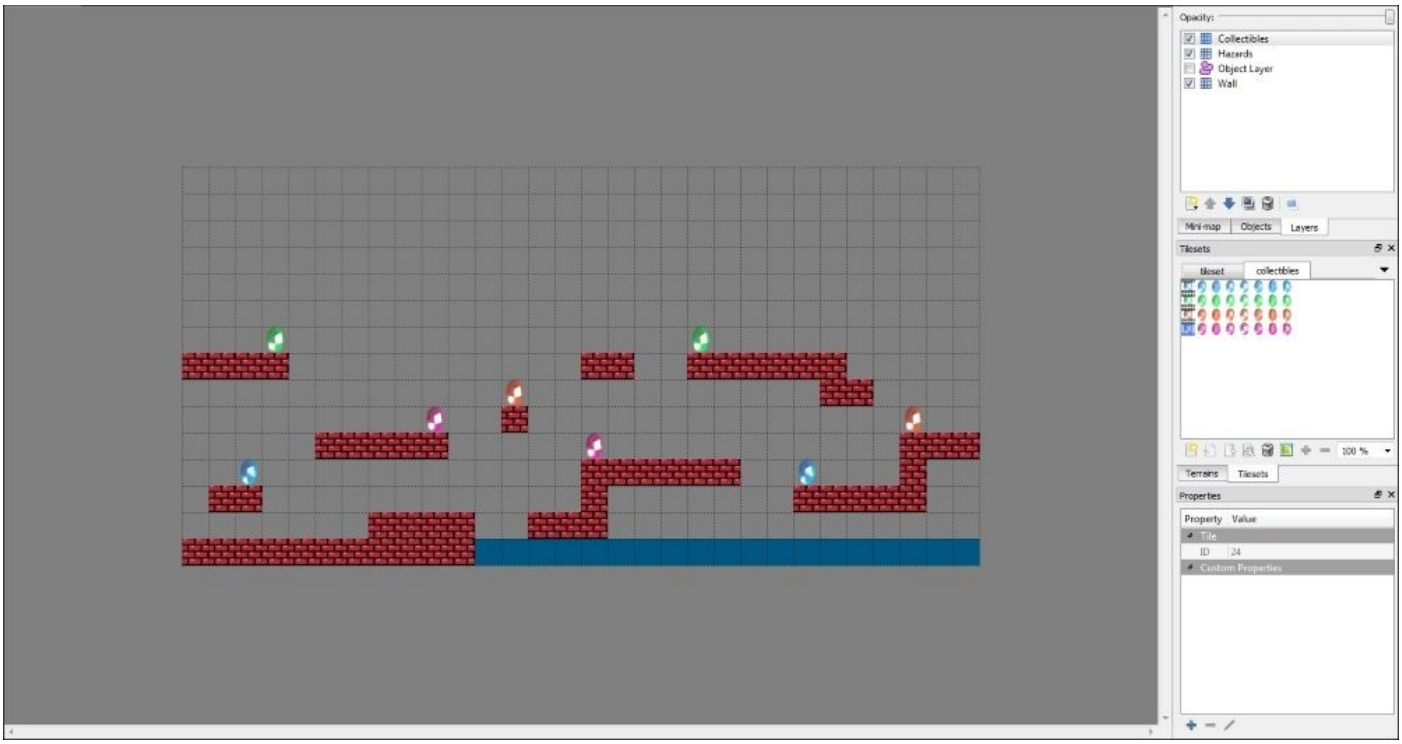
- Collecting objects and detecting hazards
- Enemies

Collecting items and detecting hazards

In this section, we will learn how to collect items in the game, such as crystals, and display scores. We will also learn how to detect hazards in the game.

Collecting objects

In this subtopic, we will learn how to collect objects such as crystals. I have made some changes to the map and it looks like the following screenshot:



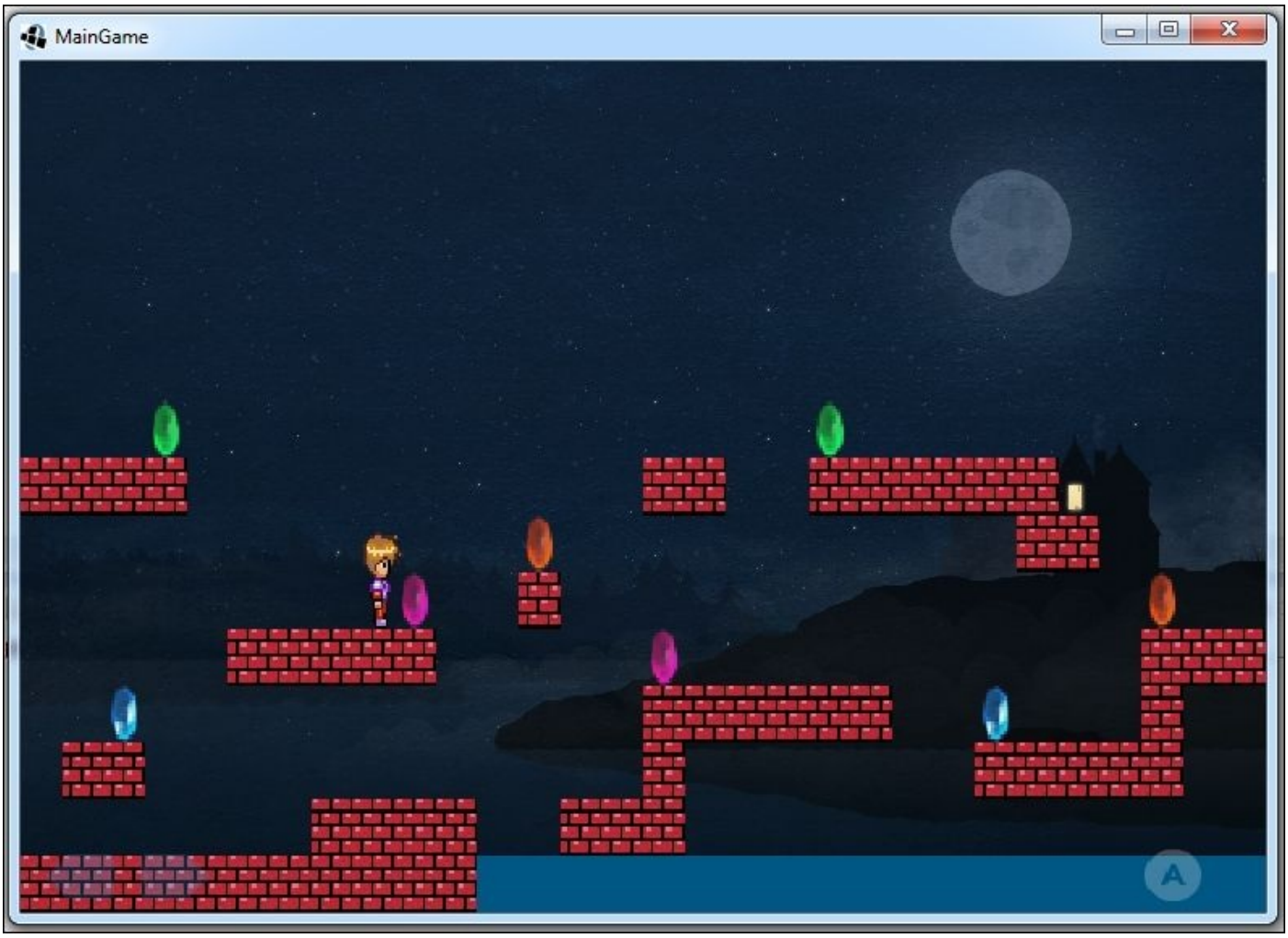
The map is resized to 30 x 15 units. I have used a different layer for the water below called **Hazards**. Tiles on this layer are supposed to be hazardous to the player. If Bob collides with them, he will die. We will see how to implement this later in this chapter. I've added four more crystal animations to the **Collectibles** layer. We will use the full size of the map so that we can view the complete level. Update the following line of code in the `GameManager` class' `initialize()` method, which will use the size of the map as read from its properties:

```
GameScreen.camera.setToOrtho(false,mapWidth,mapHeight);
```

We will also need to update Bob's position in his class' `initialize()` method:

```
setPosition(7, 5);
```

If you run the game now, it will look like the following screenshot:



Before we take a look at how to collect collectible objects, let's first do a little bit of refactoring. We will separate the logic of getting tiles that are near Bob, which we will use for the collision detection. This will come in handy as we can reuse the same logic for any other sprites and to detect collisions for other tile layers as well. We will add two methods to the `MapUtils` class using the following code:

```
/** this method returns the tiles which are near to a sprite horizontally
*/
public static Array<Rectangle> getHorizNeighbourTiles(Vector2
velocity,Sprite sprite,String layerName){
    int startX, startY, endX, endY;
    //if the sprite is moving right, get the tiles to its right side
    if (velocity.x > 0) {
        startX = endX = (int)(sprite.getX() +
sprite.getWidth()+velocity.x);
    }
    //if the sprite is moving left, get the tiles to its left side
    else {
        startX = endX = (int)(sprite.getX() + velocity.x);
    }
    startY = (int)(sprite.getY());
    endY = (int)(sprite.getY() + sprite.getHeight());

    // get the tiles
```

```
    return getTiles(startX, startY, endX, endY,layerName);
}

/** this method returns the tiles which are near to a sprite vertically */
public static  Array<Rectangle> getVertNeighbourTiles(Vector2
velocity,Sprite sprite,String layerName){
    int startX, startY, endX, endY;
    //if sprite is moving up, get the tiles above it
    if (velocity.y > 0) {
        startY = endY = (int)(sprite.getY() + sprite.getHeight() );

    }
    // if sprite is moving down, get the tiles below it
    else {
        startY = endY = (int)(sprite.getY() + velocity.y);
    }
    startX = (int)(sprite.getX());
    endX = (int)(sprite.getX() + sprite.getWidth());
    // get the tiles
    return getTiles(startX, startY, endX, endY,layerName);
}
```

The `checkWallhit()` method in the `Bob` class looks like this:

```
public void checkWallHit(){

    // set the bob's bounding rectangle to its position and dimensions
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
bobSprite.getWidth(), bobSprite.getHeight());

    // get the tiles from map utilities
    Array<Rectangle> tiles = MapUtils.getHorizNeighbourTiles(velocity,
bobSprite, "Wall");

    //if bob collides with any tile while walking right, stop his
horizontal motion
    for (Rectangle tile : tiles) {
        if (bobRectangle.overlaps(tile)) {
            velocity.x = 0;
            break;
        }
    }

    bobRectangle.x = bobSprite.getX();
    tiles = MapUtils.getVertNeighbourTiles(velocity, bobSprite, "Wall");

    bobRectangle.y += velocity.y;
    for (Rectangle tile : tiles) {
        if (bobRectangle.overlaps(tile)) {
            // we reset the Bob's y-position here
            // so it is just below/above the tile we collided with

            if (velocity.y > 0) {
                bobSprite.setY( tile.y - bobSprite.getHeight() );

            }
```

```
            else if(velocity.y < 0) {
                bobSprite.setY( tile.y + tile.height);
                isGrounded=true;
            }
            velocity.y = 0;
            break;
        }
    }

}
```

To start detecting collisions with collectible objects, we will create a new method called `checkCollectibleHit()` in the `Bob` class. Add the following code to the class:

```
public void checkCollectibleHit(){
    // set bob's bounding rectangle to its position and dimensions
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
bobSprite.getWidth(), bobSprite.getHeight());

    // get the tiles from map utilities
    Array<Rectangle> tiles = MapUtils.getHorizNeighbourTiles(velocity,
bobSprite, "Collectibles");

    // get the collectibles layer
    TiledMapTileLayer layer =
(TiledMapTileLayer)GameManager.map.getLayers().get("Collectibles");

    //if bob collides with any tile while walking right, remove it
    for (Rectangle tile : tiles) {
        if (bobRectangle.overlaps(tile)){
            layer.setCell((int)tile.x, (int)tile.y, null);
            break;
            }
    }

    bobRectangle.x = bobSprite.getX();
    tiles= MapUtils.getVertNeighbourTiles(velocity, bobSprite,
"Collectibles");

    bobRectangle.y += velocity.y;
    for (Rectangle tile : tiles) {
        if (bobRectangle.overlaps(tile)) {
            layer.setCell((int)tile.x, (int)tile.y, null);
        }
    }
}
```

This method is very similar to the one that we wrote to detect collisions between the walls. In this section, if Bob is walking to the right, we gather tiles from the **Collectibles** layer, which are to his right and to his left otherwise. If Bob collides with any collectible item, it is deleted from the map using the following line of code:

```
layer.setCell((int)tile.x, (int)tile.y, null);
```

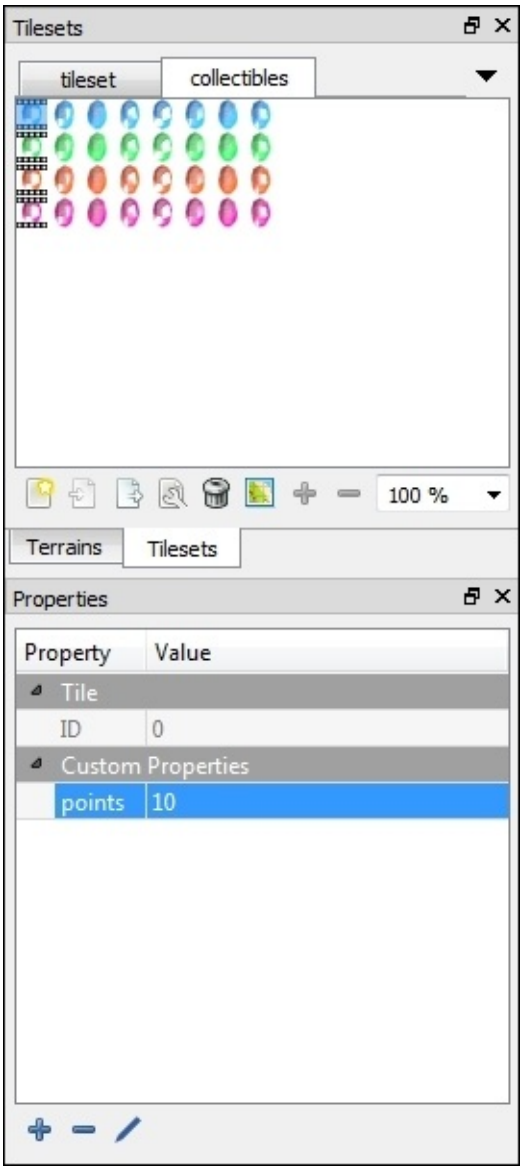
Basically, the cell at that coordinate is set to `null` (deleted) from the layer. Add a call to this method in the `update()` method of the `Bob` class:

```
checkWallHit();
checkCollectibleHit();
```

If you run the game now, you can see the crystals disappear when the player touches them. We haven't implemented a scoring system in our game as of yet. Let's create a new class to store game-related data (scores, high scores, and so on). Create a new class in the `com.packtpub.dungeonbob` package, name it `GameData`, and type the following code:

```
package com.packtpub.dungeonbob;
public class GameData {
    public static int score= 0;
}
```

Now, we'll assign different points to different crystals. In the map editor, add a new custom property called `points` to the **collectibles** tileset. Select the first (animated) tile for each crystal and give different values to `points`:



In the code, we need to update the `checkCollectibleHit()` method and check the points for a particular crystal when it is collected. Add the following lines of code to the loop

where we check for collisions. Remember to add it to both the loops:

```
for (Rectangle tile : tiles) {
    if (bobRectangle.overlaps(tile)) {
        MapProperties tilePoperties=layer.getCell((int)tile.x,
(int)tile.y).getTile().getProperties();
        int
itemPoints=Integer.parseInt(tilePoperties.get("points").toString());
        GameData.score+=itemPoints;
        layer.setCell((int)tile.x, (int)tile.y, null);
        break;
    }
}
```

We first determine the tile that we have hit by calling `getTile()` on the obtained cell. This is done as the properties are set for a tile and not its instance on the map (cell). Once the tile is determined, we get its properties and check the value of its `points` property. We can then get the value and add it to our game score. To verify this, you can add print statements after retrieving the points.

Displaying the score and adding hazards

We will display the game score, as was done previously, albeit a bit differently as we are using `assetManager` here. I have used a custom font called **dos** here. I have copied the `.fnt` file made in the Hiero tool and the corresponding image to the `data/fonts` folder of the Android project. We need to set the path in the `GameConstants` class:

```
public static final String fontPath = "data/fonts/dos.fnt";
```

Next, we need to declare the bitmap font variable in the `GameManager` class:

```
static BitmapFont font;
```

We will need to queue the font for loading in the `GameManager` class' `loadAssets()` method:

```
assetManager.load(GameConstants.fontPath,BitmapFont.class);
```

Now, in the `initialize()` method, we can load the font from `assetManager` after `loadAssets()` is called:

```
font = assetManager.get(GameConstants.fontPath);
```

To actually display the score, we will create a new class. Make a new class in the `com.packtpub.dungeonbob.managers` package, name it `TextManager`, and type in the following code:

```
package com.packtpub.dungeonbob.managers;
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.dungeonbob.GameData;

public class TextManager {
    static BitmapFont font ; // we draw the text to the screen using this
variable
    // viewport width and height
    static float width,height;
    public static void initialize(float width,float height,BitmapFont font)
{
    TextManager.font = font;
    //font = new BitmapFont();
    TextManager.width = width;
    TextManager.height= height;
    //set the font color to red
    font.setColor(Color.RED);
    //scale the font size according to screen width
    font.setScale(width/1000f);

}
public static void displayMessage(SpriteBatch batch){
    float fontWidth =  font.getBounds( "Score: "+GameData.score).width; //
get the width of the text being displayed
    //top the score display at top right corner
    font.draw(batch, "Score: "+GameData.score, width - fontWidth -
```

```
width/15f,height*0.98f);
    }
}
```

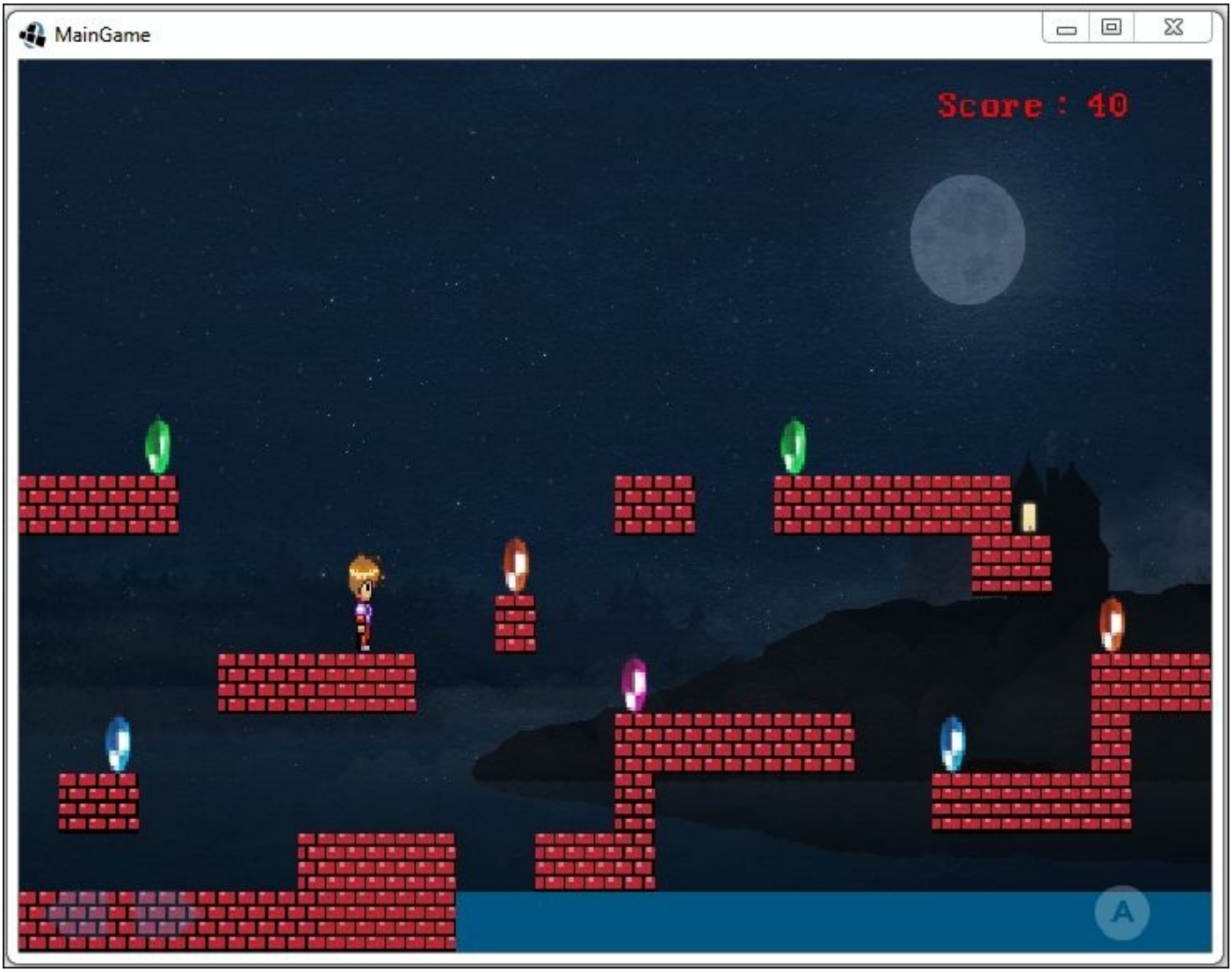
Next, we need to call the `initialize()` method of the `TextManager` class in the `GameManager` class' `initialize()` method:

```
MapUtils.initialize(map);
TextManager.initialize(width, height, font);
```

Finally, we need to call the `TextManager` class' `displayMessage()` method in the `GameManager` class' `renderGame()` method. Add this at the very end of the code, as we will need it to display the font with respect to the HUD cam:

```
jumpButtonSprite.draw(batch);
TextManager.displayMessage(batch);
```

If you run the game now, you can see the score being displayed:



Now, let's work on how to make our character die when he collides with any hazard. In this level, the water below is a hazard. So, we need to check collisions with the **Hazards** layer and act accordingly. Add a new method named `checkHazards()` to the `Bob` class:

```
public void checkHazards(){
    // set the bob's bounding rectangle to its position and dimensions
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
bobSprite.getWidth(), bobSprite.getHeight());
    // get the tiles from map utilities
    Array<Rectangle> tiles = MapUtils.getHorizNeighbourTiles(velocity,
bobSprite, "Hazards");
    //if bob collides with any tile while walking right, check the points
and update score
    for (Rectangle tile : tiles) {
        if (bobRectangle.overlaps(tile)) {
            Gdx.app.exit();
        }
    }

    bobRectangle.x = bobSprite.getX();
    tiles= MapUtils.getVertNeighbourTiles(velocity, bobSprite, "Hazards");

    bobRectangle.y += velocity.y;
    for (Rectangle tile : tiles) {
        if (bobRectangle.overlaps(tile)) {
            Gdx.app.exit();
        }
    }
}
```

In this method, we quit the game if we come across any hazard. Add this method call to the `update()` method of the `Bob` class:

```
checkCollectibleHit();
checkHazards();
```

Instead of quitting the game, we can also respawn Bob after he dies. Let's first define his spawn point in the `GameConstants` class:

```
public static final Vector2 spawnPoint = new Vector2(8,6);
```

We need to change the line where we set Bob's initial position in the `initialize()` method, as shown in the following code:

```
setPosition(GameConstants.spawnPoint.x, GameConstants.spawnPoint.y);
```

Now, to respawn Bob when he collides with any hazard, we will set his position to the spawn point instead of `Gdx.app.exit();`:

```
if (bobRectangle.overlaps(tile)) {
    setPosition(GameConstants.spawnPoint.x, GameConstants.spawnPoint.y);
```

If you run the game now, you can see Bob coming out from his starting point when he touches the water below. To further enhance this concept, we can add extra lives to Bob. He will respawn if he has any lives left; otherwise, the game will quit. First, let's add this variable to the `GameData` class:

```
public static short lives= 3;
```

Now, in the `checkHazards()` method, update the part where we check for collisions, as

shown here:

```
if(bobRectangle.overlaps(tile)) {
    if(GameData.lives>0){
        setPosition(GameConstants.spawnPoint.x,
GameConstants.spawnPoint.y);
        GameData.lives--;
        break;
    }
    else{
        Gdx.app.exit();
    }
}
```

Remember to update this in all the places where we collide with hazards. If you run the game now, you will see Bob respawning until his lives are exhausted. We also need to display the number of lives in the game, which we will do using `TextManager`. Add this line of code to its `displayMessage()` method:

```
// show the number of lives at top left corner
font.draw(batch, "Lives: "+GameData.lives, width*0.01f,height*0.98f);
```

If you run the game now, it looks like this:

Enemies

In this section, we will learn how to add enemies to the game.

Adding enemies

Let’s add enemies to our game. We will make a base enemy class and then extend this class with the classes for specific enemies. Create a new class named `Enemy` in the `com.packtpub.dungeonbob.gameobjects` class and type the following code:

```
package com.packtpub.dungeonbob.gameobjects;

import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Rectangle;
import com.badlogic.gdx.math.Vector2;
public abstract class Enemy {

    Sprite sprite; // enemy sprite
    Vector2 velocity; // velocity of the enemy
    Rectangle rectangle; // rectangle object to detect collisions
    public abstract void render(SpriteBatch batch);
    public abstract void update();
}
```

The first type of enemy that we are going to create is the zombie. First, we need to add the image to our texture atlas using the Texturepacker-GUI tool and copy the pack and the `.png` file to the project directory. We will refer to the image as `zombie`. So, let’s add a constant to the `GameConstants` class with this name:

```
public static final String zombieImage = "zombie";
```

Now, let’s make the zombie class. Create a new class in `com.packtpub.dungeonbob.gameobjects`, name it `Zombie`, and type the following code:

```
package com.packtpub.dungeonbob.gameobjects;

import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.math.Rectangle;
import com.badlogic.gdx.math.Vector2;
import com.packtpub.dungeonbob.GameConstants;

public class Zombie extends Enemy {

    private static final float RESIZE_FACTOR = 900f;
    @Override
    public void render(SpriteBatch batch) {
        sprite.draw(batch);
    }

    @Override
    public void update() {

    }
    public Zombie(float width,float height,TextureRegion zombieTexture){
```

```
        sprite = new Sprite(zombieTexture);
        sprite.setSize(sprite.getWidth()*(width/RESIZE_FACTOR),
sprite.getHeight()*(width/RESIZE_FACTOR));

sprite.setSize(sprite.getWidth()*GameConstants.unitScale,sprite.getHeight()
*GameConstants.unitScale);
        sprite.setPosition(17, 4);
        velocity = new Vector2(0, 0);

        rectangle = new Rectangle();
    }
}
```

In the `GameManager` class, add a variable to `zombie`:

```
public static Zombie zombie; // zombie instance
```

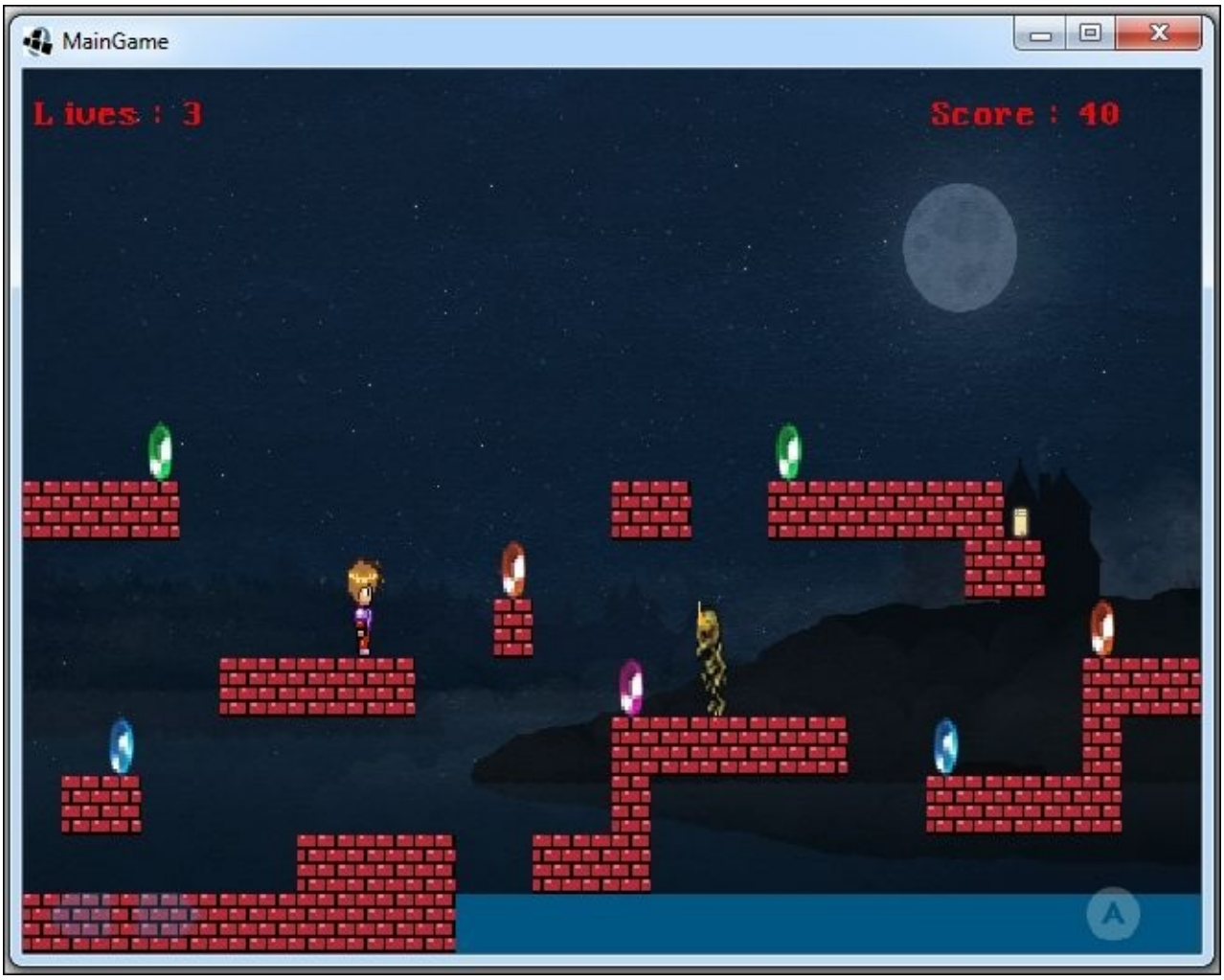
In the `initialize()` method, we will instantiate and initialize the `Zombie` class:

```
// instantiate and initialize zombie with the image from texture atlas
zombie = new Zombie(width, height,
texturePack.findRegion(GameConstants.zombieImage));
```

Now, in order to display it, we will just add a call to its `render()` method in the `renderGame()` function. We will display the zombie with respect to the main camera:

```
bob.render(batch);
zombie.render(batch);
```

Now, if you run the game, you will see the zombie on the screen:



At the moment, it doesn't do anything. The player can even walk over the zombie. We will now add the collision detection. First, let's edit the `update()` method of the `Zombie` class:

```
@Override
public void update() {
    // set the rectangle with zombie's dimensions for collisions
    rectangle.set(sprite.getX(), sprite.getY(), sprite.getWidth(),
sprite.getHeight());
}
```

We call it before we render the zombie in the `GameManager` class:

```
zombie.update();
zombie.render(batch);
```

In the `Bob` class, we create a new method called `checkEnemies()` and type the following code:

```
public void checkEnemies(){
    // set the bob's bounding rectangle to its position and dimensions
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
bobSprite.getWidth(), bobSprite.getHeight());

    // check whether bob collides with the zombie
    if(GameManager.zombie.rectangle.overlaps(bobRectangle)){
```

```
        if(GameData.lives>0){
            setPosition(GameConstants.spawnPoint.x,
GameConstants.spawnPoint.y);
            GameData.lives--;
        }
        else{
            Gdx.app.exit();
        }
    }
}
```

We call this method in the `update()` method:

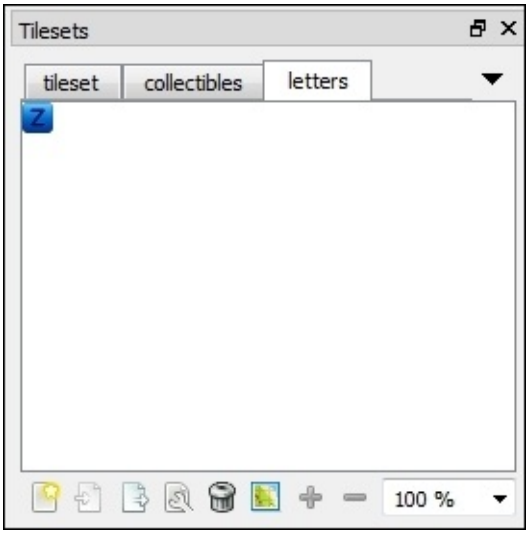
```
checkHazards();
checkEnemies();
```

We check whether Bob has collided with the zombie, and if he does, we apply the same logic as the hazards. If you run the game now, you can see the collision effect in action.

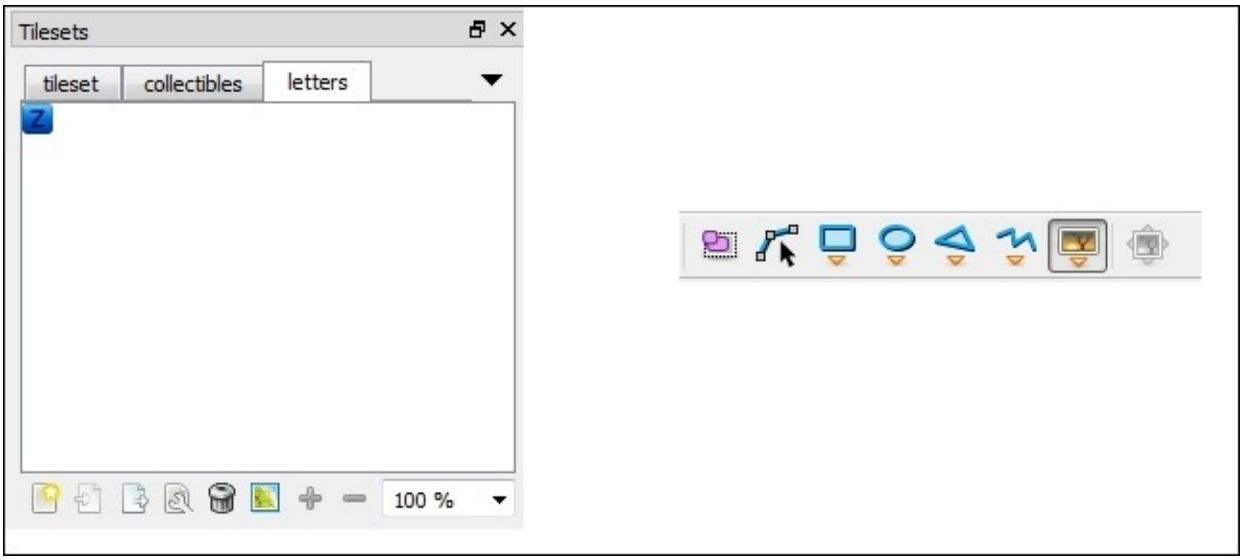
Adding enemies through Tiled

We have just added a single zombie. The game can have multiple zombies. Adding zombies one after the other and manually setting the positions is a very cumbersome task. What if we could visually add the zombies in the level? The answer is Tiled! We are going to learn how to add zombies to our level through the Tiled editor.

First of all, we would need a tileset to represent different letters for different enemies. I have just added the letter *Z* as of now (just a single image of the letter *Z*), but you can add others:

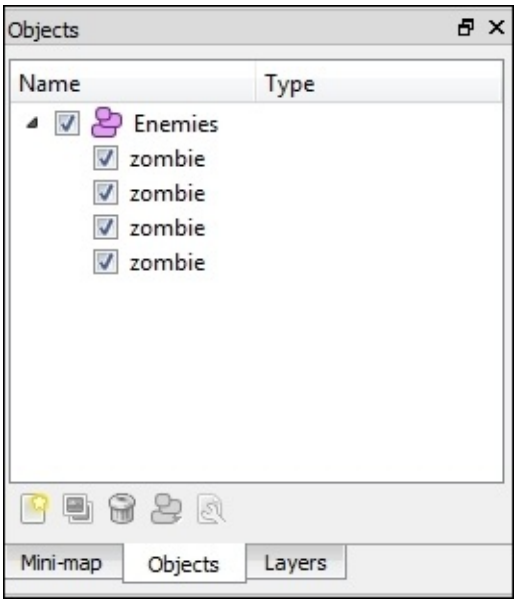


The next thing you need to do is add an object layer called **Enemies**. After selecting the layer, click on the Insert Tile option in the toolbar:



Now, select the **Z** tile and you can start placing it anywhere in the map. These are places where the zombies would spawn:

Now, select the **Objects** tab from the **Layers** section. Give the name `zombie` to these objects so that we can identify them:



Now, save the map and update it in your project. In the code, we need to parse the enemy layer, read the points where we have placed the zombie **Z** icon, and create zombie objects accordingly. In the `Zombie` class, change the initialization process a little bit, as shown in the following code:

```
public Zombie(float width,float height,TextureRegion zombieTexture,float
x,float y){

    sprite = new Sprite(zombieTexture);
    sprite.setSize(sprite.getWidth()*(width/RESIZE_FACTOR),
```

```
sprite.getHeight()*(width/RESIZE_FACTOR));

sprite.setSize(sprite.getWidth()*GameConstants.unitScale,sprite.getHeight()
*GameConstants.unitScale);
    sprite.setPosition(x,y);
    velocity = new Vector2(0, 0);

    rectangle = new Rectangle();
}
```

We now accept `x` and `y` as parameters in the constructor and set the position accordingly. In the `MapUtils` class, we will create a new method to spawn enemies:

```
public static void spawnEnemies(Array<Enemy> enemies,float width, float
height,TextureAtlas texturePack){

    Iterator<MapObject> mapObjectIterator =
map.getLayers().get("Enemies").getObjects().iterator();

    float unitScale= GameConstants.unitScale;
    while(mapObjectIterator.hasNext()){
        // get the map object from iterator
        MapObject mapObject = mapObjectIterator.next();

        // if the name of the object is "zombie" as we have given
        if(mapObject.getName().equals("zombie")){

            Rectangle rectangle =
((RectangleMapObject)mapObject).getRectangle();

            //create a zombie object and place it in that location
            Zombie zombie = new Zombie(width, height,
texturePack.findRegion(GameConstants.zombieImage),rectangle.x*unitScale,rec
tangle.y*unitScale);
            enemies.add(zombie);
        }
    }
}
```

This function iterates over the objects in the map (level) in the **Enemies** layer. It then finds the location of the objects named `zombies`. These are the **Z** blocks that we placed in the Tiled editor. Once such an object is found, we create a `zombie` object at that location and add it to the `enemies` array. This array holds all the enemies present in the map. This is useful as we don't need to keep a separate array for all enemy types as they would extend the `Enemy` base class.

Note that the coordinates returned from the map are pixel coordinates of the object locations. We multiply them with `unitScale` so that they are converted into the world coordinates. Now, we need to make some changes to them so that we can display not one but multiple zombies. Change the zombie instance in the `GameManager` class so that it represents an array of enemies:

```
public static Array<Enemy> enemies; // enemies list
```

In the `initialize()` method, replace the instantiation of a single zombie with these lines:

```
enemies = new Array<Enemy>();
MapUtils.spawnEnemies(enemies,width,height,texturePack);
```

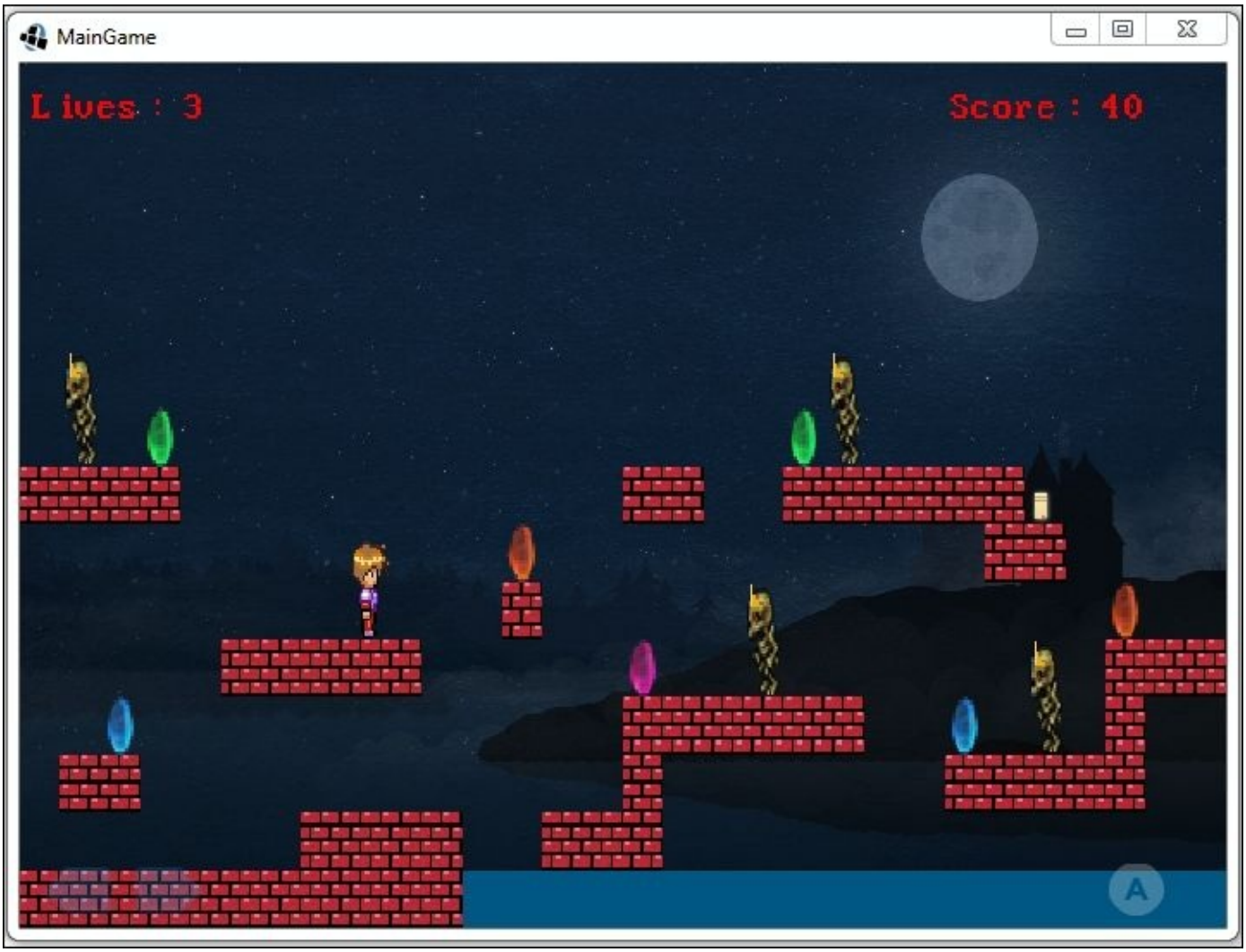
In the `render()` method, change the lines of code where we update and render a single zombie to these lines of code:

```
for(Enemy enemy :enemies){
    enemy.update();
    enemy.render(batch);
}
```

Lastly, in the `Bob` class' `checkEnemies()` method, we iterate over the `enemies` array to check whether there are any collisions:

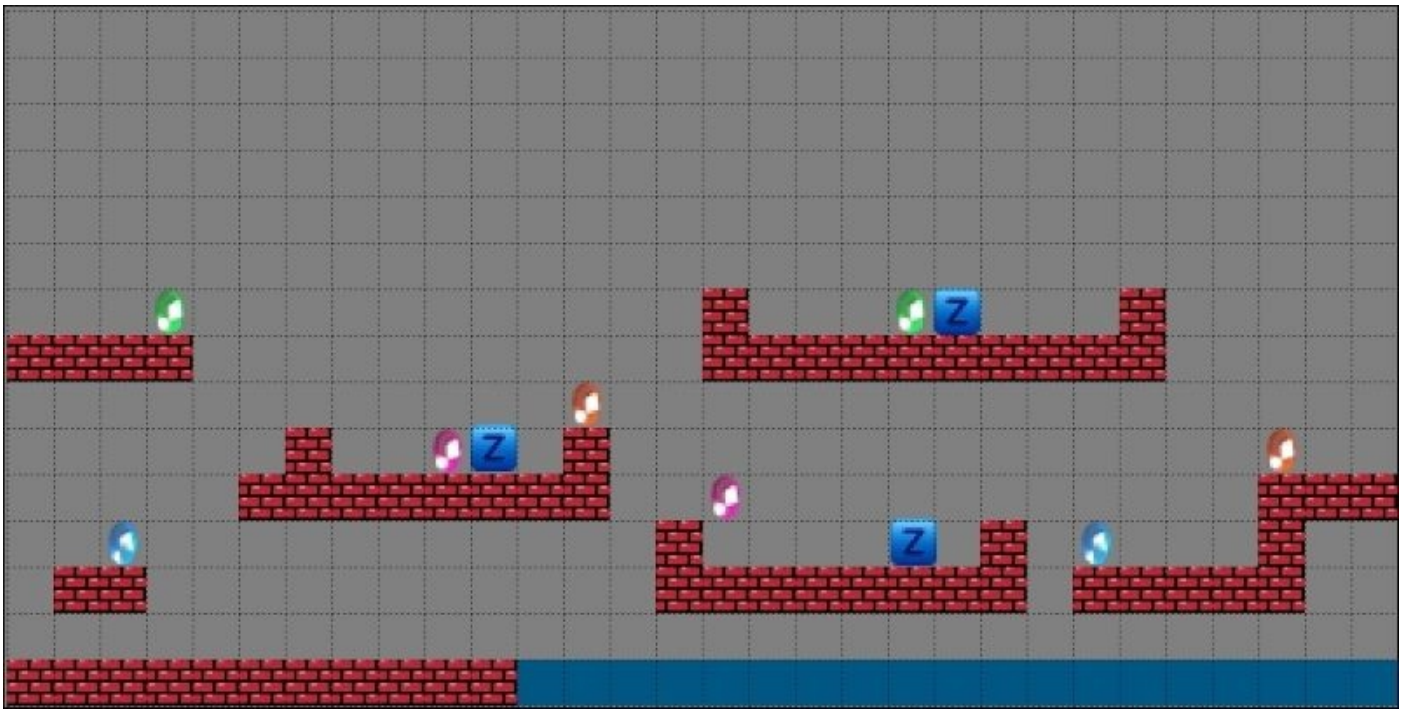
```
// check whether bob collides with the zombies
for(Enemy enemy:GameManager.enemies){
    if(enemy.rectangle.overlaps(bobRectangle)){
        if(GameData.lives>0){
            setPosition(GameConstants.spawnPoint.x,
GameConstants.spawnPoint.y);
            GameData.lives--;
            break;
        }
        else{
            Gdx.app.exit();
        }
    }
}
```

If you run the game now, you can see the zombies on the screen where you placed them:


MainGame
Lives : 3
Score : 40
A

Enemy motion

The zombies that we created are stationary. We need to add motion to them. To do this, I am changing the level a little bit:



I have placed the zombies in horizontally enclosed areas. This is done because we are going to give only a horizontal motion to the zombies. As soon as they hit a wall while walking, they will switch direction. Let's define the zombies' velocities as a constant in the `Zombie` class:

```
private static final float ZOMBIE_VELOCITY = 0.04f;
```

Then, we update Bob's spawn point in the `GameConstants` class:

```
public static final Vector2 spawnPoint = new Vector2(6,7);
```

We need to set the velocity for the zombie in the constructor of its class:

```
velocity = new Vector2(ZOMBIE_VELOCITY, 0);
```

We also need to add enums to denote the zombie's direction:

```
enum Direction{LEFT,RIGHT};
Direction direction = Direction.LEFT; //denotes zombie's direction
```

We need to create a method to detect collisions with the wall in the `Zombie` class. We create a new method called `checkWallHit()` and type the following code:

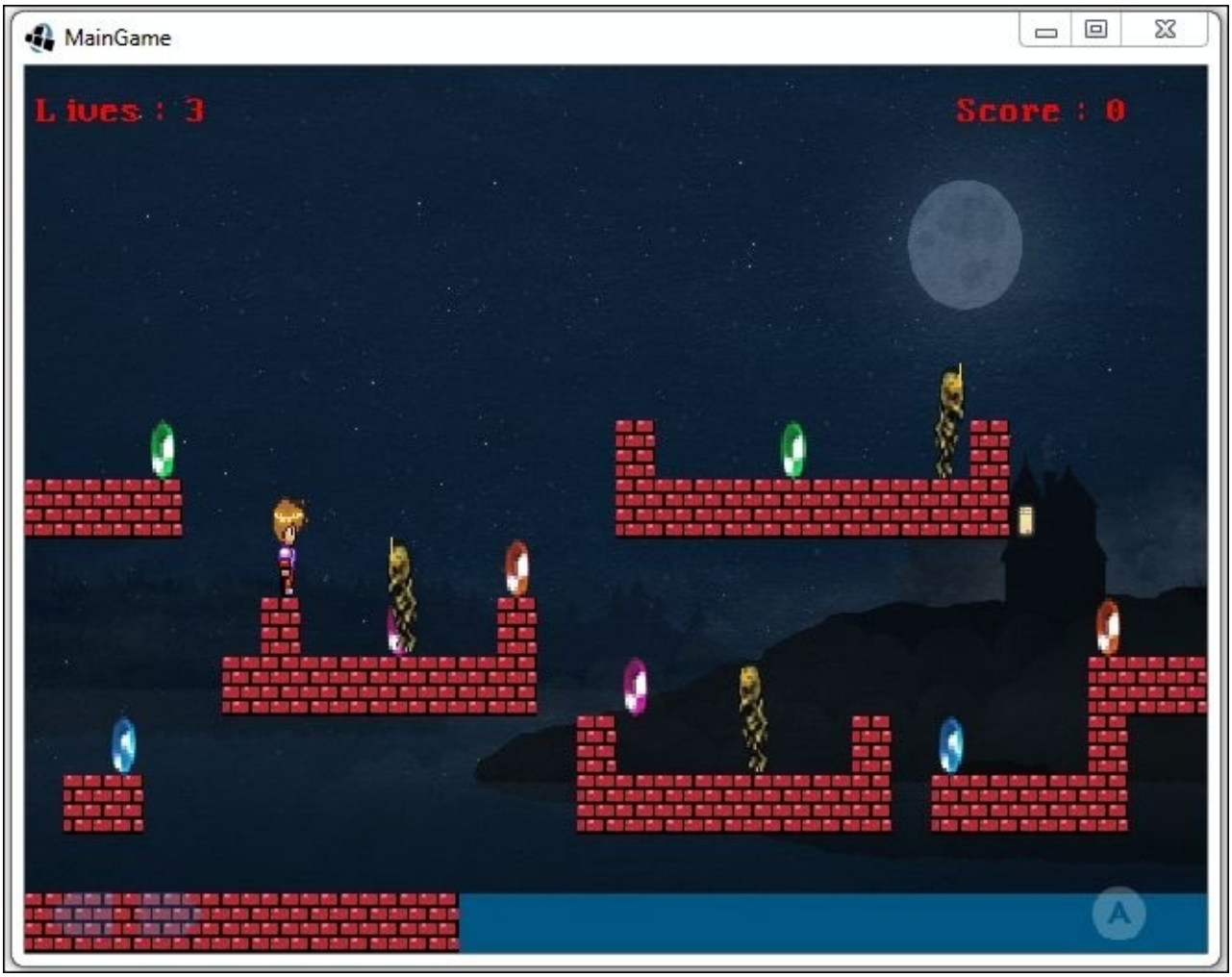
```
public void checkWallHit(){
    // get the tiles from map utilities
    Array<Rectangle> tiles = MapUtils.getHorizNeighbourTiles(velocity,
sprite, "Wall");
    //if zombie collides with any wall tile while walking right/left,
```

```
reverse his horizontal motion
    for (Rectangle tile : tiles) {
        if (rectangle.overlaps(tile)) {
        velocity.x *=-1;
        break;
        }
    }
}
```

In this method, we detect the collisions between each zombie and the wall. If they collide, we reverse their direction and velocity so that they can start moving in the opposite direction. We need to update the `update()` method as well:

```
@Override
public void update() {
    // set the rectangle with zombie's dimensions for collisions
    rectangle.set(sprite.getX(), sprite.getY(), sprite.getWidth(),
sprite.getHeight());
    checkWallHit();
    // change the direction based on velocity
    if (velocity.x < 0) {
        direction = Direction.LEFT;
    } else {
        direction = Direction.RIGHT;
    }
    sprite.setX(sprite.getX()+velocity.x);

    if(direction==Direction.RIGHT){
        sprite.setFlip(true, false);
    }
    else {
        sprite.setFlip(false, false);
    }

}
```

If you run the game now, you can see the zombies moving back and forth in the game:



All the zombies have the same velocity. If you want to have a different velocity for each zombie, you can set a custom property in the editor for each zombie object and read it in the code to set. Finally, we have to animate the zombie. The sprite sheet for the animation needs to be first packed via the Texturepacker-GUI tool:

Replace the `zombieImage` constant in the `GameConstants` class with this line of code:

```
public static final String zombieSpriteSheet = "zombie_spritesheet";
```

In the `Zombie` class, we need to add these members:

```
Animation walkAnimation;        // animation instance
    TextureRegion walkSheet;    // sprite sheet
    TextureRegion currentFrame; // current animation frame
    float stateTime;
    private static int ANIMATION_FRAME_SIZE=3;
```

We update the zombie constructor in the following code:

```
public Zombie(float width,float height,TextureRegion zombieSheet,float
```

```
x,float y){

    sprite = new Sprite();
    sprite.setPosition(x,y);
    velocity = new Vector2(ZOMBIE_VELOCITY, 0);
    rectangle = new Rectangle();
    this.walkSheet=zombieSheet; // save the sprite-sheet
    //split the sprite-sheet into different textures
    TextureRegion[][] tmp = walkSheet.split(
walkSheet.getRegionWidth()/ANIMATION_FRAME_SIZE,
walkSheet.getRegionHeight());
    // convert 2D array to 1D
    TextureRegion[] walkFrames = tmp[0];

    // create a new animation sequence with the walk frames and time period
of 0.04 seconds
    walkAnimation = new Animation(0.25f, walkFrames);

    // set the animation to loop
    walkAnimation.setPlayMode(PlayMode.LOOP_PINGPONG);
    // get initial frame
    currentFrame = walkAnimation.getKeyFrame(stateTime, true);

    sprite.setSize(((walkSheet.getRegionWidth()/ANIMATION_FRAME_SIZE)*
(width/RESIZE_FACTOR)),(walkSheet.getRegionHeight()*
(width/RESIZE_FACTOR)));

sprite.setSize(sprite.getWidth()*GameConstants.unitScale,sprite.getHeight()
*GameConstants.unitScale);

}
```

We edit the `update()` method as follows:

```
@Override
public void update() {
    // set the rectangle with zombie's dimensions for collisions
    rectangle.set(sprite.getX(), sprite.getY(), sprite.getWidth(),
sprite.getHeight());
    checkWallHit();
    if (velocity.x < 0) {
        direction = Direction.LEFT;
    } else {
        direction = Direction.RIGHT;
    }
    sprite.setX(sprite.getX()+velocity.x);
    stateTime += Gdx.graphics.getDeltaTime();
    currentFrame = walkAnimation.getKeyFrame(stateTime, true);

    sprite.setRegion(currentFrame); // set the zombie sprite's texture to
the current frame

    if(direction==Direction.RIGHT){
        sprite.setFlip(true, false);
    }
    else {
```

```
        sprite.setFlip(false, false);
    }

}
```

Finally, in the `MapUtils` class' `spawnEnemies()` method, update the instantiation line as follows:

```
Zombie zombie = new Zombie(width, height,
texturePack.findRegion(GameConstants.zombieSpriteSheet),rectangle.x*unitSca
le,rectangle.y*unitScale);
```

You can see the walking animation of the zombie if you run the game.

Summary

In this chapter, we learned the following topics:

- Collecting items in the game and keeping scores
- Displaying scores and adding hazards
- Hazard detection and keeping track of lives
- Adding enemies to the game
- Adding enemies to a level through Tiled
- Adding motion and animation to enemies

In the next chapter, we will learn how to make enemies chase the player. We will also see how to shoot bullets and how to make enemies follow a predefined path.

Chapter 9. More Enemies and Shooting

In this chapter, we will learn about two new enemy types. We will learn how to program an enemy to chase the player and follow a specified path. We will also learn how to shoot bullets and kill enemies.

In this chapter, we will cover the following topics:

- Skeletons and chasing
- Shooting and stars

Skeletons and chasing

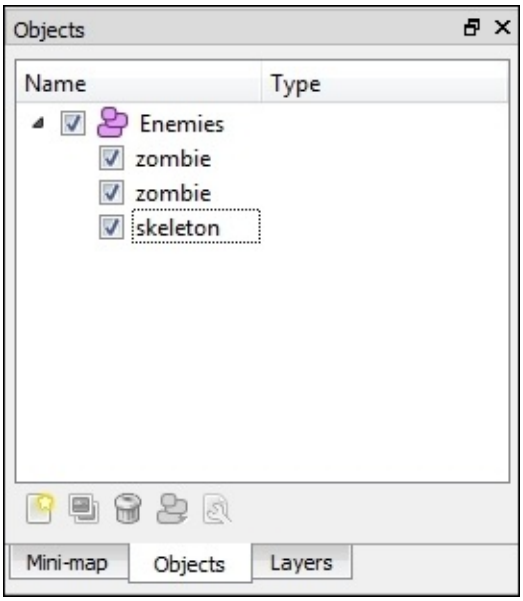
In this section, we will learn how to add a new enemy type: a skeleton. We will also learn how to chase the player with the skeleton.

Skeletons

The zombie that we made in the previous chapter was just following a straight path. It was not intelligent. Let’s make a new enemy of the skeleton type. This will be very similar to a zombie but with a twist. The skeletons will follow the player when he is closer. The sprite sheet for them looks like this:

First, we add this sheet to the packed texture and copy the files to the project so that we can use them. Next, we need to edit the map and place the skeletons in it. I have updated the map and placed a skeleton, which looks like this:

The **Objects** layer looks like this:



To access the sprite sheet, let's define its name in the `GameConstants` class:

```
public static final String skeletonSpriteSheet = "skeleton_spritesheet";
```

Before we create the skeleton class, let's move the `checkWallHit()` method from the `Zombie` class to the base `Enemy` class. Now, we'll create the `Skeleton` class. This will be similar to the `Zombie` class as of now. Add a new class named `Skeleton` to the `com.packtpub.dungeonbob.gameobjects` package and type in the following code:

```
package com.packtpub.dungeonbob.gameobjects;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.g2d.Animation;
import com.badlogic.gdx.graphics.g2d.Animation.PlayMode;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.math.Rectangle;
import com.badlogic.gdx.math.Vector2;
import com.packtpub.dungeonbob.GameConstants;

public class Skeleton extends Enemy {

    private static final float RESIZE_FACTOR = 900f;
    private static final float SKELETON_VELOCITY = 0.04f;
    private static int ANIMATION_FRAME_SIZE=9;
    enum Direction{LEFT,RIGHT};
    Direction direction = Direction.LEFT; //denotes skeleton's direction
    Animation walkAnimation;  // animation instance
    TextureRegion walkSheet;  // sprite sheet
    TextureRegion currentFrame; // current animation frame
    float stateTime;

    @Override
    public void render(SpriteBatch batch) {
        sprite.draw(batch);
```

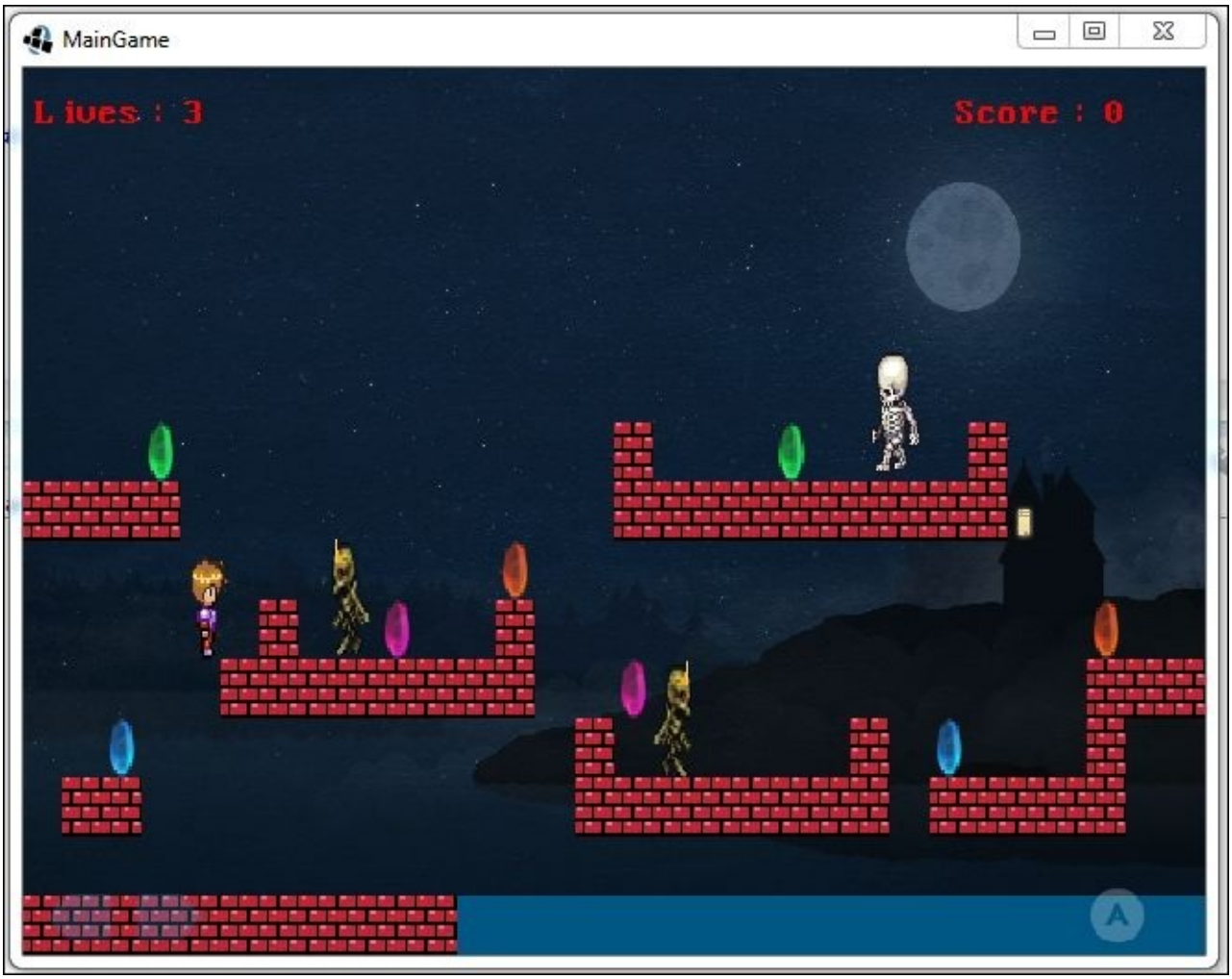
```
    }

    @Override
    public void update() {
        // set the rectangle with skeleton's dimensions for collisions
        rectangle.set(sprite.getX(), sprite.getY(), sprite.getWidth(),
sprite.getHeight());
        checkWallHit();

        // change the direction based on velocity
        if (velocity.x < 0) {
            direction = Direction.LEFT;
        } else {
            direction = Direction.RIGHT;
        }

        sprite.setX(sprite.getX()+velocity.x);
        stateTime += Gdx.graphics.getDeltaTime();
        currentFrame = walkAnimation.getKeyFrame(stateTime, true);

        sprite.setRegion(currentFrame); // set the skeleton sprite's
texture to the current frame

        if(direction==Direction.RIGHT){
            sprite.setFlip(true, false);
        }
        else {
             sprite.setFlip(false, false);
        }

    }

    public Skeleton(float width,float height,TextureRegion
skeletonSheet,float x,float y){

        sprite = new Sprite();
        sprite.setPosition(x,y);
        velocity = new Vector2(SKELETON_VELOCITY, 0);
        rectangle = new Rectangle();
        this.walkSheet= skeletonSheet; // save the sprite-sheet
        //split the sprite-sheet into different textures
        TextureRegion[][] tmp = walkSheet.split(
walkSheet.getRegionWidth()/ANIMATION_FRAME_SIZE,
walkSheet.getRegionHeight());
        // convert 2D array to 1D
        TextureRegion[] walkFrames = tmp[0];

        // create a new animation sequence with the walk frames and time
period of 0.04 seconds
        walkAnimation = new Animation(0.25f, walkFrames);

        // set the animation to loop
        walkAnimation.setPlayMode(PlayMode.LOOP_PINGPONG);
        // get initial frame
        currentFrame = walkAnimation.getKeyFrame(stateTime, true);
```

```
        sprite.setSize(((walkSheet.getRegionWidth()/ANIMATION_FRAME_SIZE)*
(width/RESIZE_FACTOR)),(walkSheet.getRegionHeight()*
(width/RESIZE_FACTOR)));

sprite.setSize(sprite.getWidth()*GameConstants.unitScale,sprite.getHeight()
*GameConstants.unitScale);

    }

}
```

To parse the map and create `skeleton` objects, we will update the `spawnEnemies()` method in the `MapUtils` class:

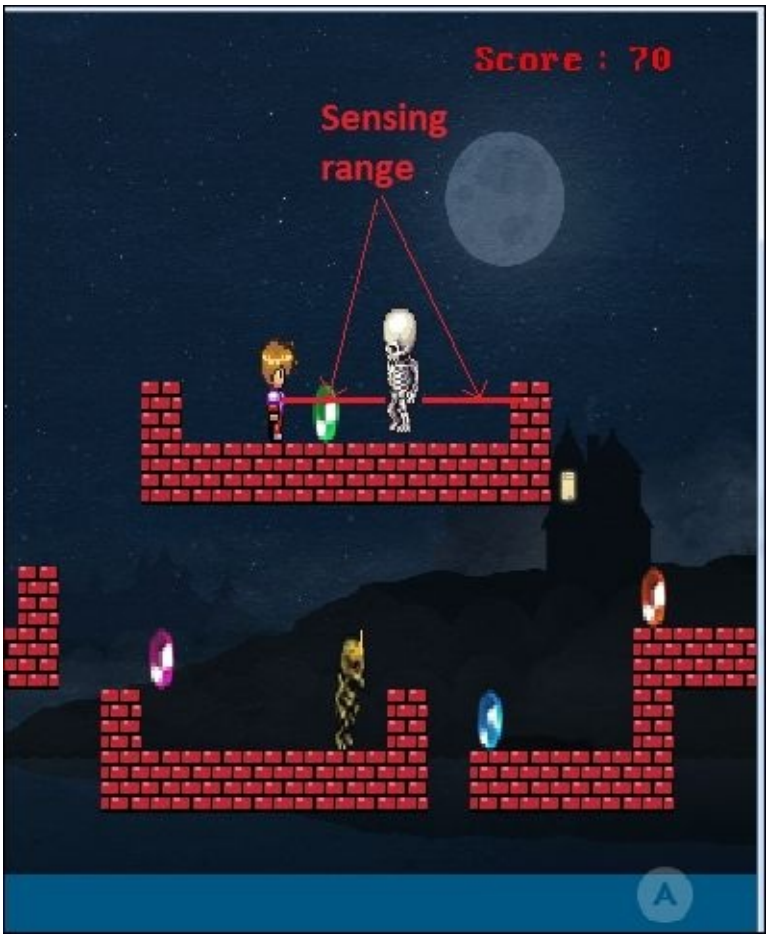
```
// if the name of the object is "zombie" as we have given
if(mapObject.getName().equals("zombie")){
    Rectangle rectangle = ((RectangleMapObject)mapObject).getRectangle();

    //create a zombie object and place it in that location
    Zombie zombie = new Zombie(width, height,
texturePack.findRegion(GameConstants.zombieSpriteSheet),rectangle.x*unitSca
le,rectangle.y*unitScale);
    enemies.add(zombie);
}
// if the name of the object is "skeleton" as we have given
else if(mapObject.getName().equals("skeleton")){
    Rectangle rectangle = ((RectangleMapObject)mapObject).getRectangle();
    //create a skeleton object and place it in that location
    Skeleton skeleton = new Skeleton(width, height,
texturePack.findRegion(GameConstants.skeletonSpriteSheet),rectangle.x*unitS
cale,rectangle.y*unitScale);
    enemies.add(skeleton);
}
```

If you run the game now, you can see the skeleton walking. If Bob runs into the skeleton, you can also see his lives getting lost. Since we are adding both a zombie and skeleton to the array of the `Enemy` type, we don’t have to render, update, or handle collisions for them separately:


MainGame
Lives : 3
Score : 0

Chasing Bob

Let’s add some chasing capabilities to our skeleton. If Bob comes near the skeleton, it should follow him. This is illustrated in the following screenshot:



We will set a sensing range for the skeleton. Whenever Bob comes within that range, the skeleton will start chasing him. First, make the `Bob` variable in `GameManager` class as public:

```
public static Bob bob; // bob instance
```

We will add a constant to the `Skeleton` class for the horizontal sense distance:

```
private static final float HORIZ_SENSE_DISTANCE = 4;
```

We will add a new method to the `Skeleton` class called `senseAndFollow()`:

```
public void  senseAndFollow(){
    // get the distance between Bob and skeleton
    float difference =GameManager.bob.bobSprite.getX()-
(sprite.getX()+sprite.getWidth()/2);

    //if the distance is between certain threshold, start chasing
    if(Math.abs(difference)<=HORIZ_SENSE_DISTANCE){

        // if bob is near and behind the skeleton switch directions
        if((direction == Direction.LEFT) && difference>0){
```

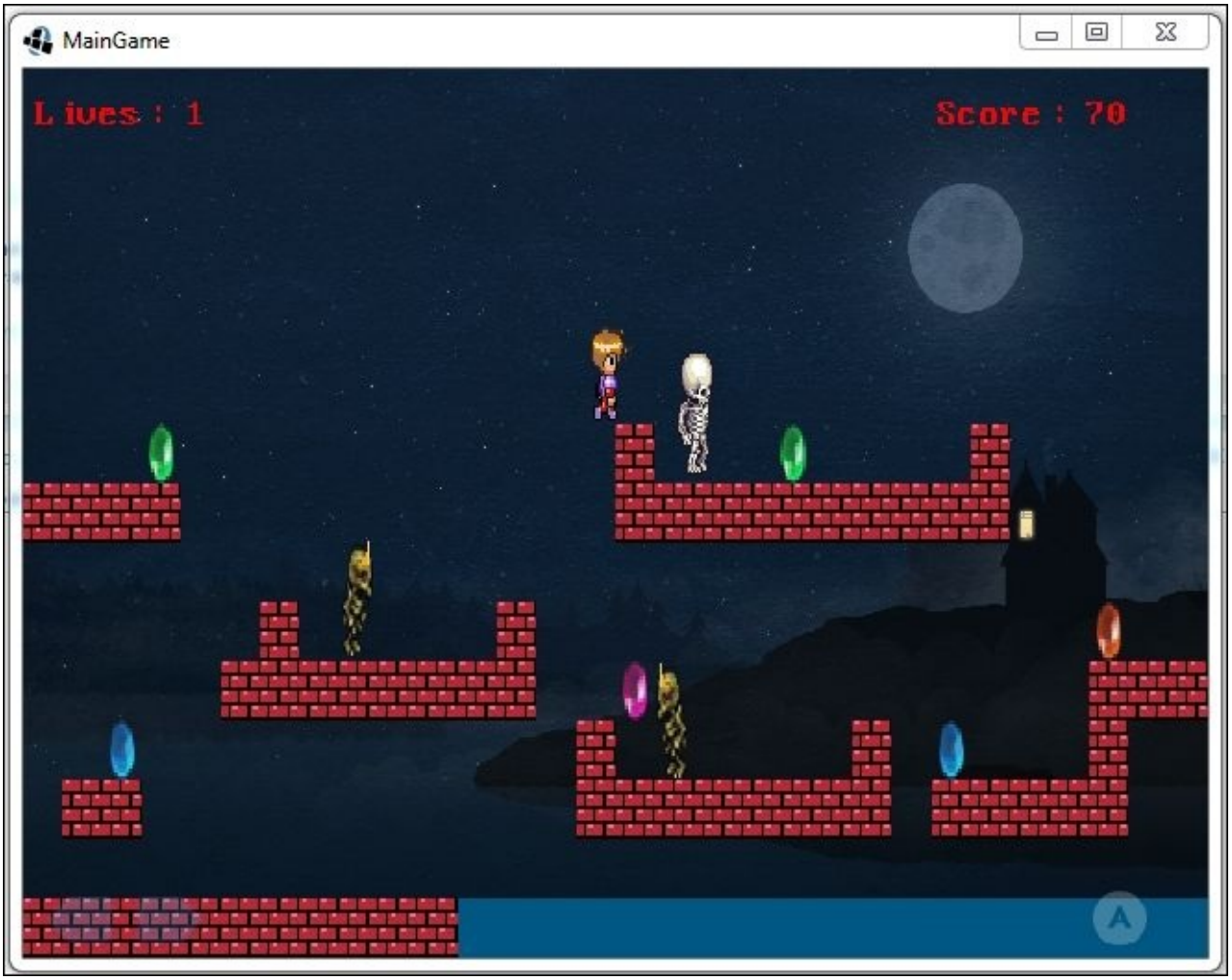
```
            velocity.x *=-1;
        }

        if((direction == Direction.RIGHT )&& difference<0){
            direction = Direction.LEFT;
            velocity.x *=-1;
        }

    }

}
```

In this method, we first get the horizontal distance between Bob and the skeleton. If it is below the threshold (4 units in this case), the skeleton decides to chase Bob. If Bob is behind the skeleton, it turns its direction and goes after him. If the skeleton is facing Bob, then it is already going in that direction. So, there is nothing to do.

We call this method in the `update()` function:

```
senseAndFollow();
checkWallHit();
```

If you run the game now and move Bob closer behind the skeleton, you can see it turning around. However, there is still a problem with this logic, which you can notice in a situation like this:



Here, we can see the skeleton stuttering while walking. This happens when the skeleton is near the edges. After hitting the wall, the logic dictates it to go back. But since it senses Bob, it tries to go back and while doing so, it hits the wall. It goes back and forth. As a result, the skeleton gets stuck. To avoid this situation, we need to make it smarter.

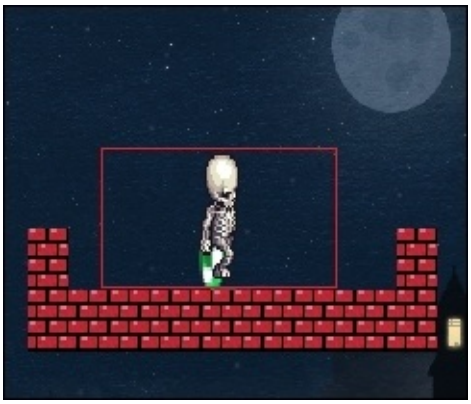
Since the skeleton would never be able to catch Bob in this case, it need not follow him. What we need to check is whether there is a wall tile between the skeleton and Bob; if so, then he need not follow.

To check this in the code, update the `senseAndFollow()` function, as follows:

```
public void  senseAndFollow(){
    // get the distance between Bob and skeleton
    float difference =GameManager.bob.bobSprite.getX()-
(sprite.getX()+sprite.getWidth()/2);

    //if the distance is between certain threshold, start chasing
    if(Math.abs(difference)<=HORIZ_SENSE_DISTANCE){

        int startX, startY, endX, endY;

        // get the tiles between bob and the skeleton
        if (difference > 0) {
            endX = (int) GameManager.bob.bobSprite.getX();
            startX = (int) sprite.getX();
        }
        else {
            startX = (int) GameManager.bob.bobSprite.getX();
            endX = (int) sprite.getX();
        }
        startY = (int) (sprite.getY());
        endY = (int) (sprite.getY() + sprite.getHeight());

        // get the tiles from map utilities
        Array<Rectangle> tiles = MapUtils.getTiles(startX, startY,
endX,endY, "Wall");

        if (tiles.size == 0) {
            // if bob is near and behind the skeleton switch directions
            if((direction == Direction.LEFT) && difference>0){
                velocity.x *=-1;
            }

            if((direction == Direction.RIGHT )&& difference<0){
                direction = Direction.LEFT;
                velocity.x *=-1;
            }
        }
    }
}
```

We get the count of wall tiles from the `MapUtils` class, and if it is 0 (nothing in between), only then we follow Bob:

Run the game to see that the skeleton does not follow Bob in this position. There is one more problem left to address though. Even if you are not near the skeleton vertically, it still tries to follow you. To fix this, we need to take the vertical distance into account as well. We basically need to make the sensing area, as shown here:



To implement this, update `senseAndFollow()` as follows:

```
float difference = GameManager.bob.bobSprite.getX() - (sprite.getX() +
sprite.getWidth() / 2);

float yDifference = GameManager.bob.bobSprite.getY() - sprite.getY();

// if the distance is between certain threshold, start chasing
if (Math.abs(difference) <= 4  && yDifference < sprite.getHeight() &&
yDifference>0) {
```

Shooting and stars

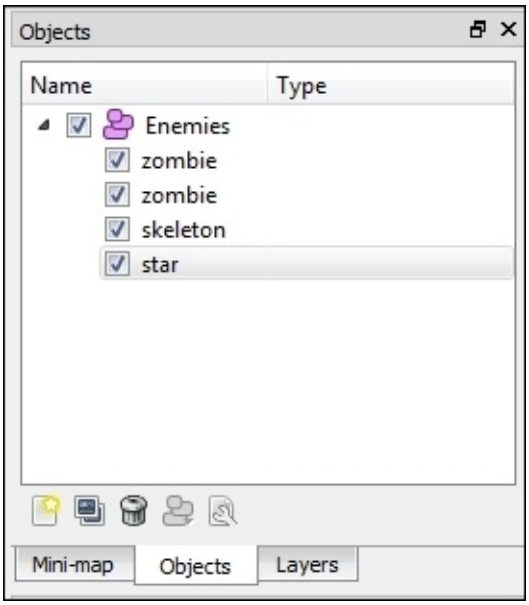
In this section, we will learn about enemies that follow a predefined path. We will also learn how to shoot bullets and kill enemies in the game.

Stars

Let’s try to make enemies who follow a path made by us. We will create a new enemy type called star:

This type of an enemy will follow the path that we create in the Tiled map editor. I have created the map, which looks like the following screenshot:

The **Objects** layer looks like this:



As you can see, we created a path in the **Objects** layer using a polygon. We named the path as **star**. The path represents a single path of the enemy of the star type. First, let's pack the star image with the Texturepacker-GUI tool. Coming back to the code, let's declare a constant in the `GameConstants` class:

```
public static final String startImage = "star";
```

Create a new class named `Star` in the `com.packtpub.dungeonbob.gameobjects` package and paste the following content:

```
package com.packtpub.dungeonbob.gameobjects;
import com.badlogic.gdx.math.Vector2;
public class Star extends Enemy{

    float path[]; // points through which the star travels
    float angle;
    Vector2 current,next; // current point and the next point
    int currentIndex; //
    int pathSize; // total no of points

    private static float SPEED = 0.1f;
    private static float SCALE_FACTOR = 3500f;
}
```

We maintain an array of floats that denote the points through which the star travels. For the iteration, we maintain two `Vector2` objects: `current` and `next`. These denote the two points through which the star is currently travelling. Let's make the constructor of the `Star` class:

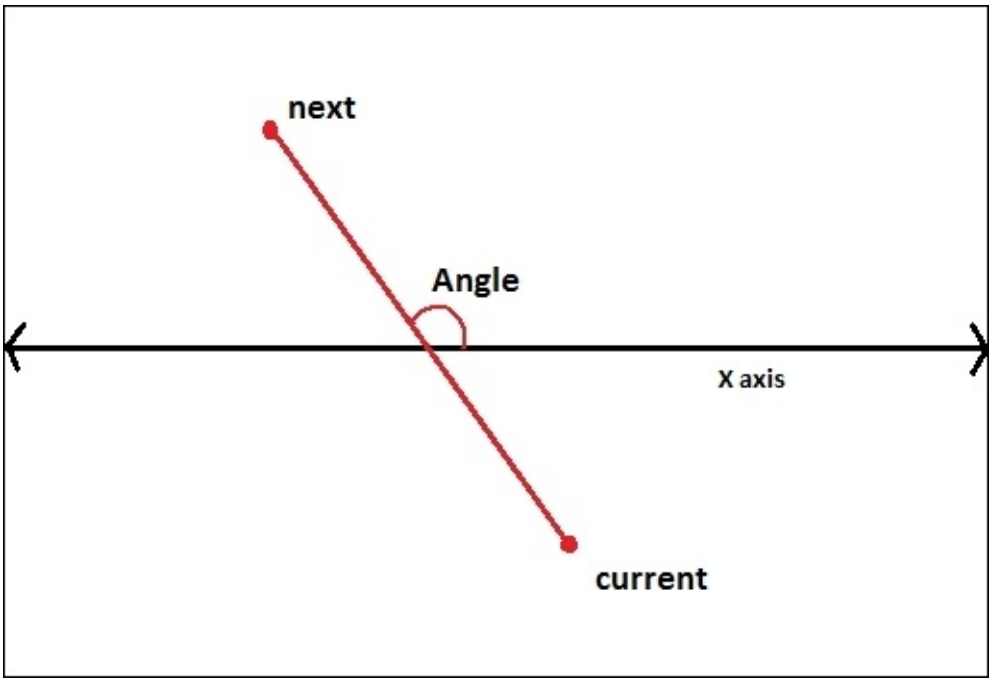
```
public Star(float width, float height, TextureRegion startTexture,float
path[]) {

    this.path = path;
    velocity = new Vector2();
```

```
    //multiply the points with unitscale to adjust them with respect to the
game's coordinates
    for(int i=0;i<path.length;i++){
        path[i]=path[i]*GameConstants.unitScale;
    }

    // initialize first and the second point
    current = new Vector2(path[0], path[1]);
    next = new Vector2(path[2], path[3]);

    sprite = new Sprite(startTexture);
    sprite.setPosition(current.x, current.y);
    sprite.setSize(sprite.getWidth() * (width /SCALE_FACTOR),
(sprite.getHeight() * (width /SCALE_FACTOR)) );
    sprite.setSize(sprite.getWidth() *
GameConstants.unitScale,sprite.getHeight() * GameConstants.unitScale);

    currentIndex=2;
    pathSize= path.length;
    rectangle = new Rectangle();
}
```

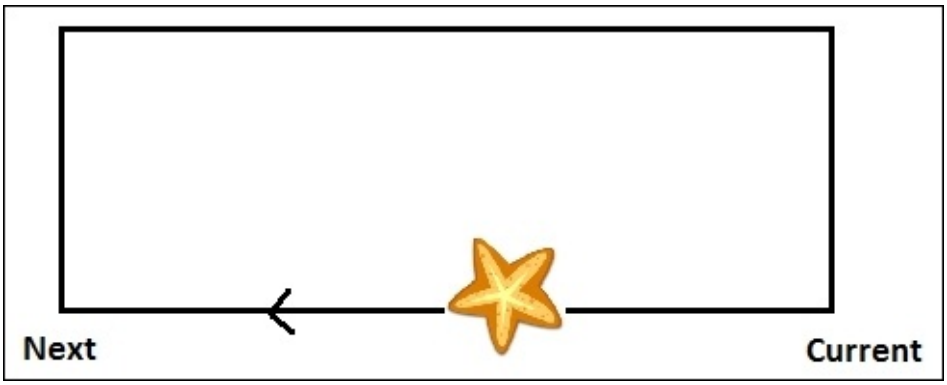
Let's add the `update()` method as well:

```
@Override
public void update(){
    // set the rectangle with star's dimensions for collisions
    rectangle.set(sprite.getX(), sprite.getY(),
sprite.getWidth(),sprite.getHeight());
    // calculate the angle between the line joining current-next with x
axis
    angle= MathUtils.atan2((next.y-current.y),(next.x-current.x));
    // set the velocity to move on that line according to the angle
    velocity.set(SPEED*MathUtils.cos(angle), SPEED*MathUtils.sin(angle));

    sprite.setPosition(sprite.getX()+velocity.x, sprite.getY()+velocity.y);

    float xDifference= next.x-sprite.getX();
    float yDifference = next.y-sprite.getY();

    // if we have reached the next point,
    if(Math.abs(xDifference) <=SPEED  && Math.abs(yDifference)<=SPEED ){
        //travel between further points
        currentIndex+=2;
        current.set(next.x, next.y);

next.set(path[currentIndex%pathSize],path[(currentIndex+1)%pathSize]);
        sprite.setPosition(current.x, current.y);
    }
}
```

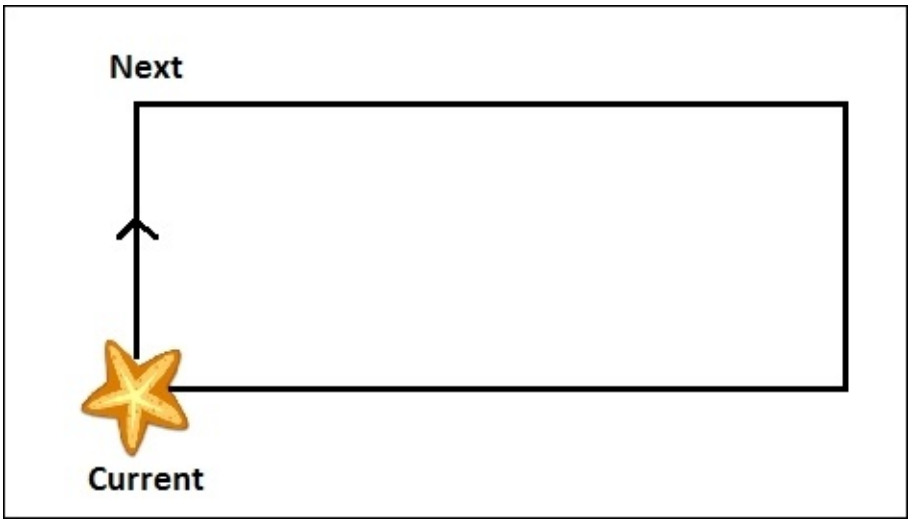
We first calculate the angle of the line that joins the current and the next point in the path with respect to the *x* axis:



Then, we set the velocity of the star with respect to the cosine and sine components of the angle that is calculated. This is done so that the star moves correctly along the line. The total velocity of that star remains equal to the speed that we have set. We calculate `xDifference` and `yDifference` and that tells us how close the star is to the next point:



If the star is very near to the destination point (next), we update the current and the next points for it. The present (next) point becomes the current and the farther point in the path becomes the next point:



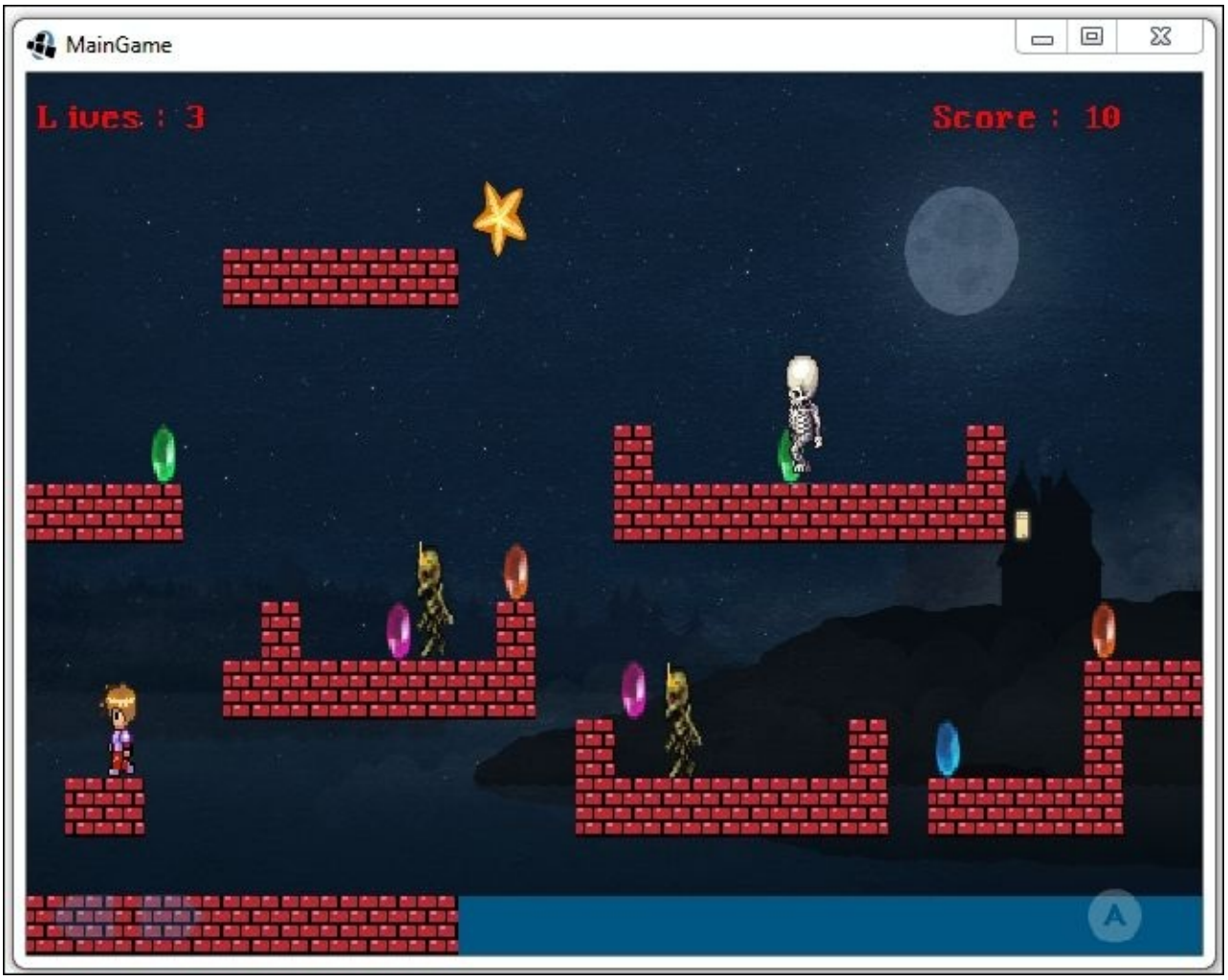
This cycle continues further and repeats. Now, we add the `render()` method:

```
@Override
public void render(SpriteBatch batch) {
    sprite.draw(batch);
}
```

In the `MapUtils` class' `spawnEnemies()` method, we need to update the code to parse the paths from the map and create stars:

```
// if the name of the object is "skeleton" as we have given
else if(mapObject.getName().equals("skeleton")){
    Rectangle rectangle = ((RectangleMapObject)mapObject).getRectangle();
    //create a skeleton object and place it in that location
    Skeleton skeleton = new Skeleton(width, height,
texturePack.findRegion(GameConstants.skeletonSpriteSheet),rectangle.x*unitS
cale,rectangle.y*unitScale);
    enemies.add(skeleton);
}
// if the name of the object is "star" as we have given
else if (mapObject.getName().equals("star")) {
    Polygon polygon = ((PolygonMapObject) mapObject).getPolygon();
    // create a star object and place it in that location
    Star star = new Star(width,
height,texturePack.findRegion(GameConstants.startImage),polygon.getTransfor
medVertices());
    enemies.add(star);
}
```

If you run the game now, you can see the star travelling along the path you created. You can also observe the collision detection between Bob and the star:


MainGame
Lives : 3
Score : 10

Shooting

Until now, the player could only get killed by the enemies. Let’s now add shooting capabilities to Bob so that he can kill the enemies as well. Let’s first add the bullet image to the texture atlas via the texture packer GUI:

Once we have done this, we need to define the bullet image name in the `GameConstants` class:

```
public static final String bulletImage = "bullet";
```

Now, we need to actually create the `Bullet` class. Create this class in the `com.packtpub.dungeonbob.gameobjects` package. After creating this class, add the following data members to it:

```
Sprite sprite; // bullet sprite

Vector2 velocity; // velocity of the bullet
Rectangle rectangle; // rectangle object to detect collisions

enum Direction{LEFT,RIGHT}; // represents in which direction the bullet is
travelling
public enum State{ALIVE,DEAD}; // represents whether the bullet is active
or not
public State state;
Direction direction;

private static final float BULLET_VELOCITY = 0.2f;
private static final float RESIZE_FACTOR = 1500f;
```

The bullet constructor is as follows:

```
public Bullet(float width, float height, TextureRegion bulletTexture) {
    sprite = new Sprite(bulletTexture);
    velocity = new Vector2();
    rectangle = new Rectangle();

    sprite.setSize((sprite.getWidth() *
(width/RESIZE_FACTOR)*GameConstants.unitScale),(sprite.getHeight() *
(width/RESIZE_FACTOR) * GameConstants.unitScale));
}
```

We are going to use only a single instance of a bullet in this level; that is, the player can shoot only one bullet at a time. We are going to need a `reset()` method that will reset the bullet’s parameters once the player shoots it:

```
public void reset(float x, float y,boolean isLeft){
    state= State.ALIVE;
    sprite.setPosition(x, y);

    if(isLeft){
```

```
        direction=Direction.LEFT;
        velocity.set(-BULLET_VELOCITY,0);
    }
    else{
        direction=Direction.RIGHT;
        velocity.set(BULLET_VELOCITY,0);
         }
}
```

Once the player shoots a bullet, we need to change the state to alive. We need to detect when the bullet hits the walls. Let's add a method named `checkWallHit()`, which will do this:

```
public void checkWallHit() {
    // get the tiles from map utilities
    Array<Rectangle> tiles = MapUtils.getHorizNeighbourTiles(velocity,
sprite, "Wall");

    // if bullet collides with any tile while walking right/left, mark it
as dead
    for (Rectangle tile : tiles) {
        if (rectangle.overlaps(tile)){
            state= State.DEAD;
            break;
        }
    }
}
```

If the bullet hits any wall, we change its state to dead. Finally, we need the `update()` and the `render()` methods:

```
public void update(){
    // set the rectangle with bullet's dimensions for collisions
    rectangle.set(sprite.getX(), sprite.getY(),
sprite.getWidth(),sprite.getHeight());
    checkWallHit();
    sprite.setX(sprite.getX() + velocity.x);
}

public void render(SpriteBatch batch) {

    if(direction==Direction.LEFT){
        sprite.setFlip(true, false);
    }
    else{
        sprite.setFlip(false, false);
    }

    sprite.draw(batch);
}
```

Now, to shoot the bullet, we need to make a `shoot()` method in the `Bob` class:

```
public void shoot(){
    if(GameManager.bullet.state==Bullet.State.DEAD){
        if(direction==Direction.RIGHT){
```

```
            GameManager.bullet.reset(bobSprite.getX()+bobSprite.getWidth(),
bobSprite.getY()+(bobSprite.getHeight()/2), false);
        }
        else{
            GameManager.bullet.reset(bobSprite.getX(), bobSprite.getY()+
(bobSprite.getHeight()/2), true);
        }
    }
}
```

We can only shoot the bullet if it's dead. The bullet's position (from where it emerges) is decided based on the player's direction. We will use the left *Ctrl* key as the key for shooting. To trigger the shoot, in the `InputManager` class' `keyDown()` method, add the following code:

```
// make bob shoot
else if(keycode==Keys.CONTROL_LEFT){
    GameManager.bob.shoot();
}
```

In the `GameManager` class, we need to declare an instance of `Bullet`:

```
public static Bullet bullet;
```

We instantiate it in the `initialize()` method:

```
bullet = new Bullet(width, height, texturePack.findRegion("bullet"));
bullet.state = Bullet.State.DEAD;
```

In the `renderGame()` method, add the following lines of code:

```
if(bullet.state==Bullet.State.ALIVE){
    bullet.update();
    bullet.render(batch);
}

//draw the paddles with respect to hud cam
batch.setProjectionMatrix(GameScreen.hudCamera.combined);
```

If you run the game now and press the left *Ctrl* button, you can see Bob shooting a bullet. It still passes through the enemies. To make the enemies die, let's add a state to it in the `Enemy` class:

```
public enum State{ALIVE,DEAD}; // represents whether the enemy is active or
not
public State state = State.ALIVE;
```

Now, we'll add some code to detect the collisions between the bullet and the enemy. The `update()` method needs to be changed, as shown in the following code:

```
public void update(){
    Bullet bullet = GameManager.bullet;
    if(rectangle.overlaps(bullet.rectangle) &&
bullet.state==Bullet.State.ALIVE ){
        bullet.state=Bullet.State.DEAD;
        state=State.DEAD;
    }
```

```
}
```

We need to modify the `update()` method of the `Star`, `Zombie`, and `Skeleton` classes in such a way that the `super` class' `update()` method will be called:

```
rectangle.set(sprite.getX(), sprite.getY(), sprite.getWidth(),
sprite.getHeight());
super.update();
```

Since `Enemy` is the base class, all the enemy types will collide with the bullet. Now, we have changed the enemy state on collision, but we will still display them irrespective of the state. Let's fix this. In the `GameManager` class' `renderGame()` method, we will only render the enemies when they are alive:

```
for(Enemy enemy :enemies){
    if(enemy.state==Enemy.State.ALIVE){
        enemy.update();
        enemy.render(batch);
    }
}
```

If you run the game now and shoot, you will notice that when the bullet hits the enemies, they disappear. You can try walking toward the position where they died. Bob still dies. This is because we detect the collision between Bob and the enemies without considering their states. Let's make the appropriate changes to the `Bob` class' `checkEnemies()` method:

```
if(enemy.state== Enemy.State.ALIVE &&
enemy.rectangle.overlaps(bobRectangle)){
```

There is one more thing to take care of. You wouldn't want your bullet to go off screen and kill someone outside the visible area. To ensure this, we will kill the bullet as it goes outside the screen bounds. First, let's declare a temporary rectangle to represent the viewport in the `Bullet` class:

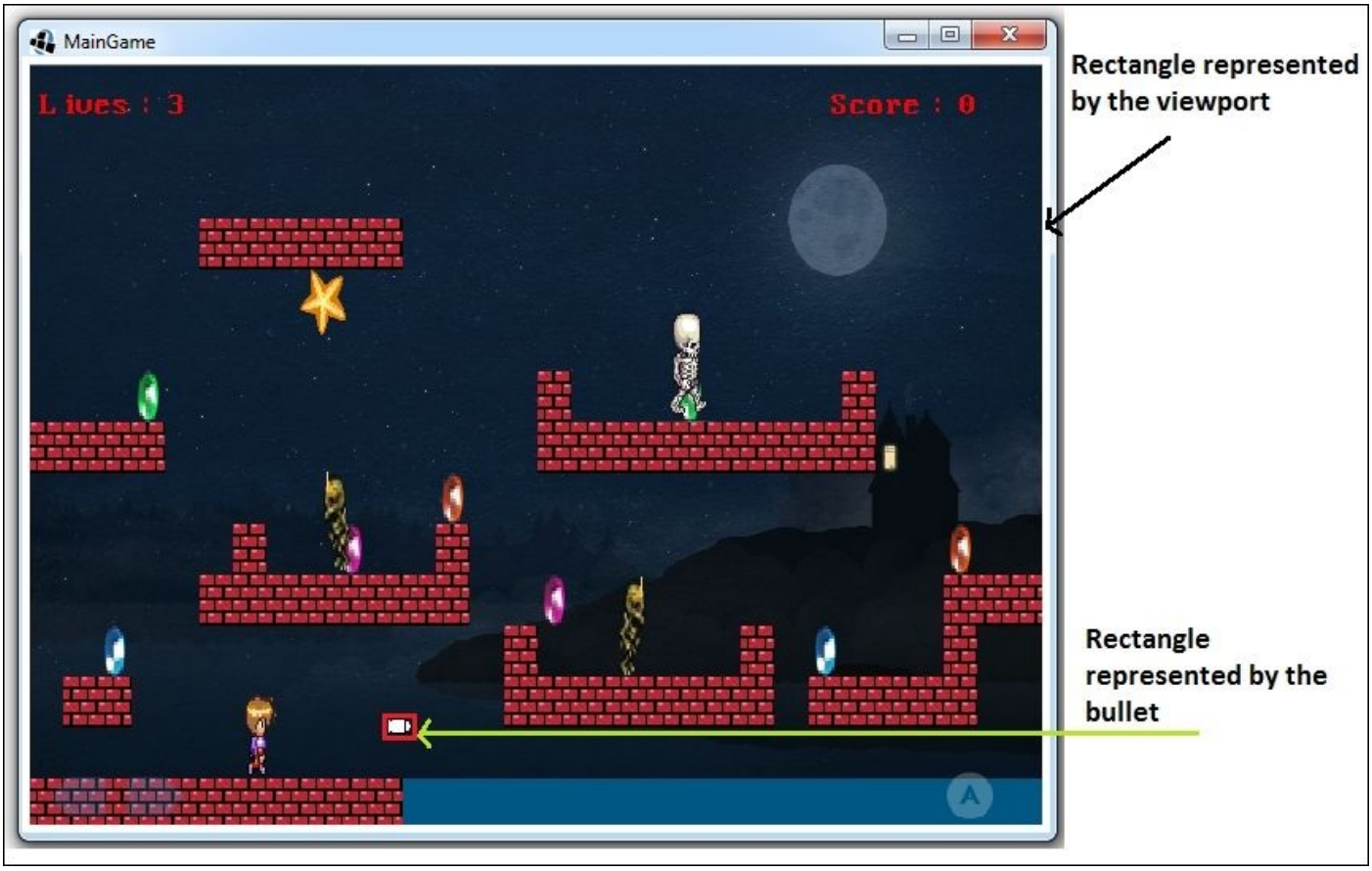
```
Rectangle temp = new Rectangle();
```

Now, we will modify the `update()` method:

```
public void update(){
    OrthographicCamera camera = GameScreen.camera;
    float camX= camera.position.x;
    float camY= camera.position.y;

    // set the rectangle with bullet's dimensions for collisions
    rectangle.set(sprite.getX(), sprite.getY(),
sprite.getWidth(),sprite.getHeight());

    temp.set((camX-camera.viewportWidth/2), (camY-camera.viewportHeight/2)
,camera.viewportWidth,camera.viewportHeight);
    if(!temp.overlaps(rectangle)){
        state = State.DEAD;
        return;
    }

    checkWallHit();
```

```
    sprite.setX(sprite.getX() + velocity.x);
}
```

We get the camera reference and represent the viewport as a rectangle object. We check whether this rectangle overlaps the bullet. If it does, then the bullet is within the screen bounds:



The only thing left now is to add an onscreen button for shooting. After adding the button icon to the texture atlas, we will define the image name for the button in the `GameConstants` class:

```
public static final String shootImage = "buttonB";
```

Add the following members to the `GameManager` class:

```
static TextureRegion shootButtonTexture;
static Sprite shootButtonSprite;
public static final float SHOOT_BTN_RESIZE_FACTOR = 700f;
```

The following code is the function used to initialize the shoot button:

```
public static void initializeShootButton(float width,float height){
    //load shoot button texture region
    shootButtonTexture = texturePack.findRegion(GameConstants.shootImage);
    //set shoot button sprite with the texture
    shootButtonSprite= new Sprite(shootButtonTexture);
    // resize the sprite
    shootButtonSprite.setSize(shootButtonSprite.getWidth()*width/
SHOOT_BTN_RESIZE_FACTOR, shootButtonSprite.getHeight()*width/
```

```
SHOOT_BTN_RESIZE_FACTOR);
    // set the position to bottom right corner with offset
    shootButtonSprite.setPosition(width*0.8f, height*0.012f);
    // make the button semi transparent
    shootButtonSprite.setAlpha(0.25f);
}
```

We call the function in the `initialize()` method of the `GameManager` class:

```
initializeShootButton(width, height);
```

We draw it in the `renderGame()` method:

```
jumpButtonSprite.draw(batch);
shootButtonSprite.draw(batch);
```

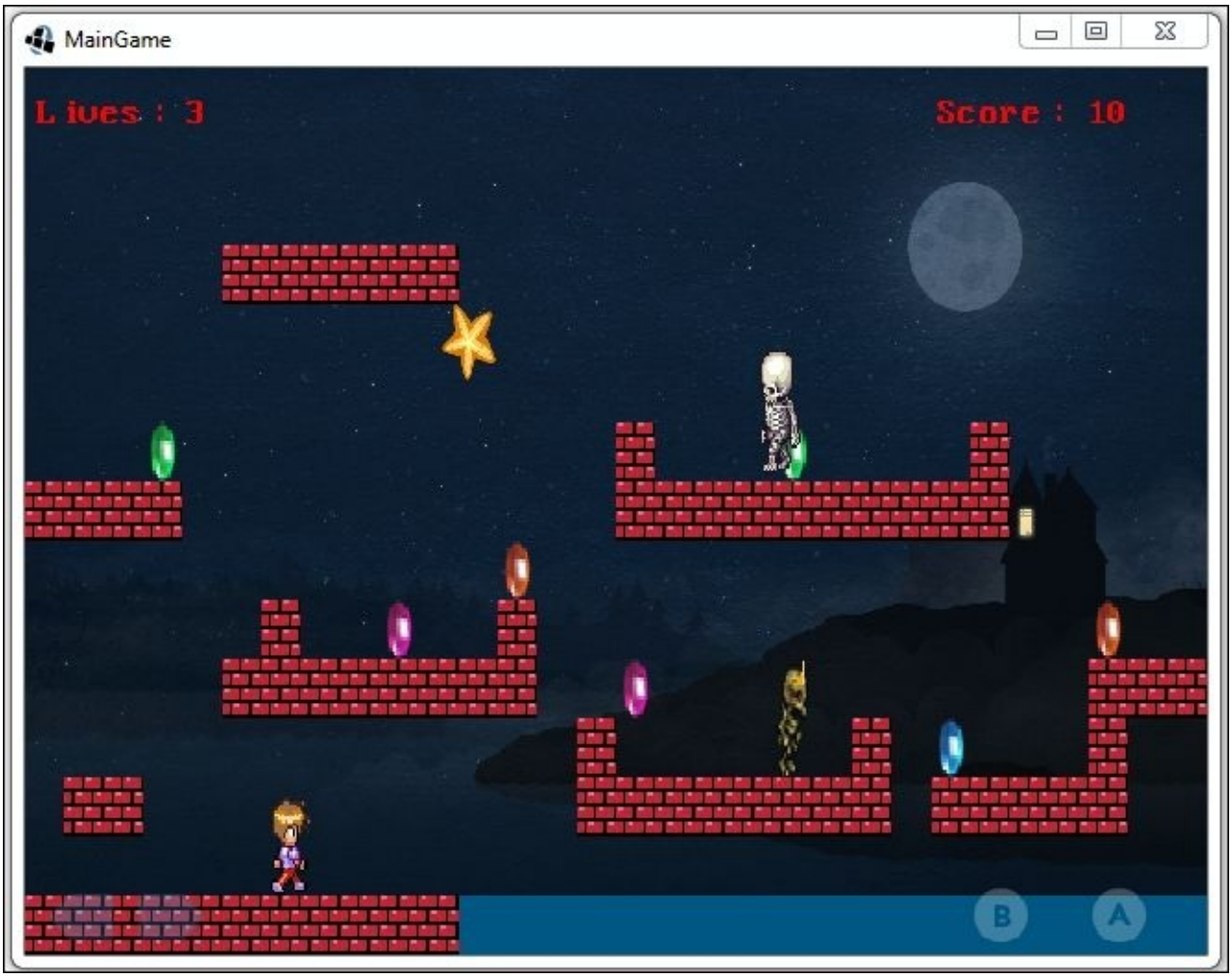
To handle the touch input for it, we will add a method to the `InputManager` class:

```
boolean isshootButtonTouched(float touchX, float touchY){
    // handle touch input on the shoot button
    if((touchX>=GameManager.shootButtonSprite.getX()) && touchX<=
(GameManager.shootButtonSprite.getX()+GameManager.shootButtonSprite.getWidt
h()) && (touchY>=GameManager.shootButtonSprite.getY()) && touchY<=
(GameManager.shootButtonSprite.getY()+GameManager.shootButtonSprite.getHeig
ht()) ){
        return true;
    }
    return false;
}
```

In the `touchdown()` method of the same class, add the following code:

```
else if(isshootButtonTouched(touchX, touchY)){
    GameManager.bob.shoot();
}
    return false;
```

If you run the game now, you will see the **B** button on the screen. It will trigger the shooting action, which is the same as that of the left *Ctrl* button:


MainGame
Lives : 3
Score : 10
B
A

Summary

In this chapter, we learned how to create some new enemy types and how to implement shooting bullets. We learned the following topics:

- Enemy sensing the player
- Enemy following the player
- Avoiding obstacles while following
- Defining enemy paths in Tiled
- Reading the paths and making an enemy follow them
- Shooting a bullet
- Killing enemies with the bullet

In the next and final chapter, we will learn about particle effects, multiple levels, and loading screens.

Chapter 10. More Levels and Effects

In this chapter, we will learn how to add different levels to our game. We will also learn how to add particle effects, such as explosions, and make them using a tool. Finally, we will take a look at how to implement a loading screen in the game.

In this chapter, we will cover the following topics:

- Multiple levels
- Particle effects
- A loading screen

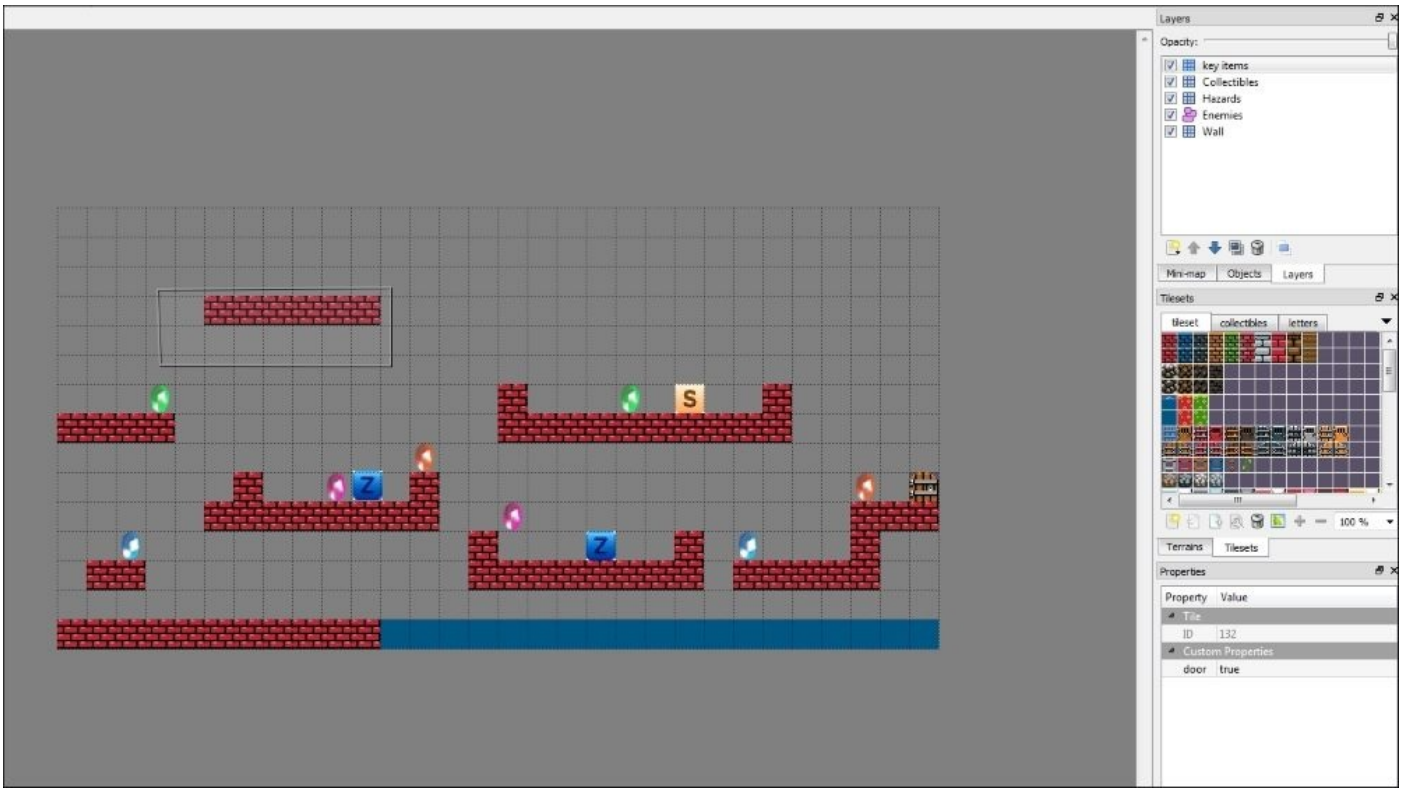
Multiple levels

In this section, we will learn how to create different levels and transitions between them.

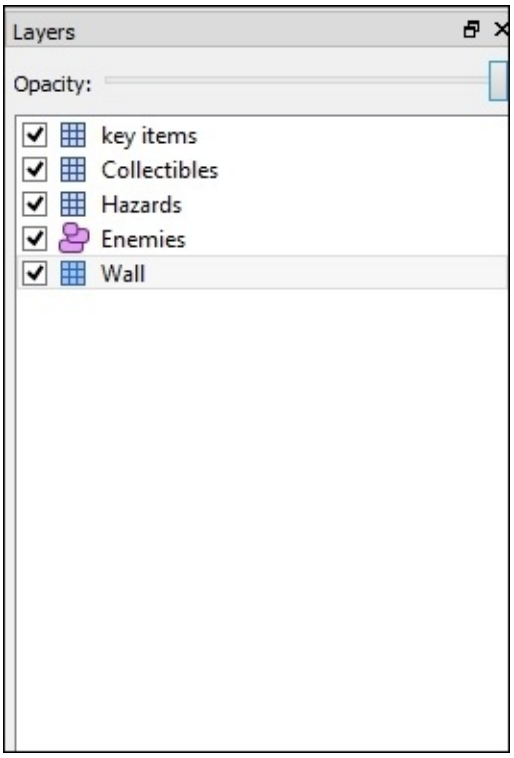
Adding the door

Until now, what we have seen is only one level in the game. Games usually have multiple levels for a good gameplay experience. We will see how to make different levels and how to make transitions between them. But first, we will need to make something that will trigger the game to move forward between different levels. We will use a door as a trigger to change levels.

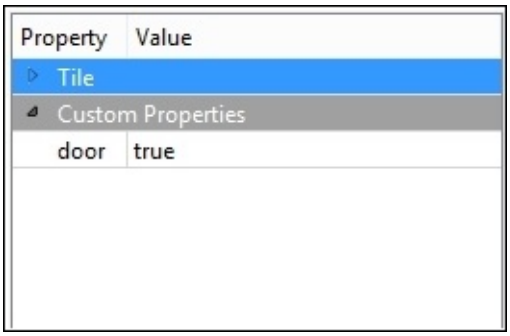
I have updated the game map as follows:



The **Objects** layer with the **key items** tile layer looks like this:



The **door** property is shown here:



I’ve made a new tile layer named **key items** and added a **door** tile to it. I’ve also added a **door** property to this tile. To be able to detect it in the game, we need to first add a new member to the `GameManager` class called `door` with the `Rectangle` type:

```
public static Rectangle door;
```

Now, we need to parse the map and initialize the `door` object with the `door` object from the map. We’ll do this in the `MapUtils` class:

```
public static Rectangle spawnDoor(int mapWidth,int mapHeight){
    TiledMapTileLayer layer = (TiledMapTileLayer) map.getLayers().get("key
items");
    if(layer==null) return null;

    for (int y = 0; y <= mapHeight; y++) {
        for (int x = 0; x <= mapWidth; x++) {
            Cell cell = layer.getCell(x, y);
            // if cell is present at a particular location in the map and
```

```
it is a door
            if (cell != null &&
cell.getTile().getProperties().containsKey("door")) {
                Rectangle rect = new Rectangle(x, y, 1, 1);
                return rect;
            }
        }
    }

    return null;
}
```

Here, we basically scan all the tiles in the **key items** layer, and if we find a door, we create a rectangle object out of it. In the `GameManager` class' `initialize()` method, we initialize the `door` object by calling the `spawnDoor()` method and passing the map's dimensions:

```
door = MapUtils.spawnDoor(mapWidth, mapHeight);
```

To detect the collisions between the door and Bob, we will add a new method called `checkDoor()` to the `Bob` class:

```
public void checkDoor(){
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
bobSprite.getWidth(), bobSprite.getHeight());
    if(GameManager.door!=null && bobRectangle.overlaps(GameManager.door)){
        // add a print statement to check collisions
    }
}
```

We add a null check as the door might not be present in the level sometimes (the last level). We call this function in the `Bob` class' `update()` method:

```
checkEnemies();
checkDoor();
```

Add a print statement in place of the comment and run the game. When the player runs into the door, you can see the output on the console.

Changing levels

Now that we have made a trigger to change the level, let's actually make the new level and write the code to switch between them. I have made a new level, which is slightly different from the old one:

We save this level with the name, `level2.tmx`. Next, instead of a constant for the level, we will use an array, which will point to the levels we have made. In the `GameConstants` class, add the following code:

```
public static final String[] levels =
{"data/maps/level1.tmx","data/maps/level2.tmx" };
```

In the `GameManager` class, we will need to declare a variable to keep track of the current level:

```
static short currentLevelIndex=0;
```

Update all the references where the level is referenced. In the `initialize()` method, add the following code:

```
map = assetManager.get(GameConstants.levels[currentLevelIndex]);
```

In the `loadAssets()` method, add the following code:

```
assetManager.load(GameConstants.levels[currentLevelIndex], TiledMap.class);
```

We need to declare the width and height of the screen as static members of the `GameManager` class:

```
public static float Width,Height;
```

Next, we will set these values to the screen's width and height in the `initialize()`

method:

```
public static void initialize(float width,float height){
    GameManager.Height = height;
    GameManager.Width = width;
```

We will now create a new method to load the levels:

```
public static void loadLevel(){
    currentLevelIndex++; // increase the level counter

    // load the next level and the assets
    assetManager.load(GameConstants.levels[currentLevelIndex],
TiledMap.class);
    assetManager.finishLoading();
    map = assetManager.get(GameConstants.levels[currentLevelIndex]);
    setMapDimensions();
    renderer.setMap(map);
    MapUtils.initialize(map);

    enemies.clear(); // remove current level's enemies
    // spawn enemies from the next level
    MapUtils.spawnEnemies(enemies, Width,Height, texturePack);
    door =  MapUtils.spawnDoor(mapWidth,mapHeight);

    GameScreen.camera.setToOrtho(false, mapWidth,mapHeight); // show
specified units horizontally and vertically by the camera
    GameScreen.camera.update();
}
```

In this method, we basically increment the level counter to point to the next level. We then proceed to load the corresponding `TiledMap` from `assetManager`. Once this is done, we repopulate the level with enemies and game objects and reset the game camera.

We set the camera's viewport to cover the whole level. This way, the whole level will be visible on a single game screen. As I mentioned earlier, usually, part of the whole level is made visible at a given time to get the scrolling effect; you can set these values as you like. Finally, we will need to call this function in the `Bob` class' `checkDoor()` method when we collide with the door:

```
public void checkDoor(){
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
bobSprite.getWidth(), bobSprite.getHeight());
    if(GameManager.door!=null && bobRectangle.overlaps(GameManager.door)){
        GameManager.loadLevel();
    }
}
```

When you run the game and hit the door, the new level should be loaded!

Respawning Bob

When Bob is killed, he is instantly respawned at the spawn point. Let's add a delay to it. First, we will need to add a way to check whether Bob is alive or not. Add the following code to the `Bob` class:

```
public enum LifeState{ALIVE,DEAD};
public LifeState lifeState = LifeState.ALIVE;
```

The `lifeState` checks whether Bob is alive or not. We will also need to maintain the delay time. In the `Bob` class, add the following code:

```
static float respawnDelay = 1; // represents maximum delay for respawn
public float respawnCounter = 0;
```

The `respawnDelay` represents the maximum delay time between Bob's death and its resurrection in seconds. The `respawnCounter` represents how much time has passed between Bob's death and its resurrection in seconds. Add a new method named `killBob()`, as follows:

```
public void killBob(){
    lifeState=LifeState.DEAD;
    // start respawn counter
    respawnCounter = respawnDelay;
}
```

Basically, when Bob dies, we change his life state to `DEAD`. After that, we set `respawnCounter` to `respawnDelay` so that we can start the countdown. We will need to call this method at all the places where Bob dies.

In the `checkHazards()` method, add the following code:

```
if (bobRectangle.overlaps(tile)) {
    if(GameData.lives>0){
        killBob();
        break;
    }
```

Remember to add this method to the second loop as well. In the `checkEnemies()` method, add the following code:

```
if(enemy.state== Enemy.State.ALIVE &&
enemy.rectangle.overlaps(bobRectangle)){
    if(GameData.lives>0){
        killBob();
        break;
    }
```

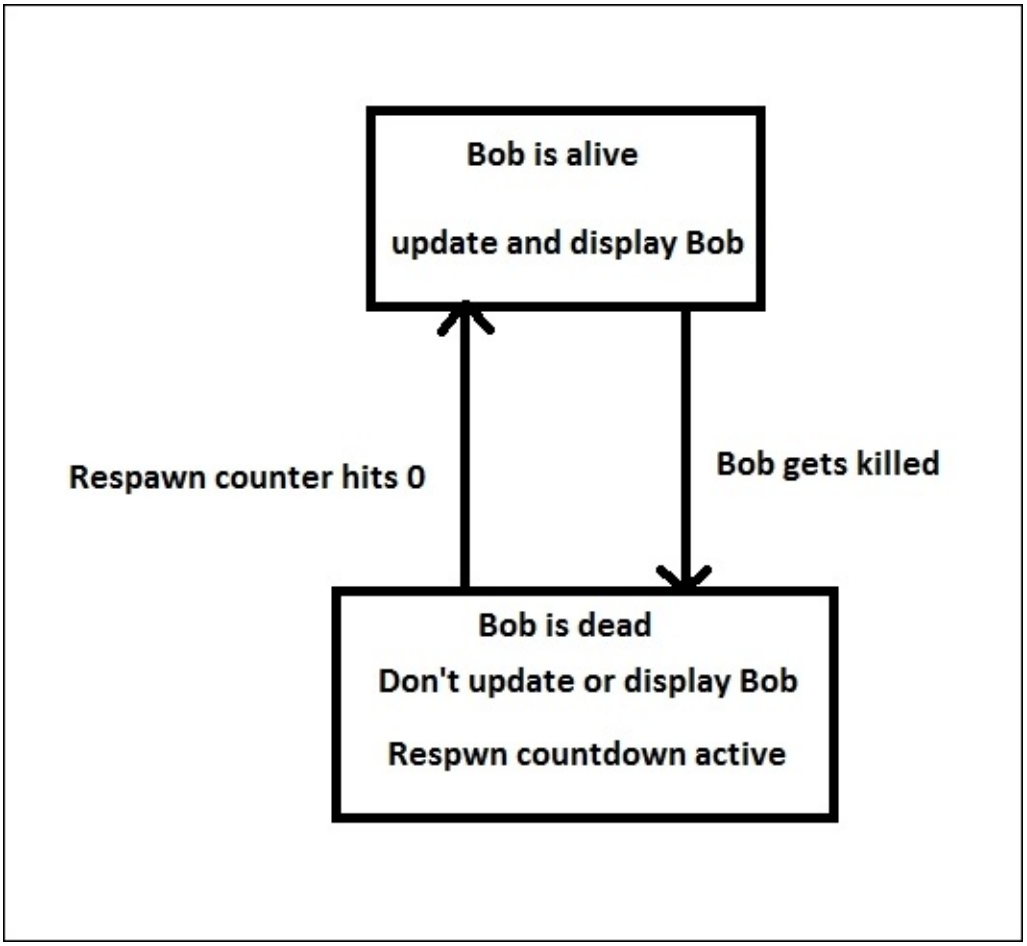
We will need to add a method to reposition Bob after he dies and decrement his lives. Add a method called `respawnBob()` to the `Bob` class:

```
public void respawnBob(){
    setPosition(GameConstants.spawnPoint.x, GameConstants.spawnPoint.y);
    GameData.lives--;
}
```

In the `GameManager` class' `renderGame()` method, make the following changes:

```
batch.setProjectionMatrix(GameScreen.camera.combined);

if(bob.lifeState==LifeState.ALIVE){
    bob.update();
    // Render(draw) the bob
    bob.render(batch);
}
else {
    if(bob.respawnCounter>0){
        bob.respawnCounter-= Gdx.graphics.getDeltaTime();
    } else{
        bob.lifeState=LifeState.ALIVE;
        bob.respawnBob();
    }
}
```

If Bob is alive, we update and display him. If he is dead, we start the respawn countdown. In this state, we keep on decreasing the `respawnCounter` until it hits 0. When it hits 0, we change his state back to `ALIVE` and respawn him:



When you run the game now and collide against any hazard or enemy, you will notice a delay after which you can see Bob getting resurrected. But, if you notice, you will see that Bob can still fire the bullet during its `DEAD` state. This is because we accept the input even after Bob is dead. To restrict this, add the following code to the start of the `Bob` class'

`shoot()` method:

```
public void shoot(){
    if(this.lifeState==LifeState.DEAD){
        return;
    }
```

We don't shoot the bullet if Bob is in the `DEAD` state. One more check should be added to the skeletons so that they don't try to follow Bob when he is dead. In the `Skeleton` class' `update()` function, add the following code:

```
if(GameManager.bob.lifeState==LifeState.ALIVE){
    senseAndFollow();
}
```

This check should be added to all input functions as well.

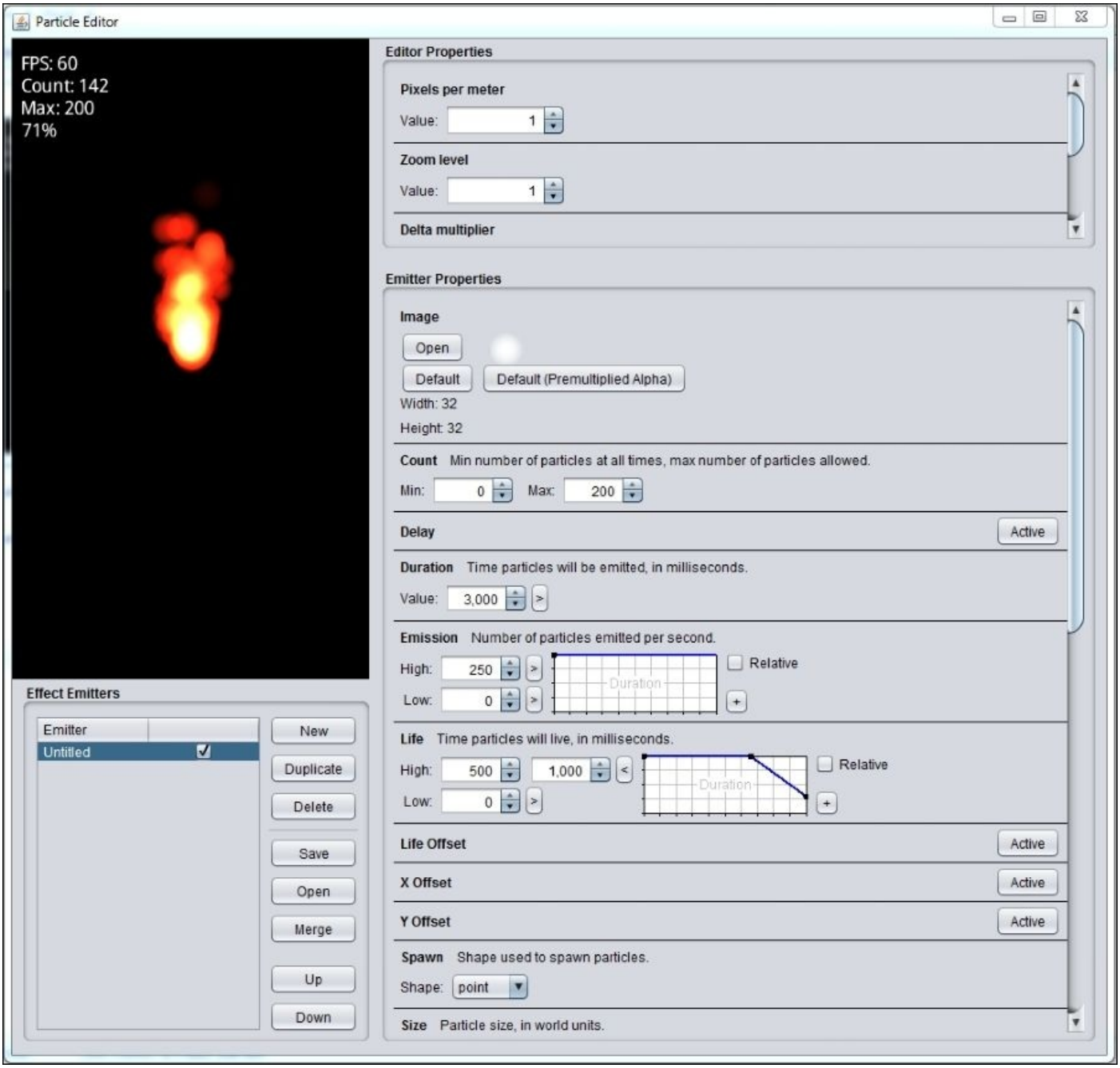
Particle effects

In this section, we will learn how to create particle effects using the Particle Editor tool and use them in the game.

Editor setup and basics

When the player dies, he just disappears for some time and reappears. Let's add an explosion effect when he hits a hazard or an enemy. There is a nifty tool shipped with LibGDX called Particle Editor, which makes it easy to create such effects. Let's see how to run this utility. Head over to https://libgdx.badlogicgames.com/tools.html to download it.

Click on the **Download** link below the **2D Particle Editor** section. Once you have downloaded it, you will get a JAR file, which you can double-click to open the utility:



A single particle is represented by an image. The editor allows you to add multiple particles and play around with their properties, such as velocity, rotation, and transparency, to achieve various effects, such as smoke, fire, explosions, and so on. The result of these

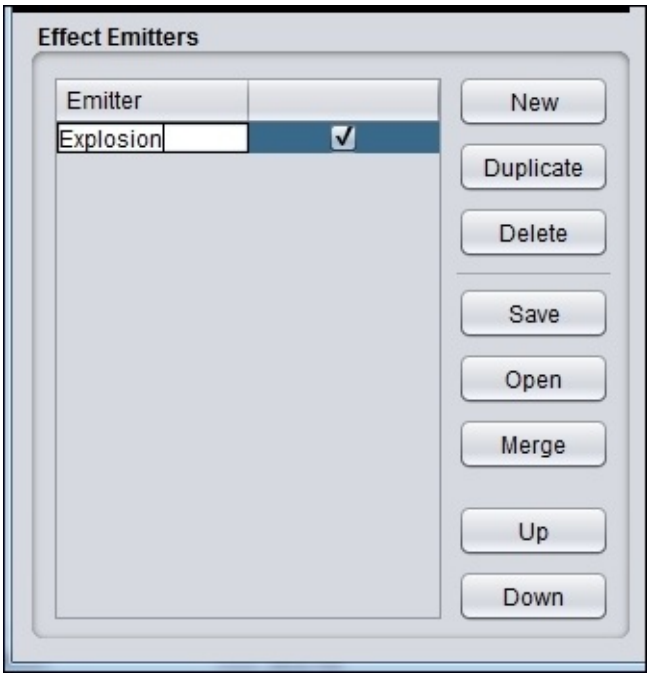
effects is instantly shown to you in the editor as well.

There are an overwhelming number of options in this tool. I’ll just cover what is needed in this case.

The Effect Emitters section

In the bottom-left corner of the screen, you will see the **Effect Emitters** section. Basically, an emitter is something that generates particles of a type. If you want multiple types of particles, you can add a new emitter. All the different properties in the **Emitter Properties** section correspond to a single emitter.

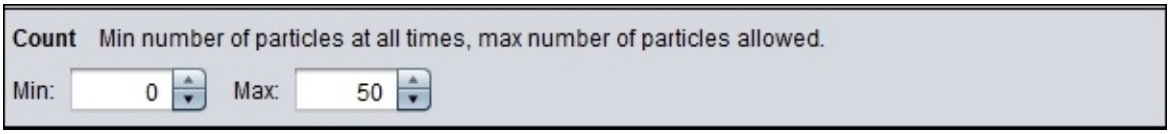
For our purpose, we will just use a single emitter. Double-click on the **Emitter** column to rename it to `Explosion`:



The Emitter Properties section

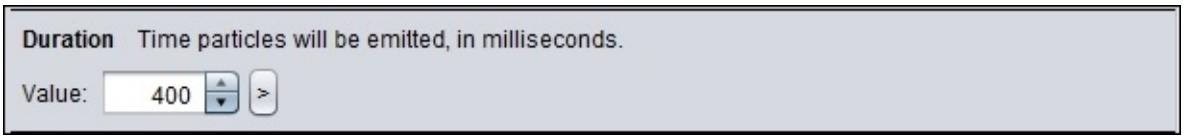
Each emitter has an associated image that represents a particle. If you click on the **Open** button by navigating to the **Emitter Properties** | **Image** section, you can use any image to represent a particle. We are going to use the default image included by the tool:

Our whole effect is just going to be a manipulation of numbers of this image. Next, we will take a look at the **Count** section. There are two options: **Min** and **Max**. As the name suggests, **Min** denotes the minimum number of particles that will always be visible on the screen at all times. **Max** denotes the maximum possible count of the particles at a given time. We will set **Min** to 0 and **Max** to 50:

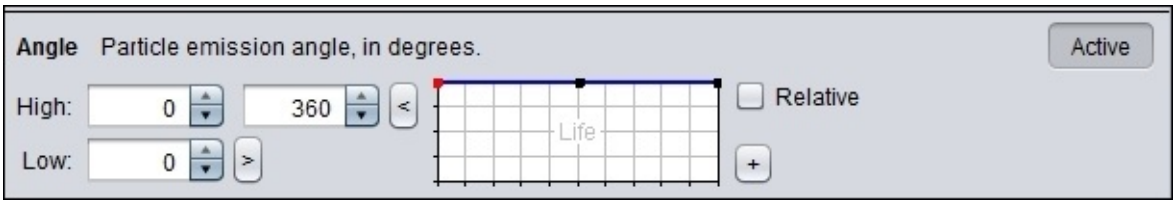


The next section is the **Duration** section. This is the total duration of the particle emission. In this case, where we are using a single emitter, it would be the total duration of our effect. We will set it to 400 ms or 0.4 sec. On the right-hand side of the screen, there is a > symbol to the right, which specifies the range of the duration.

If we set it to 400, which means that the duration of the emission of particles will vary between 400 ms and 800 ms. This means that some particles will be alive for 400 ms, some for 600 ms, some for 800 ms, and so on. Set it to **400** and **600**:

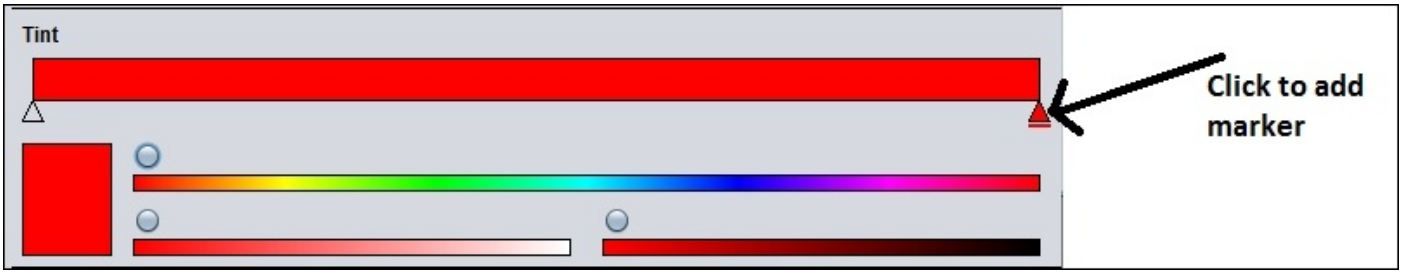


In the **Angle** section, we can set the angle with which the particles are emitted. Set the **High** values to **0** and **360** so that the particles are spread out evenly in all directions as they are emitted. The emission angle will vary between 0 and 360 for the particles:

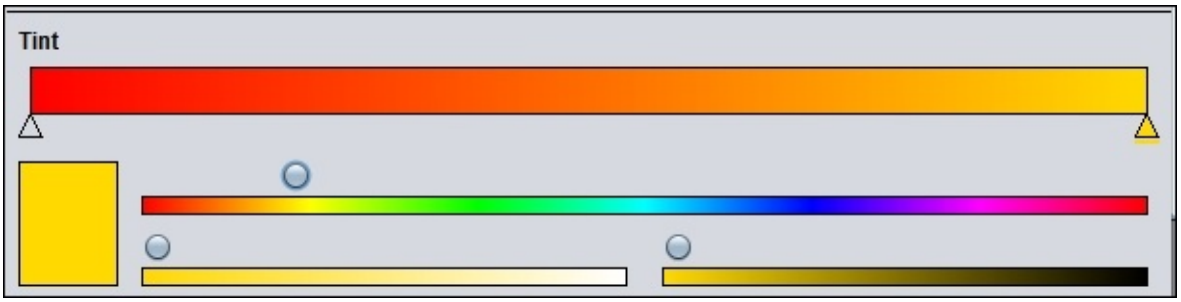


The **Tint** section specifies the tint color to be applied to our particles. In the following screenshot, the upper slider chooses the color, and you can choose the shade with the two sliders below the first slider. The effective color is shown in the box on the left-hand side of the screen:

You can have a variation between particle colors during the lifetime of your effect:



For example, if you want to transition between the red and yellow particle colors, simply click at the end of the first slider, as shown in the screenshot, to get a marker. You can then set the color of that marker:

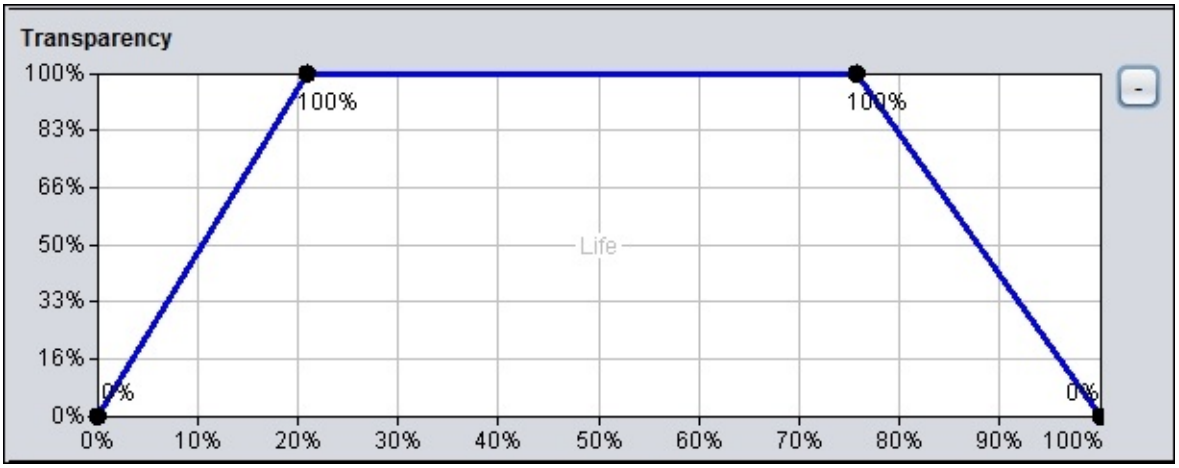


As you can see, the colors are interpolated between the first and the last marker. You can have multiple markers. To delete a marker, double-click on it. For our purpose, let’s keep the color slightly orange and keep only one shade.

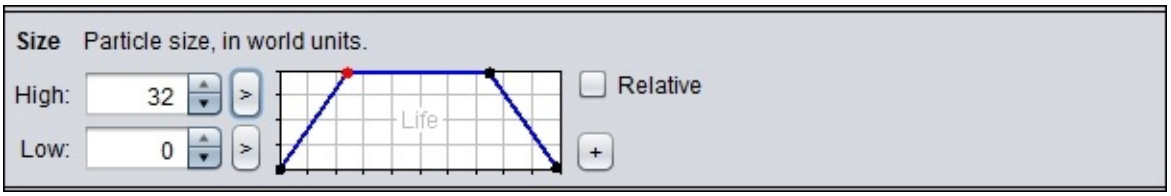
Next, the **Transparency** section is used to control the transparency of the particles. Click on the + button to open the chart view. In this view, we can control how a particular

property changes over its lifetime. The *x* axis represents the time and the *y* axis represents the values of that property.

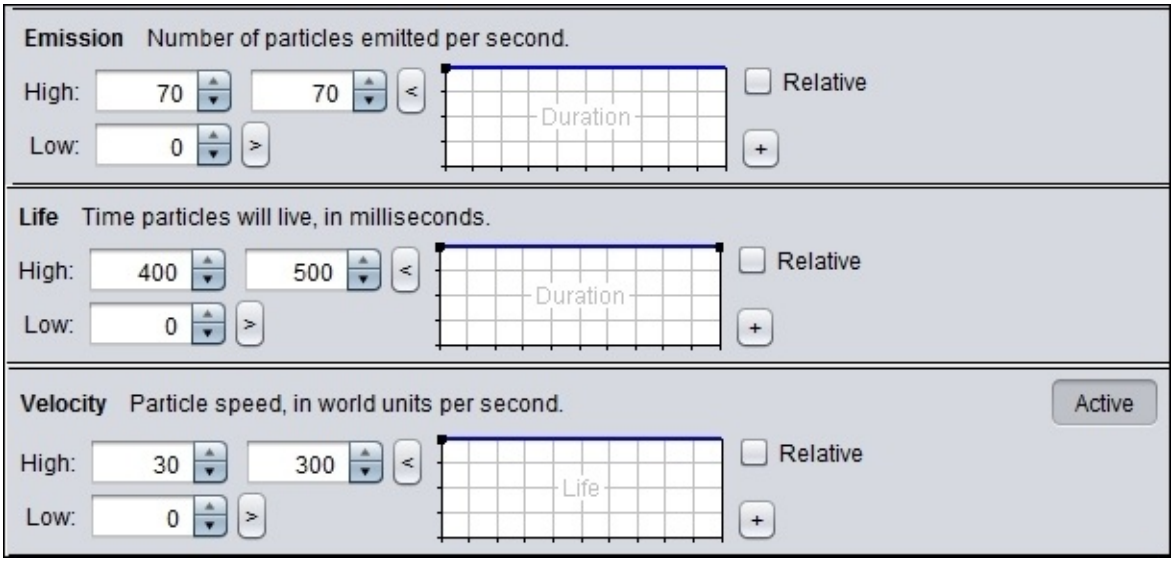
You can add points to the graph by clicking anywhere in the area. The points can be removed by double-clicking and the lines can be dragged with the mouse. We will set the graph in such a way that the particles are invisible when they start, get more opaque, and stay completely opaque in the latter portion of their lifetime. After that, their opacity decreases and they become transparent as they die:



In the **Particle size** option, let’s keep the **High** value to **32** and the graph similar to the transparency one:



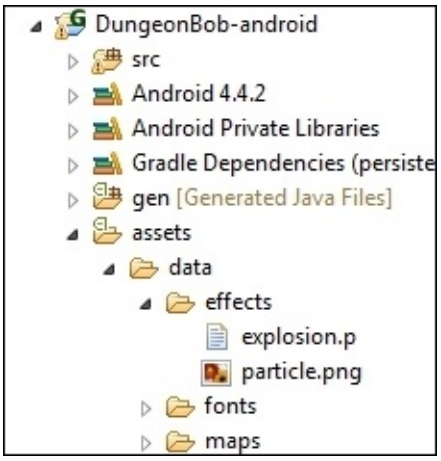
Update the **Emission**, **Life**, and **Velocity** options, as follows:



Loading the effect into the game

We need to save the effect now. Click on the **Save** button and in the next dialog window, give the file name as `explosion.p`. Now that we have saved the particle effect, let's see how to load the effect into our game and display it. First, we need to copy the effect file to the game.

The effect has an image associated with it. Since we have used the default image provided by Particle Editor, we don't know its path. You can download the image file from the code files provided with this book. Create a new folder in your Android project's `assets/data` folder and name it `effects`. Copy the image and the effect file to this folder:



In the `GameManager` class, add a variable to represent the particle effect:

```
public static  ParticleEffect explosionEffect;
```

In the `initialize()` method, add the following code:

```
explosionEffect = new ParticleEffect();
explosionEffect.load(Gdx.files.internal("data/effects/explosion.p"),
Gdx.files.internal("data/effects/"));
explosionEffect.scaleEffect(0.2f*GameConstants.unitScale);
```

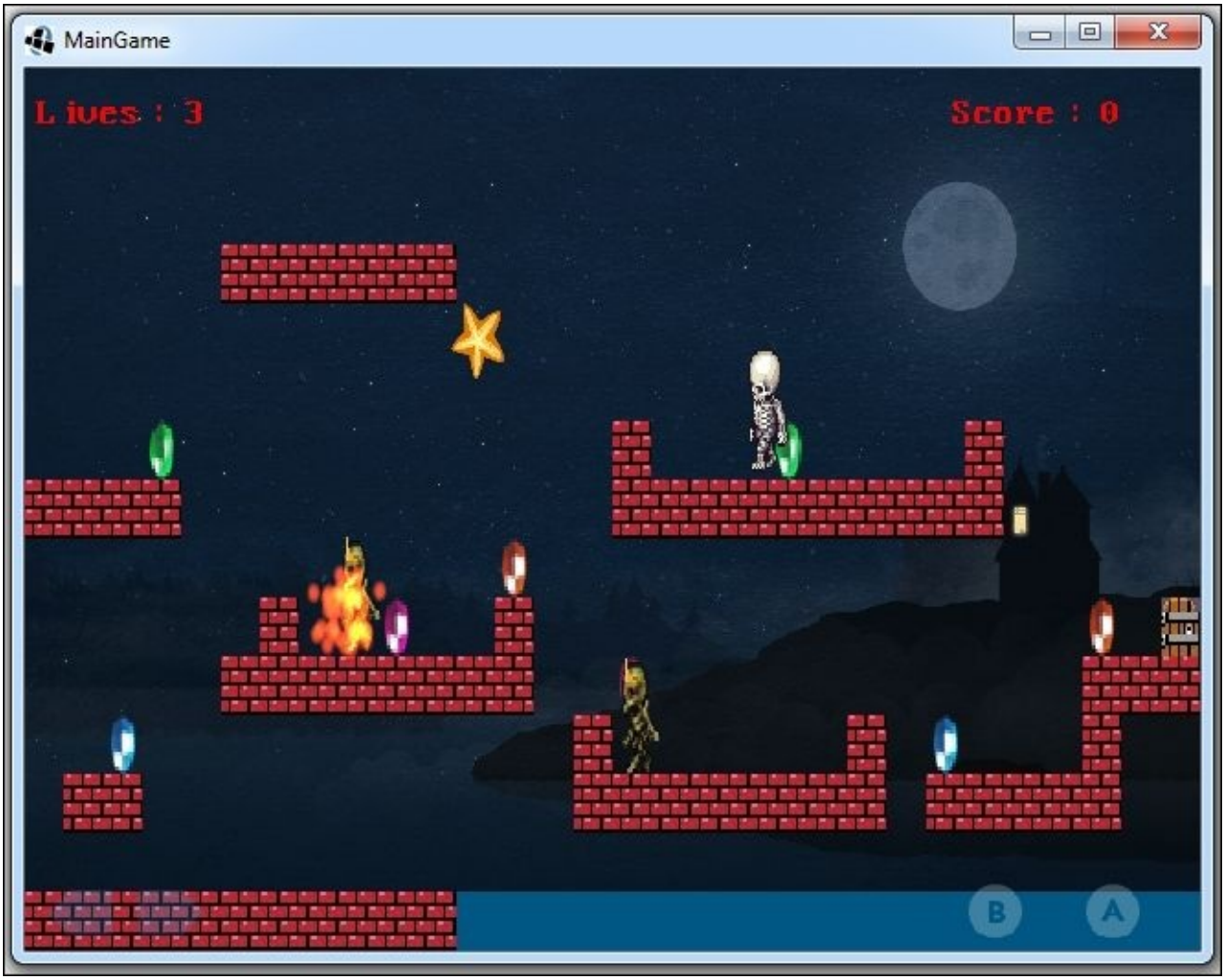
Once we have instantiated the effect, we need to load the particular file of the effect that we created. The `load()` method takes the path of the effect file as the first argument and its parent directory as the second. We then scale the effect to get the appropriate size. To trigger the effect, as we want it after Bob dies, we will update the `killBob()` method, as follows:

```
public void killBob(){
    lifeState=LifeState.DEAD;
    // start respawn counter
    respawnCounter = respawnDelay;
    // set the effect location at Bob's center
    GameManager.explosionEffect.setPosition(bobSprite.getX()+
(bobSprite.getWidth()/2), bobSprite.getY()+(bobSprite.getHeight()/2));
    GameManager.explosionEffect.reset();
}
```

We still haven't displayed the effect in the game. To do this, add the following lines of code to the end of the `GameManager` class' `renderGame()` method:

```
TextManager.displayMessage(batch);

// use main camera while drawing the effect
batch.setProjectionMatrix(GameScreen.camera.combined);
explosionEffect.draw(batch, Gdx.graphics.getDeltaTime());
```

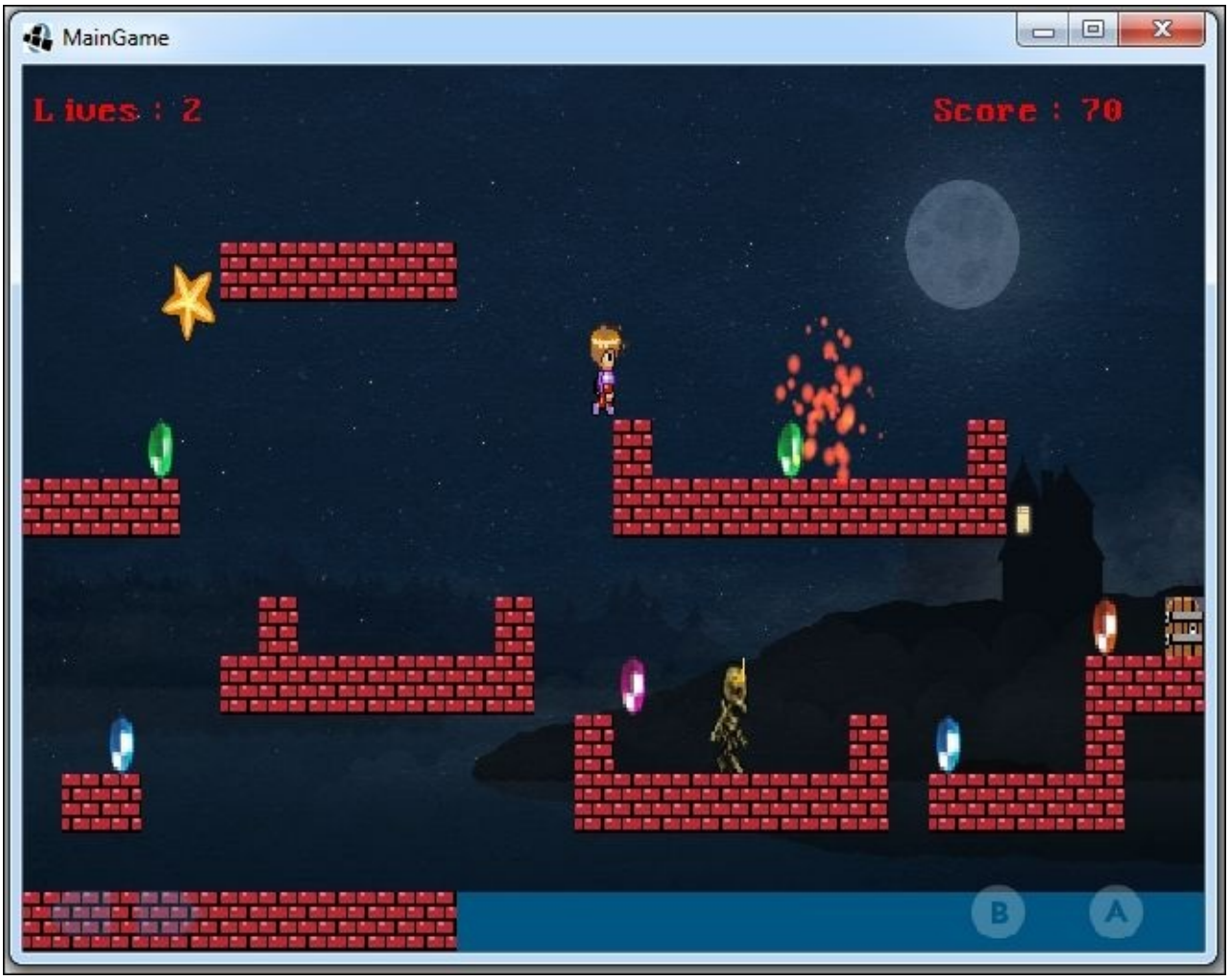
If you run the game now and run into enemies or hazards, you will see something like this:



This effect can also be applied when the enemies are killed. In the `Enemy` class' `update()` method, add the following lines to the end:

```
state=State.DEAD;
GameManager.explosionEffect.setPosition(sprite.getX()+
(sprite.getWidth()/2), sprite.getY()+(sprite.getHeight()/2));
GameManager.explosionEffect.reset();
```

Now, when the bullet hits any enemy, we need to observe the effect:



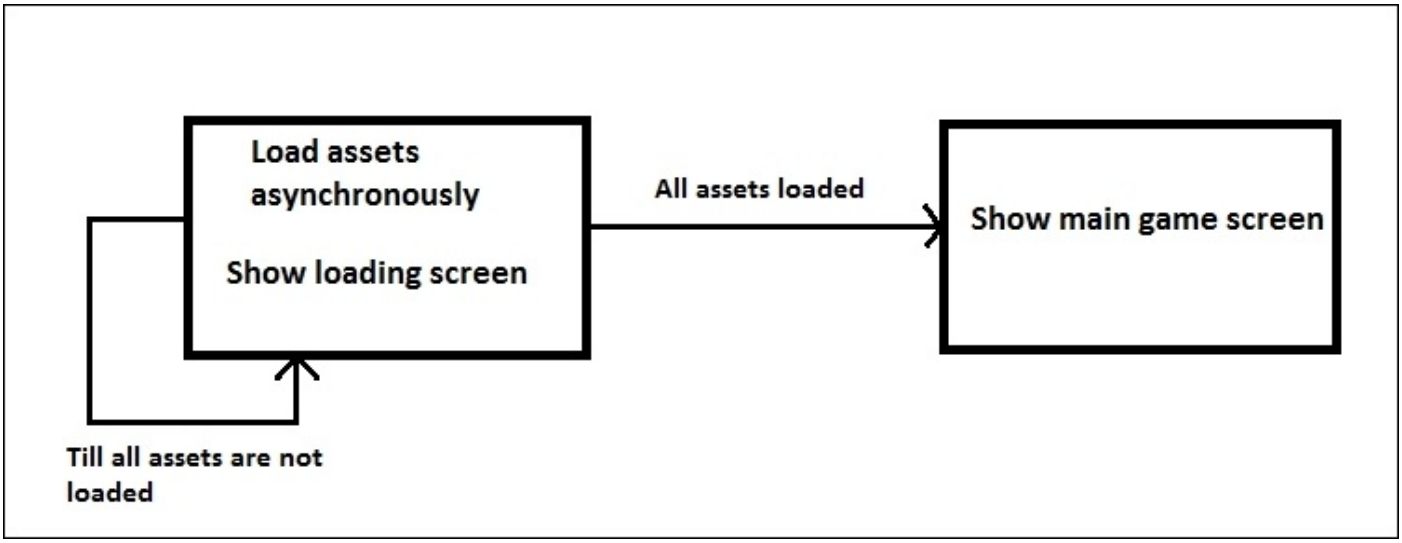
In this case, we are just moving one explosion effect wherever we want. This would not work in cases where there are multiple simultaneous effects. If you’re using the same effect in multiple scenarios, you would need to use object pooling for performance.

The loading screen

We will now learn how to implement a loading screen in the game when we transition between different levels.

Game states

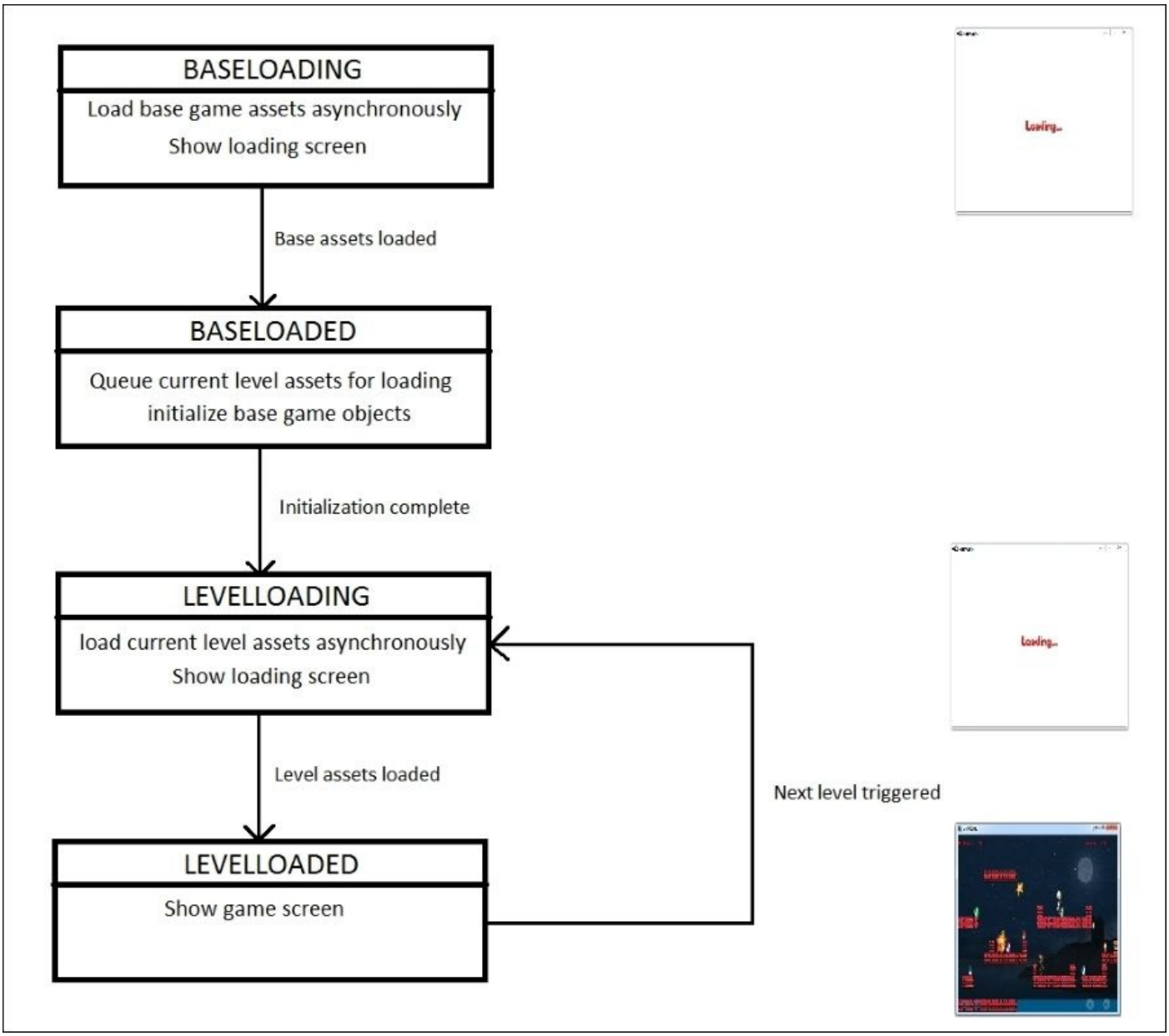
Usually, when a level is being loaded in games, we are presented with a loading screen. This is a strategy used in cases where the number of images or initializing data is so high that it takes some amount of time to load them. This strategy gives time for the game to load while simultaneously providing a visual cue to the user that something is happening in the game. In our case, we are going to asynchronously load assets into the game while showing the loading screen:



In LibGDX, `assetManager` provides the functionality of loading the assets asynchronously. We used the `finishLoading()` blocking method of `assetManager` to load all the assets at once. In this case, we will use the `update()` nonblocking method to asynchronously load the assets. We have four states in the game: `BASELOADING`, `BASELOADED`, `LEVELLOADING`, and `LEVELLOADED`. I have briefly described each state as follows:

- `BASELOADING`: Before this state, we queue all the base assets for loading, which are common to all levels in the game. These include the background image, texture pack, and the fonts. During this state, we load them. We display the loading screen in this state.
- `BASELOADED`: Once all the base assets are loaded, we change the state to `BASELOADED`. In this state, we queue the assets that are specific to the current level for loading. After that, we change the state to `LEVELLOADING`.
- `LEVELLOADING`: In this state, we load all the assets for the current level and display the loading screen.
- `LEVELLOADED`: Once all the level assets are loaded, the state changes to `LEVELLOADED`. In this state, we run the main logic of the game.

The following flowchart describes the different states, as follows:


BASELOADING
Load base game assets asynchronously
Show loading screen
Base assets loaded
BASELOADED
Queue current level assets for loading
initialize base game objects
Initialization complete
LEVELLOADING
load current level assets asynchronously
Show loading screen
Level assets loaded
LEVELLOADED
Show game screen
Next level triggered
Loading...
Loading...

Integrating the screen in the game

Let’s add the states to the `GameScreen` class:

```
public enum GameState{BASELOADING,BASELOADED,LEVELLOADING, LEVELLOADED};

public static GameState gameState = GameState.BASELOADING;
```

Since we would be making a lot of changes to our code (addition/removal/updates), I’ll write all the code for updated functions instead of just describing what to change. First, we will update the `loadLevel()` function from the `GameManager` class:

```
public static void loadLevel(){
    map = assetManager.get(GameConstants.levels[currentLevelIndex]);
    setMapDimensions();
    renderer.setMap(map);
    MapUtils.initialize(map);

    enemies.clear(); // remove current level's enemies
    // spawn enemies from the next level
    MapUtils.spawnEnemies(enemies, Width,Height, texturePack);
    door = MapUtils.spawnDoor(mapWidth,mapHeight);

    GameScreen.camera.setToOrtho(false, mapWidth,mapHeight); // show
specified units horizontally and vertically by the camera
    GameScreen.camera.update();
    GameScreen.gameState = GameState.LEVELLOADED;
}
```

You will notice that the `load()` method calls `assetManager` and the blocking method to load all the assets that have been removed. Here, we are assuming that all the assets will be loaded prior to this method. Next, rename the `loadAssets()`method to `queueBaseAssets()` and update the code, as follows:

```
public static void queueBaseAssets(){
    // queue the assets for loading
    assetManager.load(GameConstants.backGroundImage, Texture.class);

    assetManager.load(GameConstants.texturePack, TextureAtlas.class);
    assetManager.load(GameConstants.fontPath,BitmapFont.class);
}
```

Here, we are queuing the base assets of our game into `assetManager` for loading. This is also the reason why we have removed the `finishLoading()`blocking call. We will add two more methods: one to load the level assets and the other to unload them:

```
public static void  queueLevelAssets(){
    //load the tiled map
    assetManager.load(GameConstants.levels[currentLevelIndex],
TiledMap.class);
}

public static void  unloadLevelAssets(){
    assetManager.unload(GameConstants.levels[currentLevelIndex]);
```

```
}
```

We will also need to add a method, which will be responsible for actually loading the assets from `assetManager`:

```
public static void loadAssets(){
    if(assetManager.update()){
        if(GameScreen.gameState==GameState.BASELOADING){
            GameScreen.gameState=GameState.BASELOADED;
        }
        else {
            loadLevel();
            GameScreen.gameState=GameState.LEVELLOADED;
        }
    }
}
```

This method is going to be continuously called during `render()`. Once the assets are queued for loading, this method is responsible for calling the nonblocking `update()` method on `assetManager` so that it can load the assets incrementally and asynchronously. The call to the `update()` method returns `false` if the assets are still loading.

When they have finished loading, it returns `true`. When all the assets for a particular state are loaded, we move on to the next state. We make the `assetManager` variable of the `GameManager` class public and update the `initialize()` method, as follows:

```
public static void initialize(float width,float height){
    GameManager.Height = height;
    GameManager.Width = width;

    font = assetManager.get(GameConstants.fontPath);
    renderer = new OrthogonalTiledMapRenderer(map,
GameConstants.unitScale);

    GameScreen.camera.setToOrtho(false, mapWidth,mapHeight);
    GameScreen.camera.update();
    // set the renderer's view to the game's main camera
    renderer.setView(GameScreen.camera);

    texturePack = assetManager.get(GameConstants.texturePack); // get the
packed texture from asset manager

    // instantiate the bob
    bob = new Bob();
    // load the bob sprite sheet from the packed image
    bobSpriteSheet = texturePack.findRegion(GameConstants.bobSpriteSheet);
    // initialize Bob
    bob.initialize(width,height,bobSpriteSheet);

    //load background texture
    backgroundTexture = assetManager.get(GameConstants.backGroundImage);
    //set background sprite with the texture
    backgroundSprite= new Sprite(backgroundTexture);
    // set the background to completely fill the screen
    backgroundSprite.setSize(width, height);
```

```
    initializeLeftPaddle(width,height);
    initializeRightPaddle(width,height);
    initializeJumpButton(width, height);
    initializeShootButton(width, height);

    MapUtils.initialize(map);
    TextManager.initialize(width, height, font);

    // instantiate and initialize zombies
    enemies = new Array<Enemy>();

    bullet = new Bullet(width, height, texturePack.findRegion("bullet"));
    bullet.state = Bullet.State.DEAD;
    // set the tiled map loader for the assetmanager
    assetManager.setLoader(TiledMap.class, new TmxMapLoader(new
InternalFileHandleResolver()));

    explosionEffect = new ParticleEffect();
    explosionEffect.load(Gdx.files.internal("data/effects/explosion.p"),
Gdx.files.internal("data/effects/"));
    explosionEffect.scaleEffect(0.2f*GameConstants.unitScale);
}
```

Once we have determined that the level needs to progress, we first unload all the assets pertaining to the current level, as they are no longer needed. We update the level counter so that it points to the next level and queues the next level's assets to be loaded.

In the `GameScreen` class, we add two members for the height and width:

```
float width,height;
```

Now, update the constructor as follows:

```
public GameScreen (MainGame game){
    this.game=game;
    // get window dimensions and set our viewport dimensions
    height= Gdx.graphics.getHeight();
    width = Gdx.graphics.getWidth();
    // set our camera viewport to window dimensions
    camera = new OrthographicCamera(width,height);
    // center the camera at w/2,h/2
    camera.setToOrtho(false);

    batch = new SpriteBatch();
    // set our hud camera's viewport to window dimensions
    hudCamera = new OrthographicCamera(width,height);
    // center the camera at w/2,h/2
    hudCamera.setToOrtho(false);

    GameManager.assetManager = new AssetManager();
    TextManager.initializeLoadingFont(width,height);
    GameManager.queueBaseAssets();
}
```

We also need to make some changes to the `TextManager` class' `initialize()` method:

```
public static void initialize(BitmapFont font){
    TextManager.font = font;
    //set the font color to red
    font.setColor(Color.RED);
    //scale the font size according to screen width
    font.setScale(width/1000f);
}
```

We will also need to add a member to represent the font for the loading screen:

```
static BitmapFont loadingFont ; // we draw the text to the loading screen
using this variable
```

We add a method to initialize that member:

```
public static void initializeLoadingFont(float width,float height){
    TextManager.width = width;
    TextManager.height= height;

    loadingFont = new BitmapFont();
    loadingFont.setColor(Color.RED);
    loadingFont.setScale(width/300f);
}
```

Finally, we add a method to render the loading text:

```
public static void displayLoadingMessage(SpriteBatch batch){
    float fontWidth = loadingFont.getBounds( "Loading…").width; // get the
width of the text being displayed
    float fontHeight =  loadingFont.getBounds( "Loading…").height; // get
the width of the text being displayed
    loadingFont.draw(batch, "Loading…", (width/2)-fontWidth/2,(height/2)
+fontHeight/2);
}
```

Before the game enters the `BASELOADING` state, we queue all the base game assets for loading. The `render()` method of `GameScreen` looks like this:

```
@Override
public void render(float delta) {
    // Clear the screen
    Gdx.gl.glClearColor(1, 1, 1, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    camera.update();
    switch(gameState){

        case BASELOADING:
            GameManager.loadAssets();
            batch.setProjectionMatrix(hudCamera.combined);
            batch.begin();
            TextManager.displayLoadingMessage(batch);
            batch.end();
            break;

        case BASELOADED:
```

```
            //initialize the game
            GameManager.initialize(width, height);
            Gdx.input.setInputProcessor(new InputManager(hudCamera)); //
enable InputManager to receive input events
            GameManager.queueLevelAssets();
            gameState= GameState.LEVELLOADING;
            break;

        case LEVELLOADING:
            GameManager.loadAssets();
            batch.setProjectionMatrix(hudCamera.combined);
            batch.begin();
            TextManager.displayLoadingMessage(batch);
            batch.end();
            break;

        case LEVELLOADED:
            // set the spritebatch's drawing view to the hud camera's view
            batch.setProjectionMatrix(hudCamera.combined);

            batch.begin();
            GameManager.renderBackground(batch);
            batch.end();

            // set the renderer's view to the game's main camera
            GameManager.renderer.setView(camera);
            GameManager.renderer.render();

            batch.begin();
            GameManager.renderGame(batch);
            batch.end();
            break;
    }
}
```

We basically perform different operations based on the current state. As you can see from the preceding diagram, in the `BASELOADING` state, we load the base assets of the game asynchronously through the `loadAssets()` method of the `GameManager` class. While they are being loaded, we show the loading screen to the user. Once the loading is complete (`assetManager.update()` returns `true`), the state changes to `BASELOADED` in the `loadAssets()` method.

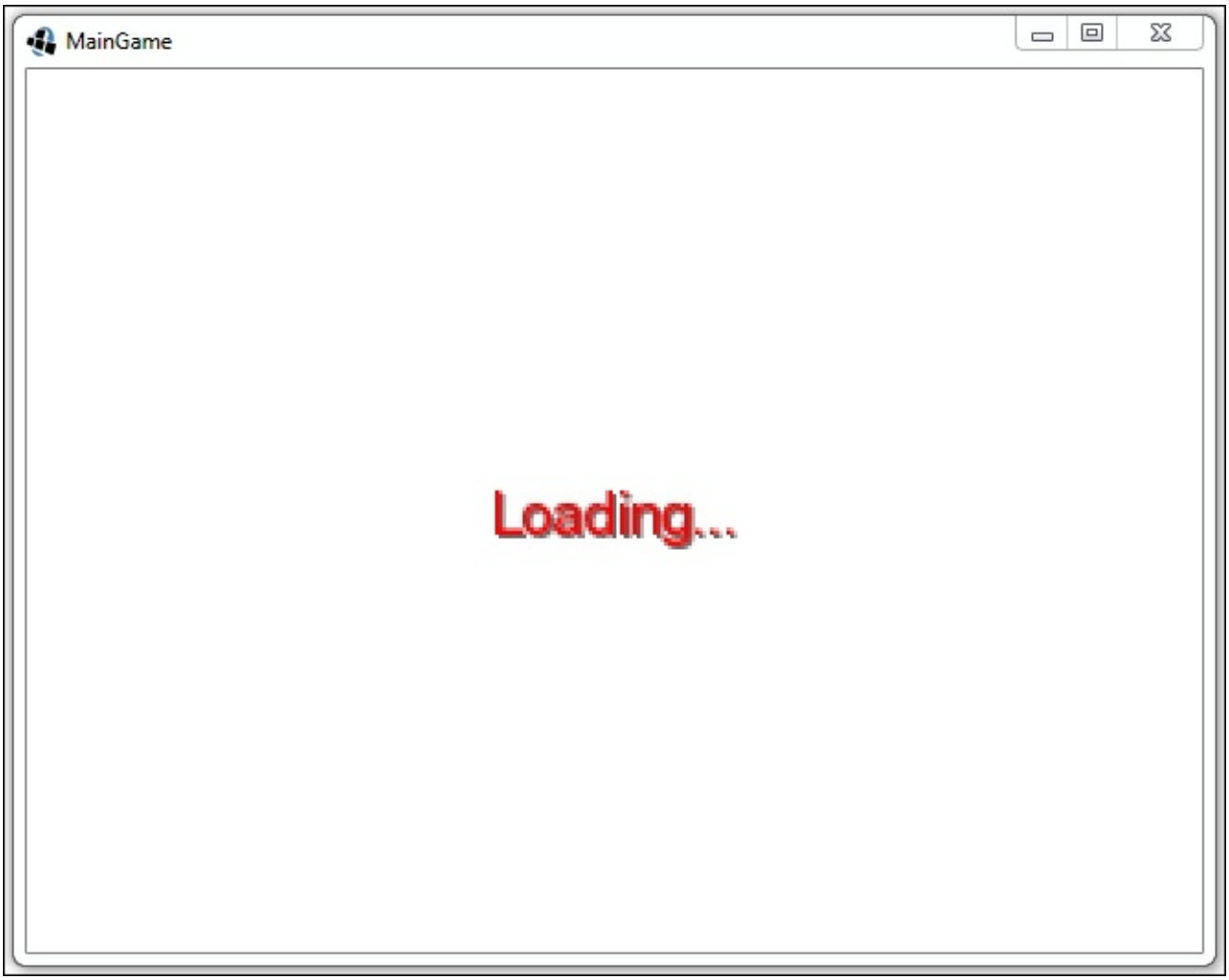
In the `BASELOADED` state, we initialize the base game objects, set the `InputProcessor`, and queue the current level assets. This state is just a staging state for the initialization, after which we change it to `LEVELLOADING`. In this state, we load the current level's assets and change the state to `LEVELLOADED` once done.

In the `LEVELLOADED` state, we run the game's main logic. The state changes to `LEVELLOADING` when the next level has to be loaded. That trigger point is set in the `Bob` class when Bob collides with the door. We update the `Bob` class' `checkDoor()` method as follows:

```
public void checkDoor(){
    bobRectangle.set(bobSprite.getX(), bobSprite.getY(),
```

```
bobSprite.getWidth(), bobSprite.getHeight());
    if(GameManager.door!=null && bobRectangle.overlaps(GameManager.door)){
        // add a print statement to check collisions
        GameManager.unloadLevelAssets(); // unload previous level's assets
        GameManager.currentLevelIndex++; // increase the level counter
        GameScreen.gameState = GameState.LEVELLOADING;
        GameManager.queueLevelAssets();
    }
}
```

When the next level is to be loaded, we unload the previous level's assets as they are not needed anymore. We increase the level counter and then queue the assets of the next level for loading. Run the game now to see the loading screen appear just before level 1 loads and also before level 2 loads:



The loading screen appears for a very short duration of time since we don't have any heavy assets to be loaded in the game.

Summary

This is it. The last section! Although we skipped implementing some of the things, such as playing sounds, saving high scores, and a menu screen in Dungeon Bob, I am sure that with the knowledge gained from the previous chapters, you would be able to implement them with ease. Here are some of the things that I think you can try implementing in the game, just as food for thought:

- A gameover screen
- A winning screen
- Enemies shooting bullets
- Flying enemies
- Bonus levels

We learned game development concepts while making four games in this book. I hope you had fun while reading this book and making games. Keep experimenting and happy coding!

Index

A

- asset management
  - about / Asset management
  - texture packer / Texture packer
  - AssetManager class / The AssetManager class
- AssetManager class
  - about / The AssetManager class

B

- background, Whack-A-Mole style game
  - adding / Adding the background
- ball throwing, Catch the Ball game
  - about / Throwing the ball
  - ball, creating / Making the ball
  - movements, adding / Adding movement
  - gravity, adding / Adding gravity
- basic game screen, Monty hall simulation
  - creating / Making the initial screen
  - Door class, implementing / Implementing the Door class
  - GameManager class, implementing / Implementing the GameManager class
  - Monty class, implementing / Implementing the Monty class
- basic game screen, Whack-A-Mole style game
  - implementing / Making the initial screen
  - Mole class, implementing / Implementing the Mole class
  - GameManager class, implementing / Implementing the GameManager class
  - WhackAMole class, implementing / Implementing the WhackAMole class
- Bob
  - chasing / Chasing Bob

C

- Catch the Ball game
  - moving basket, creating / Making a moving basket
  - ball, throwing / Throwing the ball
  - collisions, detecting / Detecting collisions
  - multiple balls, throwing / Throwing multiple balls
  - score, saving / Keeping the score and maintaining the high score
  - high score, maintaining / Keeping the score and maintaining the high score
  - screens, implementing / Implementing screens
  - sound effects, adding / Adding sound effects and music, Adding sound effects
  - background music, adding / Adding background music
- character animation, Dungeon Bob game
  - about / Character animation
  - walking Bob 1 / Walking Bob 1
  - walking Bob 2 / Walking Bob 2
- collision detection
  - game objects, integrating with Tiled map / Scaling objects and adding a secondary camera
  - realistic physics, adding to game / Physics and collision
- collision detection, Catch the Ball game
  - about / Detecting collisions
  - collision with ground / Colliding with the ground
  - collision with basket / Colliding with the basket
- color, Whack-A-Mole style game
  - adding / Adding some color
- core game classes, Monty hall simulation
  - Door / Summary of classes
  - InputManager / Summary of classes
  - TextManager / Summary of classes
  - GameManager / Summary of classes

D

- Dungeon Bob game
  - player, creating / Creating the player
  - player, moving / Moving the player
  - character animation / Character animation

E

- effects, Whack-A-Mole style game
  - adding / Adding more effects
  - mole, stunning / Stunning the mole
  - stun sign, adding to mole / Adding the stun sign
- enemies
  - adding, to game / Enemies, Adding enemies
  - adding, through Tiled / Adding enemies through Tiled
  - motion, adding / Enemy motion
- event handling / Moving the basket

G

- game objects, integrating with Tiled map and camera control
  - about / Scaling objects and adding a secondary camera
  - Bob, integrating / Integrating Bob
  - camera control / Camera control
  - paddles and background, integrating in game / Integrating game objects
- game score
  - displaying / Displaying the score and adding hazards
- game states
  - about / Game states
  - BASELOADING / Game states
  - BASELOADED / Game states
  - LEVELLOADING / Game states
  - LEVELLOADED / Game states
- game states, Monty hall simulation
  - START / Adding game states
  - CONFIRM / Adding game states
  - END / Adding game states
- Gradle
  - about / Installing the Gradle plugin
- Gradle plugin
  - installing / Installing the Gradle plugin

H

- hazards
  - detecting / Collecting items and detecting hazards
  - adding / Displaying the score and adding hazards
- Hiero
  - about / Custom fonts
  - download link / Custom fonts

I

- installing
  - Gradle plugin / Installing the Gradle plugin
- items
  - collecting / Collecting items and detecting hazards

L

- loading screen
  - implementing / The loading screen
  - game states / Game states
  - integrating in game / Integrating the screen in the game
- logic, Monty hall simulation
  - adding / Adding game logic
  - doors, finding with goats / Finding doors with goats

M

- map rendering
  - about / Rendering maps
  - basic map rendering / Basic map rendering
  - map, reading / Reading the map
  - map objects / Map objects
- moles, Whack-A-Mole style game
  - animating / Animating the mole
  - jumping up and down / Jumping up and down
  - waiting underground / Waiting underground
- Monty hall simulation
  - about / Introduction to the game
  - general workflow / General flow of the game
  - core game classes / Summary of classes
  - basic game screen, creating / Making the initial screen
  - touch/click input, capturing / Taking input
  - logic, adding / Adding game logic
  - game states / Adding game states
  - text messages, displaying / Displaying text and implementing restart, Displaying text
  - restart functionality, implementing / Implementing restart
  - background, displaying / Displaying the background
- moving basket, Catch the Ball game
  - creating / Making a moving basket
  - Basket class, implementing / Implementing the Basket class
  - GameManager class, implementing / Implementing the GameManager class
  - CatchTheBall class, implementing / Implementing the CatchTheBall class
  - basket, moving / Moving the basket
- multiple balls throwing, Catch the Ball game
  - about / Throwing multiple balls
  - after specific intervals / Throwing the balls after specific intervals
  - randomizing / Randomizing and optimizing
  - optimizing / Randomizing and optimizing
- multiple levels
  - creating / Multiple levels
  - door, adding / Adding the door
  - levels, changing / Changing levels
  - Bob, respawning / Respawning Bob

O

- objects
  - collecting / Collecting objects

P

- Particle Editor tool
  - for creating particle effects / Particle effects
  - setup / Editor setup and basics
  - Effect Emitters section / The Effect Emitters section
  - Emitter Properties section / The Emitter Properties section
- particle effects
  - creating, with Particle Editor tool / Particle effects
  - loading, into game / Loading the effect into the game
- physics
  - adding, to game / Adding physics
  - collision detection / Collision detection – 1, Collision detection – 2
  - jump action, implementing / Jumping
- player, creating in Dungeon Bob game
  - Bob class, implementing / Implementing the Bob class
  - GameManager class, implementing / Implementing the GameManager class
  - GameScreen class, implementing / Implementing the GameScreen class
- player, moving in Dungeon Bob game
  - Bob's movement, on desktop / Bob's movement on desktop
  - continuous movement / Continuous movement
  - Bob's movement, on mobile / Bob's movement on mobile
- polling / Moving the basket
- project
  - setting up / Setting up
  - prerequisites / Prerequisites
  - importing / Importing projects

S

- score, Catch the Ball game
  - saving / Keeping the score
  - custom fonts, using / Custom fonts
  - high scores, saving / Saving high scores
- scores, Whack-A-Mole style game
  - keeping / Keeping scores
- screens, Catch the Ball game
  - implementing / Implementing screens
  - menu screen, implementing / Implementing the menu screen
  - screen transitions, implementing / Implementing screen transitions
  - Back button, implementing / Implementing the Back button
  - Back button, catching / Catching the Back button
- setup app
  - using / Using the setup app
  - download link / Using the setup app
- shooting capabilities
  - adding, to Bob / Shooting
- skeletons
  - about / Skeletons
- sounds, Whack-A-Mole style game
  - adding / Adding sound effects
  - WAV format / Adding sound effects
  - MP3 format / Adding sound effects
  - OGG format / Adding sound effects
- star enemy
  - creating / Stars

T

- texture atlas / Texture packer
- Texturepacker-GUI
  - about / Texture packer
  - download link / Texture packer
- Tiled map editor
  - installing / Installing and setting up Tiled
  - URL / Installing and setting up Tiled
  - setting up / Installing and setting up Tiled
  - map, creating / Installing and setting up Tiled
  - map layers / Map layers and drawing
  - drawing / Map layers and drawing
  - erasing / Map layers and drawing
  - layers, show/hide / Map layers and drawing
  - opacity of layer, changing / Map layers and drawing
  - miscellaneous / Miscellaneous
  - custom properties, adding to map / Custom properties
  - objects, drawing / Drawing objects
  - animations, creating / Tile animations and images
  - images, adding to map / Tile animations and images
- touch/click input, Monty hall simulation
  - capturing / Taking input
  - GameManager class, implementing / Updating the GameManager class
  - InputManager class, implementing / Implementing the InputManager class

W

- Whack-A-Mole style game
  - basic game screen, implementing / Making the initial screen
  - color, adding / Adding some color
  - background, adding / Adding the background
  - holes, implementing / Implementing the holes
  - moles, adding in holes / Adding moles in holes
  - moles, animating / Animating the mole
  - randomness, adding / Adding randomness and taking input
  - wait times, randomizing / Randomizing wait times
  - input, capturing / Taking input
  - effects, adding / Adding more effects
  - scores, keeping / Keeping scores and adding sounds, Keeping scores
  - sounds, adding / Keeping scores and adding sounds, Adding sound effects